Guilherme Miguel Teixeira Rito

Bachelor of Computer Science and Engineering

Scheduling computations

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Engineering

Adviser: Hervé Miguel Cordeiro Paulino, Assistant Professor,
NOVA University of Lisbon

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2016

Dissertação apresentada para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Informática, realizada sob a orientação científica de Professor Doutor Hervé Miguel Cordeiro Paulino.

Declaro que esta Dissertação é o resultado da minha investigação pessoal e independente. O seu conteúdo é original e todas as fontes consultadas estão devidamente mencionadas no texto, nas notas e na bibliografia.

O candidato,

Declaro que esta Dissertação se encontra em condições de ser apreciada pelo júri a designar.

O orientador,

Scheduling computations

This document was created unsing the (pdf)LaTeX processor, based in the "unlthesis" template[1], developed at the Dep. Informática of FCT-NOVA [2]. [1] https://github.com/joaomlourenco/unlthesis [2] http://www.di.fct.unl.pt

This thesis is dedicated to my mom, who, in addition to everything else, taught me how to program, to my grandmother who, with an endless patience, taught me how to read and write, to my grandfather who, with intelligence and wisdom, always challenged me with math problems when I was a child, to my grandfather Jaulino for all his unconditional love and support during mine and his hardest moments, to my uncle and aunt for their support and advice, to my cousins, who are awesome!, to my half-brother, to my high school math teacher Giselia Canteiro, and to all my friends, obviously including Minnie and Floppy, who have accompanied me through this long journey, and who always helped and encouraged me to pursue my dreams.

Acknowledgements

This thesis could not have been developed without the assistance and guidance of many people. The most decisive of these was my advisor, Hervé Paulino, for his support and constant availability to listen to my countless half-baked ideas.

Right after my advisor, come João Martins, Joana Páris, Nuno Martins and Tiago Lopes for all their support and precious help in my application for the research internship at Carnegie Mellon University.

I would also like to thank Manuel Sousa Ribeiro, João Leite, Margarida Mamede, Vitor Silvestre, João Leitão, Nuno Preguiça, Rodrigo Rodrigues, João Lourenço, Jorge Cruz, Paulo Lopes, José Maria Gomes and to all the members of NOVA LINCS who gave their insightful feedback on the work presented in this thesis. I would also like to thank Umut Acar and Stefan Muller for their valuable feedback on some of my early ideas for this work.

I would also like to acknowledge the following institutions for their hosting and financial support: *Departamento de Informática* and *Faculdade de Ciências e Tecnologia* of the *Universidade NOVA de Lisboa* (DI-FCT-UNL); the *NOVA Laboratory of Computer Science and Informatics* (NOVA LINCS); and the *CMU Portugal Program*.

Abstract

For quite some time, the Work Stealing algorithm has been the *de facto* standard for scheduling multithreaded computations. To ensure scalability and achieve high performance, work is scattered through processors. In turn, each processor owns a concurrent work queue that uses to keep track of its assigned tasks. When a processor's work queue becomes empty, it becomes a thief and starts targeting victims uniformly at random, from which it attempts stealing tasks. This strategy was proved to be efficient in both theory and practice, and is currently used in state-of-the-art Work Stealing algorithms.

Nevertheless, purely receiver initiated load balancing schemes, such as Work Stealing's, are known not to be suitable for scheduling computations with few or unbalanced parallelism. More, due to the concurrent nature of work queues, even local operations require memory fences that are extremely expensive on modern computer architectures. Consequently, even when a processor is busy, it may incur in costly overheads caused by local accesses to its work queue. Finally, as the scheduler's load balancer relies on random steals, its performance when executing memory bound computations is very limited. Despite all efforts, no silver-bullet has been found, and, even worse, all these limitations still exist in state-of-the-art Work Stealing algorithms.

In this thesis we make three major theoretical contributions, addressing each of the aforementioned limitations. First, we prove that Work Stealing can easily be extended to make use of custom load balancers, that, for various classes of workloads (*e.g.* memory bound computations), can greatly boost the scheduler's performance, while, at the same time, maintaining Work Stealing's high performance for the general setting. Then, we present a provably efficient scheduler that mixes both receiver and sender-initiated policies, and theoretically show that it successfully overcomes Work Stealing's limitations for the execution of computations with few or irregular parallelism. Finally, we present a novel scheduling algorithm, whose expected runtime bounds are optimal within a constant factor, and that avoids most of the costs associated with memory fences, bounding the total expected overheads incurred by memory fences to $O(PT_\infty)$, where T_∞ is the critical-path length of a computation, and P is the number of processors. This contrasts with state-of-the-art Work Stealing algorithms where the total overheads incurred by these synchronization mechanisms can grow proportionally with the total amount of work. From this perspective, our proposal greatly improves the state-of-the-art Work

Stealing algorithm. In fact, as we will prove, for several classes of computations, the overheads incurred by our algorithm are exponentially smaller than the overheads incurred by state-of-the-art Work Stealing algorithms.

Keywords: Scheduling algorithms, Randomized algorithms, Parallel computing, Distributed computing, Dynamic load balancing.

Resumo

O algoritmo *Work Stealing* é considerado, já há vários anos, o *standard* no que toca à execução de computações paralelas. Para garantir escalabilidade e alcançar altos desempenhos, o trabalho é distribuido por processadores. Por sua vez, cada processador tem uma fila de trabalho concorrente, que utiliza para guardar as tarefas que lhe foram atribuídas. Quando a fila de trabalho de um processador fica vazia, este torna-se ladrão e começa a escolher vítimas, de forma uniformemente aleatória, das quais tenta roubar tarefas. Esta estratégia foi provada ser eficiente, tanto em teoria como na prática, e é actualmente utilizada nos algoritmos de *Work Stealing* do estado da arte.

Contudo, estratégias de balanceamento de carga como a do *Work Stealing* em que apenas os recipientes tomam a iniciativa de balancear a carga são conhecidas por não serem adequadas para o escalonamento de computações cujo paralelismo é reduzido, ou mesmo desíquilibrado. Além disso, devido à natureza concorrente das filas de trabalho, até operações locais requerem o uso de barreiras de memória, cujos custos são extremamente elevados em arquiteturas de computadores modernas. Por conseguinte, mesmo quando um processador está ocupado, este pode, frequentemente, incorrer em *overheads* bastante significativos causados por simples acessos locais à sua própria fila de trabalho. Finalmente, como o balanceamento de carga do escalonador se baseia apenas em roubos aleatórios, o seu desempenho enquanto executa computações *memory bound* é bastante limitado. Apesar de todos os esforços, não foi ainda descoberta nenhuma solução que consiga resolver estes problemas e, ainda pior, todas estas limitações existem nos algoritmos de *Work Stealing* do estado da arte.

Nesta tese, fazemos três grandes contribuições teóricas, cada uma endereçando uma das limitações acima referidas. Primeiro, provamos que o *Work Stealing* pode ser facilmente estendido para usar mecanismos personalizados de balanceamento de carga que, para inúmeras classes de computações, conseguem melhorar significativamente o desempenho do escalonador e, ao mesmo tempo, continuar a garantir altos desempenhos para o caso geral. De seguida apresentamos um novo algoritmo de escalonamento que provamos ser eficiente e que utiliza, não só estratégias de roubo, mas também de distribuição de trabalho. Mostramos também, teoricamente, que esta estratégia de balanceamento de carga consegue ultrapassar com grande sucesso as limitações do *Work Stealing*, para computações cujo paralelismo é reduzido ou desíquilibrado. Por último, apresentamos um novo algoritmo de escalonamento para o qual o tempo esperado de execução de computações é óptimo segundo um factor constante, e que consegue ainda evitar a grande maioria dos *overheads* associados às barreiras de memória causadas por acessos locais dos processadores às suas próprias filas de trabalho. Provamos ainda que os *overheads* totais esperados causados por estas barreiras são $O(PT_\infty)$, onde T_∞ corresponde ao comprimento do caminho-crítico de uma computação, e em que P denota o número de processadores. Estes resultados contrastam com o estado da arte de algoritmos de *Work Stealing*, em que os *overheads* causados por estas mesmas barreiras podem crescer proporcionalmente com a quantidade total de trabalho. Nesta perspectiva, a nossa proposta melhora substancialmente o atual algoritmo de *Work Stealing* do estado da arte. Tal como vamos mostrar, para inúmeras classes de computações, os *overheads* incorridos pelo nosso algoritmo são exponencialmente menores, quando comparados com os *overheads* incorridos pelos algoritmos do estado da arte de *Work Stealing*.

Palavras-chave: Algoritmos de escalonamento, Algoritmos aleatórios, Computação paralela, Computação distribuída, Balanceamento dinâmico de carga.

Contents

List of Figures

List of Algorithms

Acronyms

CAS	Compare-and-swap.
CFLFWS	Canonical Flexible Lock Free Work Stealing.
CFSAWSS	Canonical Flexible Synchronous Adaptive Work Stealing and Sharing.
CFWSS	Canonical Flexible Work Stealing and Sharing.
CPU	Central Processing Unit.
CWS	Classical Work Stealing.
dag	Direct Acyclic Graph.
DEQ	Dynamic Equipartitioning.
deque	Double Ended Queue.
FIFO	First In First Out.
GHz	GigaHertz.
LFWS	Lock Free Work Stealing.
LIFO	Last In First Out.
PC	Personal Computer.
semi-private-deque	Semi-private Double Ended Queue.
SLFWS	Synchronous Lock Free Work Stealing.
SPDR Semantics	Semi-Private Deque Relaxed Semantics.
WS	Work Stealing.
WSS	Work Stealing and Sharing.
WSSPD	Work Stealing with Semi-Private Deques.

CHAPTER 1

INTRODUCTION

Since the first programmable computer was invented, much effort has been made towards increasing computers' performance. The earliest commercialized machines were only available to institutions or laboratories with large budgets, and their use was, most of the time, to perform calculations of arithmetic *nature*. As time went by, several companies started acquiring programmable computers to perform tasks that were previously assigned to their employees, greatly boosting their productivity. This wide adoption lead to the creation of computer industry. Together with technology improvements and cost reductions, the birth of this industry occasioned the beginning of the Personal Computer (PC) *era*.

As companies migrated more and more work to these machines, their performance rapidly became the distinguishing property that was taken into account when acquiring new equipments. Consequently, computer builders and sellers started to work towards improving its processing capabilities. At that time, to increase a computer's performance, companies simply had to increase the frequency of the microprocessor leading to gains in the number of executed operations per time unit. Nowadays, Central Processing Unit (CPU)s operate on a frequency in the magnitude of GigaHertz (GHz).

Unfortunately, further scaling the processor's clock frequency is not feasible due to the so called *Power Wall* [PH12]. This physical limitation has forced a gigantic change in what regards to computer processing: instead of continuing to increase the number of operations per second a single processing unit can execute, microprocessors have become multiprocessors.

1.1 A parallel world

As microprocessors reached their limit in what regards to frequency, manufacturers started creating chips with multiple processors. Today, numerous machines now support multiple microprocessors, which in turn have multiple cores. Operating systems are responsible for managing the available resources, scheduling computations to the available processors. However, if applications themselves do not take advantage of parallel architectures, they get stuck at the *Power Wall*. For that reason, when performance is mandatory, CPU bound applications often recur to parallel processing. This way, rather than only depending on the throughput rate of a single processor, they distribute the required computing effort among the available processors.

Today, users are extremely demanding for low response times. In fact, studies have revealed quite interesting results: by injecting small delays in user requests to search engines, the number of daily searches made by the affected users decreased severely [Bru09; Tee+13]. This is a rather important indicator that shows the users' high demands for efficiency: the time for processing received search requests directly and severely impacts the usage of the service. Thus, now, more than ever, developers are compelled to create applications that harness the parallel computing hardware available.

Concurrent programming has become mainstream. To take advantage of modern computer architectures, developers need to focus on both the correction and performance of applications. As the implementation of parallel applications proved to be, most of the time, arduous, several standards have been created aiming to ease software development. Despite all efforts, ensuring both program correctness and performance is still quite challenging: while in sequential programming, developers only have to focus on solving a problem, to use this paradigm, they also have to take into account the correctness and performance of the concurrent programs they are creating. Regarding correctness, while in sequential programming developers only had to ensure the correctness of a single execution flow, in the development of concurrent applications all the possible flows have to be considered, and for each, correctness must be guaranteed.

Ensuring correctness is mandatory: no company wants to provide fast applications that do not behave as they should. As an analogy, consider a team of four engineers that wants to develop a car. To do it correctly, they have to coordinate themselves. Ideally, each one would work on distinct parts of the car, ensuring great performance. However, more important than building the car in a short amount of time is to make it work. Thus, coordination is necessary. If the team simply started working with no communication at all, in the end it could happen that wheels did not fit into the chassis. In the same way, the lack of synchronization could lead to unwanted situations such as finishing to build the car but forgetting to setup the steering wheel. Hence the engineers have to coordinate with each other, ensuring that these kinds of bad situations do not occur. Nonetheless, synchronization costs can be very expensive when non adequate scheduling decisions are taken: If the whole team had to wait for a single worker that was taking too long to complete his assigned tasks, their productivity would be sorely reduced. Similarly, developers have to ensure that parallel programs not only are correct but also efficient.

1.2 Performance of parallel computations

Achieving high performance when executing parallel programs is, often, challenging. To develop efficient applications, one has to ensure proper load balancing among the workers, avoiding situations in which a single processor causes everyone else to stall, waiting for the former to conclude its part of the job. Before continuing, we first need to understand what types of parallelism exist and their properties.

Usually, parallelism comes in two forms: data or task (*a.k.a.* functional) parallelism. In the second case, tasks are distributed among workers and are the minimum execution unit. Usually, each task has a different purpose. Recalling the previous analogy, task parallelism is equivalent to the task distribution for the engineers building the car. In contrast, data parallelism means the data itself is spread across the workers, which in turn perform the same operations, but over different data sets. In the car development analogy, it would correspond to assign to each engineer the construction of a different wheel: all of them would perform the sequence of operations, but each would build a wheel from his assigned feedstock. We now move to explain what types of scheduling decisions are associated with these types of parallelism.

1.2.1 Static scheduling

The easiest way to schedule data parallel computations is by simply partitioning data into smaller sets, which are then evenly distributed through processors. Many tools and frameworks support this scheduling methodology [DE98; GNU16]. As a matter of fact it has proved to be extremely efficient for uniformly grained data parallel computations. However, this very particular type of computations is only a small subset of the ones for which scheduling algorithms are required. Indeed, in several cases data parallel computations are not uniform: even though processors execute the same task over different inputs, it may take disparate processing time for computing two inputs with the same size. For these scenarios, dynamic scheduling algorithms are more efficient, as explained next.

1.2.2 Dynamic scheduling

The main motivation for dynamic strategies is related to the fact that it allows processors to load balance *online* (*i.e.* during the execution of a computation). This is a great advantage when considering applications with non uniform work distributions, such as most task parallel applications and numerous data parallel computations. A simplistic approach to dynamic scheduling is to create a single task queue at the beginning of a computation's execution, from which workers fetch and further execute tasks. In fact, some libraries and language extensions implement this exact approach [DE98].

However, this simplistic model has numerous disadvantages: a single shared data structure does not allow the number of workers to scale, as accesses to the structure have to be serialized. To overcome this limitation, the task bucket got partitioned through workers and new algorithms based on this idea were developed. Of these, some performed load balancing by means of task spreading (commonly referred to as *sender-initiated* strategies) while others assigned the responsibility of load balancing to idle workers (also known as *receiver-initiated* strategies).

1.3 Work Stealing

Much research was made towards creating the most efficient and scalable computation scheduler. For some time now, the Work Stealing (WS) algorithm [BL99] has shown to perform quite well: it avoids communication costs as much as possible while providing load balancing when necessary.

The main idea of work stealing is that each worker (also referred to as processor) maintains a pool of tasks that are assigned to it. Whenever the execution of an instruction by a worker leads to the creation of a new task, the worker adds this task to its pool. On the other hand, when a processor finishes executing some task, it attempts to fetch a task from its pool. If the pool is empty, the worker becomes a *thief* and starts trying to steal work from other processor pools. The *victims* (*i.e.* the targeted processors) are picked uniformly at random, as studies have proved that it is a provably efficient policy [BL99; Aro+01]. As one can conclude, this algorithm is optimal in the sense that it avoids, as much as possible, communication among workers.

1.3.1 Drawbacks of work stealing

The wide adoption of WS led to the identification of some limitations. First, for some classes of workloads it can be the case that most, if not all, of the work is generated by a single processor. In these scenarios, even if the worker is continuously being stolen, the work generation can always be higher than the steal rate. After some time, that processor's pool becomes a bottleneck, as every other worker is stealing tasks from it [HS02a].

Another identified limitation is related to modern computer architectures. It has been shown that due to the lack of data locality, WS does not perform as efficiently as possible when executing memory-bound computations [Aca+02].

Additionally, for applications in which there is few parallelism, workers are, most of the time, attempting to load balance. Imagine the scenario in which only a small subset of workers is actually working. Then, every other worker would be continuously performing steal attempts. Moreover, most of these steal attempts would target processors with empty task pools, and consequently be useless. It has been shown by Eager *et al* that in scenarios in which there is few parallelism, it is preferable to opt for sender initiated load balancing strategies [Eag+85]. Also according to that work, for cases in which there is plenty parallelism, receiver initiated load balancing mechanisms are more appropriate.

1.4 Proposal

We propose a flexible work stealing algorithm that allows custom load balancing policies to be used while still providing equivalent guarantees as for WS. This flexibility aims to allow custom load balancing techniques, that are effective for certain types of workloads, to be designed, greatly improving the scheduler's performance when executing such computations. For example, processors, when become idle, may try to steal work back from their latest thief, in an attempt to increase data locality.

Additionally, we propose a provably efficient algorithm (based on the former) that combines *sender-initiated* and *receiver-initiated* strategies, allowing processors to search for work when they are idle, and, at the same time, attempting to distribute generated tasks to idle processors, whenever there is few parallelism. The associated spreading mechanism is also flexible, leaving room for further optimizations. For example, one may choose to distribute generated tasks among neighbor processors, increasing data locality. Another possibility is to randomly spread generated tasks, evading situations in which a processor's work pool constantly grows and inevitably becomes a bottleneck.

Finally, we propose a variation of the work stealing algorithm that avoids most of the overheads caused by the use of expensive synchronization mechanisms when processors access their own work pools.

1.5 Contributions

The major contributions of this *thesis* are:

1. A proof that the *state-of-the-art* WS algorithm can be extended to support custom load balancing algorithms while still being provably efficient.

2. A novel method to analyze the performance of scheduling algorithms.

3. A synchronous algorithm that combines sender-initiated and receiver-initiated load balancing schemes, that is provably efficient and whose performance is strictly better than the performance of modern WS algorithms. The analysis of that synchronous algorithm which we claim to be the first one that is both provably efficient and that can achieve stability, depending on the ratio of idle processors.

4. An asynchronous scheduler, also combining both sender-initiated and receiver-initiated load balancing schemes, that is provably efficient, and, in addition, also supports custom load balancers.

5. An asynchronous scheduler that overcomes the problem related to the overheads incurred by processors when accessing their own work pools.

All these contributions will be carefully detailed throughout the thesis.

1.6 Document structure

The remaining part of this document is organized as follows: in Chapter 2, we detail the main WS algorithms. Then, we present the related work, together with the problem each contribution is trying to solve and the associated drawbacks. In Chapter 3, we establish some fundamental mathematical results that will be used through the rest of the thesis. We begin the following chapter (*i.e.* Chapter 4) by carefully describing the fundamental concepts, assumptions and conventions related to the scheduling of parallel computations. Next, in Chapter 5 we present a synchronous scheduling algorithm, and show it is provably efficient. After that, we obtain numerous results that will be used not only to compare the performance of our proposed algorithm against WS, but, also to claim that our algorithm is the first provably efficient one that can achieve stability. Chapter 6 focus on addressing real world problems. We start by presenting an asynchronous algorithm that uses both receiver and sender initiated load balancing mechanisms, and that can even be extended to support receiver and sender initiated custom load balancing mechanisms. Then, we present a variant of WS that avoids most of the overheads related to the expensive synchronization mechanisms incurred by processors when these access their own work pools. Next, we exemplify how the flexibility of our scheduler can be used in order to overcome a known limitation of WS, caused by its locality obliviousness Finally, in Chapter 7, we sum up the results achieved with our study and describe possible future directions emerging from this work.

Chapter 2

Related work

Throughout this chapter we detail numerous variations of WS, their contributions and limitations. To acquaintance the reader with all the background terminology, we start by briefly defining how a computation is executed. Next, we describe the first provably efficient WS algorithm. We further characterize the *lock-free* version of that same algorithm and its contributions. After that, we move to the description of all related work, the problems addressed, their solutions and the limitations of each approach. We conclude the chapter by discussing what properties our solution must have in order to successfully address the identified limitations.

The Work Stealing algorithm is a fully distributed *online* scheduler [BL99]. It has been proved to be an efficient strategy for scheduling fine-grained parallel computations [Aca+13b]. Multiple libraries such as *Wool* [Fax09], *Hood* [BP99], *Intel TBB* [KV07] and *Pasl* [Aca+16], languages like *Cilk* [Blu+96] and *Cilk++* [Lei09] and language extensions such as *pSystem* [LS94; Sil+99] implement this algorithm.

2.1 Background

To truly understand WS, we first have to detail how a computation is executed. A computation is represented as a Direct Acyclic Graph (dag) $G = (V, E)$. Each node in V corresponds to a single instruction, while each edge in E represents an ordering constraint between two instructions. We denote the number of processors used to execute a computation by P. The total number of nodes within a dag is expressed by T_1 and the length of a longest path, also within the graph, by T_∞. An edge from a node α to another node α' represents an ordering constraint, meaning that α' only executes after α. The *in-degree* of a node is the total number of edges pointing to it. In the same way, the *out-degree* of a node corresponds to the total number of edges starting at that node.

Nodes with *in-degree* of 0 are referred to as *roots*, while nodes with *out-degree* of 0 are called *sinks*. A node is said to be *ready* if and only if it has not yet been executed while all of its ancestors already have. To respect ordering constraints, at each step only *ready* nodes can be executed. Consequently, in the beginning of a computation's execution, only the *root* nodes are *ready*. As the execution progresses some nodes get executed, possibly enabling one or more of its successors. When all the *sinks* of a computation get executed, the execution terminates.

A *thread* is a contiguous sequence of nodes within the dag, connected by a chain of edges. The first node within a thread Γ (*i.e.* the node in which the chain begins) is referred to as the *root* of Γ. The last node of a Γ is said to be the *sink* of Γ. If some thread Γ_0 creates a child Γ_1, we say Γ_0 *spawned* Γ_1. Furthermore, the edge starting at Γ_0 and ending at the *root* of Γ_1 is referred to as a *spawn edge*. If a thread Γ_0 waits for a dependency from another thread $\Gamma 1$, we say Γ_0 *joins* with Γ_1. In this case, the edge starting at some node within Γ_1 and pointing at some node of Γ_0 is said to be a *join edge*. The notion of a *thread* is just syntactic sugar.

The execution of a computation can be partitioned into steps: at each step, a processor executes exactly one instruction. Each node of a computation's dag corresponds to one instruction. Thus, if the number of processors executing a computation is P, then, at most P nodes get executed at a single step. Decisions related with the dag's execution depend on the scheduling algorithm in use.

2.2 The classic work stealing algorithm

Blumofe *et al* presented the first provably efficient scheduler [BL99]. Nonetheless, the proposed algorithm constrains the computations in several ways. First, it is assumed that the execution of an instruction (*i.e.* a node of the dag) can enable at most two other nodes (*i.e.* a single node can spawn at most one thread). Second it obligates computations to be *fully strict* (*i.e.* well-structured). A *fully strict* dag is one in which every join edge starts at some thread Γ_1 and finishes at its parent thread Γ_0 (implying that Γ_0 had spawn Γ_1 previously). It is also assumed that a parent thread cannot immediately stall after spawning a child (*i.e.* a node spawning a new thread also enables a new node within its thread). Another assumption is that a parent thread must stay alive at least until all of its children die. Finally, the computation only has one root and one sink. As a consequence of previous assumptions, both root and sink belong to the same thread.

Each of the P processors executing a computation owns a blocking Double Ended Queue (deque). Processors operate on their deques under a Last In First Out (LIFO) policy, pushing and popping threads from the bottom side. Whenever the execution of some node spawns a new child, the processor pushes its currently executing thread into the bottom of its deque and starts executing the child. If the execution of a node does not enable any new nodes, the processor pops the *bottommost* thread from its deque and commences work on it. In the case that its deque is empty, the processor becomes a *thief* and starts targeting, uniformly at random, *victims*. It then tries to pop the *topmost* thread from the *victim's* deque. If the steal attempt is successful the *thief* starts working on the stolen thread. Otherwise, if it could not fetch any work, the processor repeats the whole stealing procedure, picking a new *victim*.

The space used by a P processor execution of any fully strict computation under this version of WS algorithm is bounded by $S_1.P$, in which S_1 denotes the minimum sequential execution requirements.

The expected number of steps to execute a fully strict computation under the WS using P processors is at most $\frac{T_1}{P} + O(T_\infty)$. Moreover, for any $\epsilon > 0$ with probability at least $1 - \epsilon$ the execution time is $\frac{T_1}{P} + O\left(T_\infty + \log(P) + P.\log\left(\frac{1}{\epsilon}\right)\right)$. Taking into account that an optimal schedule takes $\max\left(T_\infty, \frac{T_1}{P}\right)$ steps to execute, we conclude that WS is optimal within a constant factor when executing fully strict computations.

In what regards to expected communication costs, the total number of bytes communicated between the P processors executing a fully strict computation under the WS algorithm is $O(P.T_\infty.(1+n_d).S_{\max})$, in which n_d is the maximum number of join edges from a thread to its parent, and $S_{\max}$ is the size of the largest thread in bytes. As in the execution time bounds, for any $\epsilon > 0$, with probability at least $1 - \epsilon$ the total number of bytes communicated are $O\left(P.\left(T_\infty + \log\left(\frac{1}{\epsilon}\right)\right).(1+n_d).S_{\max}\right)$.

We will refer to this version of the WS algorithm as the Classical Work Stealing (CWS). It is used in *Cilk* [Blu+96] and *Cilk++* [Lei09].

2.3 The Non-blocking Work Stealing algorithm for multiprogrammed environments

A new version of WS was later proposed by Arora *et al* [Aro+01; Aro+98]. The main difference of the algorithm when compared with CWS is that, instead of using blocking deques, it uses *lock-free* ones [Blu+99]. One of its major contributions consists on the proof that WS is optimal within a constant factor under multiprogrammed environments, regardless of the underlying operating system's scheduler used. This is a rather important result as previous studies only took into consideration fully dedicated environments. Another addition of this paper is that the bounds are valid to a more broad set of computations. Concerning the proofs, a novel approach was introduced and has been widely adopted when proving bounds on variations of this algorithm [Aca+02; Aca+13b; Tch+10; MA16].

As in the previous case, it is assumed that each node of the dag represents a single instruction, implying that it takes exactly one step to execute a node. Only two assumptions are made related with the dag's structure: (i) Computations can only have one root and one sink; (ii) Every node within the computation has an *out-degree* of at most two.

The first constraint means that a computation has an unique start node and an unique finish node. Regarding the second premise, note that each node in the dag represents an instruction, and thus, the assumption is consistent with the computation's model.

As one of the objectives of this work was to prove bounds on multiprogrammed environments, some new nomenclatures were introduced: it is assumed that processors are scheduled to execute the computation in *rounds*, rather than in steps, where a round corresponds to a contiguous sequence of steps. The reason for this assumption is related to the environment in which the algorithm operates: it ensures the kernel scheduler does not *bias* steal attempts made by the processors.

Using P processors, the expected time to execute a computation under this version of WS is $O\left(\frac{T_1}{P_a} + \frac{P.T_\infty}{P_a}\right)$, where P_a represents the average number of processors scheduled for each round. Moreover, for any $\epsilon > 0$, with probability at least $1 - \epsilon$ the execution time to execute a computation under this version of WS with P processors is $O\left(\frac{T_1}{P_a} + \left(T_\infty + \log\left(\frac{1}{\epsilon}\right)\right).\frac{P}{P_a}\right)$.

In contrast to the CWS, no space nor expected communication bounds have been proved. We denote this version of WS as the Lock Free Work Stealing (LFWS).

2.4 Related work

Much work has been carried out to improve the performance of WS algorithms. One of the main issues that has been extensively studied is the lack of data locality when work stealing. Other topics that have been subject of study are related with the costs of expensive memory fences in modern weak-memory architectures, efficient load balancing techniques for certain types of workloads, and the costs and gains of different work policies.

2.4.1 Data locality

Handler *et al* suggest a completely different approach to increase data locality when scheduling computations [HS02b]. The main idea behind their proposal is to change the way in which processors keep track of their assigned threads. Instead of using deques, each processor is assigned with P work piles. Each work pile w_i within the structure owned by some processor p_j contains a set of tasks that were sent by another processor p_i. The main objective of this study is to allow flexible scheduling strategies that could increase the performance of the scheduler. Furthermore, its load balancing mechanism ensures a *wait-free* property when accessing work piles; for each work pile, there is only a single producer and another single consumer. However, this approach does not behave efficiently when processors have no assigned tasks, as they simply stall. Furthermore, to search for a task to execute, a processor might have to search through all its P processor piles, and thus, not scaling. Finally, as P^2 work piles are created, the system also does not scale with the number of processors due to its space requirements.

Acar *et al* [Aca+02] have carefully studied the data locality of WS and proposed a novel approach to overcome the issues of previous solutions. To achieve better data locality, the scheduler attempts to ensure that processors work on data in which they have already worked on. This heuristic helps processors reusing data that has already been fetch onto their caches. To achieve this objective, each processor keeps track of the threads that have affinity with the data on which it already worked upon, using a mailbox. Whenever some processor generates work that is assigned to another one, it not only pushes the created thread to its deque, but also puts a pointer to the thread in the other processor's mailbox. It has been shown that this policy can achieve substantial performance gains. However, their study lacks bounds on the expected execution time, as well as space requirements. This is due to the delays of inserting threads into the mailbox, which are not negligible. Furthermore, it could happen that a processor's mailbox was filled in with already executed threads. This issue could cause severe problems for some computation classes, as the mailbox could grow up to $O(T_1)$. Finally, as work is duplicated, synchronization mechanisms are necessary to ensure correctness, avoiding situations in which the same thread is executed twice.

An interesting proposal towards reducing steal costs in WS algorithms was to perform steals in which *victims* were picked according to their distance from the *thief* [QW10]. Even though this approach was first meant to improve load balancing on distributed environments, it has been studied as a way to increase data locality when load balancing [Suk+16]. The rationale behind this idea is to bias the probabilities for targeting each victim, such that they are inversely proportional to the distance between each possible target and the *thief*. However, no theoretical bounds were provided to this version of the algorithm. Furthermore, to the best of our knowledge, no clear evaluation results show the benefits of using this heuristic to increase data locality in multiprocessor environments.

Guo *et al* [Guo+10] propose an adaptive WS that groups processors sharing some kind of affinity. The two main contributions of their work are the proposal of an algorithm that dynamically adapts its scheduling policy and a locality aware load balancing mechanism. The evaluation of their approach shows speedups up to 2.6 times when comparing their locality aware mechanism with CWS. In contrast to previous contributions, rather than taking advantage of temporal locality, their solution attempts to improve spacial locality by joining processors in groups, sharing some affinity between them. To these groups the authors call *places*. Each of these *places* owns a mailbox. If a thread (*i.e.* a task) with affinity to some *place* is created, it is sent to the corresponding mailbox. When a processor has no assigned tasks, it starts work-stealing, but targeting only processors that are in the same *place*. If there are no threads in any of its neighbors' deques, it then checks the mailbox of the group. The version presented in the work does not allow steals of threads across processors in distinct *places*. For that reason, in most workloads the solution does not scale properly: it could happen, in an extreme scenario, that all the work was assigned to a single group.

Other approaches aiming to increase data locality are suggested by Chen *et al* [Che+11; Che+12]. The first proposal attempts to provide a provably efficient and locality aware load balancing scheme for multi-socket architectures. The presented solution partitions the dag into two parts: one corresponds to the inter socket computation while the other is a collection of small dags that are executed by the processors of a single socket. These subcomputations allow to increase data locality as the processors of a single socket share some affinity between them (after all, they are all within the same chip!). Groups of processors sharing the same socket are referred to as *places*. A deque is assigned to each *place* in order to keep track of inter socket threads. The inter socket part of the computation is executed by one processor of each chip. To keep track of intra socket tasks, each processor within a *place* also has an assigned deque. Processors only target *victims* within the same group, increasing data locality. Each *place* has a head processor that, whenever detects all the processors within the group idle (*i.e.* work stealing), pops the bottom most task from the bottom of the deque assigned to the *place*. However, in the theoretical evaluation, the authors often take as granted results that are not trivial to achieve at all.

In the second approach, also by Chen *et al*, instead of partitioning the dag at the beginning of the execution, the authors use an *online* partitioning tool [Che+12]. The objective of this work is to take advantage of processor caches: instead of simply dividing the dag into smaller dags, the cache size of each socket is taken into account, and, together with profiling statistics, the computation is partitioned such that each part of the dag fits in the cache of a socket. The only difference between this approach and the previous one is related to this *online* profiling mechanism. The authors show substantial performance gains, with speedups up to 74.4% for certain memory-bound computations when compared with the CWS. The authors rely on the proofs from their previous work to bound the execution time when using this algorithm. However, not only the previous proofs were not clear, but this new approach also is not equivalent to the previous one: the dag partitioning is now made in an *online* fashion. Furthermore, the profiling mechanism also was not used in the previous proposal. The overheads related to this profiling phase are not taken into account in the presented time bounds. Moreover, the authors do not take into account the overheads of *online* partitioning the dag, not even mentioning them.

Acar *et al* [Aca+15] have suggested to couple memory and computation management for increased data locality. In this work, only purely functional programming languages are considered, as they provide useful abstractions (*e.g.* memory disentanglement). The study is aimed to workloads consisting of nested parallel computations, like a recursive multiplication of matrices. The rationale behind this approach is to take advantage of memory disentanglement in *divide-and-conquer* approaches so that the computation's heap is partitioned hierarchically. The study also aims to facilitate the parallel garbage collection of heap chunks allocated to threads. However, no benchmarks nor any theoretical bounds were established in the report.

Recently, Suksompong *et al* propose a variant of WS which they denote by *localized work-stealing algorithm* [Suk+16]. The execution of a computation using this variant of WS has an expected execution time of $\frac{T_1}{P} + O(T_\infty.P)$. The strategy used to increase locality is somewhat related with the one proposed by Acar *et al* [Aca+02]: to ensure a processor works as much as possible in the same data, it makes *steal-backs* to recover work that was stolen from it. The computation is represented as a tree, allowing the authors to distribute work more easily: a new tree is created over the first one, in which the leaves represent each of the P processor's. Each processor then works on the subtree starting at its leaf, increasing data locality. When the subtree of some processor p gets fully executed, p starts work-stealing as in the classical algorithm. Many strategies were proposed to perform these steal-backs, however none had yet been formally studied. The major drawbacks of this proposal are related to the structure assumptions made to computations, which, as stated by the authors, are much less general than dags, and that its execution time bounds are not optimal.

Even more recently, Muller *et al* [MA16] have addressed the problem of scheduling latency bound computations. Their work assumes that threads can block, being the first handling both CPU and latency bound computations. The expected runtime to execute one such computation is $O\left(\frac{T_1}{P} + T_\infty.U.(\log(1+U))\right)$, where U represents the maximum suspension width. An interesting fact to note is that T_1 does not include the time during which threads were suspended. In other work [Mul+], Muller *et al* propose a programming language that not only aims to schedule computations efficiently, but also to ensure responsiveness, by handling requests as fast as possible.

2.4.2 Efficient Scheduling for parallel computations

Most work related with the scheduling of parallel computations aimed to improve, in one way or another, the performance of scheduling strategies. However, in most, if not all, the scenarios, there is not sufficient parallelism to keep all the processors busy during the whole execution of a computation. For example, in scenarios where there are not enough ready threads to keep all the processors busy, *thieves* waste numerous processing cycles making useless attempts to load balance. For that reason, a great effort has been made towards reducing wasted processor cycles [Agr+06; Agr+07; Agr+08; Sun+11].

An early theoretical approach, by Agrawal *et al*, studies the behavior of an adaptive greedy scheduling algorithm [Agr+07]. At each round q, the thread scheduler lets the kernel's job scheduler know the number of processors it desires for that round, d_q. The desired number of processors for some round q is calculated based on the desire of the previous round d_{q-1} together information related to round $q-1$. The authors prove that a *greedy* scheduler can achieve nearly optimal speedups: $O\left(\frac{T_1}{\widetilde{P}} + T_\infty + L\log(P)\right)$, in which $\widetilde{P}$ denotes the trimmed availability $O(T_\infty + L\log(P))$ and L denotes the number of steps of a round. Furthermore, the authors also prove that at most $\rho.T_1$ processor cycles are wasted during the execution of a computation.

In subsequent work, Agrawal *et al* study the behavior of WS in that same environment [Agr+08]. The authors propose a variant of WS that automatically adapts to the available parallelism in the computation. They further prove that a computation executed under the WS algorithm has expected execution time bounded by $O\left(\frac{T_1}{\widetilde{P}} + T_\infty + L\log(P)\right)$, that is the exact same bound as for the *greedy* approach. Moreover, it is shown that the wasted number of cycles when using WS scheduler with parallelism feedback is $O(T_1)$ in the worst possible scenario.

Sun *et al* [Sun+11] propose a variant of the previous approach by Agrawal *et al*, arguing that the instability related with the computation of the desired number of processors for each time step can lead to unnecessary scheduling overheads. The authors propose an adaptive scheduler *ACDEQ* that, taking into account the parallelism feedback by an adaptive controller *A-CONTROL* and using a well known algorithm (Dynamic Equipartitioning (DEQ)), allocates processors for each step of the computation's execution. The main contribution behind this study is a two level scheduling algorithm that guarantees control related properties and algorithmic ones as well. As part of this work is related to job scheduling, rather than the way threads are scheduled, within each job, to the available processors, we do not further detail its properties.

2.4.3 Cost of memory fences

In modern computer architectures, the costly memory fences deteriorate the performance of computations executed under WS [Aca+13b; CL05; Hen+06; Mic+09; MA14]. The reason why these operations are harmful for WS implementations is due to the fact that processors work on their deque as it was a call stack. In various scenarios, they push and pop threads at extremely high rates, meaning that any overheads incurred on deque operations can severely limit the performance of schedulers. For that reason, numerous proposals have been made, attempting to avoid these operations as much as possible.

