Universidad Nacional Autonóma de México
Posgrado en Ciencias e Ingeniería De La Computación
Instituto en Investigaciones en Matemáticas Aplicadas y En Sistemas
Teoría de la Computación

A Study Of Concurrent Data Structures With Relaxed Semantics

TESIS

QUE PARA OPTAR POR EL GRADO DE:

Doctor en Ciencia E Ingeniería en Computación

PRESENTA
Miguel Angel Piña Avelino

TUTOR
Dr. Armando Casteñeda Rojano
Instituto de Matemáticas

COMITÉ TUTOR
Dr. Sergio Rajsbaum Gorodezky
Instituto de Matemáticas

Dr. Ricardo Marcelín Jiménez
Universidad Autónoma Metrópolitana, Departamento de Ingeniería Eléctrica

Ciudad Universitaria, Ciudad de México, Septiembre, 2024

PROTESTA UNIVERSITARIA DE INTEGRIDAD Y

HONESTIDAD ACADÉMICA Y PROFESIONAL

De conformidad con lo dispuesto en los artículos 87, fracción V, del Estatuto General, 68, primer párrafo, del Reglamento General de Estudios Universitarios y 26, fracción I, y 35 del Reglamento General de Exámenes, me comprometo en todo tiempo a honrar a ala institución y a cumplir con los principios establecidos en el Código de Ética de la Universidad Nacional Autónoma de México, especialmente con los de integridad y honestidad académica.

De acuerdo con lo anterior, manifiesto que el trabajo escrito titulado A Study Of Concurrent Data Structures With Relaxed Semantics, que presenté para obtener el grado de Doctor en Ciencia e Ingeniería de la Computación, es original, de mi autoría y lo realicé con el rigor metodológico exigido por mi Programa de Posgrado, citando las fuentes, ideas, textos, imágenes, gráficos u otro tipo de obras empleadas para su desarrollo.

En consecuencia acepto que la falta de cumplimiento de las disposiciones reglamentarias y normativas de la Universidad, en particular las ya referidas en el Código de Ética, llevará a la nulidad e los actos de carácter académico administrativo del proceso de titulación/graduación

Atentamente

Miguel Angel Piña Avelino, 30611383-3

Posgrado en Ciencia e Ingeniería de la Computación
Universidad Nacional Autónoma de México

Tesis para obtener el grado de Doctor en Ciencias e Ingeniería en Computación
Primera Edición, September 24, 2024

Acknowledgements

To the National Autonomous University of Mexico

During the last 18 years, I have been part of this University, which has allowed me to know many people, from teachers to classmates and coworkers, who eventually become great friends. They all contributed to the knowledge and values that I acquired, helping in my personal and professional formation.

To my family

Thank you for being with me at every moment and for your support while I was pursuing my PhD. I especially want to thank my mother, Reina, who is a great source of motivation and inspiration.

To the tutoring comittee and synodals

I am grateful to Dr. Armando Castañeda, director of this thesis, for the support, advice, and patience provided while I was pursuing this PhD. He has been a great advisor to me for the last few years, sharing all his experience and knowledge. Also, I am grateful to Dr. Sergio Rajsbaum and Dr. Ricardo Marcelin for all their advice and the time spent in supervisor duties. Finally, I thank Dr. David Flores and Dr. Jorge Ortega for being members of the jury in my candidacy exam and the thesis defense.

To my friends

To Juan José Lopéz, Daniel Becerra, Olga Villagran, Eduardo López, Carlos Romero, Ricardo Rivas, and Juan Camacho, thank you for all the time you shared with me, which boosted me to be better every day.

To CONAHCYT and DGAPA

I am grateful to CONAHCYT for the national fellowship I received during my PhD and to DGAPA for the financial support from UNAM PAPIIT projects IN108720 and IN108723, which helped to buy the infrastructure where all the experiments of this thesis were performed.

Abstract

Concurrent computing is about the interactions between multiple computing entities over shared resources. It is considered one of the most challenging topics in computer science. Tackling the discipline requires a good imagination. We are used to thinking sequentially, and imagining multiple things happening simultaneously and randomly intermixing is not easy. This thesis explores the shift from traditional to more flexible approaches in concurrent computing for programming concurrent algorithms. It takes a theoretical approach but with practical applications in mind, particularly focusing on how relaxation can be applied to practical environments such as work-stealing and data structures (FIFO queues).

The thesis covers the state-of-the-art on classical classical concurrent computing, relaxations in concurrent computing, the problem of work-stealing, and FIFO queues. It then presents the theoretical preliminaries and the methodology used to analyze two case studies. The first case study is the problem of work-stealing. It presents two relaxed algorithms based on multiplicity and weak multiplicity based solely on read/write operations where fences are not required. The second case study delves into the problem of concurrent FIFO queues and presents a modular approach to building queue algorithms. The work concludes with an experimental evaluation of the different algorithms presented in this thesis and the conclusions and future work.

Contents

List of Figures

Listings

CHAPTER 1

Introduction

In these days, it is very common to hear about new processors that increase the number of cores in each one. Tasks like gaming, data processing, rendering animation, and video edition are becoming more natural daily. These tasks take advantage of the new processors and their multi-core architectures. It is worth mentioning that these multi-core processors are already present in laptops, smartphones, PCs, tablets, smart TVs, game consoles, multiple IoT[1] devices, smartwatches, and even devices like keyboards![2] No matter if we are specialized programmers like those who work in embedded systems or are working on back-end software or developing games, it is really important to design and code algorithms that take advantage of these multi-core architectures.