Chase *et al* propose a lock free deque that removes, as much as possible, memory fences for private accesses to the structure [CL05]. Hendler *et al* extended that proposal adding the possibility of changing the size of the deque dynamically [Hen+06]. However, Attiya *et al* proved that it is impossible to totally avoid these memory fences in the implementation of lock free data structures (*e.g.* deques) while still maintaining correctness [Att+11].

Michael *et al* propose a relaxed version of worker's data structures that provides *at-least-once* (alternatively to *exactly-once*) semantics [Mic+09]. Instead of ensuring each task is stolen by a single processor, they show that it can be beneficial to relax this property allowing duplicate executions of the same task. An immediate consequence of this scheme is that it can only be used for scheduling idempotent computations. Even though in certain cases this approach might behave better than the regular one (with *exactly-once* semantics), for certain types of workloads it can become quite inefficient: if a thread with a great amount of work gets stolen multiple times, the computation's execution could also take much longer, as a part of the work would have to be executed multiple times.

Tzannes *et al* [Tza12] proposed an algorithm where each processor keeps most of its tasks private in a deque, except for the topmost, that is kept in a shared memory cell. However, Tzannes' algorithm showed severe limitations for various types of computations in which processors had, most of the time, the private part of their deque empty, implying they would constantly have to incur in expensive synchronization costs, ensuring that the topmost task of the deque could be stolen at any moment. Moreover, the study did not analyze the expected runtime bounds of computations using their proposed algorithm, which, for the best of our knowledge, remain unknown.

Acar *et al* propose a synchronous WS model in which load balancing is only performed at the end of each round [Aca+13b]. During a round, each processor works on its deque without any interference from idle workers. This scheme allows to remove all the memory fences present in concurrent implementations of deques, totally dodging their associated overheads. The empirical evaluation of this scheme shows that even though some overheads were introduced due to the synchronous *nature* of the proposal, the approach is still quite competitive with a *state-of-art* implementation of the WS algorithm [Lei09]. They further prove that the expected runtime computations executed under this synchronous version of WS is the bounded by $O\left(\left(1+\frac{1}{\delta-1}\right)\cdot\left(\frac{T_1}{P}+T_\infty+\delta F\right)\right)$, where δ is associated with the delay of the synchronization introduced and F corresponding to the idle time associated with load balancing, which, unfortunately, are not close to be negligible overheads. Finally, we remark for the fact that the overheads associated with the polling mechanism proposed in this paper increase proportionally with the total amount of work.

Another proposal by Acar *et al* attempts to bypass memory fences, allowing data races to occur [Aca+13a]. This approach avoided any atomic *read-modify-write* operations, using instead, a complex mechanism that guarantees correctness even in the presence of such race conditions. However, the obtained empirical results did not corroborate the expectations, not showing any benefits of using the suggested approach: most of the evaluated benchmarks showed worse performance by using the proposed technique.

Moreover recently, Morrison *et al* have proposed techniques also to avoid the cost of memory fences on local pop operations [MA14]. The authors introduced a window within the deque such that steal operations can only succeed in the case that there are more than δ tasks (*i.e.* a window of size δ) between the latest read values of top and bottom of the deque. It is assumed that this δ is sufficiently large to avoid any correction problems. The main advantage of this technique is the removal of overheads related with memory barriers. However, as the authors note in their work, correctness depends on the values of δ: if the number of local pops exceeds what is expected, erroneous behaviors can occur.

2.4.4 Sender vs Receiver initiated load balancing

As mentioned in Section 1.3.1, for certain workload types, it may be desirable to opt for sender initiated load balancing mechanisms. Early work by Eager *et al* suggests that sender initiated load balancing is preferable when executing computations with few parallelism [Eag+85]. On the other hand, for scenarios in which there is plenty parallelism, receiver initiated schemes are more appropriate.

Another approach, towards achieving the best of both worlds, was a year later proposed, also by Eager *et al* [Eag+86]. Their work studies adaptive load balancing mechanisms that mix both sender and receiver initiated policies. The rationale behind their contributions lies in the creation of load balancing mechanisms that adapt to the system's state. Three load sharing mechanisms are considered:

Random
: This load sharing policy simply picks targets using a uniform random distribution. This version is the only one that requires no information about the system at all.

Threshold
: In order to avoid sending work to very busy processors, when distributing tasks, processors randomly pick targets, and check if the target has less than the defined threshold of tasks assigned to it. If such is the case, then it simply sends the work to the target. After a defined number of, at most, L_p failed attempts, the processor simply reclaims the task to itself and starts working on it.

Shortest
: Even though the previous approach already attempted to ensure a uniform task distribution among workers, it was not optimal (in what regards to uniformity). For that reason, another approach is suggested: probing L_p processors and sending the task to the one who has the shortest queue length. The major drawback of this approach is that it requires a processor to probe multiple targets until it finally decides to where it can send the task, thus increasing communication costs, as well as load balancing overheads.

Recently, Acar *et al* have compared the performance of sender and receiver initiated approaches [Aca+13b]. The authors concluded that the preference for each of the alternatives depends on the system's load, corroborating the results obtained by Eager *et al* in their early studies [Eag+85].

2.4.5 Stability of Work Stealing

Although WS is provably efficient with nearly optimal bounds, it does not efficiently handle unbalanced scenarios, where only a small fraction of the processors actually generate work. Numerous studies have argued that the performance of the scheduler for such workloads can drastically improve by allowing steals to take multiple items at once [HS02a; Ber+03; LM93; Mit98].

Berenbrink *et al* identified limiting scenarios in which WS does not scale [Ber+03]. The study considers a system with n processors and n work generators, where each generator creates, with probability λ, a unit time task. Then, from an arbitrary (but fixed) random distribution, a *generator-allocation function* is picked, according which tasks are distributed. A *generator-allocation function* corresponds to a mapping (*i.e.* a function) from the work generators to processors. To perform load balancing, idle processors use WS, targeting victims uniformly at random, and stealing from them. However, instead of stealing a single task, processors steal tasks according to a function f that takes as input the number of tasks a *victim* has. Using *Markov chains*, it is proved that, depending on the stealing strategy f, the system can achieve stability (*i.e.* the expected amount of work per processor over time is bounded).

However, the study assumes that the number of tasks generated at each step is at most the number of processors, which, considering our computation's structure, is not realistic. Furthermore, to guarantee the system's stability, the maximum number of nodes stolen by a steal attempt has to be at least $O(\lambda.n)$, which, regardless to mention, does not scale with the number of processors.

The most ingenious strategy that guarantees stability, known as *steal-half*, has been widely studied and is often used for scheduling computations with few or unbalanced parallelism (*e.g.* depth first computations). Under this scheme, thieves can take up to half of the items a victim has. Hendler *et al* propose a lock-free deque implementation [HS02a] that allows successful steal attempts to take up to half of the tasks. Nonetheless, stealing half of the items from a processor's deque is not cheap: if the deque contains $O(T_\infty)$ items, the steal will take $O(T_\infty)$ steps to migrate the stolen nodes. For this reason, the expected run-time bounds for general computations using this algorithm may not be optimal, and, for the best of our knowledge, remain unknown. Moreover, this strategy can incur in high, and fruitless communication overheads where tasks may *'fly'* across processors[1], greatly increasing the communication costs of load balancing.

2.4.6 Analysis of load balancing schemes using the balls-in-bins model

For last, we introduce a widely studied mathematical model, known as *balls-in-bins*, that is often used to analyze load balancing schemes. Suppose we are given n balls and bins. For the classic model, we then pick the n balls and toss them to the *bins* independently and uniformly at random. What is the expected number of balls in the fullest bin?

If we think of each ball as a job and each bin as a processor, then we are essentially scheduling jobs to processors. Numerous studies have been carried out seeking to reduce the expected maximum load of each bin [BL99; Mit98; Bye+04; Mit+02; Dri+02; Alb+01; Mit01; Mit99; Adl+98; Mit97].

Azar *et al* propose a variation of the classical model: instead of immediately throwing the ball into a bin, it first picks two bins and checks the number of balls in each [Aza+99]. It then throws the ball into the least full of these bins. This approach has exponentially improved the previous ones, in which balls were simply thrown uniformly and independently at random into bins. In what regards to the previous approach, the fullest bin would have, with high probability, at most $1 + o(1) + \frac{\ln(n)}{\ln(\ln(n))}$ balls, in which n denotes

[1] *i.e.* where each task can be migrated from processor to processor, multiple times

the total number of balls in the system. With this proposal, however, with high probability, at any step the fullest bin has at most $O(1)\frac{\ln(\ln(n))}{\ln(2)}$ balls. Byers *et al* extended this result [Bye+04] to an arbitrary number d of bins that are among the possible choices as the target of a ball. The authors proved that, with high probability, the fullest bin would have at most $O(1)\frac{\ln(\ln(n))}{\ln(d)}$ balls.

Mitzenmacher *et al* then proposed a variation of this model that uses memory to choose the least loaded bin known as the target of the ball [Mit+02]. In this scheme, when placing a ball into a bin, a least loaded bin is memorized and will be accounted (*i.e.* will be considered as one of the possible destinations) when tossing the next ball. Before tossing a ball, a bin is picked uniformly at random. Then, taking the information of previous throws, we choose the least loaded bin, comparing the one we just picked with the memorized ones, as the target of the ball. For this situation, with high probability, the most loaded bin has at most $O(1)+\frac{\ln(\ln(n))}{2.\ln(\phi)}$, in which ϕ is the golden ratio[2].

[2] $\phi = \frac{(1+\sqrt{5})}{2}$.

Chapter 3

Balls and bins

In this chapter, we introduce the classic *balls-in-bins* model, along with some technical results that will later be used in the theoretical analyzes of the proposed algorithm. The *balls-in-bins* has been very extensively studied [Bye+04; Mit+02; Dri+02; Alb+01; Mit01; Mit99; Adl+98; Mit97; Aza+99; GM01] and is often used to analyze randomized algorithms [BL99; Aro+01; Aca+02; BL98; Sto+03; God08]. In the original version of this model, balls get tossed, independently and uniformly at random, into the bins. We now move to introduce and study some variants of this classic model.

3.1 Disposable balls and Weighted bins

In this section, we analyze a variant of the balls and bins model in which each bin has an assigned weight. Previous work on this model [Aro+01; Aca+02; MA16] assumes that, given n balls and n bins, each of the n balls is tossed, independently and uniformly at random into one of the bins. We extend this model by assuming that, with some probability θ_r, a ball can be discarded (*i.e.* it is not tossed to any bin).

The following lemma is then a generalization of the original result presented in [Aro+01, Balls and Weighted Bins].

Lemma 3.1 (Disposable Balls and Weighted Bins)**.** *Consider an arbitrary, but fixed, θ_r, such that $0 \leq \theta_r < 1$, and suppose we are given at least $\frac{B}{1-\theta_r}$ balls, and exactly B bins. Each of the balls can either be discarded, with probability θ_r, or tossed, independently and uniformly at random, into the bins. Alike tosses, balls are discarded independently from each other.*

Suppose the balls are then handled, each being either discarded or tossed into one of the B bins, where for $i = 1, \ldots, B$, bin i has a weight W_i. The total weight is $W = \sum_{i=1}^{B} W_i$. For each *bin i, we define the random variable X_i as*

$$X_i = \begin{cases} W_i & \textit{if some ball lands in bin } i \\ 0 & \textit{otherwise} \end{cases}$$

If $X = \sum_{i=1}^{B} X_i$, then, for any β in the range $0 < \beta < 1$, we have

$$P\{X \geq \beta W\} \geq 1 - \frac{1}{(1-\beta)e}$$

Proof. First, let us consider a simpler problem in which we are given a single ball j. For a fixed bin i, let S_i denote a random variable, corresponding to the number of balls landing in i. Ball j will not land in i if it is either discarded, or lands in some other bin

$$\begin{aligned} P\{S_i = 0\} &= \theta_r + (1-\theta_r)\left(1 - \frac{1}{B}\right) \\ &= 1 - \frac{1-\theta_r}{B} \end{aligned}$$

Let B' denote the total number of balls we have been given. Generalizing for all B' balls, we then have

$$P\{S_i = 0\} = \left(1 - \frac{1-\theta_r}{B}\right)^{B'}$$

Now, consider the random variable $W_i - X_i$, that takes the value of W_i when no ball lands in bin i, and 0 otherwise. Since $B' \geq \frac{B}{1-\theta_r}$, we have

$$\begin{aligned} E[W_i - X_i] &= W_i\left(1 - \frac{1-\theta_r}{B}\right)^{B'} \\ &\leq W_i\left(1 - \frac{1-\theta_r}{B}\right)^{\frac{B}{1-\theta_r}} \\ &\leq \frac{W_i}{e^{\frac{1-\theta_r}{1-\theta_r}}} \\ &= \frac{W_i}{e} \end{aligned}$$

From the linearity of expectation, we have $E[W - X] \leq W/e$. From Markov's Inequality it follows that

$$P\{W - X > (1-\beta)W\} \leq \frac{E[W-X]}{(1-\beta)W}$$

implying

$$\begin{aligned} P\{X < \beta W\} &\leq \frac{E[W-X]}{(1-\beta)W} \\ &\leq \frac{W/e}{(1-\beta)W} \\ &= \frac{1}{(1-\beta)e} \end{aligned}$$

Thus, we conclude $P\{X \geq \beta W\} \geq 1 - \frac{1}{(1-\beta)e}$. ■

3.2 Bounds on the expected number of non-empty bins

In this section, we obtain both upper and lower bounds on the expected number of non-empty bins (*i.e.* bins with at least one ball) for certain variants of the original *balls-in-bins* model. Consider B bins, each of which being either *red* or *green*. We denote the number of red bins by B_R and the number of green ones by B_G. Alternatively, the number of red bins can be defined as a fraction α of the whole set of bins

$$B_R = \alpha B$$

implying

$$B_G = (1-\alpha)B$$

Lemma 3.2. *Suppose we are given B_R balls, each of which is either discarded, with probability θ, or is tossed, independently and uniformly at random into the bins. Consider a fixed bin b_i, and let S_i denote a random variable, corresponding to the number of balls landing in i. Then,*

$$E[S_i] = \alpha(1-\theta)$$

Proof. Let X denote a random variable, corresponding to the number of balls tossed. Consequently, X follows a binomial distribution

$$X \sim Binomial(B_R, 1-\theta)$$

In turn, S_i also follows a binomial distribution where the number of attempts corresponds to the number of balls that were actually tossed

$$S_i \sim Binomial\left(X, \frac{1}{B}\right)$$

Suppose $X = m$. Then,

$$E[S_i|X=m] = m\frac{1}{B}$$

By the *law of total expectation*, we have

$$\begin{aligned}
E[S_i] &= \sum_{m=0}^{B_R} E[S_i|X=m].P\{X=m\} \\
&= \sum_{m=0}^{B_R} m\frac{1}{B}.P\{X=m\} \\
&= \frac{1}{B}\sum_{m=0}^{B_R} m.P\{X=m\} \\
&= \frac{1}{B}.E[X] \\
&= \frac{B_R(1-\theta)}{B} \\
&= \alpha(1-\theta)
\end{aligned}$$

■

Corollary 3.2.1. *Consider Lemma 3.2. If θ is 0,*

$$E[S_i] = \alpha$$

Corollary 3.2.2. *Consider some bin i. By Lemma 3.2*

$$E[S_i] = \alpha(1-\theta)$$

Let S denote number expected number of ball in the green bins. By the linearity of expectation, it follows

$$\begin{aligned}
E[S] &= E\left[S_1 + S_2 + \ldots + S_{B_G}\right] \\
&= \sum_{i=1}^{B_G} E[S_i] \\
&= \sum_{i=1}^{B_G} \alpha(1-\theta) \\
&= B_G\alpha(1-\theta) \\
&= B(1-\alpha)\alpha(1-\theta)
\end{aligned}$$

Lemma 3.3. *Consider Lemma 3.2. Let Y_i denote an indicator variable, defined as*

$$Y_i = \begin{cases} 1 & \text{if } S_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

Then,

$$P\{Y_i = 1\} \geq 1 - e^{-\alpha(1-\theta)}$$

Proof. First, let us consider a simpler problem in which we are given a single ball j. If j does not land on i, then it has either been discarded or landed in some other bin. Thus,

$$\begin{aligned}
P\{S_i = 0\} &= \theta + (1-\theta)\left(1 - \frac{1}{B}\right) \\
&= 1 - \frac{1-\theta}{B}
\end{aligned}$$

Generalizing for B_R balls, we have

$$\begin{aligned}
P\{S_i = 0\} &= \left(1 - \frac{1-\theta}{B}\right)^{B_R} \\
&= \left(1 - \frac{1-\theta}{B}\right)^{B\alpha} \\
&\leq e^{-\alpha(1-\theta)}
\end{aligned}$$

Noting $P\{Y_i = 0\} = P\{S_i = 0\}$, it follows

$$\begin{aligned}
P\{Y_i > 0\} &= P\{Y_i = 1\} \\
&= 1 - P\{Y_i = 0\} \\
&\geq 1 - e^{-\alpha(1-\theta)}
\end{aligned}$$

■

Corollary 3.3.1. *Consider Lemma 3.3. Hence*

$$\begin{aligned} E[Y_i] &= 0.P\{Y_i = 0\} + 1.P\{Y_i = 1\} \\ &= P\{Y_i = 1\} \\ &\geq 1 - e^{-\alpha(1-\theta)} \end{aligned}$$

Corollary 3.3.2. *Consider Corollary 3.3.1. If θ is* 0,

$$E[Y_i] \geq 1 - e^{-\alpha}$$

Until here, we have only considered scenarios where a single set of balls is tossed to the bins. However, in the following lemmas we make multiple series of tosses targeting the bins, and so, as a way to distinguish between sets of tosses, we classify the objects that are thrown in each of these sets by different three-dimensional geometric figures. Concretely, in Lemma 3.4 we toss balls and cubes, and, in Lemma 3.5, balls, cubes and pyramids.

Lemma 3.4. *For this lemma, B_R and B_G denote the initial number of red and green bins, respectively. In the same way, α corresponds to the initial ratio of red bins.*

Suppose we are given B_R cubes, each of which is either discarded, with probability θ_r, or gets tossed, independently and uniformly at random, into the bins. After tossing all the cubes, we count the number of cubes that landed in green bins, and, for each such cube, a red bin is painted green.

After finishing all the paintings, we take B_G balls, each of which is either discarded, with probability θ_s, or gets tossed, independently and uniformly at random, into the bins.

Let Y denote the number of bins that are still red, with at least one ball. Then

$$E[Y] \geq B\alpha(1 - (1-\alpha)(1-\theta_r))\left(1 - e^{-(1-\alpha)(1-\theta_s)}\right)$$

Proof. Let $B_{R\mapsto G}$ denote a random variable, corresponding to the number of red bins that turned green. To prove this lemma, we start by computing $E[B_{R\mapsto G}]$ (*i.e.* the expected number of red bins that were painted green). Then, we compute the probability that an arbitrary (but fixed) bin has at least one ball. Finally, we obtain lower bounds on the expected number of red bins that have not been painted, with at least one ball.

Let C_{Ghit} denote the number of cubes that landed in the green bins. Each of the B_R cubes we have been given is either discarded, with probability θ_r, or gets tossed, independently and uniformly at random, into the bins. If we think of these cubes as balls, we have an instance of Corollary 3.2.2 (having $\theta = \theta_r$), implying

$$E[C_{Ghit}] = B(1-\alpha)\alpha(1-\theta_r)$$

Noting that $B_{R\mapsto G}$ corresponds to the number of cubes that landed in the bins that were originally green (*i.e.* $B_{R\mapsto G} = C_{Ghit}$), we then conclude

$$\begin{aligned} E[B_{R\mapsto G}] &= E[C_{Ghit}] \\ &= B(1-\alpha)\alpha(1-\theta_r) \end{aligned}$$

Now, consider a simpler problem in which we are given a single ball j. For a fixed (but arbitrary) bin i, let S_i denote a random variable, corresponding to the number of balls landing in i. Ball j will not land in i if it is either discarded, or lands in some other bin. Thus,

$$\begin{aligned} P\{S_i = 0\} &= \theta_s + (1-\theta_s)\left(1 - \frac{1}{B}\right) \\ &= 1 - \frac{1-\theta_s}{B} \end{aligned}$$

Generalizing for B_G balls, we then have:

$$\begin{aligned} P\{S_i = 0\} &= \left(1 - \frac{1-\theta_s}{B}\right)^{B_G} \\ &= \left(1 - \frac{1-\theta_s}{B}\right)^{B(1-\alpha)} \\ &\le e^{-(1-\alpha)(1-\theta_s)} \end{aligned}$$

Let Y_i be an indicator variable for bin i, such that

$$Y_i = \begin{cases} 1 & \text{if } S_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

Thus, $P\{Y_i = 0\} = P\{S_i = 0\}$, implying $P\{Y_i = 1\} = 1 - P\{S_i = 0\}$.

$$\begin{aligned} P\{Y_i = 1\} &= 1 - P\{S_i = 0\} \\ &\ge 1 - e^{-(1-\alpha)(1-\theta_s)} \end{aligned}$$

Since the probability that no ball lands in bin i is independent from the number of red bins turning green (*i.e.* Y_i is independent from $B_{R\mapsto G}$), then, for any m

$$P\{Y_i = 0 | B_{R\mapsto G} = m\} = P\{Y_i = 0\}$$

and

$$P\{Y_i = 1 | B_{R\mapsto G} = m\} = P\{Y_i = 1\}$$

Suppose $B_{R\mapsto G} = m$. It follows

$$\begin{aligned} E[Y_i | B_{R\mapsto G} = m] &= 0.P\{Y_i = 0 | B_{R\mapsto G} = m\} + 1.P\{Y_i = 1 | B_{R\mapsto G} = m\} \\ &= P\{Y_i = 1 | B_{R\mapsto G} = m\} \\ &= P\{Y_i = 1\} \\ &\ge 1 - e^{-(1-\alpha)(1-\theta_s)} \end{aligned}$$

Y denotes the number of bins that are still red with at least one ball:

$$Y = \sum_{i=1}^{B_R - B_{R\mapsto G}} Y_i$$

Then, by the linearity of expectation we have

$$\begin{aligned}
E\left[Y|B_{R\mapsto G}=m\right] &= E\left[Y_1+Y_2+\ldots+Y_{B_R-m}|B_{R\mapsto G}=m\right] \\
&= \sum_{i=1}^{B_R-m} E\left[Y_i|B_{R\mapsto G}=m\right] \\
&\geq \sum_{i=1}^{B_R-m} \left(1-e^{-(1-\alpha)(1-\theta_s)}\right) \\
&= (B_R-m)\left(1-e^{-(1-\alpha)(1-\theta_s)}\right)
\end{aligned}$$

By the law of total expectation, it follows

$$\begin{aligned}
E[Y] &= \sum_{m=0}^{B_R} E\left[Y|B_{R\mapsto G}=m\right] P\left\{B_{R\mapsto G}=m\right\} \\
&\geq \sum_{m=0}^{B_R} (B_R-m)\left(1-e^{-(1-\alpha)(1-\theta_s)}\right) P\left\{B_{R\mapsto G}=m\right\} \\
&= \left(1-e^{-(1-\alpha)(1-\theta_s)}\right)\sum_{m=0}^{B_R}(B_R-m)P\left\{B_{R\mapsto G}=m\right\} \\
&= \left(1-e^{-(1-\alpha)(1-\theta_s)}\right)\left(\sum_{m=0}^{B_R} B_R P\left\{B_{R\mapsto G}=m\right\} - \sum_{m=0}^{B_R} mP\left\{B_{R\mapsto G}=m\right\}\right) \\
&= \left(1-e^{-(1-\alpha)(1-\theta_s)}\right)\left(B_R - E\left[B_{R\mapsto G}\right]\right) \\
&= \left(1-e^{-(1-\alpha)(1-\theta_s)}\right)\left(B_R - B(1-\alpha)\alpha(1-\theta_r)\right) \\
&= \left(B\alpha - B\alpha(1-\alpha)(1-\theta_r)\right)\left(1-e^{-(1-\alpha)(1-\theta_s)}\right) \\
&= B\alpha\left(1-(1-\alpha)(1-\theta_r)\right)\left(1-e^{-(1-\alpha)(1-\theta_s)}\right)
\end{aligned}$$

■

Corollary 3.4.1. *Consider Lemma 3.4. If neither the balls nor cubes can be discarded (i.e. if $\theta_r=\theta_s=0$), we have*

$$\begin{aligned}
E[Y] &\geq B\alpha\left(1-(1-\alpha)\right)\left(1-e^{-(1-\alpha)}\right) \\
&= B\alpha^2\left(1-e^{-(1-\alpha)}\right)
\end{aligned}$$

The proof of the following result follows the same general traits as the proof of Lemma 3.4.

Lemma 3.5. *As for the previous lemma, B_R and B_G denote the initial number of red and green bins, respectively. In the same way, α corresponds to the initial ratio of red bins.*

Suppose we are given B_R cubes, each of which is either discarded, with probability θ_r, or gets tossed, independently and uniformly at random, into the bins. After tossing all the cubes, *we count the number of cubes that landed in green bins, and, for each such cube, a red bin is painted green.*

Next, we are given B_R pyramids, each of which is either discarded, with probability θ_r, or gets tossed, independently and uniformly at random, into the bins. After tossing all pyramids, we count the number of bins that were originally green with at least one pyramid, and, for each such bin, a ball is collected. Let X denote a random variable, corresponding to the number of balls collected.

After collecting all the balls and finishing all the paintings, we handle the X collected balls. Each of these balls is either discarded, with probability θ_s, or is tossed, independently and uniformly at random, into the bins.

Let Y denote the number of bins that are still red, with at least one ball. Then

$$E[Y] \geq B\alpha\left(1-(1-\alpha)(1-\theta_r)\right)\left(1-e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}\right)$$

Proof. Let $B_{R\mapsto G}$ denote a random variable, corresponding to the number of red bins that turned green. Using the same reasoning as in the proof of Lemma 3.4, we have

$$E[B_{R\mapsto G}] = B(1-\alpha)\alpha(1-\theta_r)$$

For an arbitrary (but fixed) bin j, let S_{Pj} denote a random variable, corresponding to the number of pyramids landing in j. Furthermore, we define the indicator variable X_j as

$$X_j = \begin{cases} 1 & \text{if } S_{Pj} > 0 \\ 0 & \text{otherwise} \end{cases}$$

If we think of each pyramid as a ball of Lemma 3.3 (having $\theta = \theta_r$), it follows

$$P\left\{X_j = 1\right\} \geq 1 - e^{-\alpha(1-\theta_r)}$$

Let us now consider a simpler problem in which we may only collect a ball from some bin j. For a fixed (but arbitrary) bin i, let S_i denote a random variable, corresponding to the number of balls landing in i. The ball collected from bin j will not land in i if and only if one of the following scenarios takes place:

1. The ball is not collected. The probability associated with this situation is

$$1 - P\left\{X_j = 1\right\}$$

2. The ball is collected, but then gets discarded. The probability for this scenario to take place is then

$$P\left\{X_j = 1\right\}\theta_s$$

3. The ball is collected and does not get discarded, but lands in some other bin. The probability associated with this last case is

$$P\left\{X_j = 1\right\}(1-\theta_s)\left(1-\frac{1}{B}\right)$$

Thus,

$$\begin{aligned}
P\{S_i = 0\} &= \left(1 - P\left\{X_j = 1\right\}\right) + P\left\{X_j = 1\right\}\theta_s + P\left\{X_j = 1\right\}(1-\theta_s)\left(1 - \frac{1}{B}\right) \\
&= 1 - P\left\{X_j = 1\right\} + P\left\{X_j = 1\right\}\theta_s + \left(P\left\{X_j = 1\right\}(1-\theta_s) - \frac{P\left\{X_j = 1\right\}(1-\theta_s)}{B}\right) \\
&= 1 - P\left\{X_j = 1\right\} + P\left\{X_j = 1\right\}\theta_s + P\left\{X_j = 1\right\} - P\left\{X_j = 1\right\}\theta_s - \frac{P\left\{X_j = 1\right\}(1-\theta_s)}{B} \\
&= 1 - \frac{P\left\{X_j = 1\right\}(1-\theta_s)}{B} \\
&\leq 1 - \frac{\left(1 - e^{-\alpha(1-\theta_r)}\right)(1-\theta_s)}{B}
\end{aligned}$$

Generalizing for B_G balls, we then have:

$$\begin{aligned}
P\{S_i = 0\} &= \left(1 - \frac{P\left\{X_j = 1\right\}(1-\theta_s)}{B}\right)^{B_G} \\
&\leq \left(1 - \frac{\left(1 - e^{-\alpha(1-\theta_r)}\right)(1-\theta_s)}{B}\right)^{B(1-\alpha)} \\
&\leq e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}
\end{aligned}$$

The remainder of the proof is similar to the one of Lemma 3.4.

Let Y_i be an indicator variable for bin i, such that

$$Y_i = \begin{cases} 1 & \text{if } S_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

Thus, $P\{Y_i = 0\} = P\{S_i = 0\}$, implying $P\{Y_i = 1\} = 1 - P\{S_i = 0\}$.

$$\begin{aligned}
P\{Y_i = 1\} &= 1 - P\{S_i = 0\} \\
&\geq 1 - e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}
\end{aligned}$$

Since the probability that no ball lands in bin i is independent from the number of red bins turning green (*i.e.* Y_i is independent from $B_{R\mapsto G}$), then, for any m

$$P\{Y_i = 0 | B_{R\mapsto G} = m\} = P\{Y_i = 0\}$$

and

$$P\{Y_i = 1 | B_{R\mapsto G} = m\} = P\{Y_i = 1\}$$

Suppose $B_{R\mapsto G} = m$. It follows

$$\begin{aligned}
E[Y_i | B_{R\mapsto G} = m] &= 0.P\{Y_i = 0 | B_{R\mapsto G} = m\} + 1.P\{Y_i = 1 | B_{R\mapsto G} = m\} \\
&= P\{Y_i = 1 | B_{R\mapsto G} = m\} \\
&= P\{Y_i = 1\} \\
&\geq 1 - e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}
\end{aligned}$$

Y denotes the number of bins that are still red with at least one ball:

$$Y = \sum_{i=1}^{B_R - B_{R\mapsto G}} Y_i$$

Then, by the linearity of expectation we have

$$\begin{aligned}
E[Y|B_{R\mapsto G} = m] &= E\left[Y_1 + Y_2 + \ldots + Y_{B_R - m}|B_{R\mapsto G} = m\right] \\
&= \sum_{i=1}^{B_R - m} E[Y_i|B_{R\mapsto G} = m] \\
&\geq \sum_{i=1}^{B_R - m} \left(1 - e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}\right) \\
&= (B_R - m)\left(1 - e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}\right)
\end{aligned}$$

By the law of total expectation, it follows

$$\begin{aligned}
E[Y] &= \sum_{m=0}^{B_R} E[Y|B_{R\mapsto G} = m] P\{B_{R\mapsto G} = m\} \\
&\geq \sum_{m=0}^{B_R} (B_R - m)\left(1 - e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}\right) P\{B_{R\mapsto G} = m\} \\
&= \left(1 - e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}\right) \sum_{m=0}^{B_R} (B_R - m) P\{B_{R\mapsto G} = m\} \\
&= \left(1 - e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}\right) \left(\sum_{m=0}^{B_R} B_R P\{B_{R\mapsto G} = m\} - \sum_{m=0}^{B_R} m P\{B_{R\mapsto G} = m\}\right) \\
&= \left(1 - e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}\right) (B_R - E[B_{R\mapsto G}]) \\
&= \left(1 - e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}\right) (B_R - B(1-\alpha)\alpha(1-\theta_r)) \\
&= (B\alpha - B\alpha(1-\alpha)(1-\theta_r))\left(1 - e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}\right) \\
&= B\alpha(1 - (1-\alpha)(1-\theta_r))\left(1 - e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}\right)
\end{aligned}$$

■

Corollary 3.5.1. *Consider Lemma 3.5. If neither the balls nor cubes nor pyramids can be discarded (i.e. if $\theta_r = \theta_s = 0$), we have*

$$\begin{aligned}
E[Y] &\geq B\alpha(1 - (1-\alpha))\left(1 - e^{-(1-\alpha)(1-e^{-\alpha})}\right) \\
&= B\alpha^2\left(1 - e^{-(1-\alpha)(1-e^{-\alpha})}\right)
\end{aligned}$$

3.3 Future Work

Problem 3.1. *As for Lemma 3.4, let B_R and B_G denote the initial number of red and green bins, respectively. In the same way, α corresponds to the initial ratio of red painted bins.*

Now, suppose we are given B_R cubes, each of which is either discarded, with probability θ_r, or gets tossed, independently and uniformly at random, to one of the B bins. After tossing all the cubes, we count the number of green bins in which at least one cube landed, and, for each of these bins, a red bin is painted green and we collect a ball. Let $B_{R \mapsto G}$ denote a random variable, corresponding to the number of red bins that turned green, and, thus, also corresponding to the number of balls collected.

After all the paintings, we take all the $B_{R \mapsto G}$ balls. Each one is either discarded, with probability θ_s, or gets tossed, independently and uniformly at random, to one of the B bins.

- *What is the expected number of bins that are still red, with at least one ball?*
- *Are there tight lower bounds for this expectation?*

The reason why the answer to Problem 3.1 is of significant importance for our study is explained in the subsequent chapters.

Chapter 4

Scheduling computations

In this chapter, we acquaintance the reader with several concepts that will be used through the rest of the document, and, present our first contribution, together with an analysis of its performance. We start by recalling, in a more precise manner, the base definitions of computations, and, after that, specify some basic concepts related with schedules. Next, the *state-of-the-art* scheduling algorithm is carefully described, and, along with its definition, we present the deque implementation's semantics, which are crucial to understand the analysis related with the runtime bounds of the algorithm. After that, we present our first proposal, the Canonical Flexible Lock Free Work Stealing (CFLFWS) algorithm, and, finally, we obtain bounds on the expected runtime of computations using the algorithm.

4.1 Modeling computations

In this section, we present the formal background for our work. First, recall that, as with much previous work, a computation is modeled by a dag $G = (V, E)$, where each node $v \in V$ represents a unit time instruction (*i.e.* an instruction that takes exactly one time step to execute), and each edge $(\mu_1, \mu_2) \in E$ denotes an ordering constraint, implying μ_2 can only be executed after μ_1. The *in-degree* of a node is the number of edges directed to it, while its *out-degree* corresponds to the total number of edges directed out of that same node. Nodes with no ancestors (*i.e.* with *in-degree* of 0) are referred to as *roots*, whereas the ones with no successor (*i.e.* the nodes whose *out-degree* is 0) are called *sinks*.

Through the rest of the document, we make two assumptions related with the structure of computations, equivalent to the ones made by Arora *et al* in [Aro+01]. Let G denote a computation:

1. There exists only one *root* and one *sink* in G.

2. The *out-degree* of any node within G is at most two.

Both these assumptions are reasonable. Regarding the first constraint, it is legitimate to assume that any computation has an unique root and an unique sink. In fact, computations with multiple roots and sinks can be extended to obey this assumption. Concerning the second constraint, as each node corresponds to a single instruction, it is appropriate to assume that a node's execution can enable at most two other ones.

Modeling computations with multiple roots and/or sinks.
Modeling computations with multiple roots can easily be achieved by creating a binary tree of instructions, starting at a new single root and whose leaves correspond to the multiple roots of the extended computation. On the other hand, transforming a computation with multiple sinks to one with a single one is even more straightforward: it suffices to create a new sink node and add a dependency from every sink of the computation being expanded to the new sink.

The number of nodes within a computation is expressed by T_1 and its span (*i.e.* the length of a longest directed path) by T_∞. We further denote the number of processors executing a computation by P. A node is said to be *ready* if all its ancestors have already been executed, but not the node itself. To respect ordering constraints and ensure correction, only ready nodes can be executed. Consequently, in the beginning of a computation's execution, only the root can be executed. As the execution progresses, nodes get executed, possibly causing one or more of its successors to become ready. When the *sink* gets executed, the computation's execution terminates.

The execution of a computation can be partitioned into discrete steps: at each step, every processor executes an instruction. The assignment of a node μ to a processor p means that μ will be the next node that p executes. Thus, any processor can only have one assigned node at each step, and, each node can only be assigned by a single processor (as otherwise, nodes could be executed more than once). Since the number of processors is denoted by P, then, at most P nodes get executed in a single step. Decisions related with the dag's execution depend on the scheduling algorithm in use.

The dag of a computation is dynamically unfolded during its execution. If the execution of a node u *enables* another node u', then, the edge (u, u') is an *enabling edge*. Furthermore, we name node u as the *designated parent* of u'. Since each node, other than the root, has exactly one designated parent, then, the set of all enabling edges together with the computation's nodes form an *enabling tree*, whose root is the computation's root. As one might conclude, this tree is a sub-graph of the computation. Given that the dag is dynamically unfolded, the enabling trees formed in two consecutive executions of the same computation can differ. The depth of a node u is denoted by $d(u)$, and is equal to the length of the directed path starting on the root node of the enabling tree and ending on u. The weight of u is defined as $w(u) = T_\infty - d(u)$.

The analysis we make about the execution of computations is made in an *a posteriori* fashion. Thus, we are able to refer to the enabling tree generated by a computation's execution.

4.1.1 An alternative model of computations

Until here, we have been specifying how we model computations. Now, we describe an alternative way of modeling computations that has been widely used in other related work (see [BL99; Agr+08; BL98; He+06]).

Consonantly to our representation, a computation corresponds to a dag. However, in this alternate model it is embodied by a set of *threads*, each of which is composed by a contiguous sequence of nodes within the dag, connected by a chain of edges that define the thread's execution order. As for our model, nodes also correspond to unit time instructions, whose out-degree is at most two. Each edge within the chain of some thread is referred to as a *continue* edge. Moreover, a *spawn* edge corresponds to one starting in a node of some thread Γ_1 and terminating in the first node of the spawned thread Γ_2. In such case, we say that Γ_2 is a *child* of Γ_1, or, contrarily, that Γ_1 is the *parent* of Γ_2. If a thread has no child we say it is a *leaf*. To ensure a correct execution of computations, threads can be connected by *join* edges, establishing ordering constraints for the dag's execution. For example, if a thread Γ_1 had to read a value written by another thread Γ_2, then, there would be a join edge starting at the node of Γ_1 where the value would be read, and, ending on the node of Γ_2 where the value would, respectively, be written. When a join edge starts at the last node of some thread Γ_1 and ends in some node of another thread Γ_2, we say that Γ_1 joins Γ_2.

In the literature, there also exists a notion of *strict* and *fully strict* computations. A strict computation is one in which every join edge goes from a thread to one of its ancestors. On the other hand, a fully strict computation is one where every join edge goes from a thread to its parent. With this, it is straightforward to conclude that fully strict computations are less powerful and generic than strict ones. These, in turn, are also less powerful and generic than the ones we consider.

The out-degree assumption.

As one might note, a node that spawns a new thread has an out-degree of two. In the same way, if a thread Γ_1 has to synchronize, due to some dependency, with another thread Γ_2, then, in one of the nodes within the chain of Γ_2 starts a join edge that is directed to the node of Γ_1 with the respective dependency. If, for example, n threads depend on the execution of some node μ within the chain of some other thread Γ_0, then, succeeding μ, Γ_0 would have n nodes within its chain such that, for each dependent thread, a join edge would start, directed to the corresponding dependent nodes of the respective (dependent) thread. From this perspective, it gets more obvious why the assumption that each node has an out-degree of at most two is reasonable.

4.2 Schedules

A schedule of a computation is no more than a mapping of (*processor*, *step*) pairs to dag nodes. As only ready nodes can be executed, the ordering constraints defined in E are always respected. It has been proved that, for any computation executed by P processors, there exists a schedule that takes exactly $\max\left(T_\infty, \frac{T_1}{P}\right)$ steps until termination. However, discovering such optimal schedules is an NP-complete problem [Ull75]. We now move to describe *greedy* schedules.

4.2.1 Greedy schedules

A greedy schedule is one such that, at each step i during the execution of a computation, if R_i nodes are ready, then, exactly $\min(P, R_i)$ nodes get executed during the step. These schedules are usually studied in order to compare their runtime bounds (which do not account with load balancing overheads) with the ones of practical algorithms.

The following result was proved by Arora *et al* (see [Aro+01, Theorem 2]) and assumes multiprogrammed environments. Under such environments, processes are scheduled by a kernel that chooses which ones execute at each time step. In this theorem, P_A denotes the average number of processes scheduled at each step.

Theorem 4.1 (Greedy schedules)**.** *Consider any multithreaded computation with work T_1 and critical-path length T_∞, any number of P processes, and any kernel schedule. Any greedy execution schedule has length at most $T_1/P_A + T_\infty(P-1)/P_A$, where P_A is the processor average over the length of the schedule.*

Noting that, under a dedicated environment every process is scheduled at each step (assuming the number of processes does not exceed the number of processors), then, these environments can be thought of as particular instances of multiprogrammed ones. As such, we now present a corollary of the previous result assuming dedicated environments (instead of multiprogrammed ones). As a consequence of this assumption, there is no longer an underlying kernel scheduler that manages processes. For that reason, rather than referring to processes, we, instead, refer to processors.

Corollary 4.1.1. *Consider any multithreaded computation with work T_1 and critical-path length T_∞, and any number of P processors. Any greedy execution schedule has length at most $T_1/P + T_\infty(P-1)/P$.*

Regarding dedicated environments, since, for any computation there is at least a schedule taking exactly $\max\left(T_\infty, \frac{T_1}{P}\right)$ steps, it is straightforward to conclude that any greedy execution schedule is optimal within a factor of two.

4.3 The Lock Free Work Stealing algorithm

In this section, we carefully specify the LFWS. The algorithm was originally designed for scheduling computations under multiprogrammed environments. Yet, instead of assuming such environments, we confine only to dedicated ones. The reason why we do not consider a multiprogrammed environment is because it would severely complicate the study of the proposed algorithm, and, most of all, because it is not the goal of this thesis. Part of the scheduler's description is based on the well known, clear and elegant specification given in [Aro+01].

To facilitate the comprehension of the algorithm, we first describe its behavior using the notion of threads. In the LFWS, each processor owns a deque that uses to keep track of its attached threads, adding and obtaining work from it as a computation's execution progresses. When a processor becomes idle, it turns into a *thief* and starts a quest for work, continuously targeting possible *victims* and attempting to steal a thread from their deque. After obtaining work, either from its own deque or from another's, the processor assigns the obtained thread and starts its execution, until it either stalls (due to some unsatisfied dependency) or dies. At such point, the processor then returns to its deque, from which it attempts to obtain the bottommost thread. If it has no attached thread (*i.e.* if its deque is empty), the processor restarts its quest for work, becoming a thief and starting a work stealing phase. During work stealing phases, a thief repeatedly targets, uniformly at random, possible victims, from whose deque it attempts to steal the topmost thread. These phases only terminate when either the computation's execution ends, or, when the processor obtains a thread on which it can work on, thus ceasing to be a thief. When, during a thread's execution, a processor spawns a new thread, it either pushes the child (*i.e.* the spawned thread) into the bottom of its deque and resumes the execution of its assigned thread, or, alternatively, pushes its current thread into the bottom of its deque and then assigns (and consequently, starts to work on) the spawned child[1]. Finally, alike in the spawn case, when, a processor enables a blocked thread, it can either start working on the unblocked thread, pushing the current into its deque, or, the processor can push, instead, the unblocked thread into its deque, then continuing the execution of its assigned thread.

As one might note, since a processor pushes and pops threads from the bottom of its deque, it operates on it in a LIFO fashion. On the other hand, as thieves attempt to steal the topmost nodes from their victims' deques, stealing works in a First In First Out (FIFO) style.

4.3.1 Unthreaded specification of the scheduler

We now move to specify the algorithm, this time without using the notion of threads. In this scenario, each thread in a processor's deque gets replaced by its current ready

[1] As mentioned in [Aro+01], this second approach allows to use the famous *lazy task creation* optimization [Blu+96; Lei09; Moh+91; Gol+96].

node (*i.e.* the node of the thread that should be executed next). Furthermore, rather than assigning threads, under this scenario processors assign ready nodes. The specification of the scheduler that operates on nodes, rather than threads, is defined in Algorithm 1.

Before going into further details of the algorithm, we first introduce a concept that will ease the comprehension of the scheduler's behavior. We define a *scheduling iteration* as the sequence of executed instructions, including sub-routine calls, corresponding to one iteration of the scheduling loop[2]. For the LFWS algorithm, it corresponds to the sequence of instructions starting at line 2 and terminating at line 16 of Algorithm 1. Accordingly, the full sequence of instructions executed by a processor during a computation's execution can be partitioned into smaller chunks, each corresponding to a scheduling iteration.

Before a computation's execution starts, every processor sets its *assigned* node to NONE, indicating it has no work. Then, to begin the execution, the root node is assigned to an arbitrary processor.

In the course of executing a computation, if, at the beginning of some scheduling iteration's execution a processor has an assigned node, it executes such node. From the node's execution, either none, one or two nodes can be enabled. The behavior associated to each of these cases is:

0 nodes enabled In this case, the processor attempts to pop a ready node from the bottom of its deque. If a node is returned, it becomes the processor's assigned node[3]. However, if the deque is empty, the processor does not get any node assign, implying it will start the execution of the next iteration without an assigned node.

1 node enabled The node becomes the processor's assigned node.

2 nodes enabled For the threaded scheduler description, this case corresponds to when a processor either spawned a thread or unblocked a stalled thread. In this case, one of the nodes becomes the processor's new assigned node. Then, the processor pushes the other one into the bottom of its deque.

At this point, it only remains to describe the algorithm's behavior for the case when a processor starts a scheduling iteration without an assigned node. In such case, and, as already mentioned, the thief (*i.e.* the processor) targets, uniformly at random, a victim, from whose deque it attempts to steal the topmost ready node. If the steal is successful, it immediately assigns the node, terminating the stealing phase. Otherwise, the processor starts the next iteration with no assigned node, continuing to be a thief.

[2]Note that, due to the nature of computations, a scheduler has to, somehow, loop, ensuring that every node is executed.

[3]As it will be described later, the semantics of the deque's implementation guarantee that if it is not empty, it will always return its bottommost node.

```
procedure SCHEDULER
    while computation not terminated do
        if ValidNode($assigned$) then
            $enabled \leftarrow$ execute($assigned$)
            if length($enabled$) > 0 then
                $assigned \leftarrow enabled[0]$
                if length($enabled$) = 2 then
                    $self.deque$.pushBottom($enabled[1]$)
                end if
            else
                $assigned \leftarrow self.deque$.popBottom()
            end if
        else
            $self$.WorkMigration()
        end if
    end while
end procedure

procedure WORKMIGRATION
    $victim \leftarrow$ UniformlyRandomProcessor()
    $assigned \leftarrow victim.deque$.popTop()
end procedure

function VALIDNODE($node$)
    return $node \neq$ EMPTY and $node \neq$ ABORT and $node \neq None$
end function
```

Algorithm 1: The Lock Free Work Stealing algorithm. The meaning of the values EMPTY and ABORT will be explained in Section 4.4.

4.4 The lock free deque

In this section, we present the specification of the deque's implementation, along with its associated relaxed semantics.

Ideally, a deque object would support three methods:

1. *pushBottom* — Adds an item to the bottom of the deque and does not return.

2. *popBottom* — Returns and removes the bottommost item from the deque, if any. Otherwise, the invocation has no effect and EMPTY is returned.

3. *popTop* — Returns and removes the topmost item from the deque, if any. Otherwise, the invocation has no effect and EMPTY is returned.

A deque implementation is said to be *constant-time* if and only if any invocation to any of these methods takes at most a constant number of steps to return. Taking into account that a processor executes an instruction per step, it is straightforward to conclude that for any constant-time implementation of the deque, the sequence of executed instructions associated with any invocation has constant length. Sadly, as a consequence of the result proved in [Att+11, Lemma 6.4], there is no constant-time implementation of a deque satisfying the semantics defined above. In order to allow possible implementations, the overall semantics of the deque methods have to be weakened. We refer to these weakened semantics as the *relaxed semantics*[4]. Informally, under these semantics, a *popTop* invocation is now allowed to abort under certain circumstances. Fortunately, any implementation that obeys the relaxed semantics is fit to be used by the LFWS algorithm (and, as we will explain later, by our proposed algorithm as well).

A deque object meeting the relaxed semantics supports three methods:

1. *pushBottom* — Adds an item to the bottom of the deque and does not return.
2. *popBottom* — Returns and removes the bottommost item from the deque, if any. Otherwise, the invocation has no effect and EMPTY is returned.
3. *popTop* — Attempts to return and remove the topmost item from the deque. If the deque is empty, the invocation has no effect and the value EMPTY is returned. Otherwise, if the attempt fails, the invocation has no effect and the special value ABORT is returned.

An invocation to a deque method is defined by a 4-tuple establishing:

1. Which method was invoked.
2. The invocation's beginning time.
3. The invocation's completion time.
4. The return value, if any.

We say that a set of invocations to a deque's methods meets the relaxed semantics if and only if there is a set of *linearization times* for the corresponding non-aborting invocations such that:

1. Every non-aborting invocation's linearization time lies within the beginning and completion times of the respective invocation.
2. No linearization times associated with distinct non-aborting invocations coincide.
3. The return values for each non-aborting invocation are consistent with a serial execution of the methods in the order given by the linearization times of the corresponding non-aborting invocations.
4. For each aborted *popTop* invocation x to a deque d, there exists another invocation removing the topmost item from d whose linearization time falls between the beginning and completion times of x's invocation.

[4]The relaxed semantics were originally introduced by Arora *et al* in [Aro+01].

A set of invocations is said to be *good* if and only if *pushBottom* and *popBottom* are never invoked concurrently[5]. The deque implementations given in [Aro+01; Hen+06] have been proved to be constant-time and to meet the relaxed semantics for any good set of invocations[6]. Thus, we can assume any of them as the deque's implementation.

Regarding the LFWS algorithm, since only the owner of each deque may invoke the *pushBottom* and *popBotom* methods, we conclude that the relaxed semantics will always be satisfied for this algorithm, during a computation's execution.

4.5 The Flexible Lock Free Work Stealing algorithm

We now present our first contribution: a variant of the LFWS scheduler, entitled Canonical Flexible Lock Free Work Stealing (CFLFWS), that can be extended to support custom load balancing strategies, while maintaining provably good expected runtime bounds. The main idea behind this contribution is to allow processors to use alternative scheduling decisions that, if well designed, may greatly improve the performance of certain computations. The inclusion of such an alternative is enabled by associating a probability to the use of the original WS balancer. Whenever the latter mechanism is not chosen, the alternative algorithm is executed and, consequently, a potentially different scheduling decision may be made.

The CFLFWS algorithm provides the foundation for defining schedulers that build upon this flexibility. It does not compromise itself with any alternative scheduling decision, establishing only the base algorithm and the formal framework atop which these algorithms may be specified. For example, as mentioned in Section 2.4.1, WS often renders poor performances when executing data intensive computations. This limitation is a consequence of the locality obliviousness of its underlying load balancing mechanism. To overcome it, Acar *et al* suggested a sophisticated load balancing scheme aiming to ensure that processors work on data in which they have already worked on [Aca+02]. However, not only the proposal is not provably efficient, but it also is not appropriate for scheduling arbitrary, non memory bound, computations. Fortunately, as we will show in Chapter 6, this scheme can easily be implemented on top of the CFLFWS scheduler, by taking advantage of its flexibility. By doing so, it is possible to benefit from the locality awareness of the proposed load balancing mechanism, and, at the same time, still ensuring high performance when executing arbitrary computations. In Chapter 6 we present several other possible extensions to this algorithm, each addressing a known limitation of WS.

The CFLFWS scheduler (depicted in Algorithm 2) is based on LFWS, and assumes the same deque implementation. In fact, the only difference between these two schedulers lies in the *WorkMigration* procedure: while in the original algorithm a processor always

[5] *i.e.* no two invocations to *pushBottom* are concurrent, nor two invocations to *popBottom* are concurrent, and, finally, nor an invocation to *pushBottom* and another to *popBottom* are concurrent.

[6] Regarding the deque given in [Aro+01], it was proved, in [Blu+99], that it indeed meets the relaxed semantics on any good set of invocations.

picks a victim and then attempts to steal work from its deque, in the CFLFWS a processor first chooses, uniformly at random, what it will do. Then, depending on this decision, the processor either uses the classical load balancing scheme, or, simply, takes no action. To make these decisions, an argument, denoted by θ_r, is passed to the scheduler. Then, the processor picks a number δ uniformly at random from the interval $[0,1]$, and, if $\delta < \theta_r$ it takes no action[7]. Otherwise (*i.e.* if $\delta \geq \theta_r$), the processor relies on the original WS strategy to load balance, picking a victim uniformly at random and then invoking the *popTop* method to its deque.

As for LFWS, in the CFLFWS algorithm only the owner of each deque may invoke the *pushBottom* and *popBotom* methods. Consequently, any set of invocations made during a computation's execution by the CFLFWS algorithm is good. We conclude that the relaxed semantics, afore-described in Section 4.4, will always be satisfied by the algorithm when executing a computation.

```
procedure SCHEDULER($\theta_r$)
    while $computation$ $not$ $terminated$ do
        if ValidNode($assigned$) then
            $enabled$ ←execute($assigned$)
            if length($enabled$)> 0 then
                $assigned$ ← $enabled$[0]
                if length($enabled$) = 2 then
                    $self.deque$.pushBottom($enabled$[1])
                end if
            else
                $assigned$ ← $self.deque$.popBottom()
            end if
        else
            $self$.WorkMigration($\theta_r$)
        end if
    end while
end procedure

procedure WORKMIGRATION($\theta_r$)
    $choice$ ← $UniformlyRandomNumber$([0;1])
    if $choice \geq \theta_r$ then
        $victim$ ← $UniformlyRandomProcessor$()
        $assigned$ ← $victim.deque$.popTop()
    end if
end procedure

function VALIDNODE($node$)
    return $node \neq$ EMPTY  $and$  $node \neq$ ABORT  $and$  $node \neq None$
end function
```

Algorithm 2: The Canonical Flexible Lock Free Work Stealing algorithm.

[7]As one might note, θ_r corresponds to the probability that a processor takes no action.

4.5.1 An analysis of the expected runtime bounds

Now, we move to establish the expected runtime bounds of the CFLFWS algorithm. As aforementioned, the scheduler's design aims to simplify the development of extensions to the LFWS algorithm[8], while maintaining provably good expected runtime bounds. As such, even though the following analysis only applies to the canonical version of the algorithm, it can conveniently be extended[9] to obtain bounds on the expected runtime of computations for countless possible extensions. The analysis we make follows the same traits as the one given in [Aro+01], but assuming, for the sake of simplicity, a dedicated environment.

Alike in [Aro+01], we introduce the concept of *milestone*. Due to the extensible nature of the CFLFWS, our definition of milestone differs from the original[10]. An instruction within the sequence executed by some processor is a milestone if and only if it corresponds to the execution of a node, or, to the completion of a call to the *WorkMigration* procedure.

Recall the definition of a scheduling iteration. It is straightforward to conclude that any scheduling iteration of the algorithm includes a milestone. For instance, at the beginning of a scheduling iteration, a processor either has an assigned node or not. In the first case, the iteration's execution reaches the milestone when the node gets executed (line 4 of Algorithm 2). In the latter, the processor calls the *WorkMigration* procedure. The milestone is then the last instruction within the sequence of executed instructions corresponding to the *WorkMigration* call (line 24).

We distinguish (scheduling) iterations according to the type of milestone they contain. Iterations whose milestone corresponds the execution of a node are termed *busy iterations*, while the remainder are labeled as *idle iterations*. Thereupon, if a processor has an assigned node at the beginning of an iteration's execution, the iteration is a busy one, and, otherwise, is an idle one.

As argued in [Aro+01], by observing the scheduling loop (lines 2 to 16 of Algorithm 2), it is straightforward to conclude that any processor executes at most a constant number of instructions within two consecutive milestones. Consequently, there exists a constant C such that, for any sequence of instructions executed by a processor whose length is at least C, at least one of such instructions is a milestone.

We now state a trivial variant of [Aro+01, Lemma 5] for the CFLFWS algorithm, that bounds the execution time in terms of the number of idle iterations processors execute.

Lemma 4.2. *Consider any computation with work T_1 being executed by the CFLFWS. Then, the execution time is $O\left(\frac{T_1}{P} + \frac{I}{P}\right)$, where I denotes the total number of idle iterations executed by processors.*

[8] To design an extension, one only has to extend and/or modify the CFLFWS scheduler.

[9] Evidently, the ease of the analysis' modifications or extensions is dependent to the extension.

[10] Originally, an instruction within the sequence of instructions executed by a processor would be a milestone if and only if it either corresponded to a node's execution, or, to the completion of a *popTop* invocation.

Proof. Recall that there are two types of scheduling iterations: busy and idle ones. Consider two buckets to which we add tokens during the computation's execution. We call to the first one busy bucket, and, to the second one idle bucket. Each processor places, at the beginning of each iteration, a token in one of these buckets. Concretely, if the processor has an assigned node (implying it is starting the execution of a busy iteration), it places the token into the busy bucket. Otherwise, if the processor does not have an assigned node (meaning it is starting the execution of an idle iteration), it places a token into the idle bucket. Since we are in the presence of a dedicated environment, for each C consecutive steps, at least P tokens are placed into the buckets.

Taking into account that, by definition, the computation has T_1 nodes, then, there will be exactly T_1 tokens in the busy bucket at the end of the computation's execution. Further, since I denotes the number of idle iterations, it also corresponds to the number of tokens in the idle bucket at the end of the computation's execution. Thus, exactly $T_1 + I$ tokens are collected during the computation's execution. Taking into account that for each C consecutive steps at least P tokens are placed into the buckets, we conclude the number of steps required to collect all the tokens is bounded by $C.\left(\frac{T_1}{P} + \frac{I}{P}\right)$. After collecting all the T_1 tokens, the computation's execution terminates, implying the execution time is at most $O\left(\frac{T_1}{P} + \frac{I}{P}\right)$. ∎

The following lemma, its proof and respective corollary are transcribed from [Aro+01, Lemma 3]. We only place the full proof because it will later be extended.

Lemma 4.3 (Structural Lemma)**.** *Let k be the number of nodes in a given deque at some time in the (linearized) execution of the CFLFWS algorithm, and let $v_1, \ldots, v_k$ denote those nodes ordered from the bottom of the deque to the top. Let v_0 denote the assigned node if there is one. In addition, for $i = 0, \ldots, k$ let u_i denote the designated parent of v_i. Then for $i = 1, \ldots, k$, node u_i is an ancestor of u_{i-1} in the enabling tree. Moreover, though we may have $u_0 = u_1$, for $i = 2, 3, \ldots, k$, we have $u_{i-1} \neq u_i$ that is, the ancestor relationship is proper.*

Proof. Fix a particular deque. The deque state and assigned node only change when the assigned node is executed or a thief performs a successful steal. We prove the claim by induction on the number of assigned-node executions and steals since the deque was last empty. In the base case, if the deque is empty, then the claim holds vacuously. We now assume that the claim holds before a given assigned-node execution or successful steal, and we will show that it holds after. Specifically, before the assigned-node execution or successful steal, let v_0 denote the assigned node; let k denote the number of nodes in the deque; let $v_1, \ldots, v_k$ denote the nodes in the deque ordered from the bottom to top; and for $i = 0, \ldots, k$, let u_i denote the designated parent of v_i. We assume that either $k = 0$, or for $i = 1, \ldots, k$, node u_i is an ancestor of u_{i-1} in the enabling tree, with the ancestor relationship being proper, except possibly for the case $i = 1$. After the assigned-node execution or successful steal, let $v_0{}'$ denote the assigned node; let k' denote the number of nodes in the deque; let $v_1{}', \ldots, v_k{}'$ denote the nodes in the deque ordered from bottom to top; and for $i = 1,\ldots,k'$, let $u_i{}'$ denote the designated parent of $v_i{}'$. We now show that either $k' = 0$, or for $i = 1,\ldots,k'$, node $u_i{}'$ is an ancestor of $u_{i-1}{}'$ in the enabling tree, with the ancestor relationship being proper, except possibly for the case $i = 1$.

Consider the execution of the assigned node v_0 by the owner.

If the execution of v_0 enables 0 children, then the owner pops the bottommost node off its deque and makes that node its new assigned node. If $k = 0$, then the deque is empty; the owner does not get a new assigned node; and $k' = 0$. If $k > 0$, then the bottommost node v_1 is popped and becomes the new assigned node, and $k' = k-1$. If $k = 1$, then $k' = 0$. Otherwise, $k' = k-1$. We now rename the nodes as follows. For $i = 0,\ldots,k'$, we set $v_i{}' = v_{i+1}$ and $u_i{}' = u_{i+1}$. We now observe that for $i = 1,\ldots,k'$, node $u_i{}'$ is a proper ancestor of $u_{i-1}{}'$ in the enabling tree.

If the execution of v_0 enables 1 child x, then x becomes the new assigned node; the designated parent of x is v_0; and $k' = k$. If $k = 0$, then $k' = 0$. Otherwise, we can rename the nodes as follows. We set $v_0{}' = x$; we set $u_0{}' = v_0$; and for $i = 1,\ldots,k'$, we set $v_i{}' = v_i$ and $u_i{}' = u_i$. We now observe that for $i = 1,\ldots,k'$, node $u_i{}'$ is a proper ancestor of $u_{i-1}{}'$ in the enabling tree. That $u_1{}'$ is a proper ancestor of $u_0{}'$ in the enabling tree follows from the fact that (u_0, v_0) is an enabling edge.

In the most interesting case, the execution of the assigned node v_0 enables 2 children x and y, with x being pushed onto the bottom of the deque and y becoming the new assigned node. In this case, (v_0, x) and (v_0, y) are both enabling edges, and $k' = k+1$. We now rename the nodes as follows. We set $v_0{}' = y$; we set $u_0{}' = v_0$; we set $v_1{}' = x$; we set $u_1{}' = v_0$; and for $i = 2,\ldots,k'$, we set $v_i{}' = v_{i-1}$ and $u_i{}' = u_{i-1}$. We now observe that $u_1{}' = u_0{}'$, and for $i = 2,\ldots,k'$, node $u_i{}'$ is a proper ancestor of $u_{i-1}{}'$ in the enabling tree. That $u_2{}'$ is a proper ancestor of $u_1{}'$ in the enabling tree follows from the fact that (u_0, v_0) is an enabling edge.

Finally, we consider a successful steal by a thief. In this case, the thief pops the topmost node v_k off the deque, so $k' = k-1$. If $k = 1$, then $k' = 0$. Otherwise, we can rename the nodes as follows. For $i = 0,\ldots,k'$, we set $v_i{}' = v_i$ and $u_i{}' = u_i$. We now observe that for $i = 1,\ldots,k'$, node $u_i{}'$ is an ancestor of $u_{i-1}{}'$ in the enabling tree, with the ancestor relationship being proper, except possibly for the case $i = 1$. ■

Corollary 4.3.1. *If $v_0, v_1, \ldots, v_k$ are as defined in the statement of Lemma 4.3, then we have $w(v_0) \le w(v_1) < \ldots < w(v_{k-1}) < w(v_k)$.*

Lemma 4.2 bounds the execution time in terms of the number of idle iterations. Thus, to obtain runtime bounds, we only have to bound the number of idle iterations made during the computation's execution. Alike much previous work [Aca+02; Aca+13b; Tch+10; MA16], we make use of the potential method (first introduced in [Aro+01]) to obtain bounds on the number of idle iterations executed by the processors. For instance, we also make use of the same potential function, as given in [Aro+01].

We denote the set of ready nodes at some step i by R_i. Consider any node $u \in R_i$. The potential associated with u at step i is denoted by $\phi_i(u)$ and is defined as

$$\phi_i(u) = \begin{cases} 3^{2w(u)-1} & \text{if } u \text{ is assigned} \\ 3^{2w(u)} & \text{otherwise} \end{cases}$$

The total potential at step i, denoted by Φ_i, corresponds to the sum of potentials of all the nodes that are ready at that step:

$$\Phi_i = \sum_{u \in R_i} \phi_i(u).$$

The following lemma is no more than a formalization of the arguments already given in [Aro+01]. The proof of this lemma is no more than a straightforward extension to the aforementioned arguments, but, considering every possible scenario.

Lemma 4.4. *Consider some node u, ready at step i during the execution of a computation. If u gets assigned to a processor at that step, the potential drops by at least $\frac{2}{3}\phi_i(u)$. More, if u was already assigned to a processor and gets executed at that step i, the potential drops by at least $\frac{5}{9}\phi_i(u)$.*

Proof. By the potential function's definition, if a node u gets assigned, its potential decreases from $3^{2w(u)}$ to $3^{2w(u)-1}$. Thus,

$$\begin{aligned} 3^{2w(u)} - 3^{2w(u)-1} &= \frac{2}{3}3^{2w(u)} \\ &= \frac{2}{3}\phi_i(u) \end{aligned}$$

So, we conclude that, if u is assigned, the potential decreases by at least $\frac{2}{3}\phi_i(u)$.

Regarding the second claim, note that, due to our conventions regarding computations' structure, a node can be the designated parent of at most two other ones in the enabling tree. Further, by definition, the weight of any enabled nodes is strictly smaller than the weight of their designated parent.

If no node is enabled, it is clear that the potential decreases by $\phi_i(u)$.

If a single node is enabled, it becomes the assigned node of the processor. Let x denote the enabled node (*i.e.* the child of u in the enabling tree). It follows,

$$\begin{aligned} \phi_i(u) - \phi_{i+1}(x) &= 3^{2w(u)-1} - 3^{2w(x)-1} \\ &= 3^{2w(u)-1} - 3^{2(w(u)-1)-1} \\ &= 3^{2w(u)-1}\left(1 - \frac{1}{9}\right) \\ &= \frac{8}{9}\phi_i(u) \end{aligned}$$

So, we conclude that in this case, the potential decreases by $\frac{8}{9}\phi_i(u)$.

Finally, if two nodes are enabled, then, one of them becomes the assigned node of the processor while the other is pushed into the bottom of its deque. Let x denote the enabled node that is assigned by the processor and y the other enabled node, that is pushed into the bottom of the deque. Then,

$$\begin{aligned}\phi_i(u) - \phi_{i+1}(x) - \phi_{i+1}(y) &= 3^{2w(u)-1} - 3^{2w(x)} - 3^{2w(y)-1} \\ &= 3^{2w(u)-1} - 3^{2((w(u)-1)} - 3^{2(w(u)-1)-1} \\ &= 3^{2w(u)-1}\left(1 - \frac{1}{3} - \frac{1}{9}\right) \\ &= \frac{5}{9}\phi_i(u)\end{aligned}$$

So, we conclude that in this case, the potential decreases by $\frac{5}{9}\phi_i(u)$.

Finally, taking all the possible scenarios into account, it is straightforward to conclude that, due to the execution of u, the potential decreases by at least $\frac{5}{9}\phi_i(u)$. ■

For the remainder of the analysis, we make use of a few more definitions, first introduced in [Aro+01]. We denote the set of ready nodes attached to some processor p (*i.e.* the ready nodes in p's deque together with the node it has assigned, if any) at the beginning of some step i by $R_i(p)$. The potential associated with p at step i corresponds to the sum of potentials of the nodes attached to the processor at the beginning of that step

$$\Phi_i(p) = \sum_{u \in R_i} \phi_i(u)$$

For each step i, we partition the processors into two sets D_i and A_i, where the first is the set of all processors whose deque is not empty at the beginning of step i while the second is the set of all other processors (*i.e.* the set of all processors whose deque is empty at the beginning of that step). So, the potential of some step i, Φ_i, is composed by the potential associated with each of these two partitions

$$\Phi_i = \Phi_i(D_i) + \Phi_i(A_i)$$

where

$$\Phi_i(D_i) = \sum_{p \in D_i} \Phi_i(p) \qquad \text{and} \qquad \Phi_i(A_i) = \sum_{p \in A_i} \Phi_i(p).$$

The following lemma is a direct consequence of Corollary 4.3.1 and of the potential function's properties. The lemma is transcribed from [Aro+01, Top-Heavy Deques]. However, we present the full proof, rather than the simplified version (that corresponds to the base case of our proof) given in that same study.

Lemma 4.5. *Consider any step i and any processor $p \in D_i$. The top-most node u in p's deque contributes at least $\frac{3}{4}$ of the potential associated with p. That is, we have*

$$\phi_i(u) \geq \frac{3}{4}\Phi_i(p).$$

Proof. This lemma follows from Corollary 4.3.1. We prove it by induction on the number of nodes within p's deque.

Base case As the base case, consider that p's deque contains a single node u. The processor itself can either have or not an assigned node. For the second scenario, we have $\phi_i(u) = \Phi_i(p)$. Regarding the first case, let x denote p's assigned node. Corollary 4.3.1 implies that $w(u) \geq w(x)$. It follows

$$\begin{aligned}
\Phi_i(q) &= \phi_i(u) + \phi_i(x) \\
&= 3^{2w(u)} + 3^{2w(x)-1} \\
&\leq 3^{2w(u)} + 3^{2w(u)-1} \\
&= \frac{4}{3}\phi_i(u)
\end{aligned}$$

Thus, if p's deque contains a single node we have $\Phi_i(q) \leq \frac{4}{3}\phi_i(u)$.

Induction step Consider that p's deque now contains n nodes, where $n \geq 2$, and let u, x denote the topmost and second topmost nodes, respectively, within the deque. For the purpose of induction, let us assume the lemma holds for all the first $n-1$ nodes (*i.e.* without accounting with u):

$$\Phi_i(q) - \phi_i(u) \leq \frac{4}{3}\phi_i(x).$$

Corollary 4.3.1 implies $w(u) > w(x) \equiv w(u) - 1 \geq w(x)$. It follows

$$\begin{aligned}
\Phi_i(q) &\leq \frac{4}{3}\phi_i(x) + \phi_i(u) \\
&= \frac{4}{3}3^{2w(x)} + 3^{2w(u)} \\
&\leq \frac{4}{3}3^{2(w(u)-1)} + 3^{2w(u)} \\
&= \frac{4}{3}3^{2w(u)-2} + 3^{2w(u)} \\
&= 3^{2w(u)}\left(1 + \frac{4}{27}\right) \\
&= \frac{31}{27}\phi_i(u)
\end{aligned}$$

This concludes the proof of the lemma.

■

The following result is a consequence of Lemma 4.5. The proof we present follows the exact same arguments as the one given in that same study, and is only added as, alike for the proof of Lemma 4.3, it will later be extended.

Lemma 4.6. *Suppose a thief processor p chooses a processor $q \in D_i$ as its victim at some step j, such that $j \geq i$ (i.e. a steal attempt of p targeting q occurs at step j). Then, the potential decreases by at least $\frac{1}{2}\Phi_i(q)$ due to the assignment of the topmost node within q's deque.*

Proof. Let u denote the topmost node of q's deque at the beginning of step i.

Three possible scenarios may take place due to p's attempt to steal the topmost node of q's deque. We first show that in either of these cases, u gets assigned to a processor.

The deque is empty Since $q \in D_i$, some other processor successfully removed u from q's deque. Consequently, u was assigned by a processor.

The steal attempt aborts Since the deque implementation meets the relaxed semantics on any good set of invocations, and since, as aforementioned, the CFLFWS algorithm only makes good sets of invocations, then, some other processor successfully removed a topmost node from q's deque during the aborted steal attempt made by p. If the removed node was u, then u gets assigned to a processor (that may either be q or some other thief that successfully stole u). On the other hand, if the removed node was not u, then u must have been previously stolen, and consequently, assigned to some processor.

Successful steal attempt If p stole u, then, u gets assigned to p. Otherwise, some other processor removed u before p did, implying u got assigned to that other processor.

From Lemma 4.5, we have

$$\phi_i(u) \geq \frac{3}{4}\Phi_i(q).$$

Furthermore, Lemma 4.4 proves that if u gets assigned, the potential decreases by at least

$$\frac{2}{3}\phi_i(u).$$

Because u is assigned in any case, we conclude the potential associated with q decreases by at least

$$\frac{3}{4}\cdot\frac{2}{3}\Phi_i(q) = \frac{1}{2}\Phi_i(q).$$

This completes the proof of the lemma. ∎

A weaker version of the following lemma was originally presented in [Aro+01, Lemma 8]. Its proof uses Lemma 3.1, and follows the same arguments as the one presented in that study.

Lemma 4.7. *Consider any θ_r such that $0 \leq \theta_r < 1$, and, consider any step i and any later step j such that at least $\frac{P}{1-\theta_r}$ idle iterations occur from i (inclusive) to j (exclusive). Then we have*

$$P\left\{\Phi_i - \Phi_j \geq \frac{1}{4}.\Phi_i(D_i)\right\} > \frac{1}{4}.$$

Proof. By Lemma 4.6 we know that for each processor p in D_i that is targeted by a steal attempt, the potential drops by at least

$$\frac{1}{2}\Phi_i(p).$$

We can think of each idle iteration as a ball of Lemma 3.1, where the probability for being discarded (*i.e.* the probability that a steal attempt is not made) is θ_r. Thus, for each idle iteration, with probability $1 - \theta_r$ a steal attempt is made, entailing a *popTop* invocation to the deque of a processor targeted uniformly at random (*i.e.* a victim). For each processor p in D_i, we assign it a weight

$$W_p = \frac{1}{2}\Phi_i(p),$$

and for each other processor p in A_i, we assign it a weight

$$W_p = 0.$$

The weights sum to

$$W = \frac{1}{2}\Phi_i(D_i).$$

Using $\beta = \frac{1}{2}$, P as the number of bins and θ_r as the probability of discarding a ball, then, from Lemma 3.1 it follows that with probability at least

$$1 - \frac{1}{(1-\beta)e} > \frac{1}{4},$$

the potential decreases by at least

$$\beta W = \frac{1}{4}\Phi_i(D_i).$$

This concludes the proof of this lemma. ■

Finally, we obtain bounds on the expected runtime of a computation. The following result is an extension of the one originally presented in [Aro+01, Theorem 9]. The presented proof corresponds to a straightforward extension of the one originally presented for the just mentioned theorem.

Theorem 4.8. *Consider any θ_r such that $0 \le \theta_r < 1$, any computation with work T_1 and critical-path length T_∞ being executed by the CFLFWS algorithm with P processors in a dedicated environment. Let θ_r denote the argument passed to the scheduler. Then, the expected execution time is $O\left(\frac{T_1}{P} + \frac{T_\infty}{1-\theta_r}\right)$. Moreover, for any $\varepsilon > 0$, the execution time is $O\left(\frac{T_1}{P} + \frac{T_\infty + \log\left(\frac{1}{\varepsilon}\right)}{1-\theta_r}\right)$ with probability at least $1 - \varepsilon$.*

Proof. Lemma 4.2 bounds the execution time in terms of the number of idle iterations. We shall prove that the expected number of idle iterations is $O\left(\frac{PT_\infty}{1-\theta_r}\right)$, and that the number of idle iterations is $O\left((T_\infty + \ln(1/\varepsilon))\frac{P}{1-\theta_r}\right)$ with probability at least $1 - \varepsilon$.

To analyze the number of idle iterations, we break the execution into *phases* composed by $\Theta\left(\frac{P}{1-\theta_r}\right)$ idle iterations. Then, we prove that, with constant probability, a phase leads the potential to drop by a constant factor.

A computation's execution begins when its root, which, by our conventions related with the structure of the dags, is unique, gets assigned to an arbitrary processor. Consequently, since, by definition, the root has weight T_∞, then, at the beginning of a computation's execution the potential is $\Phi_0 = 3^{2T_\infty - 1}$. Furthermore, it is straightforward to conclude that the potential is 0 after and only after a computation's execution terminates. We then use these facts to bound the expected number of phases needed to decrease the potential down to 0. The first phase starts at step $t_1 = 1$, and ends at the first step $t_1{}'$ such that, at least $\frac{P}{1-\theta_r}$ idle iterations took place during the interval $[t_1, t_1{}']$, and, such that $t_1{}' - t_1 \geq C$. The second phase begins at step $t_2 = t_1{}' + 1$, and so on.

Consider two consecutive phases starting at steps i and j respectively. We now prove that

$$P\left\{\Phi_j \leq \frac{3}{4}\Phi_i\right\} > \frac{1}{4}.$$

Recall that we can partition the potential as

$$\Phi_i = \Phi_i(A_i) + \Phi_i(D_i).$$

Since, during each phase, at least $\frac{P}{1-\theta_r}$ idle iterations take place, then, by Lemma 4.7 it follows

$$P\left\{\Phi_i - \Phi_j \geq \frac{1}{4}\Phi_i(D_i)\right\} > \frac{1}{4}.$$

Now, we have to prove the potential also drops by a constant fraction of $\Phi_i(A_i)$. Consider some processor $p \in A_i$:

- If p does not have an assigned node, then

$$\Phi_i(p) = 0.$$

- Otherwise, if p has an assigned node u at step i, then,

$$\Phi_i(p) = \phi_i(u).$$

  Noting that each phase has at least C steps, then, processor p executes node u before the next phase begins (*i.e.* before step j). Thus, the potential drops by at least $\frac{5}{9}\phi_i(u)$ during that phase.