However, concurrent computing is one of the most challenging topics in computer science. This is because we are used to thinking in a sequence of steps. It is not easy to imagine multiple things happening simultaneously and randomly intermixing. When we program sequentially, it is easy to see that things occur in the same order every time, making it deterministic. However, concurrency introduces non-determinism since processes run independently, meaning things do not necessarily happen in the same order. As a result, all kinds of unforeseen interactions can occur. When building concurrent algorithms, some things must be considered properly to

[1] Internet of the Things.

[2] My current keyboard is already a tiny computing device; it has a small screen on which small applications can be displayed and some additional controls like a knob to use such applications. Regarding the keyboard's processor specification, the provider mentioned that the processor follows a multi-core architecture without specifying which one.

avoid undesirable behaviors in concurrent executions. For example, we can consider the following reasons why concurrent programming can be challenging:

- **Complexity**: Executing concurrent algorithms involves running multiple tasks simultaneously. Such executions can result in complex interactions and interdependencies between different program parts. Managing and coordinating all these interactions in a synchronized manner can be quite challenging.
- **Race Conditions**: A race condition occurs when the expected outcome or state of a shared variable relies on a specific sequence of events that are beyond the program's design. Usually, this problem can result in errors, unpredictable behavior, or bugs that are challenging to replicate.
- **Synchronization**: To ensure that multiple processes can safely access shared resources, synchronization mechanisms such as locks, semaphores, barriers, or even concurrency primitives provided by processor architectures must be used. However, correctly managing these mechanisms can be challenging, as improper use can cause problems such as data corruption or performance issues. Therefore, it is essential to implement these mechanisms correctly to prevent such issues.
- **Performance**: Usually, we think that using multiple cores in parallel should improve the performance of a concurrent program. However, using shared resources and some factors like load balancing, synchronization, and unnecessary parallelization can degrade the performance of concurrent programs. We must carefully design and develop concurrent programs to achieve optimal performance.
- **Deadlocks**: A deadlock happens when two or more processes are waiting for each other to release resources. This results in a state where no process can make any progress. Deadlocks are usually challenging and complicated to identify and resolve, especially in complex systems
- **Scalability**: We want our concurrent programs to scale well as the number of processors or cores increases. However, ensuring that concurrent programs improve when the number of processors increases requires careful design and optimization.
- **Learning Curve**: Additional concepts concerning concurrent programming, like threads, processes, concurrency primitives, linearizability, and sequential consistency, can require a significant learning curve.

- **Debugging and Testing**: The concurrent programs' nondeterministic nature makes them difficult to debug. Order-of-events-dependent bugs are also challenging to reproduce and diagnose. Testing such programs can be time-consuming and complex.

Several techniques have been developed to manage multiple processes that access shared resources simultaneously and address the abovementioned reasons. These techniques include locks, semaphores, barriers, and primitives such as Read-Modify-Write operations, which differ in their level of granularity. Using these techniques, synchronization patterns have been designed to handle situations where data is read after being written by multiple processes, known as Read-After-Write patterns, which rely on the flag principle [44].

1.1 Motivation

Usually, to implement concurrent algorithms in the standard asynchronous shared memory model, we must use Read-After-Write synchronization patterns or atomic Read-Modify-Write instructions (e.g., Compare&Swap or Test&Set). As previously mentioned, Read-After-Write patterns rely on the flag principle [44]. Under this principle, when multiple processes write to a shared variable and then read from another variable, *memory fences* (also known as *barriers*) are necessary to prevent reordering of reads and writes by the processor or compiler. When implementing an algorithm that uses such synchronization patterns in modern multi-core architectures, using memory fences is crucial to ensure proper execution. However, it is well-known that the use of fences is highly costly, while Read-Modify-Write instructions, with high coordination power (it can be formally measured through the *consensus number* formalism [41]), are in principle slower than the simple Read/Write instructions. In practice, contention might be the dominant factor; an uncontended Read-Modify-Write instruction can be faster than contended Read/Write instructions.

The work of Attiya et al. [9] has shown that it is impossible to eliminate expensive synchronization in classic and ubiquitous algorithm specifications. This leads us to question *if it is possible to bypass this impossibility result in any way*. There are two possible ways to circumvent this result: (1) consider relaxed semantics for the algorithms and (2) make additional assumptions about the model. Considering the reasons why concurrent programming can be challenging, we are interested in studying how to design and develop concurrent algorithms that can deal with all (or at least the majority) reasons shown previously using relaxed semantics. In particular, we want to explore the shift from traditional to more flexible concurrent computing approaches to circumvent the impossibility result mentioned previously. Additionally, we want to design modular and simple concurrent algorithms that can use distinct solutions (from classic synchronization methods to relaxed and flexible solutions) as if they were Lego pieces and study when relaxations could be useful in practical settings.

1.2 Objectives And Contribution

We are interested in the following theoretical questions:

1. Are there useful relaxations that admit solutions using only synchronization mechanisms that are among the simplest ones?

2. Is it possible to build modular concurrent algorithms that use relaxed solutions and are good enough to compete with classic algorithms in the state-of-the-art?

As a first step, we explore the problem of the work-stealing in Chapter 4, seeking for Read/Write wait-free and fence-free solutions in the standard asynchronous shared memory model. Work-stealing is a popular technique for efficient task parallelization of irregular workloads by implementing dynamic load balancing. Fence-free means that the algorithm's correctness does not require any specific instruction ordering beyond what is implied by data dependence. The combination of the three requirements, Read/Write based, wait-freedom, and fence-freedom, dramatically restricts the structure of possible solutions. Every operation can only execute the Read instruction in a set of reads followed by the Write instruction in a set of writes, whose written values depend on the reads; in both cases, reads and writes instructions can be executed in any order. Despite the simplicity of the possible solution, we show that non-trivial and useful objects can be implemented.