Cumulatively, for each $p \in A_i$, it follows

$$\Phi_i - \Phi_j \geq \frac{5}{9}\Phi_i(A_i).$$

Thus, no matter how Φ_i is partitioned between $\Phi_i(A_i)$ and $\Phi_i(D_i)$, we have

$$P\left\{\Phi_i - \Phi_j \geq \frac{1}{4}\Phi_i\right\} > \frac{1}{4}.$$

We say a phase is successful if it leads the potential to decrease by at least a $\frac{1}{4}$ fraction. So, a phase succeeds with probability at least $\frac{1}{4}$. Since the potential is an integer, and, as aforementioned, starts at $\Phi_0 = 3^{2T_\infty - 1}$ and ends at 0, then, there can be at most $(2T_\infty - 1)\log_{\frac{4}{3}}(3) < 8T_\infty$ successful phases. If we think of each phase as a coin toss, where the probability that we get heads is at least $\frac{1}{4}$, then, by the binomial distribution, to get heads $8T_\infty$ times, the expected number of coins we have to toss is at most $32T_\infty$. In the same way, the expected number of phases needed to obtain $8T_\infty$ successful ones is at most $32T_\infty$. Consequently, the expected number of phases is $O(T_\infty)$. Moreover, as each phase contains $O\left(\frac{P}{1-\theta_r}\right)$ idle iterations, the expected number of idle iterations is $O\left(\frac{PT_\infty}{1-\theta_r}\right)$. Taking into account Lemma 4.2, we conclude the expected runtime of a computation is at most

$$O\left(\frac{T_1}{P} + \frac{T_\infty}{1-\theta_r}\right)$$

We now turn to the high probability bound.

Suppose the execution takes $n = 32T_\infty + m$ phases. Each phase succeeds with probability at least $p = 1/4$, so the expected number of successes is at least

$$np = 8T_\infty + \frac{m}{4}.$$

We now compute the probability that the number of X successes is less than $8T_\infty$. We use *Chernoff bound* [AS92],

$$P\{X < np - a\} < e^{-\frac{a^2}{2np}}$$

with $a = \frac{m}{4}$. It follows,

$$np - a = 8T_\infty + \frac{m}{4} - \frac{m}{4} = 8T_\infty.$$

Thus, if we choose $m = 32T_\infty + 16\ln\left(\frac{1}{\varepsilon}\right)$, it follows

$$\begin{aligned}
P\{X < 8T_\infty\} &< e^{-\frac{\left(\frac{m}{4}\right)^2}{2\left(8T_\infty + \frac{m}{4}\right)}} \\
&= e^{-\frac{\left(\frac{m}{4}\right)^2}{16T_\infty + \frac{m}{2}}} \\
&\le e^{-\frac{\left(\frac{m}{4}\right)^2}{\frac{m}{2} + \frac{m}{2}}} \\
&= e^{-\frac{(m/4)^2}{m}} \\
&= e^{-\frac{m}{16}} \\
&\le e^{-\frac{16\ln\left(\frac{1}{\varepsilon}\right)}{16}} \\
&= e^{\ln(\varepsilon)} \\
&= \varepsilon
\end{aligned}$$

Thus, the probability that the execution takes $64T_\infty + 16\ln\left(\frac{1}{\varepsilon}\right)$ phases or more, is less than ε. Finally, we conclude that the number of idle iterations is

$$O\left(\frac{\left(T_\infty + 16\ln\left(\frac{1}{\varepsilon}\right)\right)P}{1-\theta_r}\right)$$

with probability at least $1-\varepsilon$, implying that the execution time is

$$O\left(\frac{T_1}{P}+\frac{T_\infty+\log\left(\frac{1}{\varepsilon}\right)}{1-\theta_r}\right)$$

also with probability at least $1-\varepsilon$. ■

Chapter 5

The Synchronous Adaptive Work Stealing and Sharing algorithm

Although LFWS is a provably efficient scheduler, it is often not capable to achieve satisfactory performance for unbalanced scenarios, where only a small fraction of the processors actually generate all the work [HS02a; Ber+03; Mit98]. In this chapter, we present a synchronous scheduling algorithm that overcomes this limitation.

First, we acquaintance the reader with the synchronous execution environment. Then, we present a synchronized version of the LFWS scheduler, that is behaviorally equivalent to the original. Next, we specify our proposed algorithm, baptized Canonical Flexible Synchronous Adaptive Work Stealing and Sharing (CFSAWSS), and explain how it can ensure good performance for the aforementioned scenarios while maintaining provably good expected runtime bounds. After that, we show that our proposal overcomes these limitations, obtaining lower bounds on its performance gains when compared against the synchronous variant of LFWS. Finally, we prove that the algorithm is stable for scenarios with unbalanced parallelism (*i.e.* the expected amount of work per processor over time is bounded).

5.1 A synchronous execution environment

One of the main goals of this chapter is to compare WS's load balancer with the one we propose. To that end, we analyze the performance of these algorithms by modeling their behavior and studying their effectiveness. However, performing a precise and fair comparison while assuming an asynchronous environment, where different processors may be in completely distinct execution phases, is very complex. For that reason, we simulate a synchronous environment that will allow us to compare, as objectively as possible, the effectiveness of these schedulers. We now move to specify how we setup such an environment.

Recall the definitions of scheduling iterations, busy and idle iterations, and, milestones (see Section 4.3.1 and Section 4.5.1). Moreover, remind that a computation's execution can be partitioned into a set of iterations, that each iteration contains one milestone, and, that a processor can execute at most a constant number of instructions between two successive milestones. Again, let C be a large enough constant such that, for any sequence of C or more instructions executed by a processor, at least one of them is a milestone. Since each scheduling iteration contains a milestone, and, because strictly less than C instructions can be executed between two consecutive milestones, we conclude that each scheduling iteration of LFWS has constant length.

Now, we further partition the execution of a computation, splitting each scheduling iteration through a chain of *stages*. A stage then corresponds to a sub-sequence of instructions of an enclosing scheduling iteration. The mode how each iteration is partitioned into a set of stages is scheduler and study-dependent.

Remind the specification of the LFWS scheduler (depicted in Algorithm 1). For this study we partition each scheduling iteration of this algorithm into two stages:

STAGE I During the first stage of a busy iteration, the processor executes its assigned node. If the iteration is idle, however, the processor makes a steal attempt, and, in case that the attempt succeeds, the stolen node becomes assigned to the processor.

STAGE II When in the presence of a busy iteration, the node executed in the previous stage enabled either 0, 1 or 2 nodes. If no node was enabled, the processor attempts to pop one from the bottom of its deque. If one was enabled, it simply becomes the processor's assigned node. In the third case, one of the enabled nodes becomes the processor's assigned node, whist the other is pushed into the bottom of the processor's deque. Finally, if the iteration is an idle one, the processor takes no action during this stage.

Since any scheduling iteration of LFWS has constant length, then, each of the stages composing any iteration also has constant length. Moreover, even though we do not present it here, the maximum possible length of each stage composing an iteration can be computed, by statically analyzing the algorithm, implying it can be known *a priori* [1].

Measuring the maximum possible length of a stage.
As the algorithm's specification is given at a very high level, measuring the length of the longest possible sequence of instructions composing a stage becomes a painful task. In fact, the translation of the algorithm into a sequence of instructions that can be executed by a real

[1]As one might note, we have not concretely specified which deque implementation we assume. Thus, in order to compute the maximum length of each stage we would have to first fix a particular implementation.

processor is instruction-set and compiler-dependent. For that reason, we simply assume to know the maximum possible length of each stage of any scheduling iteration.

We now introduce a theoretical routine, *awaitForNextStage*, that plays a crucial role in the synchronization guarantees we require for the performance analysis. The routine takes two input parameters:

$stageLength$ — The length of a longest sequence of instructions that a given stage of any enclosing iteration may have before calling the routine.

$instructionsExecuted$ — The length of the sequence of instructions executed by the caller (*i.e.* the processor) during the stage, before calling the routine.

To synchronize every processor, the routine only has to inject an artificial delay with length $stageLength - instructionsExecuted$[2]. Due to its semantics, the routine should only be called after every other instruction of the stage has been executed. It is also required that the value of $stageLength$ is constant and known beforehand.

Consider a set of P processors that started executing the sequence of instructions corresponding to some stage s of a scheduling iteration at the exact same step, in a dedicated environment. If each of these processors calls *awaitForNextStage* right before finishing s, then they will also return from the routine at the exact same step. Consequently, the synchronous execution environment that we need for the comparison of our proposal against the synchronized variant of LFWS depends only on the following three conditions:

1. The maximum possible length of each of the stages composing a scheduling iteration must be known beforehand.

2. The *awaitForNextStage* routine must be called at the end of each stage.

3. All processors must begin the first stage of the first scheduling iteration at the same step.

Given that we are in the presence of a dedicated execution environment, it is reasonable to assume that all processors begin the execution of the computation, and, hence, of the initial stage of the first scheduling iteration, at the same step.

Having described how to obtain a synchronous execution setting (on a dedicated environment), we now specify the synchronized version of LFWS, to which we refer to as Synchronous Lock Free Work Stealing (SLFWS). The scheduler is depicted in Algorithm 3, and, alike for the original (asynchronous) algorithm, we assume a constant-time deque implementation that meets the relaxed semantics, specified in Section 4.4. Consider the original version of the scheduler, depicted in Algorithm 1, and the mode how each scheduling iteration is decomposed into two stages. We know that the length of each

[2]To inject an artificial delay of some length (or duration) l, one has to simply add a sequence of instructions with no effect (*i.e.* idle instructions) of length l.

iteration can be computed *a priori*, and, is constant. So, assuming that all processors begin a computation's execution at the same step, the addition of a call to the *awaitForNextStage* routine at the end of each stage ensures that all processors conclude the execution of the first stage at the same step, and consequently, also begin and conclude the execution of the second stage synchronized. Generalizing, all processors start and conclude every stage of each scheduling iteration at the exact same step.

```
procedure Scheduler
    while computation not terminated do
        if ValidNode(assigned) then
            enabled ← execute(assigned)
            awaitForNextStage(STAGE1_LENGTH, $\iota$())
            if length(enabled) > 0 then
                assigned ← enabled[0]
                if length(enabled) = 2 then
                    self.deque.pushBottom(enabled[1])
                end if
            else
                assigned ← self.deque.popBottom()
            end if
        else
            self.WorkMigration()
            awaitForNextStage(STAGE1_LENGTH, $\iota$())
        end if
        awaitForNextStage(STAGE2_LENGTH, $\iota$())
    end while
end procedure

procedure WorkMigration
    victim ← UniformlyRandomProcessor()
    assigned ← victim.deque.popTop()
end procedure

function ValidNode(node)
    return node ≠ EMPTY and node ≠ ABORT and node ≠ None
end function
```

Algorithm 3: The SLFWS algorithm. Constants STAGE1_LENGTH and STAGE2_LENGTH denote the maximum possible length of STAGE I and STAGE II, respectively, of any scheduling iteration. Function $\iota()$ returns the number of instructions executed by the processor, thus far, in the current stage. This value can be easily computed by resetting an internal variable at the end of each call to *awaitForNextStage*, and by counting the number of instructions the processor executed from that step up to the current.

Behavioral equivalence of SLFWS and LFWS.
As one can conclude, SLFWS is behaviorally equivalent to LFWS. In fact, the only difference between these two algorithms is that, to ensure a synchronous execution, SLFWS adds, at the end of each stage, a constant time delay.

Finally, we introduce the concept of *rounds*. Informally, a round is the time interval during which a processor (synchronously) executes a scheduling iteration. This definition builds from the synchronous execution environment we have artificially created. More precisely, a round is a sequence of steps such that, at the first of those steps every processor working on the computation begins the execution of a scheduling iteration, and, at the last one, all those processors conclude the execution of that same iteration.

As aforementioned, any scheduling iteration of LFWS has constant length. Consequently, all of the stages composing each iteration have constant length. Taking into account the specification of *awaitForNextStage*, it is straightforward to deduce that it only adds a delay of constant length at the end of each stage. Consequently, any iteration of SLFWS also has constant length. To conclude, since each processor executes one instruction per step, and, as we are assuming a dedicated execution environment, then, the sequence of steps composing each round of the SLFWS algorithm has constant length.

5.2 The Canonical Flexible Synchronous Adaptive Work Stealing and Sharing algorithm

We now present the synchronous version of our proposed scheduling algorithm. The Canonical Flexible Synchronous Adaptive Work Stealing and Sharing (CFSAWSS) scheduler is based on the CFLFWS algorithm (see Section 4.5) and can be though of as an extension to the canonical version of that same algorithm. The key difference when compared to CFLFWS is that, rather than relying on a purely receiver initiated load balancer, we allow processors that are generating work to actually donate it. As we will show, this strategy plays a crucial role in overcoming the limitations of WS for unbalanced scenarios, while still ensuring high performance for the general setting. In this algorithm processors can use custom load balancing schemes for both stealing and spreading work. Some possible uses of custom sender initiated load balancing schemes will later be exemplified.

Alike in the CFLFWS algorithm, each processor owns a lock free deque. Moreover, each processor also owns two flags and a memory cell, that uses to communicate with other processors. The first one, named the *state* flag, stores the current status of the processor: WORKING, IDLE or marked as target of a spread attempt (more on this later). The second, labeled *spreading* flag, stores a boolean value (*i.e.* TRUE or FALSE) that indicates whether the processor should attempt to spread nodes or not. Lastly, we refer to the memory cell as the *donation* cell because it is used to send ready nodes to other processors. Each processor is uniquely identified by a number and can be accessed by its corresponding *id* in constant time (implementable, for example, by using an array of processors).

The algorithm also makes use of the Compare-and-swap (CAS) instruction, with its conventional semantics, depicted in Algorithm 4. As usual, only one CAS targeting the same memory location can successfully execute at each step. We assume that, in situations where more than one processor may successfully apply this instruction over the same memory address and at the same time, one of them is chosen uniformly at random to succeed, implying the failure of the instruction for the remainder.

```
1: function CAS($m, old, new$)
2:     atomically do
3:         if Memory[$m$] = $old$ then
4:             Memory[$m$] ← $new$
5:             return SUCCESS
6:         else
7:             return FAILURE
8:         end if
9:     done
10: end function
```

Algorithm 4: The semantics of the CAS instruction.

The CFSAWSS scheduler, depicted in Algorithm 5, takes as input three parameters:

θ_r — Corresponds to the probabilities that an idle processor (*i.e.* a processor executing an idle iteration) chooses to skip a steal attempt, or, not to make an update to some processor's *spreading* flag (more on this ahead).

θ_s — The probability that a processor, with its *spreading* flag set to TRUE, chooses not to attempt to spread the node it did not assign after enabling two nodes.

heuristic — Defines in what situations processors change their status from victims to victims and donors, and vice versa. Let

$$\mathcal{D} = \{\text{TRUE}, \text{FALSE}, \text{UNCHANGED}\}$$

denote the domain of values that are used to update the *spreading* flag. Then,

$$heuristic \in (\mathcal{D} \times \mathcal{D} \times \mathcal{D}) \times (\mathcal{D} \times \mathcal{D} \times \mathcal{D} \times \mathcal{D})$$

being that the first element of the pair is a triple with the values according to which a processor must update its *spreading* flag after, respectively, an unmade[3], a successful or a failed spread attempt. For example, for values (UNCHANGED, TRUE, UNCHANGED), the processor sets its *spreading* flag to TRUE every time it succeeds attempting to spread a node, and leaves it unchanged otherwise. The

[3]We say that an unmade spread attempt is one that the processor decided not to make.

second element of *heuristic* is a quadruple featuring the values used by thieves to update the *spreading* flag of processors. Concretely, after a thief makes, or, if it is the case, skips a steal attempt, it might then update the *spreading* flag of another processor p_j, targeted uniformly at random. If the thief decides to proceed with the update, it makes it according to the outcome of its last steal opportunity:

- If the thief skipped the opportunity to make the attempt, it updates p_j's flag according to the value of the first field of the quadruple.
- However, if it did not skip, the thief updates p_j's *spreading* flag according to the outcome of the *popTop* invocation associated with that attempt: one of *success*, *empty*, or *abort*.

When a computation's execution begins, every processor, except the one to which the root is assigned, has its *state* flag set to IDLE and its *assigned* node set to NONE. The *state* flag of the processor to which the root is assigned is initially set to WORKING. Further, the *donation* cell of every processor begins with the value NONE. Finally, the initial value of the *spreading* flag varies (more on this later).

As for LFWS, it is trivial to conclude that all sets of invocations made to any deque are good[4], as only the owner of each deque may invoke the *pushBottom* and *popBottom* methods, implying the relaxed semantics are always met (see Section 4.4).

We partition the execution of each scheduling iteration of CFSAWSS into three stages. Equivalently to SLFWS, processors' synchronization is ensured by calling the *awaitForNextStage* routine at the end of each stage. For the sake of succinctness, we omit these calls from the description of the stages.

Consider some processor p participating on a computation's execution.

STAGE I —

Busy iteration p executes its currently assigned node (line 4).

Idle iteration p starts by randomly choosing whether to make a steal attempt or not (lines 59 to 61).

Decides to make a steal attempt p picks a victim uniformly at random, and invokes the *popTop* method to its deque (lines 62 and 63).

Decides not to make a steal attempt p simply skips the opportunity.

Then, p randomly chooses to either make an update or not (lines 65 and 66).

Makes an update p chooses a new processor, uniformly at random, and updates this processor's *spreading* flag according to the outcome of its last steal attempt opportunity (lines 67 to 76).

[4] As specified in Section 4.4, a set of invocations is said to be *good* if and only if *pushBottom* and *popBottom* are never invoked concurrently.

Does not make an update p simply skips the opportunity for updating another processor's *spreading* flag.

Finally, p checks if it has an assigned node, and, if it is the case, the processor sets its state to WORKING, indicating it has an assigned node (lines 78 to 80).

STAGE II —

Busy iteration The node executed by *p* in the previous stage enabled either 0, 1 or 2 other nodes.

0 nodes enabled *p* takes no action during this stage.

1 node enabled *p* gets the only enabled node assigned (line 7).

2 nodes enabled One of the enabled nodes becomes *p*'s new assigned node (line 7). Then, *p* checks its *spreading* flag (line 10).

If *p*'s *spreading* flag is set to TRUE —

The processor decides whether to attempt spreading the non assigned node or not (lines 43 and 44). If it chooses not to make the spread attempt, it takes no further action during this stage. Otherwise, the processor attempts to spread the node it did not assign (lines 45 and 46), and updates its *spreading* flag according to the outcome of that attempt (lines 48 to 55).

If *p*'s *spreading* flag is set to FALSE —

p takes no action during this stage.

Idle iteration *p* does not perform any action during this stage.

STAGE III —

Busy iteration The node executed by *p* in the first stage enabled 0, 1 or 2 other nodes.

0 nodes enabled *p* attempts to pop a ready node from the bottom of its deque, invoking method *popBottom* (line 22). If the deque is not empty, a node is returned. In such case, that node becomes *p*'s assigned node. Otherwise, *p* sets its state to IDLE, indicating it does not have an assigned node (lines 23 to 25).

1 node enabled *p* takes no action in this stage.

2 nodes enabled If *p* made a successful spread attempt in the previous stage, it takes no action during this stage. Otherwise, *p* then pushes the node into the bottom of its deque by invoking the method *pushBottom* (line 15).

Idle iteration *p* starts by checking the value of its *state* flag (line 83).

***p*'s *state* flag is set to WORKING** In this case, *p* successfully stole a ready node during the first stage. As such, *p* takes no further action during this stage.

Otherwise *p* checks if it has been offered some node (line 83). If so, it assigns the donated node and sets its state to WORKING (lines 84 to 86). If *p*'s *state* flag is set to IDLE, then no node has been offered, and thus, no further action is taken during this stage.

Finally, we assume that any call to *UniformlyRandomNumber* upon the same interval takes the same number of steps. Moreover, we also assume that any call to *UniformlyRandomProcessor* always takes the same number of steps too.

```
procedure SCHEDULER($\theta_r$, $\theta_s$, $heuristic$)
    while computation not terminated do
        if ValidNode(assigned) then                                  ▷ Busy iteration
            enabled ←execute(assigned)
            awaitForNextStage(STAGE1_LENGTH, $\iota$())
            if length(enabled)> 0 then
                assigned ← enabled[0]
                if length(enabled) = 2 then
                    result ← FAILURE
                    if self.spreading then
                        result ← self.spread(enabled[1], $\theta_s$, heuristic[0])
                    end if
                    awaitForNextStage(STAGE2_LENGTH, $\iota$())
                    if result ≠ SUCCESS then
                        self.deque.pushBottom(enabled[1])
                    end if
                else
                    awaitForNextStage(STAGE2_LENGTH, $\iota$())
                end if
            else
                awaitForNextStage(STAGE2_LENGTH, $\iota$())
                assigned ← self.deque.popBottom()
                if not ValidNode(assigned) then
                    self.state ← IDLE
                end if
            end if
        else                                                         ▷ Idle iteration
            self.WorkMigration($\theta_r$, heuristic[1])
        end if
        awaitForNextStage(STAGE3_LENGTH, $\iota$())
    end while
end procedure

function VALIDNODE(node)
    return node ≠EMPTY and node ≠ ABORT and node ≠ NONE
end function

procedure UPDATE(new)
    if new ≠ UNCHANGED then
        self.spreading ← new
    end if
end procedure
```

Algorithm 5: The CFSAWSS algorithm.

```
function SPREAD($\mu$, $\theta_s$, $spreading\_heuristic$)
    $result \leftarrow$ NONE
    $choice \leftarrow UniformlyRandomNumber([0;1])$
    if $choice \geq \theta_s$ then
        $donee \leftarrow UniformlyRandomProcessor()$
        $result \leftarrow$CAS($donee.state$, IDLE, $self.id()$)
    end if
    if $result$ = FAILURE then
        $self$.update($spreading\_heuristic[2]$)
    else if $result$ = SUCCESS then
        $self.donation \leftarrow \mu$
        $self$.update($spreading\_heuristic[1]$)
    else                                        ▷ Did not attempt spreading the node
        $self$.update($spreading\_heuristic[0]$)
    end if
    return $result$
end function

procedure WORKMIGRATION($\theta_r$, $stealing\_heuristic$)
    $choice_1 \leftarrow UniformlyRandomNumber([0;1])$
    $assigned \leftarrow$ NONE
    if $choice_1 \geq \theta_r$ then
        $victim \leftarrow UniformlyRandomProcessor()$
        $assigned \leftarrow victim.deque$.popTop()
    end if
    $choice_2 \leftarrow UniformlyRandomNumber([0;1])$
    if $choice_2 \geq \theta_r$ then
        $updated \leftarrow UniformlyRandomProcessor()$ ▷ Pick another processor to update
        if $assigned$ = ABORT then
            $updated$.update($stealing\_heuristic[3]$)
        else if $assigned$ = EMPTY then
            $updated$.update($stealing\_heuristic[2]$)
        else if ValidNode($assigned$) then                  ▷ Successful steal attempt
            $updated$.update($stealing\_heuristic[1]$)
        else                                            ▷ Did not make a steal attempt
            $updated$.update($stealing\_heuristic[0]$)
        end if
    end if
    if ValidNode($assigned$) then
        $self.state \leftarrow$ WORKING
    end if
    awaitForNextStage(STAGE1_LENGTH, $\iota()$)
    awaitForNextStage(STAGE2_LENGTH, $\iota()$)
    if $self.state \neq$ IDLE and $self.state \neq$ WORKING then
        $donor \leftarrow processor[self.state]$         ▷ $self.state$ contains the id of the donor
        $assigned \leftarrow donor.donation$
        $self.state \leftarrow$ WORKING
    end if
end procedure
```

5.2.1 Bounds on the expected runtime

Now, we obtain bounds on the expected runtime of a computation using the CFSAWSS algorithm. The analysis we make of this algorithm regarding its runtime bounds is almost equivalent to the one presented in Section 4.5.1 for the CFLFWS scheduler. For the sake of succinctness, we omit the proofs of the results for which the arguments given in the proofs of the analogous lemmas/theorems for the CFLFWS algorithm apply.

It is trivial to conclude that the execution of any scheduling iteration entails the execution of a milestone:

Busy Iteration If a processor is executing a busy iteration, then, at the beginning of that same iteration's execution it had an assigned node. Thus, by observing the algorithm (depicted in Algorithm 5), we conclude that the processor reaches the milestone when it executes its assigned node (line 4).

Idle Iteration Right after beginning the execution of an idle iteration, the processor calls the *WorkMigration* procedure. In this case, the milestone is then the last instruction executed by the processor right before returning from the procedure's call (line 88).

Assuming that the sequences of instructions associated with targeting a processor uniformly at random and with picking a number uniformly at random from a given interval have constant length, it is easy to see that the sequences of executed instructions preceding each of the calls made to the *awaitForNextStage* routine have constant length. Taking into account the semantics of the routine, it is straightforward to conclude that any of the stages composing any scheduling iteration has constant length.

With this, and by following the exact same arguments as the ones given in the proof of Lemma 4.2, we can deduce the following lemma:

Lemma 5.1. *Consider any computation with work T_1 being executed by the CFSAWSS. Then, the execution time is $O\left(\frac{T_1}{P} + \frac{I}{P}\right)$, where I denotes the total number of idle iterations executed by processors.*

Now, we prove that the statement made in Lemma 4.3 for the CFLFWS also holds for the CFSAWSS algorithm. In fact, the arguments given in the proof of that same lemma apply for this one too, except when a processor enables two nodes. We slightly adapt the proof of the aforementioned lemma (that was originally transcribed from [Aro+01, Lemma 3]) to show that the result holds for CFSAWSS.

Lemma 5.2 (Structural Lemma). *Let k be the number of nodes in a given deque at some time in the (linearized) execution of the CFSAWSS algorithm, and let $v_1, \ldots, v_k$ denote those nodes ordered from the bottom of the deque to the top. Let v_0 denote the assigned node if there is one. In addition, for $i = 0, \ldots, k$ let u_i denote the designated parent of v_i. Then for $i = 1, \ldots, k$, node u_i is an ancestor of u_{i-1} in the enabling tree. Moreover, though we may have $u_0 = u_1$, for $i = 2, 3, \ldots, k$, we have $u_{i-1} \neq u_i$ that is, the ancestor relationship is proper.*

Proof. Settle for a particular deque. The deque state and assigned node only change when the latter is executed or a thief performs a successful steal. We prove the claim by induction on the number of assigned-node executions and steals since the deque was last empty. In the base case, if the deque is empty, then the claim holds vacuously. We now assume that the claim holds before a given assigned-node execution or successful steal, and we will show that it holds afterwards. Specifically, before the assigned-node execution or successful steal, let v_0 denote the assigned node; let k denote the number of nodes in the deque; let $v_1,\dots,v_k$ denote the nodes in the deque ordered from the bottom to top; and for $i = 0,\dots,k$, let u_i denote the designated parent of v_i. We assume that either $k = 0$, or for $i = 1,\dots,k$, node u_i is an ancestor of u_{i-1} in the enabling tree, with the ancestor relationship being proper, except possibly for the case $i = 1$. After the assigned-node execution or successful steal, let $v_0{}'$ denote the assigned node; let k' denote the number of nodes in the deque; let $v_1{}',\dots,v_k{}'$ denote the nodes in the deque ordered from bottom to top; and for $i = 1,\dots,k'$, let $u_i{}'$ denote the designated parent of $v_i{}'$. We now show that either $k' = 0$, or for $i = 1,\dots,k'$, node $u_i{}'$ is an ancestor of $u_{i-1}{}'$ in the enabling tree, with the ancestor relationship being proper, except possibly for the case $i = 1$.

Consider the execution of the assigned node v_0 by the owner.

If 0 or 1 nodes are enabled, the arguments given in the proof of Lemma 4.3 for the corresponding analogous scenarios apply, meaning that the result holds for these cases.

Consider that the execution of v_0 (*i.e.* the assigned node) enables 2 children: x and y, and let y denote the new assigned node of the processor. In this case, both (v_0,x) and (v_0,y) are enabling edges. By the algorithm's specification, the processor can either successfully spread x or not.

- If it does not successfully spread x, then it pushes x onto the bottom of its deque (at STAGE III), implying $k' = k+1$. For this case, the arguments given in the proof of Lemma 4.3 for the case where 2 nodes are enabled, apply. As such, we now rename the nodes, setting $v_0{}' = y$; $u_0{}' = v_0$; $v_1{}' = x$; $u_1{}' = v_0$; and for $i = 2,\dots,k'$, setting $v_i{}' = v_{i-1}$ and $u_i{}' = u_{i-1}$. Next, we now observe that $u_1{}' = u_0{}'$, and for $i = 2,\dots,k'$, node $u_i{}'$ is a proper ancestor of $u_{i-1}{}'$ in the enabling tree. That $u_2{}'$ is a proper ancestor of $u_1{}'$ in the enabling tree follows from the fact that (u_0,v_0) is an enabling edge, concluding the proof for this case.

- Otherwise, x was successfully spread. Thus, the designated parent of y is v_0; and $k' = k$. If $k = 0$, then $k' = 0$. If not, we can rename the nodes as follows. We set $v_0{}' = y$; we set $u_0{}' = v_0$; and for $i = 1,\dots,k'$, we set $v_i{}' = v_i$ and $u_i{}' = u_i$. We now observe that for $i = 1,\dots,k'$, node $u_i{}'$ is a proper ancestor of $u_{i-1}{}'$ in the enabling tree. That $u_1{}'$ is a proper ancestor of $u_0{}'$ in the enabling tree follows from the fact that (u_0,v_0) is an enabling edge, completing the proof of the lemma.

■

Corollary 5.2.1. *If $v_0, v_1, \ldots, v_k$ are as defined in the statement of Lemma 5.2, then we have $w(v_0) \leq w(v_1) < \ldots < w(v_{k-1}) < w(v_k)$.*

Now, we reintroduce some concepts already presented in Section 4.5.1.

- The set of nodes that are ready at some step i is denoted by R_i.
- The potential associated with any node u at some step i is denoted $\phi_i(u)$ and is defined as
$$\phi_i(u) = \begin{cases} 3^{2w(u)-1} & \text{if } u \text{ is assigned} \\ 3^{2w(u)} & \text{otherwise} \end{cases}.$$
- The total potential at step i is denoted by Φ_i and is defined as
$$\Phi_i = \sum_{u \in R_i} \phi_i(u).$$

The following result is analogous to Lemma 4.4, but considering, instead the CFSAWSS algorithm.

Lemma 5.3. *Consider some node u, ready at step i during the execution of a computation. If u gets assigned to a processor at that step, the potential drops by at least $\frac{2}{3}\phi_i(u)$. More, if u was already assigned to a processor and gets executed at that step i, the potential drops by at least $\frac{5}{9}\phi_i(u)$.*

Proof. Following the reasoning made in the proof of Lemma 4.4's first claim, it is straightforward to see that the claim holds for this lemma as well.

Regarding the second one, note that, due to our conventions regarding computations' structure, a node can be the designated parent of at most two other ones in the enabling tree.

For the case when 0 or 1 nodes are enabled, by following the same arguments given in the proof of Lemma 4.4, it is trivial to conclude that the claim holds (for these two scenarios). With this, it only remains to prove that the claim continues to hold when two nodes are enabled. Let x and y denote the enabled nodes, where x is the one that the processor gets assigned. In the CFSAWSS algorithm, the processor may either successfully spread y or not. For the second case, where y is not spread, the processor then pushes y into the bottom of its deque. Following the same arguments as the ones given in the proof of Lemma 4.4, we can conclude that the potential decreases by at least $\frac{5}{9}\phi_i(u)$. Regarding the first case, since y was successfully spread, it then becomes assigned to the donee (*i.e.* the processor who received y). By the potential function we then have

$$\begin{aligned}\phi_i(u) - \phi_{i+1}(x) - \phi_{i+1}(y) &= 3^{2w(u)-1} - 3^{2w(x)-1} - 3^{2w(y)-1} \\ &\geq 3^{2w(u)-1} - 3^{2w(x)} - 3^{2w(y)-1} \\ &= \frac{5}{9}\phi_i(u)\end{aligned}$$

which concludes the proof of this lemma. ■

Again, we reintroduce more definitions already presented in Section 4.5.1.

- The set of ready nodes attached to a processor p at the beginning of a step i is denoted by $R_i(p)$.
- The potential associated with p at step i is denoted $\Phi_i(p)$ and is defined as

$$\Phi_i(p) = \sum_{u \in R_i} \phi_i(u)$$

For each step i, processors are partitioned into two sets D_i and A_i, where the first is composed by all processors whose deque is not empty at the beginning of step i while the second is the set of all other processors. So, it follows $\Phi_i = \Phi_i(D_i) + \Phi_i(A_i)$, where

$$\Phi_i(D_i) = \sum_{p \in D_i} \Phi_i(p) \qquad \text{and} \qquad \Phi_i(A_i) = \sum_{p \in A_i} \Phi_i(p).$$

The following lemma follows from Corollary 5.2.1, and from the potential function's properties. Its proof is omitted since the arguments presented in the proof of Lemma 4.5 for the CFLFWS algorithm apply for this case as well.

Lemma 5.4. *Consider any step i and any processor $p \in D_i$. The top-most node u in p's deque contributes at least $\frac{3}{4}$ of the potential associated with p. That is, we have*

$$\phi_i(u) \geq \frac{3}{4}\Phi_i(p).$$

The next result is a consequence of Lemma 5.4 and from the fact that the deque's implementation satisfies the relaxed semantics (see Section 4.4). The proof of the lemma is omitted because the reasoning made in the proof of Lemma 4.6 for CFLFWS, applies to CFSAWSS too.

Lemma 5.5. *Suppose a thief processor p chooses a processor $q \in D_i$ as its victim at some step j, such that $j \geq i$ (i.e. a steal attempt of p targeting q occurs at step j). Then, the potential decreases by at least $\frac{1}{2}\Phi_i(q)$ due to the assignment of the topmost node within q's deque.*

The following result is a consequence of Lemmas 3.1 and 5.5. Again, the proof is similar to the one of Lemma 4.7, being omitted for that reason.

Lemma 5.6. *Consider any θ_r such that $0 \leq \theta_r < 1$, and, consider any step i and any later step j such that at least $\frac{P}{1-\theta_r}$ idle iterations occur from i (inclusive) to j (exclusive). Then, we have*

$$P\left\{\Phi_i - \Phi_j \geq \frac{1}{4}.\Phi_i(D_i)\right\} > \frac{1}{4}.$$

Finally, we can bound the expected runtime of computations run under the CFSAWSS algorithm. The result follows from Lemmas 5.1 and 5.6. It can be proved using the exact same reasoning given in the proof of Theorem 4.8, and, for that reason its proof is omitted.

Theorem 5.7. *Consider any θ_r such that $0 \leq \theta_r < 1$, any computation with work T_1 and critical-path length T_∞ being executed by the CFSAWSS algorithm with P processors in a dedicated environment. Let θ_r denote the value of the first parameter passed to the scheduler. Then, the expected execution time is $O\left(\frac{T_1}{P} + \frac{T_\infty}{1-\theta_r}\right)$. Moreover, for any $\varepsilon > 0$, the execution time is $O\left(\frac{T_1}{P} + \frac{T_\infty + \log\left(\frac{1}{\varepsilon}\right)}{1-\theta_r}\right)$ with probability at least $1 - \varepsilon$.*

5.3 Modeling the performance of the schedulers

In this section, we model, analyze and compare the performance of CFSAWSS and SLFWS. To study their effectiveness, we model each one's behavior for a single round.

5.3.1 Notation and assumptions

We denote the number of processors working on a computation's execution by P. For each round, we denote the number of idle processors (*i.e.* processors executing an idle iteration) at the beginning of the round by P_i, and the number of processors with an empty deque, also at the beginning of the round by P_e. We can alternatively denote P_i and P_e as fractions of the number of processors, where $P_i = \alpha P$, and, $P_e = \beta P$. As every idle processor's deque is empty, it follows that $\beta \geq \alpha$, implying, $P_e \geq P_i$. Finally, we denote the number of processors enabling two nodes by D. Since only the busy processors (*i.e.* processors executing a busy iteration) have the opportunity to enable nodes, it is trivial to conclude that $D \leq P(1-\alpha)$.

We make the following assumptions:

- $P > 0$ — There is at least one processor executing the computation.
- $\alpha < 1$ — The number of busy processors (*i.e.* the number of processors executing a busy iteration) is not null.
- $\alpha > 0$ — If every processor was busy, no load balancing would take place. Thus, for these situations, our analyzes of the load balancers would be meaningless.
- $\theta_r < 1$ — As before, we do not allow all steal opportunities to be skipped.
- $\theta_s < 1$ — As for the previous case, we do not allow all spreading opportunities to be skipped.

5.3.2 Two theoretical heuristics

Now, we present two theoretical heuristics, according to which we will study the performance of CFSAWSS's load balancer.

Let T, F and U denote shortenings for TRUE, FALSE and UNCHANGED, respectively. Recall that

$$heuristic \in (\mathscr{D} \times \mathscr{D} \times \mathscr{D}) \times (\mathscr{D} \times \mathscr{D} \times \mathscr{D} \times \mathscr{D}),$$

where $\mathscr{D} = \{T, F, U\}$.

5.3.2.1 Greedy heuristic

For this heuristic, processors are always willing to make spread attempts. As such, the initial value of each processor's *spreading* flag is T, and *heuristic* = ((T, T, T), (T, T, T, T)).

Unfortunately, this strategy often incurs in high communication overheads, severely harming the scheduler's performance. For example, if most processors are busy, attempting to spread nodes is counterproductive, and only results in unnecessary overheads. To mitigate the excessive communication overheads incurred by this greedy spreading approach, we will next introduce the Tit-for-tat heuristic that allows to avoid unnecessary communication costs, while still benefiting from the sender-initiated nature of the spreading mechanism.

5.3.2.2 Tit-for-tat heuristic

For this heuristic, the initial value of each processor's *spreading* flag is F, and *heuristic* = ((F, F, F), (T, T, T, T)).

As one might conclude, thieves use the same strategy for choosing their victims and for choosing the processors of whom they will update the *spreading* flag. By observing the algorithm, it is clear that the expected number of steal attempts corresponds to the expected number of updates made to the processors' *spreading* flags[5]. Consequently, the expected number of steal attempts targeting each processor's deque is also the same as the expected number of times that the processor's *spreading* flag is updated to T. Moreover, note that, after having an opportunity to make a spread attempt, and regardless of its outcome, a processor sets its *spreading* flag back to F, implying it will not make any spread attempt until some thief updates the flag to T again. Thus, the number of spread attempts made by a processor is at most the number of times that its *spreading* flag was updated by a thief[6]. With this, it is easy to see that the expected number of spread attempts made by each processor is at most the expected number of times that it was targeted by a steal attempt. Finally, if we assumed that steal attempts, spread attempts and updates to the *spreading* flags only incurred in constant communication overheads, we could conclude that the communication overheads associated with CFSAWSS using this heuristic are within a constant factor from Work Stealing's, which are known to be nearly optimal [BL99]. In the next chapter we will see that, in practice, this assumption turns out to be a realistic one.

[5]Note that, due to the nature of the updates, which basically correspond to blind-writes, it is reasonable to assume that all updates succeed, ones before the others.

[6]Remind that the thief who updated the processor's *spreading* flag may not have stolen that processor.

5.3.3 A performance comparison of the load balancers

In this section, we compare the performance of CFSAWSS's load balancer against SLFWS's. To guarantee a fair comparison, we remove the flexibility of CFSAWSS, setting the input parameters θ_r and θ_s to 0. Consequently, processors always make steal attempts and updates during idle iterations and never skip a spread attempt when their *spreading* flag is set to T.

In order to compare the load balancers' efficiency, we study the number of nodes transferred from each processor during a round. Regarding the SLFWS algorithm, this value then corresponds to the number of nodes successfully stolen from the processor's deque during the round. For the CFSAWSS algorithm, however, we also have to account with the number of nodes donated by the processor (if any) during the round.

For both algorithms, consider an arbitrary (but fixed) round within the course of a computation's execution, and let p denote one of the P processors working on that computation's execution.

- If p is idle at the beginning of that round, its deque is empty and, by the definition of idle iteration, p does not have an assigned node. Consequently, during the round, no node can be stolen from its deque and it will not enable any node. Thus, the number of nodes transferred from p during the round, for both the SLFWS and CFSAWSS schedulers, is 0.

- On the other hand, we have the situation where p is busy. For both algorithms, p executes its assigned node during the first stage of the corresponding scheduling iteration, implying it may enable up to two other nodes.

  0 nodes enabled In this scenario, it is easy to see that the behavior and expected performance of the algorithms is equivalent if they are presented with an equivalent scenario (*i.e.* where the numbers of processors, idle processors and processors whose deques are empty, are the same). Again, noting that the stealing mechanism of the algorithms is equivalent and that, for this case, the number of nodes transferred from p corresponds to the number of nodes stolen from its own deque, it is trivial to conclude that the expected number of nodes transferred from p is the same for both schedulers.

  1 node enabled Following the same reasoning that we made for the previous case, it is trivial to conclude that the behavior and expected performance of the algorithms is equivalent if they are presented with an equivalent scenario.

  2 nodes enabled For last, we have the case where p enables two nodes during the round. Contrarily to the previous situations, the number of nodes transferred from p for each scheduler may now differ. To facilitate the analysis, we separately compare the performance of the schedulers depending on whether p's deque is empty or not.

Empty In this case, the number of nodes transferred from p under the SLFWS algorithm is null. However, for the CFSAWSS scheduler, depending on the value of *heuristic*, p may donate one of the enabled nodes. Consequently, for this scenario, CFSAWSS's performance is at least as good as SLFWS's (*i.e.* the number of nodes transferred from p under CFSAWSS is greater or equal to the corresponding value for SLFWS).

Non-empty We delve into the scenario where p's deque is not empty with more detail.

We now compare the performance of CFSAWSS against SLFWS for this last situation, where p enables two nodes and its deque is not empty. Again, we assume that both algorithms are presented with an equivalent scenario, where the numbers of processors, idle processors and processors whose deques are empty, are the same. To that end, we compute the expected performance gains of our scheduler when compared against SLFWS, during a round. We then have

$$\text{Gains}\left(\text{heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D\right) = \frac{E\left[\text{number of nodes transferred from } p \text{ under CFSAWSS}\left(\text{heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D\right)\right]}{E\left[\text{number of nodes transferred from } p \text{ under SLFWS}\left(\alpha, \beta, P\right)\right]}.$$

where

- heuristic defines the initial value of the processor's *spreading* flags and also defines the input parameter *heuristic*, that is passed to the CFSAWSS scheduler.
- θ_r is the second parameter given to CFSAWSS.
- θ_s is the third parameter given to CFSAWSS.
- α is the ratio of idle processors during that round.
- β is the ratio of processors whose deques are empty during that round.
- P is the number of processors.
- D is the number of processors that enabled two nodes during that round.

As aforementioned, the number of nodes transferred from p under SLFWS during a round corresponds to the number of nodes stolen from its deque during that same round. For CFSAWSS, however, we also have to account with the expected number of nodes donated by p during a round. As such, the performance gains of CFSAWSS when compared against SLFWS then correspond to

$$\text{Gains}\left(\text{heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D\right) = \frac{E\left[\text{number of nodes stolen from } p \text{ or donated by } p \text{ under CFSAWSS}\left(\text{heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D\right)\right]}{E\left[\text{number of nodes stolen from } p \text{ under SLFWS}\left(\alpha, \beta, P\right)\right]}.$$

Taking into account the assumptions we made regarding θ_r and θ_s for this comparison, it is straightforward to deduce that, for both algorithms, every idle processor makes a steal attempt in the course of the first stage's execution of the corresponding idle iteration. Moreover, as the algorithms' stealing strategy is the exact same, it is easy to see that under equivalent scenarios their performance is expected to be the same. Consequently, it follows

$$\begin{aligned}&\text{Gains}\,(\text{heuristic}, 0, 0, \alpha, \beta, P, D) = \\ &\frac{E\,[\text{number of nodes transferred from } p \text{ under CFSAWSS}\,(\text{heuristic}, 0, 0, \alpha, \beta, P, D)]}{E\,[\text{number of nodes transferred from } p \text{ under SLFWS}\,(\alpha, \beta, P)]} \\ &= 1 + \frac{E\,[\text{number of donations made by } p \text{ under CFSAWSS}\,(\text{heuristic}, 0, 0, \alpha, \beta, P, D)]}{E\,[\text{number of nodes stolen from } p \text{ under SLFWS}\,(\alpha, \beta, P)]}\end{aligned}$$

With this, we only have to obtain upper bounds on the expected number of nodes stolen from p's deque and lower bounds on the expected number of donations made by p, during the round.

Lemma 5.8. *$E\,[$number of nodes stolen from p under SLFWS$\,(\alpha, \beta, P)] \le \alpha$.*

Proof. Think of each steal attempt as a ball toss, targeting a processor's deque uniformly at random, and, of each idle processor as a red bin. Since every idle processor makes a steal attempt during the first stage's execution, the number of steal attempts is P_i. Thus, the number of tossed balls also is P_i. Additionally, think of each busy processor as a green bin. With this, it is easy to see that we have an instance of Corollary 3.2.1, where $\theta = \theta_r = 0$, $B = P$, $B_R = P_i$, and, S_i, or, more correctly, S_p corresponds to the expected number of steal attempts targeting p's deque. By that corollary, we have $E\left[S_p\right] = \alpha$, meaning that, on average, α steal attempts target p's deque. Finally, since each steal attempt can only remove at most one node, it is clear that the expected number of nodes stolen from p's deque during the round is at most the expected number of steal attempts targeting its deque, which concludes the proof of this lemma. ■

As a consequence of Lemma 5.8, it follows

$$\begin{aligned}&\text{Gains}\,(\text{heuristic}, 0, 0, \alpha, \beta, P, D) = \\ &1 + \frac{E\,[\text{number of donations made by } p \text{ under CFSAWSS}\,(\text{heuristic}, 0, 0, \alpha, \beta, P, D)]}{E\,[\text{number of nodes stolen from } p \text{ under SLFWS}\,(\alpha, \beta, P)]} \ge \\ &1 + \frac{E\,[\text{number of donations made by } p \text{ under CFSAWSS}\,(\text{heuristic}, 0, 0, \alpha, \beta, P, D)]}{\alpha}.\end{aligned}$$

Now, it only remains to obtain lower bounds on the expected number of donations made by p under CFSAWSS.

Lemma 5.9. *Suppose there is a processor q whose state flag has the value of* IDLE *at the very beginning of the second stage's execution. More, suppose that p chooses q as the target of its spread attempt (i.e. p attempts to be q's donor). Then, the more processors choose q as the target of their spread attempt, the lower will be the probability that p's spread attempt succeeds.*

Proof. By the specification of the stages composing each scheduling iteration of CFSAWSS, if q has its *state* flag set to IDLE during the execution of the second stage, then, q is executing an idle iteration. Moreover, by observing Algorithm 5, it is easy to deduce that q failed its steal attempt, as, otherwise its *state* flag would be set to WORKING during the second stage's execution. Therefore, during that stage's execution, q is available to accept node donations.

Again, from the algorithm's specification, all donation attempts (*i.e.* spread attempts) take place during the execution of the second stage. As aforementioned, we assume that any call to *UniformlyRandomNumber* for the same input interval takes the exact same number of steps. Furthermore, we also assume that any call to *UniformlyRandomProcessor* takes the same number of steps as well. Thus, the sequence of instructions executed by any possible donor from the beginning of the second stage until the step when each processor finally makes the spread attempt (corresponding to lines 6 to 11 and 41 to 46), has the exact same length. Taking into account the synchronous execution environment we have artificially created, where all processors start each stage's execution (of each iteration) at the exact same step, we can conclude that all spread attempts made during each round take place at the exact same step.

By analyzing Algorithm 5, it is easy to see that a spread attempt targeting q succeeds if and only if the execution of the associated CAS instruction succeeds. Thus, during the round, exactly one of the processors can succeed spreading a node to q. Moreover, the one who does succeed is chosen uniformly at random from the possible candidates (*i.e.* from q's possible donors). Let $Attempts_q$ denote the number of spread attempts made by processors whose target is q, during the round. Recall that p attempted to be q's donor. As a consequence of the CAS's semantics, the probability that p's spread attempt succeeds is $\frac{1}{Attempts_q}$, completing the proof of this lemma. ■

We prove the next lemma for any possible θ_r and θ_s, as we intend to use the result later.

Lemma 5.10. *For any* heuristic

$$E\left[\textit{number of donations made by p under CFSAWSS}(\text{heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D)\right] \geq$$
$$E\left[\textit{number of donations made by p under CFSAWSS}(\text{heuristic}, \theta_r, \theta_s, \alpha, \beta, P, P(1-\alpha))\right]$$

Proof. Consider an arbitrary (but fixed) scenario, at the very beginning of the second stage's execution of the round. For $i = 1, \ldots, P$, let p_i denote one of the processors executing the computation. Moreover, let $state_i$ denote the value of p_i's *state* flag; $spreading_i$ the value of p_i's *spreading* flag; $donation_i$ the value of p_i's *donation* cell; $deque_i$ the current state of p_i's deque; $working_i$ to TRUE if p_i executed a node during the first stage or to FALSE if it did not; and $enabled_i$ the set of nodes enabled by p_i during the execution of the previous stage. Let B denote the set of busy processors (*i.e.* the set composed by all the processors such that $working_i$ is set to TRUE); G denote the set of processors that enabled two nodes; and S the set of processors that can possibly make a spread attempt (*i.e.* S is composed by every processor $p_i \in G$ such that $spreading_i$ is set to TRUE).

We consider two possible scenarios.

- If the heuristic never allows processors to make donation attempts, then it is straightforward to conclude that the lemma holds

  $$E\left[\text{number of donations made by } p \text{ under CFSAWSS}(heuristic, \theta_r, \theta_s, \alpha, \beta, P, D)\right] =$$
  $$E\left[\text{number of donations made by } p \text{ under CFSAWSS}(heuristic, \theta_r, \theta_s, \alpha, \beta, P, P(1-\alpha))\right] = 0.$$

- We now prove the opposite scenario. Again, we consider two possible situations.

  $B = G$: In this case, $D = P - P_i = P(1-\alpha)$. Consequently, it follows

  $$E\left[\text{number of donations made by } p \text{ under CFSAWSS}(\text{heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D)\right] =$$
  $$E\left[\text{number of donations made by } p \text{ under CFSAWSS}(\text{heuristic}, \theta_r, \theta_s, \alpha, \beta, P, P(1-\alpha))\right]$$

  meaning, the lemma holds for this situation.

  $B \supset G$: For this case, we build a new scenario based on the original, whose only difference is that each busy processor now enables two nodes. As such, we define heuristic' = heuristic; $\theta_r{}' = \theta_r$; $\theta_s{}' = \theta_s$; $\alpha' = \alpha$; $\beta' = \beta$; $P' = P$, and, as we will see, $D' = P(1-\alpha)$. For $i = 1, \dots, P'$, let q_i denote one of the processors executing the computation. We then set q_i's *state* flag to the value of $state_i$; q_i's *spreading* flag to the value of $spreading_i$; q_i's *donation* cell to the value of $donation_i$; the current state of q_i's deque to $deque_i$; and, if $working_i$ has the value of TRUE, we set q_i's enabled nodes to $\{x_i, y_i\}$ (where both x_i and y_i are nodes we created) and, otherwise, we set q_i's enabled nodes to $enabled_i$. Finally, let B' denote the set of busy processors; G' the set of the processors that enabled two nodes; and S' the set of processors that can possibly make a spread attempt.

  First, it is trivial to deduce that $B' = B$. Moreover, since now every busy processor enables two nodes, then, for this case $G' = B' = B \supset G$, implying $G' \supset G$. For last, since we have maintained every processor's *spreading* flag and $G' \supset G$, it follows $S' \supseteq S$.

  Finally, we have two possible cases.

  $S' = S$ In this case, the expected number of donors is the exact same as before. Thus, we have

  $$E\left[\text{number of donations made by } p \text{ under CFSAWSS}(\text{heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D)\right] =$$
  $$E\left[\text{number of donations made by } p \text{ under CFSAWSS}(\text{heuristic}, \theta_r, \theta_s, \alpha, \beta, P, P(1-\alpha))\right]$$

  $S' \supset S$ To conclude, in this scenario the expected number of donors is greater than before. Since donors target processors uniformly at random, then, the expected number of processors targeting each possible donee is now greater than before. Note that the probability that p targets a possible donee is still the same. By Lemma 5.9, as the expected number of processors targeting each donee increased, we can conclude that the probability that p becomes a donor, given that it targeted a possible donee is smaller than in the previous case. Thus, we conclude

$$\begin{aligned}&E\left[\text{number of donations made by } p \text{ under CFSAWSS}\left(\text{heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D\right)\right] > \\ &E\left[\text{number of donations made by } p \text{ under CFSAWSS}\left(\text{heuristic}, \theta_r, \theta_s, \alpha, \beta, P, P\left(1-\alpha\right)\right)\right]\end{aligned}$$

completing the proof of this lemma.

■

Corollary 5.10.1. *Consider Lemma 5.10. If we assume $\theta_r = \theta_s = 0$, then, for any* heuristic

$$\begin{aligned}&E\left[\textit{number of donations made by p under CFSAWSS}\left(\text{heuristic}, 0, 0, \alpha, \beta, P, D\right)\right] \geq \\ &E\left[\textit{number of donations made by p under CFSAWSS}\left(\text{heuristic}, 0, 0, \alpha, \beta, P, P\left(1-\alpha\right)\right)\right]\end{aligned}$$

As a consequence of Corollary 5.10.1, we then have

$$\begin{aligned}&\text{Gains}\left(\text{heuristic}, 0, 0, \alpha, \beta, P, D\right) \geq \\ &1 + \frac{E\left[\text{number of donations made by } p \text{ under CFSAWSS}\left(\text{heuristic}, 0, 0, \alpha, \beta, P, D\right)\right]}{\alpha} \geq \\ &1 + \frac{E\left[\text{number of donations made by } p \text{ under CFSAWSS}\left(\text{heuristic}, 0, 0, \alpha, \beta, P, P(1-\alpha)\right)\right]}{\alpha}.\end{aligned}$$

Lemma 5.11.

$$\begin{aligned}&E\left[\textit{number of donations made by p under CFSAWSS}\left(\text{Greedy heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D\right)\right] \\ &\geq \frac{\alpha}{1-\alpha}\left(1 - \left(1-\alpha\right)\left(1-\theta_r\right)\right)\left(1 - e^{-(1-\alpha)(1-\theta_s)}\right)\end{aligned}$$

Proof. We will use the result presented in Lemma 3.4 as an analogy for the spreads.

As for the description of the algorithm, we partition the execution of each scheduling iteration into three stages.

STAGE I During this stage, each idle processor has an opportunity to make a steal attempt, and, later in this stage, will also have the opportunity for making an update.

STAGE II It is during this stage that donation offers are made: each busy processor enabling two nodes and whose *spreading* flag is set to T will have an opportunity to donate one of such nodes.

STAGE III Finally, in this stage, the idle processors who received a donation offer, get the donated node assigned.

Suppose p decided to make a spread/donation attempt. As donors target their donees during the execution of the second stage, and, uniformly at random, it is trivial to conclude that, the greater the number of successful steal attempts made during the first stage, the smaller will be the probability that p targets a processor who is accepting donation offers.

Noting that each steal attempt may only take at most a single node from a deque, it is straightforward to conclude that the expected number of nodes stolen during the first stage is bounded by the expected number of steal attempts targeting processors whose deques are not empty. Moreover, since the number of processors whose deques are not empty is at most the number of busy processors, it is trivial to conclude that the expected number of nodes stolen during the first stage is at most the expected number of steal attempts targeting busy processors.

By Lemma 5.10, it follows

$$E\left[\text{number of donations made by } p \text{ under CFSAWSS}(\text{Greedy heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D)\right] \geq$$
$$E\left[\text{number of donations made by } p \text{ under CFSAWSS}(\text{Greedy heuristic}, \theta_r, \theta_s, \alpha, \beta, P, P(1-\alpha))\right]$$

Using the same arguments as in the proof of Lemma 5.9, it is trivial to conclude that the expected number of nodes donated by each processor who makes a spread attempt during the second stage is the exact same. For the sake of completeness, we now resemble those arguments.

From the algorithm's specification, all donation attempts (*i.e.* spread attempts) take place during the execution of the second stage. As aforementioned, we assume that any call to *UniformlyRandomNumber* for the same input interval takes the exact same number of steps. Moreover, we also assume that any call to *UniformlyRandomProcessor* takes the same number of steps as well. Thus, the sequence of instructions executed by any possible donor from the beginning of the second stage until the step when each processor finally makes the spread attempt (corresponding to lines 6 to 11 and 41 to 46), has the exact same length. Taking into account the synchronous execution environment we have artificially created, where all processors start each stage's execution (of each iteration) at the exact same step, we can conclude that all spread attempts made during each round take place at the exact same step.

By analyzing Algorithm 5, it is easy to see that a spread attempt targeting some possible donee q succeeds if and only if the execution of the associated CAS instruction succeeds. Moreover, as a consequence of the synchronous execution environment, every processor invokes the CAS instruction, attempting to donate work to the donee, at the exact same step. From the semantics of the CAS instruction, it follows that, if one or more processors make a spread attempt targeting the same donee, exactly one succeeds, and, the one who does is chosen uniformly at random. With this, it is straightforward to conclude that the expected number of nodes donated by each processor during a round is the exact same.

We finally make the analogy for Lemma 3.4, allowing us to obtain the expected number donations made by all processors during that stage. We see each bin as an analogy for a processor, implying $B = P$ (*i.e.* the number of bins is the same as the number of processors); each red bin as an idle processor; the initial number of red bins B_R as the initial number of idle processors P_i; each green bin as a busy processor; the initial number of green bins B_G as the initial number of non idle processors $P - P_i$; and each cube as a steal opportunity, where

- Discarding a cube corresponds to skipping a steal attempt opportunity.
- Tossing a cube corresponds to making a steal attempt.

Furthermore, we see each red bin painted green as an idle processor that successfully stole a node during the execution of the first stage of the enclosing scheduling iteration. The initial number of cubes we are given is B_R, as each corresponds to an idle processor's opportunity for making a steal attempt.

The analogous moment for finishing all the paintings is when the first stage's execution terminates.

Since we are using the Greedy heuristic, then, every processor's $spreading$ flag is set to T. Consequently, for the second stage, we are given B_G balls, each corresponding to a spread attempt (note that, by the analogy, B_G corresponds to the number of processors executing a busy iteration). As such, we are assuming that all the initially busy processors make a spread attempt, which, as aforementioned, corresponds to the worst case scenario for p. Moreover, we see each bin that is still red as a processor that is available for receiving a donation (corresponding to a processor whose steal attempt, made during the first stage, did not succeed); each of the B_G balls we have as an opportunity for attempting a spread, where

- Discarding a ball corresponds to skipping a spread attempt opportunity.
- Tossing a ball corresponds to making a spread attempt.

and, finally, each bin that is still red with at least one ball as a donee who was targeted by at least one spread attempt.

The lemma then implies that the expected number of still red bins with at least one ball is at least $B\alpha(1-(1-\alpha)(1-\theta_r))\left(1-e^{-(1-\alpha)(1-\theta_s)}\right)$. In our analogy, this corresponds to saying that the expected number of successful spread attempts made during that round is at least $P\alpha(1-(1-\alpha)(1-\theta_r))\left(1-e^{-(1-\alpha)(1-\theta_s)}\right)$.

Finally, as a consequence of our previous assumptions, the expected number of nodes spread by each busy processor is the exact same (note that we have shown that the worst case scenario for p is when all the busy processor enable two nodes, and thus become eligible for making a spread attempt).

It then follows

$$
\begin{aligned}
&E\left[\text{number of donations made by } p \text{ under CFSAWSS}(\text{Greedy heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D)\right] \\
&\geq \frac{P\alpha\left(1-(1-\alpha)(1-\theta_r)\right)\left(1-e^{-(1-\alpha)(1-\theta_s)}\right)}{P-P_i} \\
&= \frac{\alpha}{1-\alpha}\left(1-(1-\alpha)(1-\theta_r)\right)\left(1-e^{-(1-\alpha)(1-\theta_s)}\right)
\end{aligned}
$$

This concludes the proof of the lemma. ■

Corollary 5.11.1.

$$
\begin{aligned}
&E\left[\textit{number of donations made by } p \textit{ under } \text{CFSAWSS}(\text{Greedy heuristic}, 0, 0, \alpha, \beta, P, D)\right] \\
&\geq \frac{\alpha^2}{1-\alpha}\left(1-e^{-(1-\alpha)}\right)
\end{aligned}
$$

Lemma 5.12.

$$
\begin{aligned}
&E\left[\textit{number of donations made by } p \textit{ under } \text{CFSAWSS}(\text{Tit-for-tat heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D)\right] \\
&\geq \frac{\alpha}{1-\alpha}\left(1-(1-\alpha)(1-\theta_r)\right)\left(1-e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}\right)
\end{aligned}
$$

Proof. We will use the result presented in Lemma 3.5 as an analogy for the spreads.

As for the description of the algorithm, we partition the execution of each scheduling iteration into three stages.

STAGE I During this stage, each idle processor has an opportunity to make a steal attempt, and, later in this stage, will also have the opportunity for making an update.

STAGE II It is during this stage that donation offers are made: each busy processor enabling two nodes and whose *spreading* flag is set to T will have an opportunity to donate one of such nodes.

STAGE III Finally, in this stage, the idle processors who received a donation offer, get the donated node assigned.

Suppose p decided to make a spread/donation attempt. As donors target their donees during the execution of the second stage, and, uniformly at random, it is trivial to conclude that, the greater the number of successful steal attempts made during the first stage, the smaller will be the probability that p targets a processor who is accepting donation offers.

Noting that each steal attempt may only take at most a single node from a deque, it is straightforward to conclude that the expected number of nodes stolen during the first stage is bounded by the expected number of steal attempts targeting processors whose deques are not empty. Furthermore, since the number of processors whose deques are not empty is at most the number of busy processors, it is trivial to conclude that the expected number of nodes stolen during the first stage is at most the expected number of steal attempts targeting busy processors.

Contrarily to the of the Greedy heuristic, for this one, every processor begins the computation's execution with its *spreading* flag set to F. For that reason, we assume the worst possible case for this round, which is that every processor starts the execution of the first stage with its *spreading* flag set to F. By the algorithm's definition, after a thief makes a steal attempt (or skips it, if it is the case), it will then decide, uniformly at random whether to update another processor's *spreading* flag or not. Note that the probability that a thief skips an opportunity is θ_r. More, also note that, in case a thief does not skip an update opportunity, it picks the processor whose *spreading* flag it will update, uniformly at random. Finally, since these updates still take place during the execution of the first stage, then, during the second stage, the number of processors whose *spreading* flag is set to T corresponds to the number of processors that were targeted by at least one update (note that, by the heuristic's definition, any update made by a thief implies that the processor's *spreading* flag is set to T).

By Lemma 5.10, it follows

$$E\left[\text{number of donations made by } p \text{ under CFSAWSS}\left(\text{Tit-for-tat heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D\right)\right] \geq$$
$$E\left[\text{number of donations made by } p \text{ under CFSAWSS}\left(\text{Tit-for-tat heuristic}, \theta_r, \theta_s, \alpha, \beta, P, P\left(1-\alpha\right)\right)\right]$$

Using the same arguments as in the proof of Lemma 5.9, it is trivial to conclude that the expected number of nodes donated by each processor who makes a spread attempt during the second stage is the exact same. We omit these arguments as they are the exact same as the ones given in the proof of Lemma 5.11.

We finally make the analogy for Lemma 3.5, allowing us to obtain the expected number donations made by all processors during that stage. We see each bin as an analogy for a processor, implying $B = P$ (*i.e.* the number of bins is the same as the number of processors); each red bin as an idle processor; the initial number of red bins B_R as the initial number of idle processors P_i; each green bin as a busy processor; the initial number of green bins B_G as the initial number of non idle processors $P - P_i$; each cube as a steal opportunity, where

- Discarding a cube corresponds to skipping a steal attempt opportunity.

- Tossing a cube corresponds to making a steal attempt.

and each pyramid as an update opportunity, where

- Discarding a pyramid corresponds to skipping an update opportunity.

- Tossing a pyramid corresponds to making an update to some processor (setting its *spreading* flag to T).

Furthermore, we see each red bin painted green as an idle processor that successfully stole a node during the execution of the first stage of the enclosing scheduling iteration. Collecting a ball for each green bin hit by a pyramid is an analogy for when some thief updates a busy processor's *spreading* flag, implying the targeted processor will have the opportunity for making a spread attempt during the second stage.

The number of cubes we are given is B_R, as each corresponds to an idle processor's opportunity for making a steal attempt. In the same way, the number of pyramids we are given is also B_R, as each corresponds to an idle processor's opportunity for updating some processor's *spreading* flag.

The analogous moment for finishing all the paintings is when the first stage's execution terminates.

Note that, by always collecting a ball for each (originally) green bin hit by a pyramid, we are basically assuming that every busy processor (that, in the analogy, corresponds to a green bin) enables two nodes, and thus becomes eligible for making a spread attempt during this second stage. As already mentioned, this corresponds to the worst possible scenario for p (*i.e.* for the expected number of nodes spread by p). We see each bin that is still red as a processor that is available for receiving a donation (corresponding to a processor whose steal attempt, made during the first stage, did not succeed); each of the balls we have collected as an opportunity for attempting a spread, where

- Discarding a ball corresponds to skipping a spread attempt opportunity.
- Tossing a ball corresponds to making a spread attempt.

and, finally, each bin that is still red with at least one ball as a donee who was targeted by at least one spread attempt.

Lemma 3.5 then implies that the expected number of still red bins with at least one ball is at least $B\alpha(1-(1-\alpha)(1-\theta_r))\left(1-e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}\right)$. In our analogy, this corresponds to saying that the expected number of successful spread attempts made during that round is at least $P\alpha(1-(1-\alpha)(1-\theta_r))\left(1-e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}\right)$.

Finally, as a consequence of our previous assumptions, the expected number of nodes spread by each busy processor is the exact same: note that we have shown that the worst case scenario for p is when all the busy processor enable two nodes, and thus become eligible for making a spread attempt.

It then follows

$$\begin{aligned}
&E\left[\text{number of donations made by } p \text{ under CFSAWSS}(\text{Tit-for-tat heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D)\right] \\
&\geq \frac{P\alpha(1-(1-\alpha)(1-\theta_r))\left(1-e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}\right)}{P-P_i} \\
&= \frac{\alpha}{1-\alpha}(1-(1-\alpha)(1-\theta_r))\left(1-e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}\right)
\end{aligned}$$

This concludes the proof of the lemma. ■

Corollary 5.12.1.

$$E\,[\textit{number of donations made by p under CFSAWSS}\,(\text{Tit-for-tat heuristic}, 0, 0, \alpha, \beta, P, D)]$$
$$\geq \frac{\alpha^2}{1-\alpha}\left(1 - e^{-(1-\alpha)(1-e^{-\alpha})}\right)$$

Now, we can compute the gains of our scheduler when compared against SLFWS, for each of the two theoretical heuristics.

As a consequence of Corollary 5.11.1, we then have

$$\text{Gains}\,(\text{Greedy heuristic}, 0, 0, \alpha, \beta, P, D) \geq$$
$$1 + \frac{E\,[\text{number of donations made by } p \text{ under CFSAWSS}\,(\text{Greedy heuristic}, 0, 0, \alpha, \beta, P, D)]}{\alpha} \geq$$
$$1 + \frac{\alpha}{1-\alpha}\left(1 - e^{-(1-\alpha)}\right).$$

As one might note, the lower bounds on the gains for the Greedy heuristic now only depend on the ratio of idle processors α. Let $\psi(\alpha)$ denote the lower bounds on the gains for the Greedy heuristic, as a function of α. Then,

$$\psi(\alpha) = 1 + \frac{\alpha}{1-\alpha}\left(1 - e^{-(1-\alpha)}\right)$$

Due to the result given in Corollary 5.12.1, we then have

$$\text{Gains}\,(\text{Tit-for-tat heuristic}, 0, 0, \alpha, \beta, P, D) \geq$$
$$1 + \frac{E\,[\text{number of donations made by } p \text{ under CFSAWSS}\,(\text{Tit-for-tat heuristic}, 0, 0, \alpha, \beta, P, D)]}{\alpha} \geq$$
$$1 + \frac{\alpha}{1-\alpha}\left(1 - e^{-(1-\alpha)(1-e^{-\alpha})}\right).$$

As for the Greedy heuristic, the lower bounds on the gains for the Tit-for-tat heuristic now only depend on the ratio of idle processors α. Let $\delta(\alpha)$ denote the lower bounds on the gains for the Tit-for-tat heuristic, as a function of α. Then,

$$\delta(\alpha) = 1 + \frac{\alpha}{1-\alpha}\left(1 - e^{-(1-\alpha)(1-e^{-\alpha})}\right)$$

Figure 5.1 illustrates the *lower bounds* on the theoretical advantage of CFSAWSS (for $\theta_r = \theta_s = 0$) over SLFWS, depending on the ratio of idle processors α. Clearly, the algorithm shows great advantage potential, specially for scenarios where the ratio of idle processors is higher. This theoretical result may justify the folk wisdom that Work Sharing based schemes have the upper hand in what regards to scenarios with few, or, unbalanced, parallelism. In particular, for the Greedy heuristic ($\psi(\alpha)$), the lower bounds on the gains are at least 39.34% for any ratio of idle processors greater than 0.5, and can even achieve a factor of 100%, as $\lim_{\alpha\to 1^-}\left(1 + \frac{\alpha}{1-\alpha}.\left(1 - e^{\alpha-1}\right)\right) = 2$. On the other hand, using the Tit for Tat heuristic ($\delta(\alpha)$), the lower bounds on the gains are at least 17.85% for any ratio of

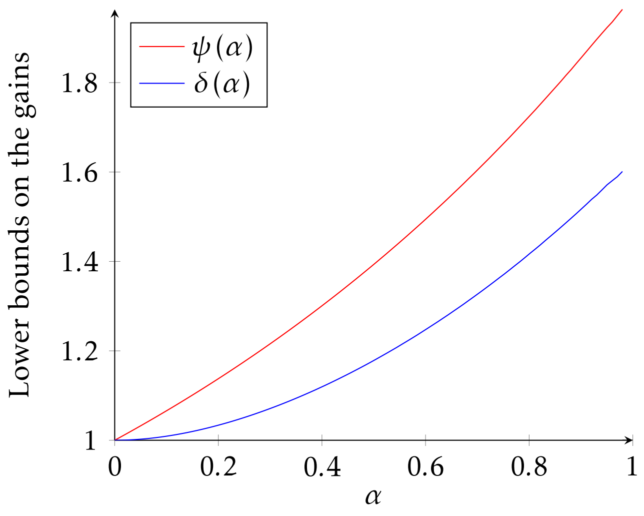

Figure 5.1: Theoretical gains of the CFSAWSS algorithm over SLFWS for a round.

idle processors greater than 0.5, and can easily achieve gains over 40%, if the ratio of idle processors is greater than 0.8.

Since the evaluation is made on a round basis, the length of each round matters: in the utmost extent, if each round of CFSAWSS took twice as much when compared to a round length of the SLFWS algorithm, it would only be fair to compare two round iterations of SLFWS with one round of the algorithm we propose. However, we remark the fact that CFSAWSS was only designed in order to prove that mixed load balancing schemes (*i.e.* load balancing schemes using both receiver and sender initiated policies) are much more effective than purely receiver initiated ones, as it is the case of WS. Furthermore, the comparison of the length of a CFSAWSS's round against the length of a SLFWS's one is pretty much meaningless, as both these algorithms are theoretical. In the next chapter, we will present an asynchronous and practical algorithm that uses a mixed load balancing strategy, similar to CFSAWSS's, and will further prove that the expected runtime of computations for that algorithm is at most $O\left(\frac{T_1}{P} + \frac{T_\infty}{1-\theta_r}\right)$.

5.3.4 A study of CFSAWSS's stability

Finally, we claim that CFSAWSS tends to become stable, depending on the ratio of idle processors α, for a given heuristic, and, for a given *"degree"* of flexibility (*i.e.* for given values of θ_r and θ_s).

Recall that a scheduler is said to be stable if and only if, during a computation's execution, the number of nodes attached to each processor is expected to be bounded by a constant. As one might note, CFSAWSS can never guarantee stability when executing an arbitrary computation: if, at some round, more than half of the processors working on the computation's execution, enable two nodes, then, the number of enabled nodes exceeds the number of processors, implying that the amount of work per processor strictly increases. In fact, for scenarios where $\alpha < \frac{1}{2}$, it is impossible to guarantee stability for the CFSAWSS scheduler. Nevertheless, depending on the fraction of idle processors at the beginning of some round (*i.e.* α), we show that, for both the Greedy and the Tit-for-tat heuristics, and, for given values of θ_r and θ_s, the number of nodes attached to each of the P processors at the end of the round is either expected to decrease, or, if it is the case, to continue being bounded by a constant.

As in Section 5.3.3, consider some round during the course of a computation's execution under CFSAWSS, using some heuristic, and, for certain values of θ_r and θ_s. More, let p be one of the P processors working on that computation's execution; let α be the ratio of idle processors; β the ratio of processors whose deques are empty; and D, the number of processors enabling two nodes during that round. Through the rest of this analysis, we assume that $\alpha \geq \frac{1}{2}$, as otherwise it would be impossible to ensure stability.

We begin by remarking that only the owner of each deque may add nodes to its deque. Now, consider each of the possible scenarios for p.

- If p is executing an idle iteration during that round, then, the processor does not execute any node. Furthermore, in such case its deque is empty, and, the number of nodes enabled by p during that round is 0. Even if p makes a successful steal attempt or is offered a node, it is straightforward to conclude that the number of nodes attached to p at the end of the round is constant.

- On the other hand, we have the situation where p is busy. In such case, p's deque may be either empty or not.

  Deque is empty Regardless of how many nodes p may enable due to the execution of its currently assigned one, it is trivial to deduce that the number of nodes attached to p at the end of the round is constant.

  Deque is not empty Because p is busy, it then executes its assigned node, implying it may enable up to two other nodes.

  0 nodes enabled — In this case, since p executed its assigned node, but no new node was enabled, the number of nodes attached to p strictly decreased.

  1 node enabled — For this scenario, the number of nodes attached to p at the end of the round is at most the same as it was at the beginning of the round. As, for the aforementioned reasons, we made the assumption that $\alpha \geq \frac{1}{2}$, then, the expected number of nodes stolen from p's deque is not null, implying that the number of nodes attached to p is expected to strictly decrease.

  2 nodes enabled — For last, we have the case where p enables two nodes during the round. We delve into this situation with more detail.

We now study the performance of CFSAWSS for this last scenario, where p enables two nodes and its deque is not empty. Informally, our goal is to prove that the number of nodes enabled by p, which in this case is 2, is strictly less than the number of nodes executed by p together with the number of nodes that were transfered from p to some other processor. More formally, we want to prove that the following inequality holds

$$2 < 1 + E\left[\text{number of nodes transferred from } p \text{ under CFSAWSS}(\text{heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D)\right]$$
$$\equiv 1 < E\left[\text{number of nodes transferred from } p \text{ under CFSAWSS}(\text{heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D)\right].$$

We now move to obtain lower bounds on the expected number of nodes transferred from p during a round of CFSAWSS. To that end, we compute lower bounds on both the expected number of nodes stolen from p's deque and the expected number of nodes donated by p, both during that round.

Lemma 5.13.

$$E\left[\textit{number of nodes stolen from } p \textit{ under CFSAWSS}(\text{heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D)\right]$$
$$\geq 1 - e^{-\alpha(1-\theta_r)}$$

Proof. To prove this result, we use Corollary 3.3.1 as an analogy for the steal attempts made by idle processors.

Note that, by the description of the algorithm, all steal attempts take place during the execution of the first stage. By observing the CFSAWSS, depicted in Algorithm 5, it is trivial to conclude that every idle processor has an opportunity for making a steal attempt during the execution of the first stage.

Now, we make the analogy for Corollary 3.3.1, allowing us to obtain lower bounds on the expected number of nodes stolen from p, whose deque is not empty. We see each bin as an analogy for a processor, implying $B = P$ (*i.e.* the number of bins is the same as the number of processors); each red bin as an idle processor; the number of red bins B_R as the number of idle processors P_i; each green bin as a busy processor; the number of green bins B_G as the number of non idle processors $P - P_i$; and each ball as a steal opportunity, where

- Discarding a ball corresponds to skipping a steal attempt opportunity (which occurs with probability θ_r).
- Tossing a ball corresponds to making a steal attempt.

Finally, for a fixed bin b_i, S_i denotes a random variable that corresponds to the number of balls landing in b_i, and Y_i denotes an indicator variable, defined as

$$Y_i = \begin{cases} 1 & \text{if } S_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

It is easy to see that Y_i corresponds to the expectation that one or more balls land in b_i.

By Corollary 3.3.1, having $\theta = \theta_r$, it then follows

$$E\left[Y_i\right] \geq 1 - e^{-\alpha(1-\theta_r)}$$

Since p's deque is not empty, then, by the relaxed semantics we know that if one or more steal attempts target p's deque, at least one succeeds. In such case, at least one node is stolen from p.