We first consider work-stealing with multiplicity [16], a relaxation in which every task is taken by *at least* one Take/Steal operation, and, differently from idempotent work-stealing [65], if several operations take on a task, they must be *pairwise concurrent*. Therefore, no more than the number of processes in the system can take the same task. We study the case where tasks are inserted/extracted in FIFO order. We present a Read/Write wait-free algorithm for work-stealing with multiplicity, whose Put operation is fence-free and Take and Steal operations are devoid of Read-After-Write synchronization patterns. The step complexity of Put is constant, while Take and Steal have logarithmic step complexity. Simplicity is a notable quality of the algorithm. It is based on a single instance of MaxRegister object [7, 51], showing that work-stealing with multiplicity reduces to MaxRegister.

Then, we study a variant of multiplicity in which Take/Steal operations extracting the same task *need not be concurrent*. However, each process extracts any task *at most once* and hence the relaxed behavior *is not allowed to happen* in sequential executions. This relaxation is called work-stealing with *weak multiplicity*. We present an algorithm inspired by our first solution, which uses only Read/Write instructions, is *fence-free*, and all its operations are wait-free. Furthermore, each operation has constant step complexity. To our knowledge, this is the first algorithm for a relaxation of work-stealing with all these properties. The algorithm is obtained by reducing work-stealing with weak multiplicity to RangeMaxRegister, a relaxation of MaxRegister proposed here.

We continue our study by addressing the second question presented at the beginning of this section. We adopt a modular approach to building concurrent algorithms to tackle this question. Specifically, we focus on the problem of *multi-producer, multi-consumer concurrent FIFO queues*. Our modular approach for FIFO queue models the queue as a pair of objects to manage the *head* and the *tail*, along with a set of container objects to store the items inserted into the queue. To deal with concurrency during insertions/extractions, we consider the idea of *baskets* [46] as the containers to store the items. Initially, baskets were considered as a way to reduce queue's Compare&Swap contention in a variant of the Michael-Scott queue [64], being defined implicitly. More recently, the concept of the basket was explicitly described as an abstract data type [74]; nevertheless, in this work, we propose a basket specification that provides stronger guarantees and allows different basket implementations to continue with the modular design. We provide two distinct implementations for the baskets, the first one that follows an approach similar to that of the LCRQ algorithm [67], while the second implementation is reminiscent of locally linearizable generic data structure implementations of [33].

In the case of the objects to manage *head* and *tail*, we propose a novel object we call Load-Link/Incremental-Conditional, which resembles the well-known instruction Load-Link/Store-Conditional, and, in a similar fashion to the baskets, it can be implemented in different ways. We even propose a solution that implements only Read/Write instructions instead of more sophisticated Read-Modify-Write instructions to continue tackling the first question of this section. Another implementation of this type of object is based on Compare&Swap instruction.

We complement our results by performing an experimental evaluation for both case studies, i.e., for work-stealing and the modular basket queues. In the first experimental evaluation, we compare our work-stealing algorithms to the standard Cilk THE [30], Chase-Lev [22], and idempotent work-stealing [65]. The algorithms were evaluated using three different benchmarks: (1) zero cost experiment, (2) parallel spanning tree, and (3) parallel SAT. While the work associated with each task is minimal in the first two benchmarks, the work associated with tasks is considerable in the third one. In the first two benchmarks, some of our algorithms exhibit similar and sometimes better performance than idempotent work-stealing algorithms, which outperform Cilk THE and Chase-Lev. However, in the third benchmark, no significant difference exists between all algorithms, either relaxed or not.

Similarly, we compare our modular queue algorithms to the Wait-Free queue by Yang and Mellor-Crummey [90], the Lock-Free LCRQ queue by Morrison and Afek [67], the Lock-Free queue by Michael and Scott [64], the Lock-Free queue by Ramalhete and Correia [77] and the Lock-Free queue by Ostrovsky and Morrison [74], which use the idea of basket, just like we do. The algorithms were evaluated using two benchmarks: (1) inner experiments and (2) outer experiments. In the first benchmark, we evaluate distinct combinations of LL/IC objects and baskets in the modular queue and compare their performance. The results show that the combination of Compare&Swap-based LL/IC object with the basket implementation that follows a similar approach to that of LCRQ performed better than the other combinations. In the second benchmark, the previous combination had a better performance with respect to the queue of Ostrovsky and Morrison and the classic queue of Michael and Scott but it is outperformed by the fastest queues known in state-of-the-art (LCRQ, Yang-Mellor Crummey's queues).

This thesis gathers results from a conference paper in 35th International Symposium on Distributed Computing, DISC 2021 [17], a journal paper in the Journal of Parallel and Distributed Computing [19] and a preprint work published in ArXiv [18]. Part of this thesis is a continuation of such a preprint.

1.3 Structure Of This Thesis

The rest of this thesis is structured as follows. In Chapter 2, we discuss the state of the art concerning concurrent computing, relaxed concurrent computing, the problem of work-stealing, and concurrent queues. Chapter 3 presents the model of computation used for this work, the linearizability and set-linearizability formalisms, a background of hardware fundamentals where is discussed the use of memory fences and some architectures like TSO and x86, the relationship between consistency models in programming languages and the hardware, and finishing with the statistical methodology used for the experiments. Chapter 4 addresses the problem of work-stealing and presents the wait-free, fence-free, Read/Write algorithms to solve this problem. Chapter 5 describes the design for the modular baskets queue and the distinct algorithms of the modules for this queue. Chapter 6 presents the experimental evaluations and the results of both case studies. Chapter 7 closes this work, presenting the final discussion about the two case studies analyzed in this thesis and future research.