Thinking of each ball as a steal attempt, it follows that the expectation that one or more steal attempts target p as their victim is at least

$$1 - e^{-\alpha(1-\theta_r)},$$

completing the proof of this lemma. ■

For the Greedy heuristic, Lemma 5.11 states that

$$\begin{aligned}&E\left[\text{number of donations made by } p \text{ under CFSAWSS}(\text{Greedy heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D)\right]\\ &\geq \frac{\alpha}{1-\alpha}\left(1-(1-\alpha)(1-\theta_r)\right)\left(1-e^{-(1-\alpha)(1-\theta_s)}\right)\end{aligned}$$

From Lemma 5.13, it follows

$$\begin{aligned}&E\left[\text{number of nodes transferred from } p \text{ under CFSAWSS}(\text{Greedy heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D)\right]\\ &\geq 1 - e^{-\alpha(1-\theta_r)} + \frac{\alpha}{1-\alpha}\left(1-(1-\alpha)(1-\theta_r)\right)\left(1-e^{-(1-\alpha)(1-\theta_s)}\right)\end{aligned}$$

As one might note, this expectation only depends on the values of α, θ_r and θ_s. To simplify, let

$$\varphi(\alpha, \theta_r, \theta_s) = 1 - e^{-\alpha(1-\theta_r)} + \frac{\alpha}{1-\alpha}\left(1-(1-\alpha)(1-\theta_r)\right)\left(1-e^{-(1-\alpha)(1-\theta_s)}\right)$$

denote the lower bounds we obtained for

$$E\left[\text{number of nodes transferred from } p \text{ under CFSAWSS}(\text{Greedy heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D)\right].$$

Thus, it is trivial to conclude that, if

$$1 < 1 - e^{-\alpha(1-\theta_r)} + \frac{\alpha}{1-\alpha}\left(1-(1-\alpha)(1-\theta_r)\right)\left(1-e^{-(1-\alpha)(1-\theta_s)}\right)$$

then

$$\begin{aligned}&1 <\\ &E\left[\text{number of nodes transferred from } p \text{ under CFSAWSS}(\text{Greedy heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D)\right].\end{aligned}$$

Claim 5.1. *Consider some computation being executed by an arbitrary (but fixed) number of processors, under CFSAWSS, using the Greedy heuristic, and, having both θ_r and θ_s set to 0. Then, if $\alpha \in [0.7375; 1[$, the expected number of nodes attached to p at the end of the round is less than the number of nodes attached to p at the beginning of that same round.*

We do not prove this claim due to the complexity of proving that

$$\forall \alpha \in [0.7375; 1[, \quad \varphi(\alpha, 0, 0) > 1.$$

In any case, we remark that the graphic shown in Figure 5.2 gives a very insightful idea of the veracity of this claim. Furthermore, we have used Euler's method in order to obtain a good approximation for $\varphi(\alpha, 0, 0) - 1 = 0$, which got us to $\alpha \approx 0.7375$. However, this result is still not enough to guarantee us that it is the only possible solution for the equation, for which reason we do not consider this claim as proved.

Regarding the Tit-for-tat heuristic, Lemma 5.12 proves that

$$\begin{aligned}&E[\text{number of donations made by } p \text{ under CFSAWSS}(\text{Tit-for-tat heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D)]\\ &\geq \frac{\alpha}{1-\alpha}(1-(1-\alpha)(1-\theta_r))\left(1-e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}\right)\end{aligned}$$

From Lemma 5.13, it follows

$$\begin{aligned}&E[\text{number of nodes transferred from } p \text{ under CFSAWSS}(\text{Tit-for-tat heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D)]\\ &\geq 1-e^{-\alpha(1-\theta_r)} + \frac{\alpha}{1-\alpha}(1-(1-\alpha)(1-\theta_r))\left(1-e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}\right)\end{aligned}$$

Again, to simplify, let

$$\sigma(\alpha, \theta_r, \theta_s) = 1-e^{-\alpha(1-\theta_r)} + \frac{\alpha}{1-\alpha}(1-(1-\alpha)(1-\theta_r))\left(1-e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}\right)$$

denote the lower bounds we obtained for

$$E[\text{number of nodes transferred from } p \text{ under CFSAWSS}(\text{Tit-for-tat heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D)].$$

Finally, it is straightforward to deduce that, if

$$1 < 1-e^{-\alpha(1-\theta_r)} + \frac{\alpha}{1-\alpha}(1-(1-\alpha)(1-\theta_r))\left(1-e^{-(1-\alpha)(1-\theta_s)\left(1-e^{-\alpha(1-\theta_r)}\right)}\right)$$

then

$$\begin{aligned}&1 <\\ &E[\text{number of nodes transferred from } p \text{ under CFSAWSS}(\text{Tit-for-tat heuristic}, \theta_r, \theta_s, \alpha, \beta, P, D)].\end{aligned}$$

Claim 5.2. *Consider some computation being executed by an arbitrary (but fixed) number of processors, under CFSAWSS, using the Tit-for-tat heuristic, and, having both θ_r and θ_s set to 0. Then, if $\alpha \in [0.8675; 1[$, the expected number of nodes attached to p at the end of the round is less than the number of nodes attached to p at the beginning of that same round.*

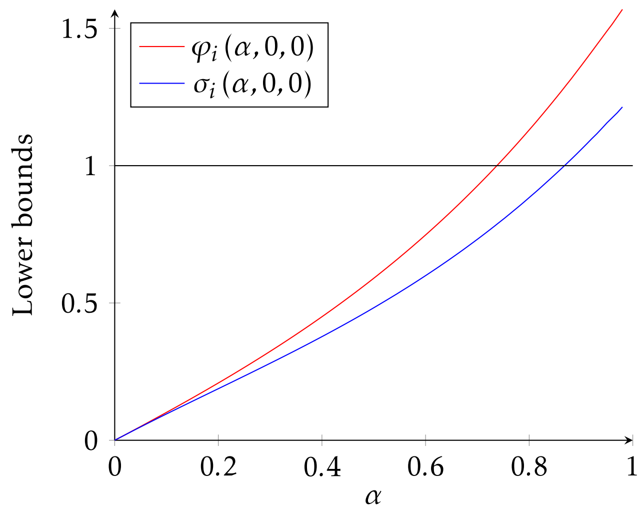

Figure 5.2: Lower bounds on the number of nodes transferred from p for each heuristic, and for $\theta_r = \theta_s = 0$. The black horizontal line shown in the graphic corresponds to the minimum number of nodes that should be transferred from p in order to guarantee that the number of nodes attached to p at the end of the round is smaller than in the beginning of that same round.

We do not prove this claim for the same reason that we have not proved Claim 5.1. It easy to see that proving that

$$\forall \alpha \in [0.8675; 1[, \quad \sigma(\alpha, 0, 0) > 1$$

is extremely hard. Again, we remark that the graphic shown in Figure 5.2 gives a strong insight of the veracity of the claim. Furthermore, we have used Euler's method in order to obtain an approximation for α such that $\sigma(\alpha, 0, 0) - 1 = 0$. The approximation we got to is $\alpha \approx 0.8674$. Nevertheless, this does not guarantee us that the claim is true, for which reason we consider it unproved.

5.3.4.1 Theoretical evaluation

We now evaluate the theoretical results we have achieved with CFSAWSS. Even though we have not been able to prove Claims 5.1 nor 5.2, we still remark that Figure 5.2 gives an insight of why CFSAWSS is effective scheduling computations with few or unbalanced parallelism. For instance, if at least $\frac{3}{4}$ of the processors are idle, Figure 5.2 shows that the expected number of nodes attached to each processor (whose deque is not empty) is expected to decrease, when using the Greedy heuristic, and, for $\theta_r = \theta_s = 0$. In the same way, when using the Tit-for-tat heuristic, and, for $\theta_r = \theta_s = 0$, if at least $\frac{7}{8}$ of the processors are idle, Figure 5.2 shows that the expected number of nodes attached to each processor (whose deque is not empty) is also expected to decrease. By the reasoning made in Section 5.3.4, it is easy to see that CFSAWSS is expected to be very efficient when executing computations with low or unbalanced parallelism. Furthermore, if either of the claims (*i.e.* Claims 5.1 or 5.2) is ever proved, then, CFSAWSS will have been the

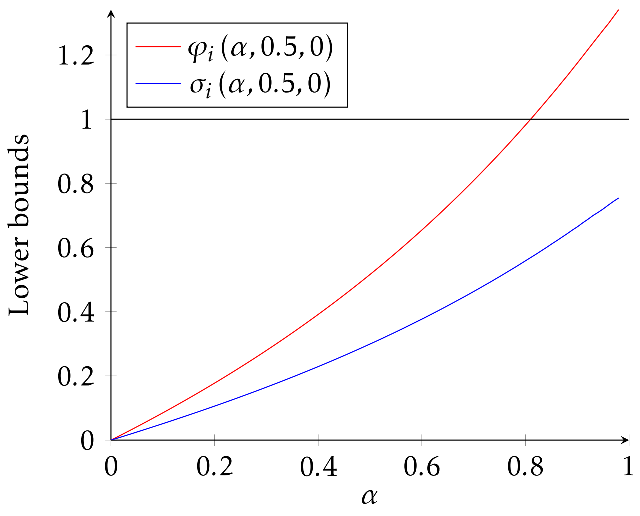

Figure 5.3: Lower bounds on the number of nodes transferred from p for each heuristic, and for $\theta_r = \frac{1}{2}$ and $\theta_s = 0$.

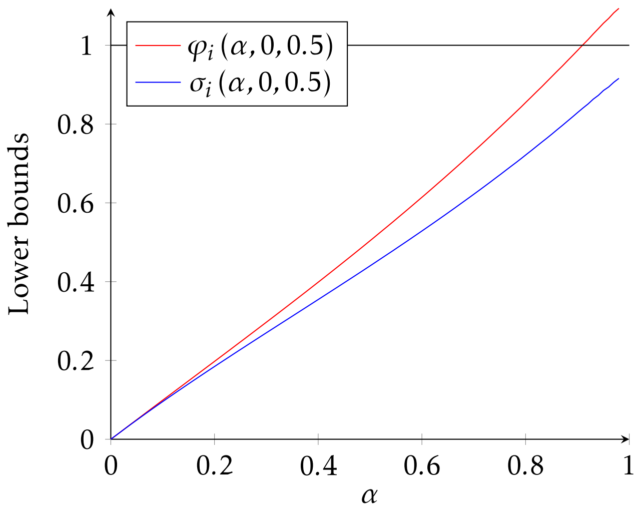

Figure 5.4: Lower bounds on the number of nodes transferred from p for each heuristic, and for $\theta_r = 0$ and $\theta_s = \frac{1}{2}$.

first scheduler with nearly optimal bounds, $O\left(\frac{T_1}{P} + T_\infty\right)$, and that can achieve stability for computations with few (or unbalanced) parallelism (note that for both those claims, θ_r and θ_s are set to 0).

Finally, we have also studied the performance of CFSAWSS varying the values of θ_s and θ_r. In Figure 5.3 it is easy to see that only when using the Greedy heuristic, it may be possible for each processor to tend towards stability. As a consequence of decreasing the number of steal attempts made, the performance of the scheduler when using the Tit-for-tat heuristic was severely harmed. This dramatic performance drop is a consequence of the Tit-for-tat heuristic's nature, as the probability that some processor is given a spread attempt opportunity is the exact same as the probability that the processor is targeted by a steal attempt. By reducing the expected number of steal attempts (*i.e.* by increasing θ_r),

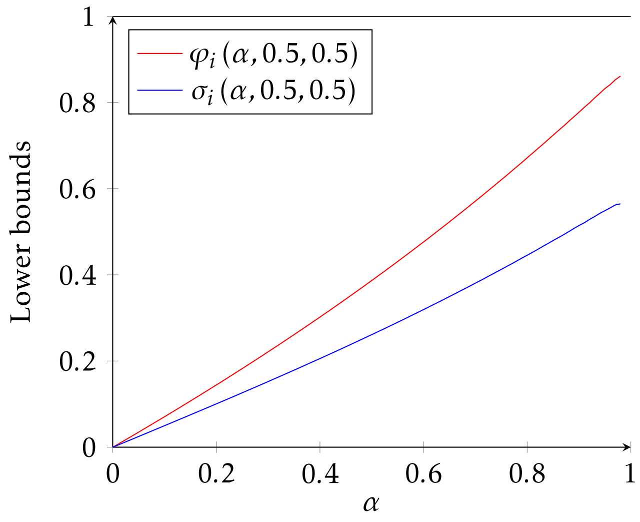

Figure 5.5: Lower bounds on the number of nodes transferred from p for each heuristic, and for $\theta_r = \theta_s = \frac{1}{2}$.

we are then also reducing the probability that processors are given the opportunity for attempting a spread. Moreover, note that Figure 5.4 corroborates our reasoning: in this case, the probability that processors are given an opportunity for attempting a spread is the same as in the first figure. Additionally, note that, for this scenario, the performance of the algorithm when using the Greedy heuristic is close to its performance when using the Tit-for-tat heuristic. This validates the fact that it is due to the spreading mechanism that we successfully *"ensured"* that the number of nodes attached to each processor is expected to decrease for the aforementioned values of α.

To conclude, as awaited, we cannot use high values for both θ_r and θ_s and still expect being able to *"guarantee"* stability. This last situation is depicted in Figure 5.5.

5.4 Future Work

As expected, the first problem we have left open is to prove Claims 5.1 and 5.2.

In addition, we are also looking forward to study the performance of CFSAWSS when using other heuristics. More concretely, we are very interested in studying the performance of the algorithm for the two following heuristics:

Continuous Tit-for-tat heuristic For this heuristic, the initial value of each processor's *spreading* flag is F, and *heuristic* = ((U, U, F), (T, T, T, T)). The rationale behind this heuristic is similar to the original Tit-for-tat heuristic, but allowing processors to continue attempting spreads, as long as these do not fail. The benefits of this heuristic reside in its continues use, an analysis we did not address in this thesis.

Continuous Abort-for-tat heuristic . For this case, each processor's *spreading* flag is initially set to F, and *heuristic* = ((U, U, F), (U, U, U, T)). The key idea behind this heuristic is that steal attempts are more likely to abort if the fraction of idle processors is large. As for the previous heuristic, we also allow processors to continue making spread attempts while these do not fail.

Finally, we explain why Problem 3.1 presented in Chapter 3 is of great importance. In CFSAWSS, an idle processor first decides to whether make, or not, a steal attempt, and, only after that (and after possibly making the steal attempt) decides again if it will update some processor's $spreading$ flag. The reason why the algorithm does this is to ensure that steals and updates to processors' $spreading$ flags are independent events. However, if, in the future, Problem 3.1 is solved, then, depending on the lower bounds that are obtained, we may then modify the algorithm, avoiding these complicated schemes that are only used to guarantee the independence of the aforementioned events.

Chapter 6

Addressing real world problems

In this chapter we show how our theoretical contributions can be used to address real world problems. First, we present an asynchronous algorithm that makes use of the spreading mechanism presented in the previous chapter, and prove it is efficient for scheduling computations. Next, we present various examples of custom load balancing mechanisms that can be implemented on our proposal. For last, we make some concluding remarks and discuss possible directions for future work.

6.1 The Flexible Work Stealing and Sharing algorithm

As shown in the previous chapter, allowing processors to donate work is crucial for achieving high performance when executing computations with unbalanced parallelism. For that reason, we now present the Work Stealing and Sharing (WSS) algorithm which implements an asynchronous version of that same spreading mechanism.

Under the synchronous execution environment, the spreading mechanism could be easily implemented thanks to the automatic coordination of processors at the end of each stage. However, under asynchronous ones, there is no coordination among processors, and, for that reason, we have to make additional requirements in order to ensure that the algorithm is efficient in both theory and practice. First, we have to guarantee that no donation ever causes the donor, the donee nor any possible thief to stall, awaiting its completion. In addition, alike for CFLFWS, the nodes stored in each deque must be kept ordered, from the deque's bottom to its top, according to their weight. Finally, the whole spreading mechanism must be extremely efficient, ensuring that any spread attempt incurs in at most constant overhead.

In WSS, each processor owns two flags, a private stack and a lock free deque. The first flag, named the *state* flag, either stores the current status of the processor (WORKING or IDLE) or a pointer to a donation offer (more on this ahead). The second one, labeled *spreading* flag, stores a boolean value (*i.e.* TRUE or FALSE) that indicates whether the processor should or should not make spread attempts. While in previous approaches processors worked directly on their deques, now, they, instead, work on their stacks, pushing and popping nodes from its bottom. Each node in a processor's deque is then replaced by a pointer to an entry of its local stack. In turn, each entry of a processor's stack contains a computation node and two additional status flags. The first, to which we call the *shared* flag, stores a boolean value that indicates if the entry is being shared with other processors (more on this later). The second one, named the *owned* flag, stores a boolean value that indicates whether the node of that entry is owned by a processor or otherwise.

6.1.1 The private stack

In this section, we present the specification of the stack's implementation, along with its associated *ideal-stack* semantics.

A stack object meeting the ideal semantics supports three methods:

1. *push* — Adds an entry to the bottom of the stack and returns a pointer to the place where the entry was stored.

2. *pop* — Removes the bottommost entry from the stack, if any, and does not return a value.

As for the deque, the stack implementation is said to be constant-time if and only if any invocation to any of these methods takes at most a constant number of steps to return.

An invocation to a stack method is defined by a 4-tuple establishing:

1. The method that was invoked.

2. The invocation's beginning time.

3. The invocation's completion time.

4. The return value, if any.

We say that a set of invocations to a stack's methods meets the ideal-stack semantics if and only if there is a set of linearization times for the corresponding invocations such that:

1. The linearization time of every invocation lies within the beginning and completion times of the respective invocation.

2. No linearization times associated with distinct invocations coincide.

3. The return values for each invocation are consistent with a serial execution of the methods in the order given by the linearization times of the corresponding invocations.

A set of invocations is said to be good if and only if *push* and *pop* are never invoked concurrently[1].

We claim the stack's implementation depicted in Algorithm 6 is constant-time and meets the ideal-stack semantics on any good set of invocations. Unfortunately, the proof of this claim is omitted as it goes beyond the scope of this work.

```
procedure PUSH($entry$)
    $pointer \leftarrow$ pointerOf($self.entries[self.bottom]$)
    $self.entries[self.bottom] \leftarrow entry$
    $self.bottom \leftarrow self.bottom + 1$
    return $pointer$
end procedure

procedure POP
    if $self.bottom > 0$ then
        $self.bottom \leftarrow self.bottom - 1$
    end if
end procedure
```

Algorithm 6: A private stack's implementation.

6.1.2 The spreading mechanism

We now acquaint the reader with the asynchronous spreading mechanism. To facilitate the comprehension of the overall scheme, we explain the mechanism's behavior without accounting with the algorithm's flexibility.

As for CFLFWS, when some processor p enables two nodes, one of them is immediately assigned to p. Then, p allocates an entry from the bottom side of its private stack, and, stores inside it the node that did not become assigned. By default, both the *shared* and the *owned* flags of each entry are set to FALSE. If p's *spreading* flag is set to TRUE, the processor then makes a spread attempt:

1. First, p starts by setting that entry's *shared* flag to TRUE.

2. Next, it targets, uniformly at random, a possible donee q, and checks if q's *state* flag is set to IDLE:

   a) In such case, p makes q's *state* flag pointing to the entry that it just allocated (and initialized) from its local stack.

[1] *i.e.* no two invocations to *push* are concurrent, nor two invocations to *pop* are concurrent, and, finally, nor an invocation to *push* and another to *pop* are concurrent.

   b) Otherwise, it resets the entry's *shared* flag to FALSE, indicating it is not going to be shared.

3. In any case, p then updates its *spreading* flag according to the apparent outcome of the spread attempt it made to q[2].

If p's *spreading* flag was initially set to FALSE, or, if it is the case, after p concludes the spread attempt, the processor then pushes into the bottom of its deque a pointer to that stack's entry.

When the execution of p's assigned node does not enable any new nodes, p attempts to pop the bottommost pointer from its deque. If the deque is not empty, the bottommost pointer is returned and p goes onto checking the value of the corresponding entry's *shared* flag.

shared = **FALSE** In this case, the node stored inside the corresponding entry becomes p's new assigned node.

shared = **TRUE** Under this situation, p first checks if the node has already been assigned, by reading the entry's *owned* flag.

- *owned* = **TRUE** In this scenario, we say that the *fetchBottom* operation failed.
- *owned* = **FALSE** For this case, p then attempts to atomically acquire ownership of the node, executing a CAS instruction.
  - **Successful CAS** In this case, p was successful getting the node's ownership, and so the node becomes p's new assigned node.
  - **Failed CAS** In this situation, p could not get the node's ownership, and, alike for a previous case, we say that the *fetchBottom* operation failed. As we will see, this scenario can only take place if the donee, concurrently, acquired the ownership of the node.

Finally, in either case, p updates the bottom index of its private stack, excluding the previous bottommost entry.

Now, we describe how thieves can steal work from their victims. As always, a thief targets victims uniformly at random. After picking one, the thief invokes the *popTop* method to the victim's deque. If the steal is successful, a pointer to an entry stored in the victim's private stack is returned. To avoid unnecessary communication overheads, the thief first checks if the node stored in that entry is already owned by some other processor, reading the entry's *owned* flag. If it is (*i.e.* in the case that the flag's value is TRUE), the thief simply gives up on that steal attempt. Otherwise, the thief proceeds as follows:

1. First, it migrates the node, copying it from the victim's entry to local memory.

[2]Note that, due to the asynchronous execution environment, it could perfectly be the case that q just had assigned a ready node, and was about to update its *state* flag to WORKING, when p read its state and thought q was idle.

2. Then, the thief reads the *shared* flag, to know whether the node is possibly shared or otherwise.

   shared = FALSE In this case, the thief simply assigns the node.

   shared = TRUE Under this situation, the thief then attempts to atomically acquire ownership of the node, executing a CAS instruction.

   Successful CAS The thief was successful in acquiring the node's ownership, and so the node becomes the thief's new assigned node.

   Failed CAS For this scenario, the thief simply gives up on the steal attempt.

At this point, it only remains to explain how idle processors (*i.e.* thieves) accept donation offers. After, for one reason or another, a thief fails stealing work from a victim's deque, it audits the value of its *state* flag. If the flag is set to IDLE, the thief simply resumes its quest for work. However, if the flag's value corresponds to a pointer, the processor then attempts to obtain the offered node. To avoid unnecessary communication overheads, the thief first checks if the node stored in that entry is already owned by some other processor, reading the entry's *owned* flag.

owned = TRUE The thief simply resets its *state* flag back to IDLE and continues its hunt for work.

owned = FALSE Under this situation, the processor proceeds as follows:

1. First, it migrates the node, copying it from the donor's entry to local memory.
2. Then, the donee attempts to atomically acquire ownership of the node, executing a CAS instruction.

   Successful CAS The donee successfully acquired the node's ownership, and so the node becomes the processor's new assigned node.

   Failed CAS In this case, the node is no longer available for donation. As such, the donee simply gives up on the offer and returns to its quest for work.

As one might have noticed, the addition of the *shared* flag to each entry is nothing more than an optimization, allowing processors to avoid executing expensive CAS instructions when synchronization is not required. We now move to the algorithm's full specification.

6.1.3 The specification of the Canonical Flexible Work Stealing and Sharing algorithm

In this section we detail the Canonical Flexible Work Stealing and Sharing (CFWSS) algorithm's specification.

As for CFSAWSS, the CFWSS scheduler, depicted in Algorithm 7, takes as input three parameters:

θ_r — Corresponds to the probabilities that an idle processor chooses to skip a steal attempt, or, to skip making an update to some processor's *spreading* flag.

θ_s — The probability that a processor skips a spreading opportunity.

heuristic — Defines in what situations processors change their statuses from victims to victims and donors, and vice versa. The specification and meaning of this parameter for CFWSS is the same as for CFSAWSS: Let

$$\mathscr{D} = \{\text{TRUE}, \text{FALSE}, \text{UNCHANGED}\}$$

denote the domain of values that are used to update the *spreading* flag. Then,

$$heuristic \in (\mathscr{D} \times \mathscr{D} \times \mathscr{D}) \times (\mathscr{D} \times \mathscr{D} \times \mathscr{D} \times \mathscr{D})$$

being that the first element of the pair is a triple with the values according to which a processor must update its *spreading* flag after, respectively, an unmade, a successful or a failed spread attempt. The second element of *heuristic* is a quadruple featuring the values used by thieves to update the *spreading* flag of processors. Concretely, after a thief makes, or, if it is the case, skips a steal attempt, it might then update the *spreading* flag of another processor p_j, targeted uniformly at random. If the thief decides to proceed with the update, it makes it according to the outcome of its last steal opportunity:

- If the thief skipped the opportunity to make the attempt, it updates p_j's flag according to the value of the first field of the quadruple.
- However, if it did not skip, the thief updates p_j's *spreading* flag according to the outcome of the *popTop* invocation associated with that attempt: one of *success*, *empty*, or *abort*.

To ease the algorithm's comprehension, we partition the execution of each scheduling iteration into three stages, each being similar to its analogous one for the CFSAWSS algorithm. Contrarily to CFSAWSS, however, the execution of some scheduling iterations may take a non constant number of steps (see lines 14 to 16 of Algorithm 7).

Consider some processor p participating on a computation's execution.

STAGE I —

Busy iteration p executes its currently assigned node (line 4).

Idle iteration p starts by randomly choosing whether to make a steal attempt or not (lines 72 and 73).

Decides to make a steal attempt p picks a victim uniformly at random, and invokes the *popTop* method to its deque (lines 74 and 75). If the steal attempt is successful, p calls *assign*, attempting to assign the node stored in the entry pointed by the value returned by the invocation (line 77). In case it successfully assigns the node, it then sets its *state* flag to WORKING.

Decides not to make a steal attempt *p* simply skips the opportunity.

STAGE II —

Busy iteration The node executed by *p* in the previous stage enabled either 0, 1 or 2 other nodes.

0 nodes enabled *p* takes no action during this stage.

1 node enabled *p* gets the only enabled node assigned (line 6).

2 nodes enabled One of the enabled nodes becomes *p*'s new assigned node (line 6). Then, *p* stores the node it did not assign in the bottom of its private stack (line 8). Next, the processor invokes the *spread* procedure: *p* checks the value of its *spreading* flag (line 32):

If *p*'s *spreading* flag is set to TRUE — The processor decides whether to attempt spreading the non assigned node or not (lines 33 and 34).

***p* chooses to make a spread attempt** *p* picks a donee uniformly at random and checks its *state* flag (lines 36 and 37).
If the donee's *state* flag is set to IDLE, *p* makes the flag point to the entry of the node that it is willing to donate (line 38). Then, *p* updates its own *spreading* flag, as if it had successfully spread the the node (line 39).
If not, *p* resets the *shared* flag of the entry in which the offered node is stored, to FALSE, avoiding unnecessary synchronization costs (line 41). Next, *p* updates its own *spreading* flag as if its spread attempt had failed (line 42).

Otherwise In this case, *p* chose skipping the spread attempt opportunity. The processor then updates its *spreading* flag according to its decision (line 45) and takes no further action during this stage.

If *p*'s *spreading* flag is set to FALSE — *p* takes no action during this stage.

Idle iteration *p* randomly chooses to either make an update or not (lines 82 and 83).

Decides to make an update *p* chooses a new processor, uniformly at random, and updates this new processor's *spreading* flag according to the outcome of its last steal attempt opportunity (lines 84 to 93).

Decides not to make an update *p* simply skips the opportunity for updating some processor's *spreading* flag.

STAGE III —

Busy iteration The node executed by *p* in the first stage enabled 0, 1 or 2 other nodes.

0 nodes enabled *p* starts executing a cycle, attempting to obtain work (lines 14 to 16). Each iteration of that cycle consists on a call to the *fetchBottom* procedure: the processor starts by trying to pop a pointer from the bottom of its deque, invoking method *popBottom* (line 50).

EMPTY is returned In this case, *p*'s deque is empty. The processor then returns from the *fetchBottom* procedure, and, as we will see, exits the cycle.

Otherwise *p* attempts to assign the node stored in the entry pointed by the pointer it just obtained. If *p* does not assign the node, we say that the *fetchBottom* call failed. It is easy to see that this situation may only occur if *p* made a donation to some processor, and that processor accepted the donation. In any case, *p* then removes the entry of that node from its stack, making an invocation to the *pop* method (line 55).

If *p*'s invocation to *popBottom* returned EMPTY, or, if *p* successfully assigned a node, it then exits the cycle. Otherwise, the processor repeats it. If, after quiting the cycle, *p* has no assigned node, it sets its *state* flag to IDLE (lines 17 to 19), reflecting the fact that it has no work.

1 node enabled *p* takes no action in this stage.

2 nodes enabled *p* pushes a pointer (pointing to the entry in which the node that did not become its assigned one, is stored) into the bottom of its deque by invoking the method *pushBottom* (line 10).

Idle iteration *p* checks if it has an assigned node.

***p* has an assigned node** The processor sets its state to WORKING (line 96).

***p* does not have an assigned node** The processor checks the value of its *state* flag (line 97).

***p*'s *state* flag is set to IDLE** *p* takes no further action during this stage.

Otherwise In this case, *p* received a donation offer. Then, it calls the *assign* procedure, attempting to assign the node it has been offered. In case it successfully assigns the node, it then sets its *state* flag to WORKING (line 101). Otherwise, *p* sets its *state* flag back to IDLE, indicating it is available for accepting donation offers (line 103).

```
procedure SCHEDULER($\theta_r$, $\theta_s$, $heuristic$)
    while computation not terminated do
        if ValidNode($assigned$) then
            $enabled$ ←execute($assigned$)
            if length($enabled$)> 0 then
                $assigned$ ← $enabled[0]$
                if length($enabled$) = 2 then
                    $pointer$ ← $self$.StoreNewNode($enabled[1]$)
                    $self$.spread($pointer$, $\theta_s$, $heuristic[0]$)
                    $self.deque$.pushBottom($pointer$)
                end if
            else
                $result$ ← FALSE
                while $result$ = FALSE do
                    $result$ ← $self$.fetchBottom()
                end while
                if not ValidNode($assigned$) then
                    $self.state$ ← IDLE
                end if
            end if
        else
            $self$.WorkMigration($\theta_r$, $heuristic[1]$)
        end if
    end while
end procedure

function STORENEWNODE($node$)
    $entry$ ← Entry($node$, FALSE, FALSE)            ▷ The new entry containing the node.
    $pointer$ ← $self.stack$.push($entry$)
    return $pointer$
end function
```

Algorithm 7: The CFWSS algorithm.

```
31: procedure SPREAD($pointer$, $\theta_s$, $spreading\_heuristic$)
32:     if $self$.spreading then
33:         $choice \leftarrow UniformlyRandomNumber([0;1])$
34:         if $choice \geq \theta_s$ then
35:             $pointer.shared \leftarrow$ TRUE
36:             $donee \leftarrow UniformlyRandomProcessor()$
37:             if $donee.state =$ IDLE then
38:                 $donee.state \leftarrow pointer$
39:                 $self$.update($spreading\_heuristic[1]$)
40:             else
41:                 $pointer.shared \leftarrow$ FALSE
42:                 $self$.update($spreading\_heuristic[2]$)
43:             end if
44:         else
45:             $self$.update($spreading\_heuristic[0]$)
46:         end if
47:     end if
48: end procedure

49: function FETCHBOTTOM
50:     $pointer \leftarrow self.deque$.popBottom()
51:     if $pointer =$ EMPTY then
52:         return TRUE
53:     end if
54:     $result \leftarrow self$.assign($pointer$)
55:     $self.stack$.pop()                ▷ Makes its private stack reflect its deque's state
56:     return $result$
57: end function

58: function ASSIGN($pointer$)
59:     $assigned \leftarrow pointer.node$
60:     if $pointer.shared =$ FALSE then
61:         return TRUE
62:     else
63:         if CAS($pointer.owned$, FALSE, TRUE) = SUCCESS then
64:             return TRUE
65:         else
66:             $assigned \leftarrow$ NONE
67:             return FALSE
68:         end if
69:     end if
70: end function
```

```
71: procedure WorkMigration($\theta_r$, $stealing\_heuristic$)
72:     $choice_1 \leftarrow UniformlyRandomNumber([0;1])$
73:     if $choice_1 \geq \theta_r$ then
74:         $victim \leftarrow UniformlyRandomProcessor()$
75:         $pointer \leftarrow victim.deque$.popTop()
76:         if $pointer \neq$ EMPTY $and\, pointer \neq$ ABORT then
77:             assign($pointer$)
78:         end if
79:     else
80:         $pointer \leftarrow$ NONE
81:     end if
82:     $choice_2 \leftarrow UniformlyRandomNumber([0;1])$
83:     if $choice_2 \geq \theta_r$ then
84:         $updated \leftarrow UniformlyRandomProcessor()$ ▷ Pick another processor to update
85:         if $pointer =$ ABORT then
86:             $updated$.update($stealing\_heuristic[3]$)
87:         else if $pointer =$ EMPTY then
88:             $updated$.update($stealing\_heuristic[2]$)
89:         else if $pointer =$ NONE then                    ▷ Did not make a steal attempt
90:             $updated$.update($stealing\_heuristic[0]$)
91:         else                                ▷ Might have been a successful steal attempt
92:             $updated$.update($stealing\_heuristic[1]$)
93:         end if
94:     end if
95:     if ValidNode($assigned$) then
96:         $self.state \leftarrow$ WORKING
97:     else if $self.state \neq$ IDLE then
98:         $pointer \leftarrow self.state$
99:         assign($pointer$)
100:         if ValidNode($assigned$) then
101:             $self.state \leftarrow$ WORKING
102:         else
103:             $self.state \leftarrow$ IDLE
104:         end if
105:     end if
106: end procedure

107: procedure UPDATE($new$)
108:     if $new \neq$ UNCHANGED then
109:         $self.spreading \leftarrow new$
110:     end if
111: end procedure

112: function ValidNode($node$)
113:     return $node \neq$ EMPTY   $and$   $node \neq$ ABORT   $and$   $node \neq$ NONE
114: end function
```

As for CFLFWS and CFSAWSS, in the CFWSS algorithm only the owner of each deque may invoke the *pushBottom* and *popBotom* methods. Consequently, any set of invocations made to each deque during a computation's execution by the CFLFWS algorithm is good. We conclude that the relaxed semantics, afore-described in Section 4.4, will always be satisfied by the algorithm when executing a computation.

Additionally, we remark the fact that, for CFWSS, only the owner of each stack may invoke the *push* and *pop* methods. Thus, as for the case of the deque, any set of invocations made to each stack during a computation's execution by the CFLFWS algorithm is good. With this, we conclude that the ideal-stack semantics, as specified in Section 6.1.1, will always be satisfied by the algorithm when executing a computation.

6.1.3.1 Some remarks regarding CFWSS

First, the scheduler we have presented in Algorithm 7 is not fully correct. Consider the case when some donor offers to some donee a node u (by setting the donee's *state* flag to point to the entry where u is stored). It could happen that, before the donee tried to assign u, the donor popped u from its stack (*i.e.* popping the entry in which u was stored from the stack), acquired its ownership, executed it. If u's execution enabled two nodes, then, a new node v would be stored in the entry where u was previously stored. Furthermore, if the donor did not attempt to make a spread this time, that entry's *shared* flag would be set to F. Then, when the donee attempted to accept the donation offer, it would assign v, a node that was not meant to be shared. In such case, in some later step, either the donor or some thief would successfully remove the pointer to v's entry from the stack, and then assign and execute v. To conclude, in this case v would have been executed twice, while it should only have been executed a single time, possibly implying an incorrect execution of the computation.

This problem is actually very similar to the famous *ABA* problem. To address it, we would only have to add an age tag to each entry of a processor's stack, and, whenever a donee tried to assigned a shared node, it would also have ensure that the age of the node it was trying to assign was the same as the one that was offered to it. Due to the complexity of CFWSS algorithm, depicted in Algorithm 7, we have decided to omit this detail, avoiding adding even more complexity to the algorithm.

More, it also worth to mention the reason why the spreading scheme is of such complexity. As mentioned in the previous chapter, allowing processors to make spread attempts can be crucial in guaranteeing high performance for scenarios with few or unbalanced parallelism. Nonetheless, every bit of overhead added to a processor that is working severely harms the scheduler's performance, as these overheads scale with the total amount of work. For that reason, we designed the spreading mechanism in such a way that it incurred in no expensive synchronization. Further, we have also presented the Tit-for-tat heuristic that, pretty much, amortizes the costs of spreading with the costs of stealing. On that account, we expect that CFWSS can be used in practice for any type of computations, of course, depending on the heuristic used.

6.1.4 Analysis of the scheduler

Now, we obtain bounds on the expected runtime of a computation using the CFWSS algorithm. The analysis we make of this algorithm regarding its runtime bounds is similar to the one presented in Section 4.5.1 for the CFLFWS scheduler.

We say that a call to the *fetchBottom* procedure fails if and only if the corresponding return value is FALSE. Now, we reintroduce the concept of *milestone*. An instruction within the sequence executed by some processor is a milestone if and only if it corresponds to the execution of a node, to the completion of a call to the *WorkMigration* procedure, or, if it corresponds to a call of the *fetchBottom* procedure that failed.

Note that, for each time a processor executes an iteration of the "*fetchBottom*" cycle (*i.e.* executes the instructions corresponding to lines 14 to 16), it either executes a milestone corresponding to a call to the *fetchBottom* procedure that failed, or, quits the cycle and executes a milestone in the next scheduling iteration's execution. By observing the algorithm, it is straightforward to conclude that any processor executes at most a constant number of instructions within two consecutive milestones. Let C denote a constant such that, for any sequence of instructions executed by a processor whose length is C or more, at least one of such instructions is a milestone.

Lemma 6.1. *Consider any computation with work T_1 being executed by the CFWSS. Then, the execution time is $O\left(\frac{T_1}{P} + \frac{I}{P}\right)$, where I denotes the total number of idle iterations executed by processors.*

Proof. Consider three buckets: the work bucket, the idle bucket and the donation bucket. Whenever a processor executes a milestone, it collects a token. If the milestone corresponds to a node's execution, it places the collected token in the work bucket. On the other hand, if it corresponds to the completion of a *WorkMigration* procedure, the processor puts the token in the idle bucket. Finally, if the milestone corresponds to a *fetchBottom*'s failed call, it places the token in the donation bucket.

As we are in the presence of a dedicated environment, for each C consecutive steps, at least P tokens are placed into the buckets.

Since the computation has T_1 nodes, then, at the end of its execution the number of tokens inside the work bucket is T_1. On the other hand, as I corresponds to the total number of idle iterations executed by processors, and, as each of such iterations entails a call to the *WorkMigration* procedure along with its consequent completion, then, the number of tokens inside the idle bucket at the end of the computation's execution is I.