CHAPTER 2

State of the Art

From the end of World War II until the 1990s, most computers had only a single processor core. Operating systems used schedulers and other techniques to simulate concurrency. In 2001, IBM created the first multicore processor, which enabled two processors to work together at high bandwidths and benefit from significant on-chip memories and high-speed buses. As time passed, processors were equipped with more cores [47]. It's important to remember Moore's Law, which states that the number of transistors in the same space keeps increasing yearly. However, this results in smaller electronic components and circuits, which cannot be made faster without overheating. As a result, many industries are now using "multicore" architectures. Several processors communicate through shared memory in this setup, using hardware caches and RAM. This allows more effective computing through parallelism, where the processors work together on a single task[44].

The advent of multiprocessors has revolutionized the way we approach software development. By exploiting parallelism, we can run complex algorithms faster by dividing them into smaller sub-tasks. This can be achieved using parallel, distributed, or concurrent computing techniques. However, programming multiprocessors can be challenging because modern computer systems are inherently asynchronous.

This thesis explores the shift from traditional to more flexible approaches in concurrent computing to programming concurrent algorithms. This takes a theoretical approach but with practical applications in mind.

2.1 Classic Concurrent Computing

Sequential computing has been the standard method of performing computations since the early days of electronic computing, before the advent of concurrent, parallel, and distributed computing. Sequential computing involves executing instructions one after the other using a processor based on the contributions of Turing [87] and Von Neumann [89]. In this way, *processes* modify *objects* through *atomic operations*, where the relationship between operations and objects can be defined in terms of *preconditions* and *postconditions*. This is similar to API documentation, which describes the state of an object before and after a method[1] is called on the object, as well as the method's output, which can be a specific value or throw an exception. This style of documentation is known as "sequential specification".

However, this way of expressing the relationship between objects and methods falls short when several processes share such objects. If many processes can invoke an object's operation concurrently, what invocation is first? What is the state after the execution of these overlapping invocations? Does it make sense to talk about operation order?

In concurrent systems, three consistency models are usually utilized as a correctness condition: *Serializability*, *Sequential Consistency*, and *Linearizability*. The concept of Serializability was initially explained by Papadimitriou [76]. Lamport introduced the notion of Sequential Consistency [57]. Herlihy and Wing introduced the idea of Linearizability [45]. Serializability in concurrent computing guarantees the correctness and isolation of transactions in a multi-user database or concurrent system. It ensures that when executing a set of transactions concurrently, the final result is equivalent to running them one after another without overlap, mimicking a serial execution order. This helps maintain consistency and prevents errors in the system. Sequential consistency requires shared variable operations in concurrent systems to appear executed sequentially according to program order. Linearizability is a stricter condition that guarantees Sequential Consistency and ensures that the global order of operations includes a specific point in time (i.e., linearization point) for each operation. This ensures that every operation seems to take effect atomically at some point between its invocation and response. Linearizability refines the concept of Sequential Consistency by imposing a stricter requirement on the sequence of methods. This ensures that the system's observed behavior aligns with a valid

[1] Since now, we will refer to "*operation*" as "*method*" like is used in the context of *Object-Oriented Programming*.

sequential execution of the processes. Therefore, while Sequential Consistency allows for multiple valid orders of operations as long as they respect program order, Linearizability enforces a stricter condition by requiring operations to appear as if they occurred instantaneously at some specific point between invocation and response.

In a concurrent multi-process system, a progress condition outlines the assurance of process progress. It sets specific requirements that ensure processes in the system will keep advancing toward completing their tasks. Progress conditions are partitioned into *blocking* and *non-blocking*. Two blocking progress conditions rely on lock-based synchronization: Deadlock-freedom and starvation-freedom [44]. *Deadlock-freedom* guarantees that processes will not deadlock and at least one process will make progress; this means that a process acquiring a lock will release it; in other words, a process trying to acquire the lock eventually succeeds. *Starvation-freedom* ensures that every thread progresses as long as no other thread holds the lock.

On the other hand, there are three *non-blocking* progress conditions: *Obstruction-Free* [43], *Lock-Free* [45] and *Wait-Free* [41]. *Lock-free* progress condition ensures that *some* method invocation finishes in a finite number of steps. *Wait-free* progress condition [41] is stronger than lock-free, where *every* method invocation finishes its execution in a finite number of steps. When using lock-free methods, the system viewed as a whole will make progress; however, there is not guarantee that any specific thread will make progress. This is because lock freedom ensures *minimal progress*. On the other hand, wait-freedom ensures the *maximal progress*: any process that continues to take steps will make progress. Obstruction-free [43] only guarantees progress only if no other processes actively interfere with the process making progress. This makes the condition strictly weaker than *lock-free*.

Consistency models and progress conditions are properties of the concurrent objects that show how they should behave and how they make progress. However, we still need other properties that tell us how powerful the methods are for solving synchronization problems. In the article Wait-Free Synchronization by Herlihy, the notion of *consensus number* was introduced [41], which is used as a measure of the computational power of concurrent objects. The *consensus number* of a concurrent object is the maximum number of processes that can solve an elementary synchronization problem known as *consensus* using concurrent objects, which are often called *synchronization primitives*. Herlihy shows that there is an infinite hierarchy of synchronization primitives. This means that a primitive at a certain level cannot be used to implement a wait-free or lock-free version of any primitive at a higher level [41].

Developing efficient and correct concurrent algorithms is widely recognized as a challenging problem. To address the issue, currently, multiprocessors provide synchronization instructions that can be expressed as Read-Modify-Write (RMW) operations[2], with high coordination power (measured through the consensus number [41]), which are in principle slower than simple Read/Write instructions[3].

Additionally, some programs may utilize Read-After-Write synchronization patterns that rely on the flag principle (see, for example, [44]). This involves writing to a shared variable and then reading another variable. To ensure proper implementation of this synchronization pattern on multicore architectures, a *memory fence* (also known as a *barrier*) should be explicitly added to prevent the compiler or architecture from rearranging Read and Write instructions.

It has been demonstrated that building concurrent implementations of classic and ubiquitous specifications[4] in the standard asynchronous shared memory model must use Read-After-Write synchronization patterns or atomic Read-Modify-Write instructions. Attiya et al. [9] addresses the fundamental limitation in concurrent algorithms, arguing that the necessity of synchronization mechanisms is intrinsic and cannot be eliminated without incurring significant costs. Ellen et al. [23] show that shared data structures are often inherently sequential and cannot be easily parallelized. Attiya et al. [8] explores the advantages and drawbacks of obstruction-free implementations over other synchronization methods. These implementations can avoid the scalability and fault-tolerance problems that arise from traditional locking-based techniques, which can become a bottleneck in highly concurrent systems. Obstruction-free implementations can perform well without step contention but have high worst-case complexity.

2.2 Relaxed Concurrent Computing

The work of Attiya et al. [9] has shown that it is impossible to eliminate expensive synchronization in classic and ubiquitous specifications. This leads us to question *if it is possible to bypass this impossibility result in any way.* There are two possible ways to circumvent this result: (1) consider relaxed semantics for the algorithms and (2) make additional assumptions about the model.

Software development has become more challenging with the widespread adop-

[2] e.g., Compare&Swap or Test&Set

[3] In practice, an uncontended Read-Modify-Write instruction can be faster than contended Read/Write instructions due to contention.

[4] Such as sets, queues, stacks, and mutual exclusion.

tion of multicore processors as the standard computing platform. It is critical to optimize the use of all available computer resources, including multiple cores, memory, and storage, for efficient performance. Most programs need data structures, and with all these new multicore computing platforms, concurrent data structures are required for implementing distributed, parallel, and concurrent programs. Designing concurrent data structures is a challenging task. The challenge arises when trying to enhance performance while maintaining correctness. As we strive to improve performance, ensuring the algorithm's correctness becomes increasingly more complex [82]. In order to improve scalability, it has been mentioned that traditional data structures must be relaxed. This often involves relaxing correctness and progress conditions[5]. By relaxing the ordering guarantees of queues and stacks, performance and scalability can be significantly increased. There are many examples of natural relaxations that demonstrate this [82]. In the work of Shavit and Taubenfeld [83], it is pointed out that relaxing the semantics of traditional data structures might be beneficial to reduce synchronization requirements and improve scalability: "There is a trade-off between synchronization and the ability of an implementation to scale performance with the number of processors. Amdahl's law implies that even a small fraction of inherently sequential code limits scaling. Using semantically weaker data structures may help reduce the synchronization requirements and improve multicore systems' scalability." Two types of relaxation are used: (1) relaxing the sequential specification of traditional data structures and (2) relaxing the requirements for correctness conditions.

An example of the first case is the k-FIFO queue presented in the work of Kirsch et al. [53, 54], where the sequential specification requirement was relaxed. The elements of this queue can be dequeued out-of-order up to a constant $k \geq 0$ (called k-Out-of-Order). Hezinger et al. [38] address such redefinition of data structure semantics. Their definition of a relaxed data structure corresponds to establishing a distance from any sequence over the alphabet to sequential specification. The k-relaxed sequential set contains all sequences over the alphabet within distance k from the original specification [38]. This semantic specification defines the distance in terms of data structures. Shavit and Taubenfeld conducted a theoretical analysis of relaxed queues, stacks, and multisets. They examine whether the relaxation of these data structures' semantics can result in more simple and scalable implementations. The authors evaluate these relaxations from a perspective of computability [83]. Also, in the work of Henzinger et al. [38], in addition to the definition of K-Out-of-Order, they define the K-Stuttering and the K-Lateness. Concurrent data structures can employ a relaxation scheme known as K-Stuttering, which allows for

[5] Often called *safety* and *liveness* respectively.

a certain amount of repetition or stuttering in operation execution. This relaxation enables an operation to be repeated up to k times before being considered a failure. While this can increase the performance of the data structure by reducing the need for synchronization, it may also compromise its correctness to some extent. The concept of K-lateness involves measuring the duration that an item remains on a stack without removal. This metric is determined by counting the number of pop operations that have occurred (k) since the item was last the youngest element added to the stack. K-lateness is particularly useful in the context of k-stuttering relaxation for concurrent data structures, as it helps to identify which items can be removed from the stack without violating the relaxation constraints.

In the second case, based on the research conducted by Afek et al. [4], the concept of quasi-linearizability is a way to quantify limited non-determinism. An object implementation is considered quasi-linearizable if each execution is at a bounded "distance" from some linear execution of the object. This definition is more flexible than linearizability and can improve the performance and scalability of concurrent object implementations. To illustrate, quasi-linearizability can be seen as a middle ground between linearizability and weaker consistency models. A quantitative relaxation framework to formally specify relaxed objects is introduced in the work of Henzinger et al. [38], and this formalism is applied in the work of Haas et al. [34], where their relaxed queue implementations are instances of a distributed queue, consisting of multiple FIFO queues k-relaxed. In the work of Talmage and Welch [86], it is shown that the relaxations: k-Out-of-Order, k-Lateness, and k-Stuttering and Linearizability studied in [38], can also be defined as consistency conditions. In the work of Henzinger et al. [33], the concept of *local linearizability* is introduced. This relaxed consistency condition applies to container data structures such as pools, queues, and stacks. The concept of distributional linearizability was first introduced in the work of Alistarh et al. [5]. This concept is used to analyze randomized relaxations and has been applied to MultiQueues [79], a group of concurrent data structures designed to implement relaxed concurrent priority queues.