With this, it only remains to bound the number of tokens in the donation bucket at the end of the computation's execution. First, note that each token in that bucket entails a failed call to the *fetchBottom* procedure. In turn, a call to the *fetchBottom* procedure, made by some processor p, fails, if and only if p successfully popped a pointer from the bottom of its deque, the pointer pointed to an entry that was marked as shared (*i.e.* the entry's *shared* flag was set to TRUE), and, p could not acquire the ownership of the node that was stored in that entry. By observing the algorithm depicted in Algorithm 7, it is easy to conclude that there are at most two pointers, pointing to the entry containing the node whose ownership p failed to acquire: one in p's deque, and, if the node was offered to some donee, in that donee's *state* flag. Since p successfully popped the pointer to that entry from its deque, and, yet, failed to acquire that node's ownership, then, the donee must have accepted the donation made by p, and, before p, successfully acquired the ownership of the node stored in that entry.

Since the computation, by definition, contains exactly T_1 node, then, at most T_1 nodes can be donated. With this, we conclude that the number of tokens in the donation bucket is at most T_1, completing the proof of the lemma. ■

We now state a variant of the Structural Lemma (*i.e.* Lemma 4.3). The proof of the lemma uses the exact same arguments as the ones given in the proof of Lemma 4.3, but additionally taking into account the fact that the deque contains pointers to the entries where the nodes are stored. For that reason, the proof is omitted.

Lemma 6.2 (Structural Lemma)**.** *Let k be the number of pointers stored in a given deque at some time in the (linearized) execution of the CFWSS algorithm, and let $v_1,\ldots,v_k$ denote the nodes stored in the entries pointed by each of the pointers, ordered from the bottom of the deque to the top. Let v_0 denote the assigned node if there is one. In addition, for $i = 0,\ldots,k$ let u_i denote the designated parent of v_i. Then for $i = 1,\ldots,k$, node u_i is an ancestor of u_{i-1} in the enabling tree. Moreover, though we may have $u_0 = u_1$, for $i = 2,3,\ldots,k$, we have $u_{i-1} \neq u_i$ that is, the ancestor relationship is proper.*

Corollary 6.2.1. *If $v_0, v_1, \ldots, v_k$ are as defined in the statement of Lemma 6.2, then we have $w(v_0) \leq w(v_1) < \ldots < w(v_{k-1}) < w(v_k)$.*

Again, we reintroduce some of the concepts already presented in Section 4.5.1.

- The set of nodes that are ready at some step i is denoted by R_i.
- The potential associated with any node u at some step i is denoted $\phi_i(u)$ and is defined as

$$\phi_i(u) = \begin{cases} 3^{2w(u)-1} & \text{if } u \text{ is assigned} \\ 3^{2w(u)} & \text{otherwise} \end{cases}.$$

- The total potential at step i is denoted by Φ_i and is defined as

$$\Phi_i = \sum_{u \in R_i} \phi_i(u).$$

The following result is analogous to Lemma 4.4. We omit its proof as the one of Lemma 4.4 straightly applies for this algorithm as well.

Lemma 6.3. *Consider some node u, ready at step i during the execution of a computation. If u gets assigned to a processor at that step, the potential drops by at least $\frac{2}{3}\phi_i(u)$. Furthermore, if u was already assigned to a processor and gets executed at that step i, the potential drops by at least $\frac{5}{9}\phi_i(u)$.*

Again, we reintroduce more of the definitions already presented in Section 4.5.1.

- For CFWSS, we say that a node is attached to a processor p if it is the p's assigned node, or, if it is stored in an entry, whose pointer is stored in p's deque. The set of ready nodes attached to a processor p at the beginning of a step i is denoted by $R_i(p)$.
- The potential associated with p at step i is denoted $\Phi_i(p)$ and is defined as

$$\Phi_i(p) = \sum_{u \in R_i} \phi_i(u)$$

For each step i, processors are partitioned into two sets D_i and A_i, where the first is composed by all processors whose deque is not empty at the beginning of step i while the second is the set of all other processors. So, it follows $\Phi_i = \Phi_i(D_i) + \Phi_i(A_i)$, where

$$\Phi_i(D_i) = \sum_{p \in D_i} \Phi_i(p) \qquad \text{and} \qquad \Phi_i(A_i) = \sum_{p \in A_i} \Phi_i(p).$$

The following lemma follows from Corollary 6.2.1, and from the potential function's properties. Its proof is omitted since the arguments presented in the proof of Lemma 4.5 for the CFLFWS algorithm apply for this case as well.

Lemma 6.4. *Consider any step i and any processor $p \in D_i$. The top-most node u in p's deque contributes at least $\frac{3}{4}$ of the potential associated with p. That is, we have*

$$\phi_i(u) \geq \frac{3}{4}\Phi_i(p).$$

The next result is a consequence of Lemma 6.4 and from the fact that the deque's implementation satisfies the relaxed semantics (see Section 4.4).

Lemma 6.5. *Suppose a thief processor p chooses a processor $q \in D_i$ as its victim at some step j, such that $j \geq i$ (i.e. a steal attempt of p targeting q occurs at step j). Then, the potential decreases by at least $\frac{1}{2}\Phi_i(q)$ due to the assignment of the node stored in the entry pointed by the topmost pointer within q's deque.*

Proof. Let u denote the node stored in the entry pointed by the topmost pointer ptr of q's deque at the beginning of step i.

If some processor removes ptr from q's deque, u becomes assigned: by observing Algorithm 7, it is clear that after a processor r pops ptr from q's deque, either from the bottom (in which case it would be q popping the node) or from the top, r will then invoke the *assign* method. Thus, either r or some donee to whom q offered u, successfully acquired u's ownership, and consequently assigned the node.

Regarding p's steal attempt to q's deque, three possible scenarios may occur. We show that in either of these cases, some processor removes ptr from q's deque, and consequently, assigns u.

The deque is empty Since $q \in D_i$, some other processor successfully removed ptr from q's deque. Since ptr was removed from q's deque, then, u was assigned by some processor.

The steal attempt aborts Since the deque implementation meets the relaxed semantics on any good set of invocations, and since, as aforementioned, the CFWSS algorithm only makes good sets of invocations, then, some other processor successfully removed a topmost pointer from q's deque during the aborted steal attempt made by p. If the removed pointer was ptr, then u gets assigned to some processor (that may either be q, some other thief that successfully stole ptr, or even a donee). On the other hand, if the removed pointer was not ptr, then ptr must have been previously stolen, implying u was assigned to some processor.

Successful steal attempt If p stole ptr, then, u gets assigned to some processor. Otherwise, some other processor removed ptr before p did. Thus, even for this case, u got assigned to some processor.

From Lemma 6.4, we have

$$\phi_i(u) \geq \frac{3}{4}\Phi_i(q).$$

Furthermore, Lemma 6.3 proves that if u gets assigned, the potential decreases by at least

$$\frac{2}{3}\phi_i(u).$$

Because u is assigned in any case, we conclude the potential associated with q decreases by at least

$$\frac{3}{4}.\frac{2}{3}\Phi_i(q) = \frac{1}{2}\Phi_i(q).$$

■

The following result is a consequence of Lemmas 3.1 and 6.5. The proof of this lemma is the same as the one of Lemma 4.7, being omitted for that reason.

Lemma 6.6. *Consider any θ_r such that $0 \leq \theta_r < 1$, and, consider any step i and any later step j such that at least $\frac{P}{1-\theta_r}$ idle iterations occur from i (inclusive) to j (exclusive). Then, we have*

$$P\left\{\Phi_i - \Phi_j \geq \frac{1}{4}.\Phi_i(D_i)\right\} > \frac{1}{4}.$$

Finally, we can bound the expected runtime of computations run under the CFWSS algorithm. The result follows from Lemmas 6.1 and 6.6. It can be proved using the exact same reasoning given in the proof of Theorem 4.8, and, for that reason, its proof is omitted.

Theorem 6.7. *Consider any θ_r such that $0 \leq \theta_r < 1$, any computation with work T_1 and critical-path length T_∞ being executed by the CFWSS algorithm with P processors in a dedicated environment. Let θ_r denote the value of the first parameter passed to the scheduler. Then, the expected execution time is $O\left(\frac{T_1}{P} + \frac{T_\infty}{1-\theta_r}\right)$. Moreover, for any $\varepsilon > 0$, the execution time is $O\left(\frac{T_1}{P} + \frac{T_\infty + \log\left(\frac{1}{\varepsilon}\right)}{1-\theta_r}\right)$ with probability at least $1 - \varepsilon$.*

6.2 A provably efficient scheduler that avoids most unnecessary memory barriers

In this section, we present the Work Stealing with Semi-Private Deques (WSSPD), which, for the best of our knowledge, is the first algorithm with nearly optimal expected runtime bounds and for which the expected number of memory barriers issued is at most $O(PT_\infty)$.

In the WSSPD algorithm (depicted in Algorithm 8), each processor owns a Semi-private Double Ended Queue (semi-private-deque) that uses to store its attached ready nodes, and, in addition, owns a *targeted* flag that stores a boolean value (*i.e.* TRUE or FALSE). The specification of the semi-private-deque will be given in Section 6.2.1.

Before a computation's execution starts, every processor sets its *assigned* node to NONE, and its *targeted* flag to FALSE. Then, to begin the execution, the root node is assigned to an arbitrary processor.

Now, we give an informal description of the algorithm, aiming to acquaintance the reader with the overall idea of the scheduler. During a computation's execution, each busy processor works locally on the private part of its semi-private-deque, adding and removing ready nodes from the private part, as necessary. When the private part becomes empty, the processor invokes the *popBottom* procedure, attempting to fetch a node from the public part of the semi-private-deque. On the other hand, idle processors are searching for work, making steal attempts targeting other processors' semi-private-deques. If a thief attempts to steal a victim's semi-private-deque whose public part is empty, it updates the victim's *targeted* flag, informing the victim that it was targeted by a steal attempt. The victim then invokes the *updateBottom* method to its semi-private-deque, adding a new node to the public part of the semi-private-deque. Then, if the victim is targeted by another steal attempt, the node that became public is either stolen by a thief, or, if it is the case, was removed by the victim as a consequence of the private part of its semi-private-deque being empty.

```
procedure SCHEDULER
    while $computation\ \ not\ \ terminated$ do
        if $self.targeted$ then
            $self.deque$.updateBottom()
            $self.targeted \leftarrow$ FALSE
        end if
        if ValidNode($assigned$) then
            $enabled \leftarrow$execute($assigned$)
            if length($enabled$) > 0 then
                $assigned \leftarrow enabled[0]$
                if length($enabled$) = 2 then
                    $self.deque$.push($enabled[1]$)
                end if
            else
                $assigned \leftarrow self.deque$.pop()
                if $assigned$ = RACE then
                    $assigned \leftarrow self.deque$.popBottom()
                end if
            end if
        else
            $self$.WorkMigration()
        end if
    end while
end procedure

procedure WORKMIGRATION
    $victim \leftarrow$ UniformlyRandomProcessor()
    $assigned \leftarrow victim.deque$.popTop()
    if $assigned$ = EMPTY then
        $victim.targeted \leftarrow$ TRUE
    end if
end procedure

function VALIDNODE($node$)
    return $node \neq$EMPTY  $and$  $node \neq$ ABORT  $and$  $node \neq None$
end function
```

Algorithm 8: The WSSPD algorithm.

We now move to present the implementation and associated semantics of the semi-private-deque.

6.2.1 The semi-private lock free deque

In this section, we present the specification of a semi-private-deque's implementation, along with its associated Semi-Private Deque Relaxed Semantics (SPDR Semantics) semantics. Our implementation is based on the deque's implementation given in [Aro+01].

A semi-private-deque is partitioned into two parts: a private and a public one. The public part corresponds to the topmost part of the semi-private-deque while the private corresponds to the rest of the semi-private-deque. As we will see, the owner of a semi-private-deque always works on the private part, except when that part becomes empty.

A semi-private-deque object contains four variables:

1. A variable $entries$, corresponding to an array of ready nodes.

2. A variable $privateBottom$ that is the index below the bottommost node of the semi-private-deque, and also the bottommost node of the structure's private part.

3. A variable $officialBottom$ that is the index below the bottommost node of the semi-private-deque's public part.

4. A variable age containing two fields: the first is top, corresponding to the index of the topmost node of the semi-private-deque's public part, and, the second is tag that is only used to ensure correction (avoiding the ABA problem).

A semi-private-deque object meeting the SPDR Semantics supports five methods:

1. *push* — Pushes a node into the bottom of the semi-private-deque's private part, and does not return a value.

2. *pop* — Removes and returns a node from the bottom of the semi-private-deque's private part, if that part is not empty. Otherwise returns the special value RACE (*i.e.* if it is the case that the private part of the semi-private-deque is empty).

3. *updateBottom* — Transfers the topmost node from the private part of the semi-private-deque to the public part, if the private part of the semi-private-deque was not empty. In other words, an invocation makes the topmost node from the private part of the semi-private-deque to become stealable, unless the private part is empty.

4. *popBottom* — Returns and removes the bottommost node from the public part of the semi-private-deque, if any. Otherwise, the invocation has no effect and EMPTY is returned.

5. *popTop* — Attempts to return and remove the topmost node from the public part of the semi-private-deque. If the public part of the semi-private-deque is empty, the invocation has no effect and the value EMPTY is returned. Otherwise, if the invocation aborts (*i.e.* if the attempt fails), the invocation has no effect and the special value ABORT is returned.

As for the deque, a semi-private-deque implementation is said to be constant-time if and only if any invocation to any of these methods takes at most a constant number of steps to return.

An invocation to a semi-private-deque method is defined by a 4-tuple establishing:

1. Which method was invoked.
2. The invocation's beginning time.
3. The invocation's completion time.
4. The return value, if any.

We say that a set of invocations to a semi-private-deque's methods meets the SPDR Semantics if and only if there is a set of *linearization times* for the corresponding non-aborting invocations such that:

1. Every non-aborting invocation's linearization time lies within the beginning and completion times of the respective invocation.
2. No linearization times associated with distinct non-aborting invocations coincide.
3. The return values for each non-aborting invocation are consistent with a serial execution of the methods in the order given by the linearization times of the corresponding non-aborting invocations.
4. For each aborted *popTop* invocation x to a semi-private-deque d, there exists another invocation removing the topmost item from d whose linearization time falls between the beginning and completion times of x's invocation.

A set of invocations is said to be *good* if and only if no invocation to *push*, *pop*, *updateBottom* and *popBottom* are never invoked concurrently, and *popBottom* can only be invoked after a *pop* invocation returned the special value RACE.

We claim that the implementation of the semi-private-deque is constant-time and meets the SPDR Semantics on any good set of invocations. The proof of the claim is left as future work, because of its extension and complexity, and, most of all because it is not the focus of this thesis. Yet, we remark the fact that this implementation is based on the implementation of the lock-free deque given in [Aro+01]. More, we also remark that, that deque's implementation has been proven, in [Blu+99], to be a correct implementation, meeting the relaxed semantics (as defined in [Aro+01]) on any good set of invocations (for the corresponding definition of good sets of invocations, given in Section 4.4).

```
procedure PUSH($node$)
    $pBot \leftarrow self.privateBottom$
    $self.entries[pBot] \leftarrow node$
    $self.privateBottom \leftarrow pBot + 1$
end procedure

procedure POP
    $pBot \leftarrow self.privateBottom$
    if $pBot = self.officialBottom$ then
        return RACE
    end if
    $pBot \leftarrow pBot - 1$
    $node \leftarrow self.entries[pBot]$
    $self.privateBottom \leftarrow pBot$
    return $node$
end procedure

procedure UPDATEBOTTOM
    $pBot \leftarrow self.privateBottom$
    $oBot \leftarrow self.officialBottom$
    if $pBot > oBot$ then
        $oBot \leftarrow oBot + 1$
    end if
    $self.officialBottom \leftarrow oBot$
end procedure
```

Algorithm 9: The five SPDeque's methods.

```
procedure popBottom
    $oBot \leftarrow self.officialBottom$
    if $oBot = 0$ then
        return EMPTY
    end if
    $oBot \leftarrow oBot - 1$
    $self.officialBottom \leftarrow oBot$
    $node \leftarrow self.entries[oBot]$
    $oldAge \leftarrow age$
    if $oBot > oldAge.top$ then
        return $node$
    end if
    $self.officialBottom \leftarrow 0$
    $self.privateBottom \leftarrow 0$
    $newAge.top \leftarrow 0$
    $newAge.tag \leftarrow oldAge.tag + 1$
    if $oBot = oldAge.top$ then
        if CAS($age, oldAge, newAge$) = SUCCESS then
            return $node$
        end if
    end if
    $self.age \leftarrow newAge$
    return EMPTY
end procedure

procedure popTop
    $oldAge \leftarrow self.age$
    $oldBottom \leftarrow self.officialBottom$
    if $oldBottom \leq oldAge.top$ then
        return EMPTY
    end if
    $node \leftarrow self.entries[oldAge.top]$
    $newAge \leftarrow oldAge$
    $newAge.top \leftarrow newAge.top + 1$
    if CAS($age, oldAge, newAge$) = SUCCESS then
        return $node$
    end if
    return ABORT
end procedure
```

It is straightforward to see that any set of invocations made to each processor's semi-private-deque, under the WSSPD algorithm, is good, since only the owner of each semi-private-deque may invoke the *push*, *pop*, *updateBottom* and *popBottom* methods. As such, we assume that the SPDR Semantics will always be satisfied for this algorithm, during a computation's execution.

6.2.1.1 Analysis of WSSPD

We now obtain bounds on the expected runtime and on the expected number of memory barriers issued by invocations to the semi-private-deque methods, for the WSSPD algorithm. The analysis we make of this algorithm regarding its runtime bounds follows the same traits as the one given in [Aro+01] for LFWS.

Recall that a scheduling iteration is defined as the sequence of executed instructions, including sub-routine calls, corresponding to one iteration of a scheduling loop. For the WSSPD scheduler, it corresponds to the sequence of instructions starting at line 2 and terminating at line 23 of Algorithm 8. Again, note that the full sequence of instructions executed by a processor during a computation's execution can be partitioned into smaller chunks, each corresponding to a scheduling iteration.

Now, we recall the concept of milestone. For this analysis, we consider the same definition of milestone as we considered in Section 4.5.1: An instruction within the sequence executed by some processor is a milestone if and only if it corresponds to the execution of a node or to the completion of a call to the *WorkMigration* procedure. Again, by observing the algorithm it is trivial to see that any processor can execute at most a constant number of instructions within two consecutive milestones. Through the rest of this analysis, let C denote a large enough constant such that, for any sequence of instructions executed by a processor whose length is C or more, at least one of such instructions is a milestone.

As for CFLFWS, it is straightforward to deduce that any scheduling iteration of the algorithm includes a milestone. We distinguish between scheduling iterations according to the type of milestone they contain. Iterations whose milestone corresponds the execution of a node are called busy iterations, while the remainder are labeled as idle iterations.

We say that a node u is *stealable* if and only if it is stored in the public part of some processor's semi-private-deque.

By following the exact same arguments as the ones given in the proof of Lemma 4.2, we can deduce the following lemma:

Lemma 6.8. *Consider any computation with work T_1 being executed by the WSSPD. Then, the execution time is $O\left(\frac{T_1}{P} + \frac{I}{P}\right)$, where I denotes the total number of idle iterations executed by processors.*

The following lemma proves that the number of memory barriers issued during the execution of a computation under WSSPD is bounded by the number of idle iterations and by the number of processors. We only present a proof sketch as it relies on the assumption that no invocation of the methods *push* and *pop* made by a processor to its own semi-private-deque ever issues a memory fence.

Lemma 6.9. *Consider any computation being executed by the WSSPD, using P processors. The number of memory fences issued by the WSSPD algorithm during the computation's execution is $O(I + P)$, where I denotes the total number of idle iterations executed by processors.*

Proof sketch. By observing Algorithm 8, it is trivial to see that the memory fences may only be issued by invocations to the methods of the semi-private-deque, or, be issued between an invocation to the *updateBottom* method and consequent update to the *targeted* flag of the processor who made the invocation.

First, note that invocations to the *push* or *pop* methods do not issue memory fences, as these only operate in the private part of the semi-private-deque, and require no synchronization.

Second, note that the number of *popTop* invocations is the same as the number of idle iterations, implying it is exactly I. If one such invocation returns EMPTY, the owner of the processor to which the corresponding *popTop* was invoked has its *targeted* flag set to TRUE. By observing Algorithm 8, it is trivial to conclude that for each *popTop* invocation that returns EMPTY, at most one call to the *updateBottom* is made. Since the number of *popTop* invocations is at most I, we conclude the number of invocations made to the *updateBottom* method is at most I as well. At this point, it only remains to bound the number of invocations made to the *popBottom* method.

A processor only invokes the *popBottom* method if the private part of its semi-private-deque is empty. Consider a processor p that invokes the *popBottom* method to its semi-private-deque. Since the semi-private-deque meets the SPDR Semantics, only two possible scenarios may occur:

A node is returned In this case, p assigns the node that was returned. Noting that a node can only be transferred to the public part of the semi-private-deque when a processor invokes the *updateBottom* method. As we have already proved, the number of invocations made to the *updateBottom* method is at most I, implying the number of times a processor successfully pops a ready node from the bottom of the semi-private-deque's public part is at most I.

EMPTY is returned In this case, p's semi-private-deque is empty. Now, consider the following two possible cases:

- The processor does not execute any milestone until the computation's execution terminates. In such case, noting there are P processors executing the computation, the number of *popBottom* invocations corresponding to this situation is at most P.
- Otherwise, p executes another milestone before the computation's execution terminates. Since the semi-private-deque meets the SPDR Semantics, we know that p will not have an assigned node at the beginning of the next iteration's execution, implying the next iteration will be an idle one. Since, by definition, there are I such iterations, we conclude that the number of times corresponding to this situation is at most I.

With this, we can conclude that the number of *popBottom* invocations made to the processor's semi-private-deques is at most $O(I+P)$. So, the total number of invocations made to *popBottom*, *updateBottom* and *popTop* is $O(I+P)$.

We now bound the number of memory fences issued during the computation's execution. That the number of memory fences issued during each invocation to those methods is constant, is a consequence of the fact that the semi-private-deque's implementation is constant-time. Furthermore, since a processor may only execute at most C instructions between two consecutive milestones (by the definition of C), the number of memory fences issued between some processor p's invocation to the *updateBottom* method and its consequent update to its *targeted* flag is constant.

With this, we conclude that the number of memory fences issued during a computation's execution by the WSSPD scheduler is at most $O(I+P)$. ■

We now state a variant of the Structural Lemma (*i.e.* Lemma 4.3), considering semi-private-deques instead of deques. The proof of the lemma uses the exact same arguments as the ones given in the proof of Lemma 4.3, but taking into account the fact that the WSSPD algorithm uses semi-private-deques rather than deques. For that reason, the proof is omitted.

Lemma 6.10 (Structural Lemma)**.** *Let k be the number of nodes in a given semi-private-deque at some time in the (linearized) execution of the WSSPD algorithm, and let $v_1, \ldots, v_k$ denote those nodes ordered from the bottom of the deque to the top. Let v_0 denote the assigned node if there is one. In addition, for $i = 0, \ldots, k$ let u_i denote the designated parent of v_i. Then for $i = 1, \ldots, k$, node u_i is an ancestor of u_{i-1} in the enabling tree. Moreover, though we may have $u_0 = u_1$, for $i = 2, 3, \ldots, k$, we have $u_{i-1} \neq u_i$ that is, the ancestor relationship is proper.*

Corollary 6.10.1. *If $v_0, v_1, \ldots, v_k$ are as defined in the statement of Lemma 6.10, then we have $w(v_0) \leq w(v_1) < \ldots < w(v_{k-1}) < w(v_k)$.*

Again, we reintroduce some of the concepts already presented in Section 4.5.1. We denote the set of ready nodes at some step i by R_i. Consider any node $u \in R_i$. The potential associated with u at step i is denoted by $\phi_i(u)$ and is defined as

$$\phi_i(u) = \begin{cases} 4^{3w(u)-2} & \text{if } u \text{ is assigned} \\ 4^{3w(u)-1} & \text{if } u \text{ is stealable} \\ 4^{3w(u)} & \text{otherwise} \end{cases}$$

The total potential at step i, denoted by Φ_i, corresponds to the sum of potentials of all the nodes that are ready at that step:

$$\Phi_i = \sum_{u \in R_i} \phi_i(u).$$

The following result is analogous to Lemma 4.4. We only present a proof sketch as its full version follows the same reasoning we made in the proof of Lemma 4.4.

Lemma 6.11. *Consider some node u, ready at step i during the execution of a computation. If u gets assigned to a processor at that step, the potential drops by at least $\frac{3}{4}\phi_i(u)$. If u becomes stealable at that step, the potential drops by at least $\frac{3}{4}\phi_i(u)$. Moreover, if u was already assigned to a processor and gets executed at that step i, the potential drops by at least $\frac{47}{64}\phi_i(u)$.*

Proof sketch. Regarding the first claim, note that, in the worst case, u was stealable, and so, the potential decreases from $4^{3w(u)-1}$ to $4^{3w(u)-2}$. Thus,

$$\begin{aligned} 4^{3w(u)-1} - 4^{3w(u)-2} &= \frac{3}{4}4^{3w(u)-1} \\ &= \frac{3}{4}\phi_i(u) \end{aligned}$$

So, we conclude that, if u is assigned, the potential decreases by at least $\frac{3}{4}\phi_i(u)$.

Regarding the second one, note that u was not stealable (because it became stealable at step i), and so, the potential decreases from $4^{3w(u)}$ to $4^{3w(u)-1}$. It follows,

$$\begin{aligned} 4^{3w(u)} - 4^{3w(u)-1} &= \frac{3}{4}4^{3w(u)} \\ &= \frac{3}{4}\phi_i(u) \end{aligned}$$

So, if u becomes stealable, the potential decreases by $\frac{3}{4}\phi_i(u)$.

Regarding the last claim, note that, due to our conventions regarding the computations' structure, a node can be the designated parent of at most two other ones in the enabling tree. Further, by definition, the weight of any enabled nodes is strictly smaller than the weight of their designated parent. In the most interesting case, two nodes are enabled. In such case, one of the enabled nodes becomes the assigned node of the processor whist the other is pushed onto the bottom of the semi-private-deque's private part. Let x denote the enabled node that is assigned by the processor and y the other enabled node. Then,

$$\begin{aligned} \phi_i(u) - \phi_{i+1}(x) - \phi_{i+1}(y) &= 4^{3w(u)-2} - 4^{3w(x)} - 4^{3w(y)-2} \\ &= 4^{3w(u)-2} - 4^{3((w(u)-1)} - 4^{3(w(u)-1)-2} \\ &= 4^{3w(u)-2}\left(1 - \frac{1}{4} - \frac{1}{64}\right) \\ &= \frac{47}{64}\phi_i(u) \end{aligned}$$

So, the potential decreases by $\frac{47}{64}\phi_i(u)$.

The remaining scenarios are checked in a similar fashion, like shown in the proof of Lemma 4.4. ■

Once more, we reintroduce more of the definitions already presented in Section 4.5.1.

- For WSSPD, we say that a node is attached to a processor p if it is the p's assigned node, or, if it is stored in p's semi-private-deque. The set of ready nodes attached to a processor p at the beginning of a step i is denoted by $R_i(p)$.
- The potential associated with p at step i is denoted $\Phi_i(p)$ and is defined as

$$\Phi_i(p) = \sum_{u \in R_i} \phi_i(u)$$

We say that a processor's semi-private-deque is empty if and only if both the private and the public parts of the semi-private-deque are empty.

For each step i, processors are partitioned into two sets D_i and A_i, where the first is composed by all processors whose semi-private-deque is not empty at the beginning of step i while the second is the set of all other processors. So, it follows $\Phi_i = \Phi_i(D_i) + \Phi_i(A_i)$, where

$$\Phi_i(D_i) = \sum_{p \in D_i} \Phi_i(p) \qquad \text{and} \qquad \Phi_i(A_i) = \sum_{p \in A_i} \Phi_i(p).$$

The following lemma follows from Corollary 6.10.1, and from the potential function's properties. We only present a proof sketch of this result as the arguments presented in the proof of Lemma 4.5 for the CFLFWS algorithm apply for this case.

Lemma 6.12. *Consider any step i and any processor $p \in D_i$. The top-most node u in p's semi-private-deque contributes at least $\frac{4}{5}$ of the potential associated with p. That is, we have*

$$\phi_i(u) \geq \frac{4}{5}\Phi_i(p).$$

Proof sketch. This lemma follows from Corollary 6.10.1. Suppose the topmost node u in p's semi-private-deque is also the only node in p's semi-private-deque. Furthermore, suppose that u is stealable and that it has the same designated parent as the node v, that is the one currently assigned to p. We then have

$$\begin{aligned}
\Phi_i(q) &= \phi_i(u) + \phi_i(v) \\
&= 4^{3w(u)-1} + 4^{3w(v)-2} \\
&= 4^{3w(u)-1} + 4^{3w(u)-2} \\
&= 4^{3w(u)-1}\left(1 + \frac{1}{4}\right) \\
&= \frac{5}{4}\phi_i(u)
\end{aligned}$$

By following similar arguments as the ones given in the proof of Lemma 4.5, we can conclude that in all other cases u contributes to an even larger fraction of q's potential. ■

The next result is a consequence of Lemma 6.12 and from the fact that the semi-private-deque's implementation satisfies the SPDR Semantics.

Lemma 6.13. *Suppose a thief processor p chooses a processor $q \in D_i$ as its victim at some step j, such that $j \geq i$ (i.e. a steal attempt of p targeting q occurs at step j). Then, at step $j + 2C$, the potential decreased by at least $\frac{3}{5}\Phi_i(q)$ due to either the assignment of the topmost node in q's semi-private-deque, or for making the topmost node of q's semi-private-deque become stealable.*

Proof. Let u denote the topmost node of q's semi-private-deque at the beginning of step i. We first prove that either u gets assigned or becomes stealable.

Three possible scenarios may take place due to p's steal attempt targeting q's semi-private-deque.

The invocation returns a node If p stole u, then, u gets assigned to p. Otherwise, some other processor removed u before p did, implying u got assigned to that other processor.

The invocation aborts (*i.e.* returns ABORT) Since the semi-private-deque implementation meets the SPDR Semantics on any good set of invocations, and because the WSSPD algorithm only makes good sets of invocations, we conclude that some other processor successfully removed a topmost node from q's semi-private-deque during the aborted steal attempt made by p. If the removed node was u, then u gets assigned to a processor (that may either be q, or, some other thief that successfully stole u). Otherwise, if the removed node was not u, then u must have been previously stolen by a thief or popped by q, and thus became assigned to some processor.

The invocation returns EMPTY This situation can only occur if either q's semi-private-deque is empty, or, if there is no node in the public part of q's semi-private-deque.

- For the first case, since $q \in D_i$, some other processor must have successfully removed u from q's semi-private-deque. Consequently, u was assigned by a processor.
- If there was no node in the public part of q's semi-private-deque, p sets q's *targeted* flag to TRUE in a later step j'. Recall that, for each C consecutive instructions executed by a processor, at least one corresponds to a milestone. It follows, $j' \leq j + C$. Furthermore, by observing Algorithm 8, we conclude that q will make and complete an invocation to *updateBottom* of its semi-private-deque in one of the C steps succeeding step j'. Thus if q's semi-private-deque's private part is not empty, a node will become stealable. From that invocation, only two possible situations can take place:

  No node becomes stealable In this case, the private part of q's semi-private-deque was empty, implying some processor (either q or some thief) assigned u.

  A node becomes stealable If the node that became stealable was not u, then either q or some other thief assigned u. Otherwise, u became stealable.

With this, we conclude that u either became assigned or became stealable until step $j+2C$.

From Lemma 4.5, we have

$$\phi_i(u) \geq \frac{4}{5}\Phi_i(q).$$

Furthermore, Lemma 4.4 proves that if u gets assigned, the potential decreases by at least

$$\frac{3}{4}\phi_i(u),$$

and if u becomes stealable, the potential also decreases by at least

$$\frac{3}{4}\phi_i(u).$$

Because u is either assigned or becomes stealable in any case, we conclude the potential associated with q at step $j+2C$ has decreased by at least

$$\frac{4}{5}.\frac{3}{4}\Phi_i(q) = \frac{3}{5}\Phi_i(q).$$

■

The following result is a consequence of Lemmas 3.1 and 6.13. The proof of this lemma follows the same traits as the one of Lemma 4.7, but without accounting with the parameter θ_r.

Lemma 6.14. *Consider any step i and any later step j such that at least P idle iterations occur from i (inclusive) to j (exclusive). Then, we have*

$$P\left\{\Phi_i - \Phi_{j+2C} \geq \frac{3}{10}.\Phi_i(D_i)\right\} > \frac{1}{4}.$$

Proof. By Lemma 6.13 we know that for each processor p in D_i that is targeted by a steal attempt, the potential drops by at least

$$\frac{3}{5}\Phi_i(p).$$

We can think of each idle iteration as a ball of Lemma 3.1, where the probability for being discarded is 0. Thus, for each idle iteration, a steal attempt is made, entailing a *popTop* invocation to the semi-private-deque of a processor targeted uniformly at random (*i.e.* a victim), and, in case the invocation returns EMPTY, also entailing an update to the victim's *targeted* flag to TRUE. For each processor p in D_i, we assign it a weight

$$W_p = \frac{3}{5}\Phi_i(p),$$

and for each other processor p in A_i, we assign it a weight

$$W_p = 0.$$

The weights sum to

$$W = \frac{3}{5}\Phi_i(D_i).$$

Using $\beta = \frac{1}{2}$, P as the number of bins and 0 as the probability of discarding a ball, then, from Lemma 3.1 it follows that with probability at least

$$1 - \frac{1}{(1-\beta)e} > \frac{1}{4},$$

the potential decreases by at least

$$\beta W = \frac{3}{10}\Phi_i(D_i).$$

This concludes the proof of this lemma. ■

Now, we bound the expected number of idle iterations taking place during a computation's execution using the WSSPD algorithm. The result follows from Lemmas 6.8 and 6.14. It is proved using similar arguments as the ones used in the proof of Theorem 4.8, but, without accounting with flexibility.

Lemma 6.15. *Consider any computation with work T_1 and critical-path length T_∞ being executed by the WSSPD algorithm with P processors in a dedicated environment. Then, the expected number of idle iterations is at most $O(PT_\infty)$.*

Proof. To analyze the number of idle iterations, we break the execution into *phases* composed by $\Theta(P)$ idle iterations. Then, we prove that, with constant probability, a phase leads the potential to drop by a constant factor.

A computation's execution begins when its root, which, by our conventions related with the structure of the dags, is unique, gets assigned to an arbitrary processor. Consequently, since, by definition, the root has weight T_∞, then, at the beginning of a computation's execution the potential is $\Phi_0 = 4^{3T_\infty - 2}$. Furthermore, it is straightforward to conclude that the potential is 0 after and only after a computation's execution terminates. We then use these facts to bound the expected number of phases needed to decrease the potential down to 0. The first phase starts at step $t_1 = 1$, and ends at the first step $t_1{}'$ such that, at least P idle iterations took place during the interval $[t_1, t_1{}' - 2C]$. The second phase begins at step $t_2 = t_1{}' + 1$, and so on.

Consider two consecutive phases starting at steps i and j respectively. We now prove that

$$P\left\{\Phi_j \leq \frac{7}{10}\Phi_i\right\} > \frac{1}{4}.$$

Recall that we can partition the potential as

$$\Phi_i = \Phi_i(A_i) + \Phi_i(D_i).$$

Since, from the beginning of each phase and until its last $2C$ steps, at least P idle iterations take place, then, by Lemma 4.7 it follows

$$P\left\{\Phi_i - \Phi_j \geq \frac{3}{10}\Phi_i(D_i)\right\} > \frac{1}{4}.$$

Now, we have to prove the potential also drops by a constant fraction of $\Phi_i(A_i)$. Consider some processor $p \in A_i$:

- If p does not have an assigned node, then

$$\Phi_i(p) = 0.$$

- Otherwise, if p has an assigned node u at step i, then,

$$\Phi_i(p) = \phi_i(u).$$

  Noting that each phase has more than C steps, then, p executes u before the next phase begins (*i.e.* before step j). Thus, the potential drops by at least $\frac{47}{64}\phi_i(u)$ during that phase.

Cumulatively, for each $p \in A_i$, it follows

$$\Phi_i - \Phi_j \geq \frac{47}{64}\Phi_i(A_i).$$

Thus, no matter how Φ_i is partitioned between $\Phi_i(A_i)$ and $\Phi_i(D_i)$, we have

$$P\left\{\Phi_i - \Phi_j \geq \frac{3}{10}\Phi_i\right\} > \frac{1}{4}.$$

We say a phase is successful if it leads the potential to decrease by at least a $\frac{3}{10}$ fraction. So, a phase succeeds with probability at least $\frac{1}{4}$. Since the potential is an integer, and, as aforementioned, starts at $\Phi_0 = 4^{3T_\infty - 2}$ and ends at 0, then, there can be at most

$$\begin{aligned}(3T_\infty - 2)\log_{\frac{10}{7}}(4) &= (3T_\infty - 2)\frac{\ln(4)}{\ln\left(\frac{10}{7}\right)} \\ &< 12T_\infty\end{aligned}$$

successful phases. If we think of each phase as a coin toss, where the probability that we get heads is at least $\frac{1}{4}$, then, by the binomial distribution, to get heads $12T_\infty$ times, the expected number of coins we have to toss is at most $48T_\infty$. In the same way, the expected number of phases needed to obtain $12T_\infty$ successful ones is at most $48T_\infty$. Consequently, the expected number of phases is $O(T_\infty)$. Moreover, as each phase contains $O(P)$ idle iterations, the expected number of idle iterations is $O(PT_\infty)$. ■

Theorem 6.16. *Consider any computation with work T_1 and critical-path length T_∞ being executed by the WSSPD algorithm with P processors in a dedicated environment. Then, the expected execution time is at most $O\left(\frac{T_1}{P} + T_\infty\right)$.*

Proof. Lemma 6.8 bounds the execution time in terms of the number of idle iterations. Taking into account Lemma 6.15, we conclude that the expected execution time of a computation under WSSPD is at most $O\left(\frac{T_1}{P} + T_\infty\right)$. ■

Theorem 6.17. *Consider any computation with work T_1 and critical-path length T_∞ being executed by the WSSPD algorithm with P processors in a dedicated environment. Then, the expected number of memory fences issued during the computation's execution is at most $O(PT_\infty)$.*

Proof. Lemma 6.9 bounds the number of memory barriers issued in terms of the number of idle iterations. Taking into account Lemma 6.15, we conclude that the expected number of memory barriers issued during a computation's execution under the WSSPD algorithm is at most $O(PT_\infty)$. ■

6.3 Some extensions to the CFWSS scheduler

In this section, we show how the CFWSS and CFLFWS algorithms can be extended to address known limitations of WS schedulers. The extensions we present can be seen as simple *proofs-of-concept*, for which reason we do not theoretically nor empirically analyze their performance. We will focus on the specific problem of scheduling memory bound computations, and show how to address the known limitations of WS for this class of computations, which, as mentioned in Section 2.4.1, are related with the lack of locality awareness of WS's load balancer. We divide the extensions we propose as receiver initiated, sender initiated, or, mixed (where both sender and receiver-initiated schemes are used).