Castañeda, Rajsbaum, and Raynal introduce the concept of *multiplicity* [16, 20], which refers to the property of a relaxed queue or stack that allows an item to be returned more than once by different operations, but only in case of concurrency. The property of *multiplicity* will be utilized for relaxation in most of the research presented in Chapter 4.

2.3 Work-Stealing

In this work, we are interested in studying how relaxation can be applied to practical environments. In particular, we are interested in applying to work-stealing and data-structures. *Work-stealing* is a popular technique for efficient task parallelization of irregular workloads by implementing dynamic *load balancing*. It has been utilized in various contexts, such as programming languages, parallel-programming frameworks, SAT solvers, and state-space exploration in model checking (e.g. [10, 12, 10, 27, 30, 58, 78]).

In the work-stealing technique, each process has a set of tasks it needs to complete. The process that owns the task set can put or take tasks from it to complete them. Once a process completes all its tasks (that is, the set is empty), it becomes a *thief* and can *steal* tasks from another process, which is called the *victim*. A work-stealing algorithm offers three main operations: Put and Take, exclusively for the owner's use, and Steal, solely for the thief's use. To guarantee correctness, *Linearizability* condition [45] is generally assumed, while *lock-freedom* [45] and *wait-freedom* [41] are the typical progress conditions. When designing work-stealing algorithms, the main objective is to ensure that the *Put* and *Take* operations are efficient and easy to use since these are the most frequently used operations by the owner. Unfortunately, it has been demonstrated that any work-stealing algorithm in the standard asynchronous shared memory model must rely on either Read-After-Write synchronization patterns or Read-Modify-Write instructions (such as Compare&Swap or Test&Set) [9]. The Read-After-Write synchronization pattern is based on the *flag principle*, which entails writing on a shared variable and reading another variable (as shown in [44]).

To properly implement an algorithm on multicore architectures using a synchronization pattern, it is crucial to include a *memory fence* (also called *barrier*) to prevent the reordering of Read or Write instructions by the compiler or the architecture. However, these fences can be costly, and atomic Read-Modify-Write instructions, with high coordination power (which can be formally measured through the *consensus number* formalism [41]), are slower than simple Read/Write instructions[6]. Take/Steal operations in work-stealing algorithms are based on the flag principle, as found in the literature [22, 30, 36, 37]. To overcome the impossibility result in [9], we must consider work-stealing with relaxed semantics or make additional assumptions on the model. Only a few works, such as [65] and [68], have explored these directions.

[6] In practice, contention might be the dominant factor, namely, an uncontended Read-Modify-Write instruction can be faster than contended Read/Write instructions.

Observing that in some contexts, it is ensured that no task is repeated (e.g., by checking first if a task is completed) or the nature of the problem solved tolerates repeatable work (e.g., parallel SAT solvers), Michael, Vechev, and Saraswat propose the concept of *idempotent* work-stealing [65]. This relaxation permits a task to be taken *at least once* instead of *exactly once*. Three idempotent work-stealing algorithms are presented in their paper [65], where tasks are inserted and extracted in different orders. The relaxation allows each of the algorithms to overcome the impossibility result in [9] in its Put and Take operations as they use only Read/Write instructions and do not require Read-After-Write synchronization patterns. However, the Steal operation uses Compare&Swap, and Put requires that certain Write instructions not be reordered, while Steal needs certain Read instructions not to be reordered either. Thus, fences are required when the algorithms are implemented. Fences between Read (respective Write) instructions are typically not overly costly in practice. As for progress guarantees, Put and Take are wait-free, while Steal is only non-blocking.

Morrison and Afek propose two work-stealing algorithms in [69] based on the TSO (Total Store Order) model [81]. Their Put operation is wait-free and uses only Read/Write instructions, while Take and Steal are either non-blocking and use Compare&Swap, or blocking and use a *lock*. Two well-known algorithms, Cilk THE, and Chase-Lev work-stealing, have been adapted here. These adaptations have been modified to work with the TSO model, which prohibits the reordering of Write and Read instructions, eliminating the need for fences between them [22, 30]. In Morrison and Afek's algorithms, each process has a local buffer for storing Write instructions until they are sent in a FIFO order to the main memory. Their correctness can be affected by reordering Write or Read instructions, but TSO prevents this. To avoid Read-After-Write patterns, they assume that the Write buffers have limited size.

2.4 FIFO Queues

These shared data structures are fundamental and used in all sorts of systems. Shared-memory implementations of concurrent queues have been proposed for more than three decades. Unfortunately, even state-of-the-art concurrent queues experience poor scalability because of high contention arising from Read-Modify-Write instructions such as Compare&Swap instruction or the (Fetch&Increment) instruction, which manipulate the head and tail of the queue [24, 25, 46, 55, 56, 64, 66, 90]. The latency of RMW instructions increases linearly with the number of contending cores as each instruction acquires exclusive ownership of its cache line.

One of the most popular ways to implement a queue is by utilizing the meaning behind the Fetch&Increment instruction, which does not fail and always makes progress [67, 90]. In many queue implementations, a queue operation retries a failed Compare&Swap until it succeeds [24, 25, 46, 55, 56, 90]. The basket queue approach lies in the middle, where a failed Compare&Swap in an enqueue operation implies concurrency with other enqueue operations. Therefore, the items of all these operations do not need to be ordered, and instead, they are stored in a basket where the items can be dequeued in any order [46]. A recent implementation of hardware transactional Compare&Swap has been proposed to overcome this bottleneck, exhibiting better performance than the usual Compare&Swap [74].

In Section 2.2, examples of First-In-First-Out (FIFO) queues that applied relaxations were provided [34, 53, 54]. Following the work of Castañeda, Rajsbaum, and Raynal [20] about the multiplicity relaxation, Johnen et al. [52] proposed a wait-free FIFO queue that supports multiple enqueuers and multiple dequeuers where. They show that by relaxing the semantics of the queue, such as allowing concurrent *dequeue* operations (multiplicity relaxation), they can achieve $O(\log n)$ worst-case complexity for both *enqueue* and *dequeue* operations.

CHAPTER 3

Preliminaries and Methodology

This chapter will cover all the necessary tools required for the thesis. As we are considering asynchronous shared memory systems, the first section will describe the mathematical foundation for the computational model. This will describe formal concepts such as *process*, *algorithm*, and *Linearizability* [45] in concurrent systems.

In the following section, we will explore the hardware foundations of concurrent systems. Specifically, we will discuss concepts such as *cache memory*, *consistency memory model*, *cache coherence*, and *memory fences*, which are crucial for correctly implementing concurrent algorithms. We will also discuss the concept of a memory model at the programming language level and examine the Java and C++ memory models that define the allowable behavior of multithreaded programs.

Finally, we will conclude this chapter by discussing the statistical experimental methodology used to analyze program implementations and compare performance between them.

3.1 Computation Model

In the realm of computing, concurrent computation stands as a cornerstone for achieving efficient and scalable systems. It enables the execution of multiple tasks simultaneously, which is crucial for modern software applications such as web servers and programs exploiting multi-core processors' power. However, the complexity of concurrent systems requires precise mathematical models to reason about their behavior accurately. This section will discuss the mathematical model in detail, including its multiple components and assumptions for the essential topics developed in this thesis.

We consider the standard concurrent shared memory with $n \geq 2$ *asynchronous* processes, $p_0, \ldots, p_{n-1}$, which may *crash* at any time during execution [40, 44, 45]. The *index* of process p_i is i. Processes communicate with each other by invoking *atomic* instructions of base objects: either simple Read/Write, or more powerful Read-Modify-Write, such as Swap or Compare&Swap.

An *algorithm* for a high-level concurrent object T (e.g., a queue or a stack) is a distributed algorithm $\mathcal{A}$ consisting of local state machines $A_1, \ldots, A_n$. Local machine A_i specifies which instruction of base objects execute to return a response when it invokes a (high-level) operation of T; each of these instructions is a *step*.

An *execution* of $\mathcal{A}$ is a (possibly infinite) sequence of steps, namely, instructions of base objects, plus invocations and responses of (high-level) operations of the concurrent object T with the following properties:

1. Each process first invokes an operation, and only when it has a corresponding response can it invoke another operation, i.e., executions are well-formed and

2. For any invocation to an operation *op* of a process p_i, denoted as $inv_i(op)$, the steps of p_i between that invocation and its correspondent response (if there is one), denoted $res_i(op)$, are the steps specified by A_i when p_i invokes *op*.

An operation in an execution is *complete* if both its invocation and response appear in the execution. An operation is *pending* if only its invocation appears in the execution. It is assumed that after a process completes an operation, it non-deterministically picks the operation it executes next. An execution E is an *extension* of an execution F, if E is a prefix of F, namely, $E = F \cdot F'$ for some F'.

For any finite execution E and any process p_i, $E|p_i$ denotes the sequence of invocations and responses of p_i in E. Two finite executions E and F are equivalent if $E|p_i = F|p_i \; \forall p_i$. For any execution E, *comp*(E) denotes the execution obtained by removing from E all steps and invocations of pending operations.

A process is *correct* in an infinite execution if it takes infinitely many steps. An implementation is lock-free if, in every infinite execution, infinitely many operations are complete [45]. An implementation is *wait-free* if, in every infinite execution, every correct process completes infinitely many operations [41]. Thus, a wait-free implementation is lock-free but not necessarily vice-versa. *Bounded wait-freedom* [40] additionally requires a bound on the number of steps needed to terminate. The *step complexity* is the maximum number of steps a process needs to execute to return.

The step complexity of an algorithm is the maximum among the step complexity of its operations.

In the Read-After-Write synchronization pattern, a process first writes in a shared variable and then reads another shared variable, maybe executing other instructions in between. For example, this mechanism is widely used in the classic Lamport's bakery mutual exclusion algorithm (see [44]). The correctness of the mechanism requires that the write and read instructions of a process are executed in a specific order, although there is no data dependence relation between them. In Section 3.2, we will discuss thoroughly how current processor architectures can reorder instructions, how they can alter the correctness of concurrent algorithms, and how to avoid this problem using *fences*.

An algorithm, or one of its operations, is *fence-free* if it does not require any specific ordering among its steps beyond what is implied by data dependence (e.g., the value written by a Write instruction depends on the value read by a previous Read instruction). Note that a fence-free algorithm does not use Read-After-Write synchronization patterns. In our algorithms, we use notation $\{O_1.inst_1, \ldots, O_x.inst_x\}$ to denote that the instructions $O_1.inst_1, \ldots, O_x.inst_x$ can be executed in any order. Observe that memory fences (also known as memory barriers) are not required to correctly implement a fence-free algorithm in a concrete language or multi-core architecture since any reordering of non-data-dependent instructions does not affect the correctness of the algorithm.

Linearizability [45] is the standard correctness condition for concurrent objects. Intuitively, an execution is linearizable if its (high-level) operations can be ordered sequentially, without reordering non-overlapping operations, so that their responses satisfy the specification of the implemented object.

A *sequential specification* of a concurrent object T is a state machine specified through a transition function δ. Given a state q and an invocation $inv_i(op)$ of process p_i, $\delta(q, inv_i(op))$ returns the tuple $(q', res_i(op))$ (or a set of tuples if the machine is *non-deterministic*) indicating that the machine moves to state q' and the response to op is $res_i(op)$. The sequences of invocation-responses tuples $\langle inv_i(op) : res_i(op)\rangle$ produced by the state machine are its *sequential executions*. For the sake of clarity, a tuple $\langle inv_i(op) : res_i(op)\rangle$ is simply denoted op. Also, the subscripts of invocations and responses are omitted.