6.3.1 Receiver initiated extensions

As already mentioned, receiver initiated load balancing schemes are most appropriate when executing computations with high parallelism. Thus, the receiver initiated extensions we now propose aim to address memory bound computations with high degrees of parallelism. In order to increase WS's locality awareness, we present two strategies that can be used by both CFLFWS and CFWSS.

This first scheme was introduced by Suksompong *et al* in [Suk+16], and attempts to ensure that processors work on the same data during the computation's execution, thus taking advantage of the temporal locality principle [PH12]. To that end, each time a thief steals a victim, it lets the victim know who stole from its deque. In order to implement this mechanism, each processor has an additional flag, the $lastThief$ flag, whose value corresponds to the last thief who successfully stole the processor. A sample algorithm using this strategy is depicted in Algorithm 10, where θ_r represents the probability that a processor attempts to make a *stealback*.

In that same study, the authors assumed that the computations were represented as trees instead of dags, which, as mentioned in Section 2.4.1 and also in that study, is a very specific subtype of dags. For such computations, the authors proved that the expected runtime when using the proposed algorithm is $\frac{T_1}{P} + O(T_\infty.P)$, which is far from the bounds[3] we could easily achieve by extending CFLFWS or CFWSS to use this strategy, while also considering generic dag computations, rather than tree-shaped ones.

[3]Recall that the expected runtime bound for the CFLFWS and also for the CFWSS is at most $O\left(\frac{T_1}{P} + \frac{T_\infty}{1-\theta_r}\right)$.

```
procedure WorkMigration($\theta_r$)
    $choice \leftarrow UniformlyRandomNumber([0;1])$
    if $choice > \theta_r$ then
        $victim \leftarrow UniformlyRandomProcessor()$
        $assigned \leftarrow victim.deque$.popTop()
        if ValidNode($assigned$) then
            $victim.lastThief \leftarrow self$
        end if
    else
        $victim \leftarrow self.lastThief$
        if $victim =$ NONE then
            $victim \leftarrow UniformlyRandomProcessor()$ ▹ The processor was never stolen.
        end if
        $assigned \leftarrow victim.deque$.popTop()
    end if
end procedure
```

Algorithm 10: The Stealback load balancing mechanism on CFLWFS.

Another possibility for coping with the locality obliviousness of WS schedulers would be to ensure that thieves prioritize their neighbors, when choosing their victims. As one might conclude, this class of extensions prioritizes the spacial locality principle, rather than the temporal locality one. For certain memory bound computations, biasing steal attempts in order to target close neighbors can drastically improve the scheduler's performance. First, note that the steal attempts would incur in less expensive communication overheads, as the thief would be probably closer to its victim. Second, for numerous data intensive computations, the thief would, most probably, start to work on data that was already close to it, thus avoiding extra communication costs. In Algorithm 11, we give a concrete example of how this strategy could be implemented, assuming a shared memory environment. As in for the previous extension, it would be straightforward to obtain expected runtime bounds for this algorithm, depending on the value of θ_r: $O\left(\frac{T_1}{P} + \frac{T_\infty}{1-\theta_r}\right)$.

```
procedure WorkMigration($\theta_r$)
    $choice \leftarrow UniformlyRandomNumber([0;1])$
    if $choice > \theta_r$ then
        $victim \leftarrow UniformlyRandomProcessor()$
        $assigned \leftarrow victim.deque$.popTop()
    else
        if $choice > (\theta_r/2)$ then
            $victim \leftarrow UniformlyRandomNeighbor(Neighborhood_{L1})$
            $assigned \leftarrow victim.deque$.popTop()
        else if $choice > (\theta_r/4)$ then
            $victim \leftarrow UniformlyRandomNeighbor(Neighborhood_{L2})$
            $assigned \leftarrow victim.deque$.popTop()
        else
            $victim \leftarrow UniformlyRandomNeighbor(Neighborhood_{L3})$
            $assigned \leftarrow victim.deque$.popTop()
        end if
    end if
end procedure
```

Algorithm 11: A custom load balancer that attempts thieves to steal work from their closest neighbors.

6.3.2 Sender initiated extensions

Now, we propose a very simple sender initiated load balancing strategy that allows to ensure good data locality when scheduling computations with few, or unbalanced parallelism. Because the scheme relies on work sharing mechanisms, it can only be applied to the CFWSS' algorithm, since it is the only (practical) scheduler supporting work spreading.

The mechanism presented in Algorithm 12 is based in the same idea as the one given in [Suk+16], and used on the first extension we proposed. As in the previous scheme, each time a thief steals a victim, it lets the victim know who stole from its deque. To that end, each processor owns a $lastThief$ flag, whose value corresponds to the its last thief. However, as, in this case, we aim towards increasing the data locality for computations with few, or unbalanced parallelism, simply attempting to steal work back from a thief is certainly not the best approach, due to the fact that it corresponds to a receiver initiated load balancing policy. In fact, for scenarios with such type of parallelism, it would be more likely that the node the thief stole did not generate much work, implying it was possibly idle again. For that reason, we have to take a fundamentally different approach for increasing locality awareness for these scenarios. Concretely, instead of making the processor steal work back from its thief, we, instead, make it attempt spreading work to the thief. Algorithm 12 depicts our scheme, where θ_s represents the probability that a processor attempts to make a *stealback*.

As in the previous cases, obtaining expected runtime bounds for this extension would be trivial thanks to the flexibility of the CFWSS algorithm. In fact, if no custom load balancing receiver initiated strategy was used (*i.e.* if $\theta_r = 0$), then we could easily prove that the expected runtime of a computation run by this scheduler would be at most $O\left(\frac{T_1}{P} + T_\infty\right)$, which, taking into account the results presented in Section 4.2.1, is a constant factor away from being optimal.

```
procedure SPREAD($pointer$, $\theta_s$, $spreading\_heuristic$)
    if $self$.spreading then
        $choice \leftarrow UniformlyRandomNumber([0;1])$
        if $choice \geq \theta_s$ then
            $pointer.shared \leftarrow$ TRUE
            $donee \leftarrow UniformlyRandomProcessor()$
            if $donee.state =$ IDLE then
                $donee.state \leftarrow pointer$
                $self$.update($spreading\_heuristic[1]$)
            else
                $pointer.shared \leftarrow$ FALSE
                $self$.update($spreading\_heuristic[2]$)
            end if
        else
            $self$.update($spreading\_heuristic[0]$)
            $donee \leftarrow self.lastThief$                ▷ Attempts donating to its last thief.
            if $donee \neq$ NONE then
                if $donee.state =$ IDLE then
                    $donee.state \leftarrow pointer$
                else
                    $pointer.shared \leftarrow$ FALSE
                end if
            end if
        end if
    end if
end procedure
```

Algorithm 12: A spread-back strategy for sharing work.

6.3.3 Mixed extensions

One of the most notable strategies for scheduling memory bound computations was proposed by Acar *et al* in [Aca+02]. To ensure better data locality, the scheduler attempts to ensure that processors work on data in which they have already worked on, helping to ensure that the processors reuse the data that has already been fetch onto their caches. To that end, each processor keeps track of the threads that have affinity with the data on which it had already worked upon, using a mailbox, which, in practice, is implemented as a queue. Whenever some processor generates work that has affinity to another one, it pushes the work onto its deque, and, additionally also places a pointer to that work in the other processor's mailbox. The empirical analysis of this policy validated its efficiency, showing it can achieve substantial performance gains for the execution of memory bound computations. However, the expected time for the execution of computations using this strategy is unknown.

Now, we show how we could implement this whole mechanism while still guaranteeing good performance when executing any kind of computations (including non memory bound ones). In order to guarantee that adding work to a processor's mailbox takes constant time, we assume that each mailbox is implemented as a queue whose implementation is given in Algorithm 13. Each queue object contains three variables:

entries — The array of the queue.

bottom — An index pointing to the position bellow the bottommost entry of the queue.

top — An index pointing to the topmost entry of the queue.

Furthermore, each queue object supports two methods:

enqueue — Tries to place an item into the bottom of the queue.

dequeue — Tries to remove an item from the top of the queue.

The only claims we make are:

- The implementation of the queue given in Algorithm 13 is constant-time.
- A *dequeue* invocation either returns EMPTY, or some valid entry of the queue.

```
1: procedure ENQUEUE($pointer$)
2:     $lBot \leftarrow self.bottom$
3:     $self.entries[lbot] \leftarrow pointer$
4:     $self.bottom \leftarrow lbot + 1$
5: end procedure

6: procedure DEQUEUE
7:     $ltop \leftarrow self.bottom$
8:     if $self.bottom = ltop$ then
9:         return EMPTY
10:    end if
11:    $ret \leftarrow self.entries[ltop]$
12:    $self.top \leftarrow ltop + 1$
13:    return $ret$
14: end procedure
```

Algorithm 13: The two queue's methods.

As one might note, the queue may not behave correctly when the *enqueue* method is invoked concurrently[4]. That is indeed true. However, we remark the fact that it is far

[4]Regarding the *dequeue* method, note that only the owner of each queue may invoke the method, implying it is not invoked concurrently.

more preferable to, from time to time, fail sending work to some processor's mailbox, than incurring in extremely expensive synchronization costs that, otherwise, would have to be used, in order to ensure the correct operation of the queue's implementation. As for the original scheme, presented by Acar *et al*, we have to ensure that each node is only executed once, and, for that reason, synchronization is required. Fortunately, such synchronization perfectly fits the synchronization mechanisms used in the CFWSS scheduler: it suffices setting to TRUE the *shared* flag of the entry containing the node that was to be offered to the other processor.

In Algorithm 14, we present how this mechanism could be implemented on CFWSS. We omit the full algorithm for the sake of succinctness. Yet, we claim that, for a given value of θ_r, where, as always $\theta_r \in [0;1[$, the expected runtime of computations under this extended scheduler (that is based on CFWSS) is at most $O\left(\frac{T_1}{P} + \frac{T_\infty}{1-\theta_r}\right)$.

```
procedure SCHEDULER($\theta_r$, $\theta_s$, $heuristic$)
    while $computation\ not\ terminated$ do
        if ValidNode($assigned$) then
            ...
            if length($enabled$) = 2 then
                $pointer \leftarrow self$.StoreNewNode($enabled[1]$)
                $affinityOfNode \leftarrow getAffinity(enabled[1])$
                if $affinityOfNode \neq$ NONE then
                    $pointer.shared \leftarrow$ TRUE
                    $affinityOfNode.mailbox$.enqueue($pointer$)
                else
                    $self$.spread($pointer$, $\theta_s$, $heuristic[0]$)
                end if
                $self.deque$.pushBottom($pointer$)
            end if
            ...
        else
            $self$.WorkMigration($\theta_r$, $heuristic[1]$)
        end if
    end while
end procedure

procedure WORKMIGRATION($\theta_r$)
    $choice \leftarrow UniformlyRandomNumber([0;1])$
    if $choice > \theta_r$ then
        $victim \leftarrow UniformlyRandomProcessor()$
        $pointer \leftarrow victim.deque$.popTop()
    else
        $pointer \leftarrow self.mailbox$.dequeue()
    end if
    if $pointer \neq$ EMPTY and $pointer \neq$ ABORT then
        assign($pointer$)
    end if
end procedure
```

Algorithm 14: The mailbox strategy implemented on the CFWSS.

6.4 Future Work

As one might have noticed, we have skipped non trivial proofs, which, as we mentioned, were out of the scope of this thesis. Concretely, we leave the following proofs as future work:

1. The proof of correction of the CFWSS algorithm.
2. The proof of correction of the semi-private-deque.
3. The proof that the semi-private-deque meets the SPDR Semantics on any good set of invocations, as defined in Section 6.2.1.
4. A full version of the proof of Lemma 6.9.

Additionally, we also leave the empirical analysis of all these algorithms as future work.

Chapter 7

Conclusions and Future Work

We now outline the principal contributions made in this thesis, discussing their advantages, disadvantages and possible directions for future work. The contributions are presented in the same order as they have been introduced through the thesis.

The first main contribution we have made is related with the flexibility of the WS algorithm. We showed that WS can be easily extended to make use of custom load balancing policies, that, for various classes of workloads, can greatly boost the scheduler's performance. Furthermore, our analysis of the canonical version of the Flexible Work Stealing algorithm shows a great potential in overcoming some of the known limitations of state-of-the-art WS algorithms, while still ensuring nearly optimal expected runtime bounds. In Chapter 6 we have even shown how to address the famous limitation of WS algorithms related with the locality obliviousness of the scheduler's load balancer. Being the focus of this thesis on the formal study of the proposed algorithms, we have not yet confirmed if, in practice, our approach can effectively be used to overcome the limitations of WS while still maintaining high performances for the general setting. We are looking forward to empirically evaluate the impact of using customized load balancers when attempting to overcome WS's limitations, and, hopefully, validate our hypothesis.

Second, we have proposed a theoretical scheduling algorithm, CFSAWSS, that combines sender and receiver initiated load balancing schemes, in an attempt to overcome the limitations of WS schedulers for scenarios with few or unbalanced parallelism. One of the key features of CFSAWSS is that it supports both sender and receiver initiated custom load balancing mechanisms, providing even more opportunities in overcoming the known limitations of WS algorithms. We proved that this algorithm is provably efficient, also with nearly optimal expected runtime bounds, and then delved into a careful analysis of the performance of its load balancer. The analysis we made assumed a synchronous environment, which allowed us to properly compare the performances of CFSAWSS's and WS's load balancers. In order to be as impartial as possible when comparing the performance of both algorithms, we have removed all the flexibility of CFSAWSS, meaning that, alike for WS, processors never skip load balancing opportunities. The comparison showed that our algorithm's load balancer achieves gains of at least 40% if the ratio of idle processors is 60% or more. In other words, we have shown that if there is a considerable ratio of processors that are idle (60% or more), the performance of our load balancer is expected to be at least 40% better than the performance of WS's load balancer. Nevertheless, sender initiated load balancers, such as the one CFSAWSS uses, are known to incur in high communication overheads, that often hamper the scheduler's performance. To overcome this limitation, we have introduced the Tit-for-tat heuristic that attempts to mitigate all these expenses by amortizing them with the communication costs associated with Work Stealing, which are known to be low. We have then shown that, even when using this heuristic, if the ratio of busy processors is at most 20%, the algorithm's load balancer is expected to be at least 40% more effective than Work Stealing's.

After this comparison, we obtained lower bounds on the amount of work that was load balanced by CFSAWSS, depending on the ratio of idle processors, and, also depending on the degrees of flexibility used by the scheduler. Then, we claimed that, for both the Greedy and the Tit-for-tat heuristics, if the degrees of flexibility are null, the amount of work attached to any processor is expected to decrease, or, if it is the case, to continue being bounded by a constant. In particular, we have claimed that, for any computation being executed by some arbitrarily fixed number of processors, using the Greedy heuristic, and having both degrees of flexibility set to null (*i.e.* having both θ_r and θ_s set to 0), the expected amount of work attached to each processor is expected to decrease, as long as $\alpha \in [0.7375;1[$ (*i.e.* as long as the ratio of idle processors is at least 0.7375, up to, but, excluding, 1). In the same way, we also claimed that, for any computation being executed by an arbitrary, but fixed, number of processors, using the Tit-for-tat heuristic, and having both θ_r and θ_s set to 0, the expected amount of work attached to each processor is expected to decline, as long as $\alpha \in [0.8675;1[$. Unfortunately, we did not manage to prove the veracity of either of these claims, despite all the evidences that suggest it indeed holds. We have even used *Wolfram alpha*[1] whose solutions confirm our allegations. However, as the tool did not show the steps it took to compute such solution, we cannot in rigor considered proved. In case any of these claims is proved, our algorithm will have been, to the best of our knowledge, the first provably efficient one that, depending on the ratio of idle processors, can tend towards stability. The proof of these claims is then left as an open problem.

Despite all these promising results, CFSAWSS is only a theoretical algorithm. For that reason, we have also developed CFWSS, an asynchronous version of CFSAWSS that intends to be used in practice. This scheduler then combines sender and receiver initiated

[1] Available at *https://www.wolframalpha.com/*.

load balancing mechanisms, and can also be extended to support custom strategies for any of these policies. As for CFSAWSS, we have proved that the expected runtime of computations under CFWSS is nearly optimal. Nevertheless, to implement the spreading mechanism we have added a few extra overheads when processors generate work. Unfortunately, even if these overheads are very small, they scale with the total amount of work, implying the overall performance of the scheduler may be severely harmed. For that reason, we have presented the Tit-for-tat heuristic that, pretty much, amortizes the costs of spreading with the costs of stealing. Moreover, we designed the spreading mechanism in such a way that it does not incur in expensive operations when attempting to spread work. Due to all these reasons, we expect CFWSS to be a practical scheduler. However, as we have not yet implemented CFWSS, we did not empirically validate our claim, that mixing both receiver and sender initiated strategies (in the way we have proposed it) can drastically improve the performance of schedulers. Therefore, the experimental confirmation of this claim is left as future work. In addition, we also leave the proofs of CFWSS's correction as future work.

The last main contribution we make in this thesis is a variant of LFWS, the WSSPD algorithm, that avoids most of the overheads caused by the use of memory fences, that are very expensive in modern relaxed-memory architectures, when processors access their own work pools. Regarding its efficiency, we prove that, for any computation with work T_1, critical-path length T_∞ and for any number of processors P, the expected time that WSSPD takes to execute the computation is at most $O\left(\frac{T_1}{P} + T_\infty\right)$, which is a constant factor away from being optimal.

In state-of-the-art WS algorithms, each processor owns a concurrent deque that uses to keep track of its attached work. Unfortunately, as mentioned in [Aca+13b], even when processors are operating locally on their own deques, costly memory fences are required in order to ensure correction (avoiding, for example, that a thief and the deque's owner both steal the same node). Even more unfortunate is the result proved in [Att+11], which implies that it is impossible to remove all these synchronization costs in the access to concurrent deques, while ensuring correctness. For this reason, the algorithm we propose cannot avoid all the expenses incurred by the use of these synchronization mechanisms (in particular, of memory fences). It can, however, avoid most of the unnecessary memory fences when processors are working locally on their deques, during the execution of computations. For instance, while in modern WS algorithms the total overhead incurred by the use of memory fences can proportionally grow with the size of the computation (*i.e.* the total overhead can be $O(T_1)$), for our proposal, it only grows with the computation's span, and, with the number of processors participating in its execution. We show that, for the WSSPD algorithm, the total expected overhead associated with these synchronization mechanisms is at most $O(T_\infty.P)$. We remark the fact that there are numerous types of computations whose amount of work grows exponentially with its span (*e.g.* Optimization problems, whose corresponding decision problems are NP-complete, parallel-fors, parallel-scans, etc). In other words, there are numerous classes of computations where $T_1 = O\left(2^{T_\infty}\right)$. For such cases, the total expected overheads associated with the memory fences issued by our algorithm are, basically, exponentially smaller when compared to the ones for modern work stealing algorithms.

The design of a practical and efficient implementation of the WSSPD algorithm does not look to be an easy task, for which reason we consider it to be a very interesting direction for future work. Another fundamental implication of this result is that, for sequential executions, our algorithm incurs in absolutely no synchronization overheads. When an efficient implementation of this algorithm is developed, it will then be particularly interesting to study the serial execution time of parallel programs using such implementation, comparing them against their sequential versions (*i.e.* non parallel versions), without using the parallel environment. If the execution times of both versions turn out to be very similar, it would be interesting to see current sequential execution environments to become parallel by default.

Finally, incorporating the scheme used by this algorithm into CFWSS while, at the same time, maintaining good bounds on the total expected overheads caused by memory fences, seems a very challenging problem. We leave it for future work, but, yet, remark that the Tit-for-tat heuristic will hopefully allow to amortize the costs of the memory fences associated to work spreading, with the ones incurred by work stealing.

Bibliography

[PH12] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2012. ISBN: 978-0-12-374750-1. URL: http://www.elsevierdirect.com/product.jsp?isbn=9780123747501.

[Bru09] J. Brutlag. "Speed matters for Google web search". In: *Google. June* (2009).

[Tee+13] J. Teevan, K. Collins-Thompson, R. W. White, S. T. Dumais, and Y. Kim. "Slow search: Information retrieval without time constraints". In: *Proceedings of the Symposium on Human-Computer Interaction and Information Retrieval*. ACM. 2013, p. 1.

[DE98] L. Dagum and R. Enon. "OpenMP: an industry standard API for shared-memory programming". In: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55.

[GNU16] GNU. *Automatic parallelization in GCC*. [Online; accessed 19-January-2016]. 2016. URL: \url{https://gcc.gnu.org/wiki/AutoParInGCC}.

[BL99] R. D. Blumofe and C. E. Leiserson. "Scheduling Multithreaded Computations by Work Stealing". In: *J. ACM* 46.5 (1999), pp. 720–748. DOI: 10.1145/324133.324234. URL: http://doi.acm.org/10.1145/324133.324234.

[Aro+01] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. "Thread Scheduling for Multiprogrammed Multiprocessors". In: *Theory Comput. Syst.* 34.2 (2001), pp. 115–144. DOI: 10.1007/s00224-001-0004-z. URL: http://dx.doi.org/10.1007/s00224-001-0004-z.

[HS02a] D. Hendler and N. Shavit. "Non-blocking steal-half work queues". In: *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing, PODC 2002, Monterey, California, USA, July 21-24, 2002.* 2002, pp. 280–289. DOI: 10.1145/571825.571876. URL: http://doi.acm.org/10.1145/571825.571876.

[Aca+02] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. "The Data Locality of Work Stealing". In: *Theory Comput. Syst.* 35.3 (2002), pp. 321–347. DOI: 10.1007/s00224-002-1057-3. URL: http://dx.doi.org/10.1007/s00224-002-1057-3.

[Eag+85] D. L. Eager, E. D. Lazowska, and J. Zahorjan. “A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing”. In: *SIGMETRICS*. 1985, pp. 1–3. DOI: 10.1145/317795.317802. URL: http://doi.acm.org/10.1145/317795.317802.

[Aca+13b] U. A. Acar, A. Charguéraud, and M. Rainey. “Scheduling parallel programs by work stealing with private deques”. In: *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ’13, Shenzhen, China, February 23-27, 2013*. 2013, pp. 219–228. DOI: 10.1145/2442516.2442538. URL: http://doi.acm.org/10.1145/2442516.2442538.

[Fax09] K.-F. Faxén. “Wool-a work stealing library”. In: *ACM SIGARCH Computer Architecture News* 36.5 (2009), pp. 93–100.

[BP99] R. D. Blumofe and D. Papadopoulos. *Hood: A user-level threads library for multiprogrammed multiprocessors*. Tech. rep. Citeseer, 1999.

[KV07] A. Kukanov and M. J. Voss. “The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks.” In: *Intel Technology Journal* 11.4 (2007).

[Aca+16] U. A. Acar, A. Charguéraud, and M. Rainey. *PASL: Parallel Algorithm Scheduling Library*. [Online; accessed 21-January-2016]. 2016. URL: \url{http://www.chargueraud.org/softs/pasl/}.

[Blu+96] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. “Cilk: An Efficient Multithreaded Runtime System”. In: *J. Parallel Distrib. Comput.* 37.1 (1996), pp. 55–69. DOI: 10.1006/jpdc.1996.0107. URL: http://dx.doi.org/10.1006/jpdc.1996.0107.

[Lei09] C. E. Leiserson. “The Cilk++ concurrency platform”. In: *Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009*. 2009, pp. 522–527. DOI: 10.1145/1629911.1630048. URL: http://doi.acm.org/10.1145/1629911.1630048.

[LS94] L. M. B. Lopes and F. M. A. Silva. “Scheduling Algorithms Performance with the pSystem Parallel Programming Environment”. In: *PARLE ’94: Parallel Architectures and Languages Europe, 6th International PARLE Conference, Athens, Greece, July 4-8, 1994, Proceedings*. 1994, pp. 827–830. DOI: 10.1007/3-540-58184-7_167. URL: http://dx.doi.org/10.1007/3-540-58184-7_167.

[Sil+99] F. M. A. Silva, H. Paulino, and L. M. B. Lopes. “di_pSystem: A Parallel Programming System for Distributed Memory Architectures”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 6th European PVM/MPI Users’ Group Meeting, Barcelona, Spain, September 26-29, 1999, Proceedings*. Ed. by J. Dongarra, E. Luque, and T. Margalef. Vol. 1697. Lecture Notes in Computer Science. Springer, 1999, pp. 525–532. DOI: 10.1007/3-540-48158-3_65. URL: http://dx.doi.org/10.1007/3-540-48158-3_65.

[Aro+98] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. “Thread Scheduling for Multiprogrammed Multiprocessors”. In: *SPAA*. 1998, pp. 119–129. DOI: 10.1145/277651.277678. URL: http://doi.acm.org/10.1145/277651.277678.

[Blu+99] R. D. Blumofe, C. G. Plaxton, and S. Ray. “Verification of a concurrent deque implementation”. In: *University of Texas at Austin, Austin, TX* (1999).

[Tch+10] M. Tchiboukdjian, N. Gast, D. Trystram, J. Roch, and J. Bernard. “A Tighter Analysis of Work Stealing”. In: *Algorithms and Computation - 21st International Symposium, ISAAC 2010, Jeju Island, Korea, December 15-17, 2010, Proceedings, Part II*. 2010, pp. 291–302. DOI: 10.1007/978-3-642-17514-5_25. URL: http://dx.doi.org/10.1007/978-3-642-17514-5_25.

[MA16] S. K. Muller and U. A. Acar. “Latency-Hiding Work Stealing: Scheduling Interacting Parallel Computations with Work Stealing”. In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11-13, 2016*. 2016, pp. 71–82. DOI: 10.1145/2935764.2935793. URL: http://doi.acm.org/10.1145/2935764.2935793.

[HS02b] D. Hendler and N. Shavit. “Work dealing”. In: *SPAA*. 2002, pp. 164–172. DOI: 10.1145/564870.564900. URL: http://doi.acm.org/10.1145/564870.564900.

[QW10] J. Quintin and F. Wagner. “Hierarchical Work-Stealing”. In: *Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part I*. 2010, pp. 217–229. DOI: 10.1007/978-3-642-15277-1_21. URL: http://dx.doi.org/10.1007/978-3-642-15277-1_21.

[Suk+16] W. Suksompong, C. E. Leiserson, and T. B. Schardl. “On the efficiency of localized work stealing”. In: *Inf. Process. Lett.* 116.2 (2016), pp. 100–106. DOI: 10.1016/j.ipl.2015.10.002. URL: http://dx.doi.org/10.1016/j.ipl.2015.10.002.

[Guo+10] Y. Guo, J. Zhao, V. Cavé, and V. Sarkar. “SLAW: A scalable locality-aware adaptive work-stealing scheduler”. In: *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*. 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470425. URL: http://dx.doi.org/10.1109/IPDPS.2010.5470425.

[Che+11] Q. Chen, Z. Huang, M. Guo, and J. Zhou. “CAB: Cache Aware Bi-tier Task-Stealing in Multi-socket Multi-core Architecture”. In: *International Conference on Parallel Processing, ICPP 2011, Taipei, Taiwan, September 13-16, 2011*. 2011, pp. 722–732. DOI: 10.1109/ICPP.2011.32. URL: http://dx.doi.org/10.1109/ICPP.2011.32.

[Che+12] Q. Chen, M. Guo, and Z. Huang. "CATS: cache aware task-stealing based on online profiling in multi-socket multi-core architectures". In: *International Conference on Supercomputing, ICS'12, Venice, Italy, June 25-29, 2012*. 2012, pp. 163–172. DOI: 10.1145/2304576.2304599. URL: http://doi.acm.org/10.1145/2304576.2304599.

[Aca+15] U. A. Acar, G. E. Blelloch, M. Fluet, S. K. Muller, and R. Raghunathan. "Coupling Memory and Computation for Locality Management". In: *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*. 2015, pp. 1–14. DOI: 10.4230/LIPIcs.SNAPL.2015.1. URL: http://dx.doi.org/10.4230/LIPIcs.SNAPL.2015.1.

[Mul+] S. Muller, U. A. Acar, and R. Harper. *Responsive Parallel Computation*. https://www.cs.cmu.edu/~rwh/papers/resppar/draft.pdf. Accessed: 2016-08-27.

[Agr+06] K. Agrawal, Y. He, and C. E. Leiserson. "An Empirical Evaluation ofWork Stealing with Parallelism Feedback". In: *26th IEEE International Conference on Distributed Computing Systems (ICDCS 2006), 4-7 July 2006, Lisboa, Portugal*. 2006, p. 19. DOI: 10.1109/ICDCS.2006.14. URL: http://dx.doi.org/10.1109/ICDCS.2006.14.

[Agr+07] K. Agrawal, Y. He, W. Hsu, and C. E. Leiserson. "Adaptive Scheduling with Parallelism Feedback". In: *21th International Parallel and Distributed Processing Symposium (IPDPS 2007), Proceedings, 26-30 March 2007, Long Beach, California, USA*. 2007, pp. 1–7. DOI: 10.1109/IPDPS.2007.370496. URL: http://dx.doi.org/10.1109/IPDPS.2007.370496.

[Agr+08] K. Agrawal, C. E. Leiserson, Y. He, and W. Hsu. "Adaptive work-stealing with parallelism feedback". In: *ACM Trans. Comput. Syst.* 26.3 (2008). DOI: 10.1145/1394441.1394443. URL: http://doi.acm.org/10.1145/1394441.1394443.

[Sun+11] H. Sun, Y. Cao, and W. Hsu. "Efficient Adaptive Scheduling of Multiprocessors with Stable Parallelism Feedback". In: *IEEE Trans. Parallel Distrib. Syst.* 22.4 (2011), pp. 594–607. DOI: 10.1109/TPDS.2010.121. URL: http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.121.

[CL05] D. Chase and Y. Lev. "Dynamic circular work-stealing deque". In: *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA*. 2005, pp. 21–28. DOI: 10.1145/1073970.1073974. URL: http://doi.acm.org/10.1145/1073970.1073974.

[Hen+06] D. Hendler, Y. Lev, M. Moir, and N. Shavit. “A dynamic-sized nonblocking work stealing deque”. In: *Distributed Computing* 18.3 (2006), pp. 189–207. DOI: 10.1007/s00446-005-0144-5. URL: http://dx.doi.org/10.1007/s00446-005-0144-5.

[Mic+09] M. M. Michael, M. T. Vechev, and V. A. Saraswat. “Idempotent work stealing”. In: *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*. 2009, pp. 45–54. DOI: 10.1145/1504176.1504186. URL: http://doi.acm.org/10.1145/1504176.1504186.

[MA14] A. Morrison and Y. Afek. “Fence-free work stealing on bounded TSO processors”. In: *Architectural Support for Programming Languages and Operating Systems, ASPLOS ’14, Salt Lake City, UT, USA, March 1-5, 2014*. 2014, pp. 413–426. DOI: 10.1145/2541940.2541987. URL: http://doi.acm.org/10.1145/2541940.2541987.

[Att+11] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. T. Vechev. “Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated”. In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. 2011, pp. 487–498. DOI: 10.1145/1926385.1926442. URL: http://doi.acm.org/10.1145/1926385.1926442.

[Tza12] A. Tzannes. “Enhancing productivity and performance portability of general-purpose parallel programming”. In: (2012).

[Aca+13a] U. A. Acar, A. Charguéraud, S. Muller, and M. Rainey. *Atomic Read-Modify-Write Operations are Unnecessary for Shared-Memory Work Stealing*. Research Report. Sept. 2013. URL: https://hal.inria.fr/hal-00910130.

[Eag+86] D. L. Eager, E. D. Lazowska, and J. Zahorjan. “Adaptive Load Sharing in Homogeneous Distributed Systems”. In: *IEEE Trans. Software Eng.* 12.5 (1986), pp. 662–675. DOI: 10.1109/TSE.1986.6312961. URL: http://dx.doi.org/10.1109/TSE.1986.6312961.

[Ber+03] P. Berenbrink, T. Friedetzky, and L. A. Goldberg. “The Natural Work-Stealing Algorithm is Stable”. In: *SIAM J. Comput.* 32.5 (2003), pp. 1260–1279. DOI: 10.1137/S0097539701399551. URL: http://dx.doi.org/10.1137/S0097539701399551.

[LM93] R. Lüling and B. Monien. “A Dynamic Distributed Load Balancing Algorithm with Provable Good Performance”. In: *SPAA*. 1993, pp. 164–172. DOI: 10.1145/165231.165252. URL: http://doi.acm.org/10.1145/165231.165252.

[Mit98] M. Mitzenmacher. “Analyses of Load Stealing Models Based on Differential Equations”. In: *SPAA*. 1998, pp. 212–221. DOI: 10.1145/277651.277687. URL: http://doi.acm.org/10.1145/277651.277687.

[Bye+04] J. W. Byers, J. Considine, and M. Mitzenmacher. “Geometric generalizations of the power of two choices”. In: *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain*. 2004, pp. 54–63. DOI: 10.1145/1007912.1007921. URL: http://doi.acm.org/10.1145/1007912.1007921.

[Mit+02] M. Mitzenmacher, B. Prabhakar, and D. Shah. “Load Balancing with Memory”. In: *43rd Symposium on Foundations of Computer Science (FOCS 2002), 16-19 November 2002, Vancouver, BC, Canada, Proceedings*. 2002, pp. 799–808. DOI: 10.1109/SFCS.2002.1182005. URL: http://dx.doi.org/10.1109/SFCS.2002.1182005.

[Dri+02] E. Drinea, A. M. Frieze, and M. Mitzenmacher. “Balls and bins models with feedback”. In: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 6-8, 2002, San Francisco, CA, USA*. 2002, pp. 308–315. URL: http://dl.acm.org/citation.cfm?id=545381.545422.

[Alb+01] S. Albers, M. Charikar, and M. Mitzenmacher. “Delayed Information and Action in On-Line Algorithms”. In: *Inf. Comput.* 170.2 (2001), pp. 135–152. DOI: 10.1006/inco.2001.3057. URL: http://dx.doi.org/10.1006/inco.2001.3057.

[Mit01] M. Mitzenmacher. “The Power of Two Choices in Randomized Load Balancing”. In: *IEEE Trans. Parallel Distrib. Syst.* 12.10 (2001), pp. 1094–1104. DOI: 10.1109/71.963420. URL: http://doi.ieeecomputersociety.org/10.1109/71.963420.

[Mit99] M. Mitzenmacher. “On the Analysis of Randomized Load Balancing Schemes”. In: *Theory Comput. Syst.* 32.3 (1999), pp. 361–386. DOI: 10.1007/s002240000122. URL: http://dx.doi.org/10.1007/s002240000122.

[Adl+98] M. Adler, S. Chakrabarti, M. Mitzenmacher, and L. E. Rasmussen. “Parallel randomized load balancing”. In: *Random Struct. Algorithms* 13.2 (1998), pp. 159–188. DOI: 10.1002/(SICI)1098-2418(199809)13:2<159::AID-RSA3>3.0.CO;2-Q. URL: http://dx.doi.org/10.1002/(SICI)1098-2418(199809)13:2<159::AID-RSA3>3.0.CO;2-Q.

[Mit97] M. Mitzenmacher. “On the Analysis of Randomized Load Balancing Schemes”. In: *SPAA*. 1997, pp. 292–301. DOI: 10.1145/258492.258521. URL: http://doi.acm.org/10.1145/258492.258521.

[Aza+99] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. “Balanced Allocations”. In: *SIAM J. Comput.* 29.1 (1999), pp. 180–200. DOI: 10.1137/S0097539795288490. URL: http://dx.doi.org/10.1137/S0097539795288490.

[GM01] E. Gafni and M. Mitzenmacher. "Analysis of Timing-Based Mutual Exclusion with Random Times". In: *SIAM J. Comput.* 31.3 (2001), pp. 816–837. DOI: 10.1137/S0097539799364912. URL: http://dx.doi.org/10.1137/S0097539799364912.

[BL98] R. D. Blumofe and C. E. Leiserson. "Space-Efficient Scheduling of Multithreaded Computations". In: *SIAM J. Comput.* 27.1 (1998), pp. 202–229. DOI: 10.1137/S0097539793259471. URL: http://dx.doi.org/10.1137/S0097539793259471.

[Sto+03] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. "Chord: a scalable peer-to-peer lookup protocol for internet applications". In: *IEEE/ACM Trans. Netw.* 11.1 (2003), pp. 17–32. DOI: 10.1109/TNET.2002.808407. URL: http://dx.doi.org/10.1109/TNET.2002.808407.

[God08] B. Godfrey. "Balls and bins with structure: balanced allocations on hypergraphs". In: *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*. 2008, pp. 511–517. URL: http://dl.acm.org/citation.cfm?id=1347082.1347138.

[He+06] Y. He, W. Hsu, and C. E. Leiserson. "Provably Efficient Two-Level Adaptive Scheduling". In: *Job Scheduling Strategies for Parallel Processing, 12th International Workshop, JSSPP 2006, Saint-Malo, France, June 26, 2006, Revised Selected Papers*. 2006, pp. 1–32. DOI: 10.1007/978-3-540-71035-6_1. URL: http://dx.doi.org/10.1007/978-3-540-71035-6_1.

[Ull75] J. D. Ullman. "NP-complete scheduling problems". In: *Journal of Computer and System sciences* 10.3 (1975), pp. 384–393.

[Moh+91] E. Mohr, D. A. Kranz, and R. H. H. Jr. "Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs". In: *IEEE Trans. Parallel Distrib. Syst.* 2.3 (1991), pp. 264–280. DOI: 10.1109/71.86103. URL: http://doi.ieeecomputersociety.org/10.1109/71.86103.

[Gol+96] S. C. Goldstein, K. E. Schauser, and D. E. Culler. "Lazy Threads: Implementing a Fast Parallel Call". In: *J. Parallel Distrib. Comput.* 37.1 (1996), pp. 5–20. DOI: 10.1006/jpdc.1996.0104. URL: http://dx.doi.org/10.1006/jpdc.1996.0104.

[AS92] N. Alon and J. Spencer. *The Probabilistic Method*. John Wiley, 1992. ISBN: 0-471-53588-5.

2016

Scheduling computations

Guilherme Rito