Given an execution E, we write $op <_E op'$ if and only if $res(op)$ precedes $inv(op')$ in E. Two operations are *concurrent* denoted $op||_E op'$, if neither $op <_e op'$ nor $op' <_E op$. The execution is sequential if $<_E$ is a total order.

Definition 3.1 (Linearizability [45])
Let be $\mathcal{A}$ an algorithm for a concurrent object T. A finite execution E of $\mathcal{A}$ is *linearizable with respect to* T, or just *linearizable* if T is clear from the context, if there is a sequential execution S of T and E can be extended to an execution E' by appending zero or more responses such that:

1. $comp(E')$ and S are equivalent and
2. for every two complete operations op and op' in E, if $op <_E op'$ then $op <_S op'$.

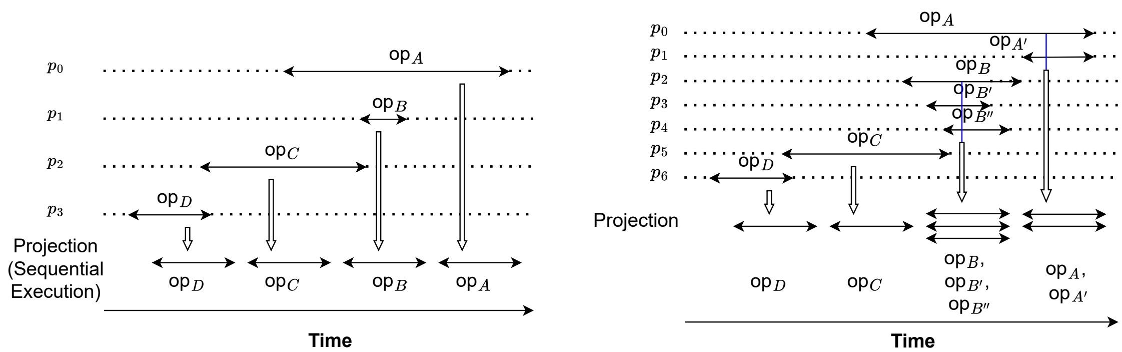
We say that $\mathcal{A}$ *is linearizable with respect to* T or just *linearizable* if T is clear from the context if each of its executions is linearizable.

Another correctness condition for concurrent objects is *Set-Linearizability*. Set-Linearizability [15, 71] is an extension of linearizability, allowing multiple operations to be linearized at the same linearization point, whereas linearizability requires a total order of operations.

A *set-sequential specification* of a concurrent object differs from a sequential execution in that δ receives as input the current state q of the machine and set $Inv = \{inv_{id_1}(op_1), \ldots, inv_{id_t}(op_t)\}$ of operation invocations, and $\delta(q, Inv)$ returns (q', Res), where q' is the next state and $Res = \{res_{id_1}(op_1), \ldots, res_{id_t}(op_t)\}$ are the responses to the invocations in Inv and each id_i denotes the index of the invoking/responding process. Intuitively , all operations $op_1, \ldots, op_t$ are performed concurrently and move the machine from state q to q'. The sequence of sets Inv, Res is a *concurrency class* of the machine. The state machine's sequences of concurrency classes are its *set-sequential executions*. In our set-sequential specifications, invocations will be subscripted with the index of the invoking process only when there is more than one invocation in a concurrency class. Observe that a set-sequential specification in which all concurrency classes have a single element corresponds to a sequential specification.

Given a set-sequential execution S of a set-sequential object, the partial order $<_S$ on the operations of S is defined as above: $op <_S op'$ if and only if $res(op)$ precedes $inv(op')$ in S, namely, the concurrency class of op appears before of the concurrency class of op'.

Definition 3.2 (Set-linearizability [15, 71])
Let be $\mathcal{A}$ an implementation of a concurrent object T. A finite execution E of $\mathcal{A}$



(a) Linearizability (b) Set-Linearizability

Figure 3.1: Graphical description of linearizability and set-linearizability.

is *set-linearizable with respect to* T, or just *set-linearizable* if T is clear from the context, if there is a set-sequential execution S of T and E can be extended to an execution E' by appending zero or more responses such that:

1. $comp(E')$ and S are equivalent,
2. for every two completed operations op and op' in E, if $op <_E op'$ then $op <_S op'$.

We say that $\mathcal{A}$ is *set-linearizable with respect to* T, or just *set-linearizable* if T is clear from the context if each of its executions is set-linearizable.

A comparison between Linearizability and Set-Linearizability can be found in Figures 3.1a and 3.1b, respectively.

3.2 Hardware Foundations

Knowing the hardware foundations behind concurrent computing besides the mathematical computation model is important. When we translate concurrent algorithms from the mathematical computation model to a program in a particular programming language, we need to deal with a lot of assumptions and rules that do not necessarily current processor architectures follow. For example, when we design our algorithms, we suppose linearizability is the correctness condition in asynchronous shared memory. Still, current processors cannot implement linearizability in the real world because this property incurs significant costs[1]. Instead of stronger correctness conditions for asynchronous shared memory, manufacturers provided distinct types of memory consistency (also known as memory model) [70, 80] for their processors and machines. A memory model is a precise, architecturally visible definition of shared memory correctness [70]. Knowing about the hardware foundations helps us to implement correct algorithms. This will help us understand why specific tools like "*fences*" are necessary to ensure correct concurrent computing. In this section, we will discuss the functioning of concurrent programs on computer hardware at a high level and focus on memory interactions at various levels.

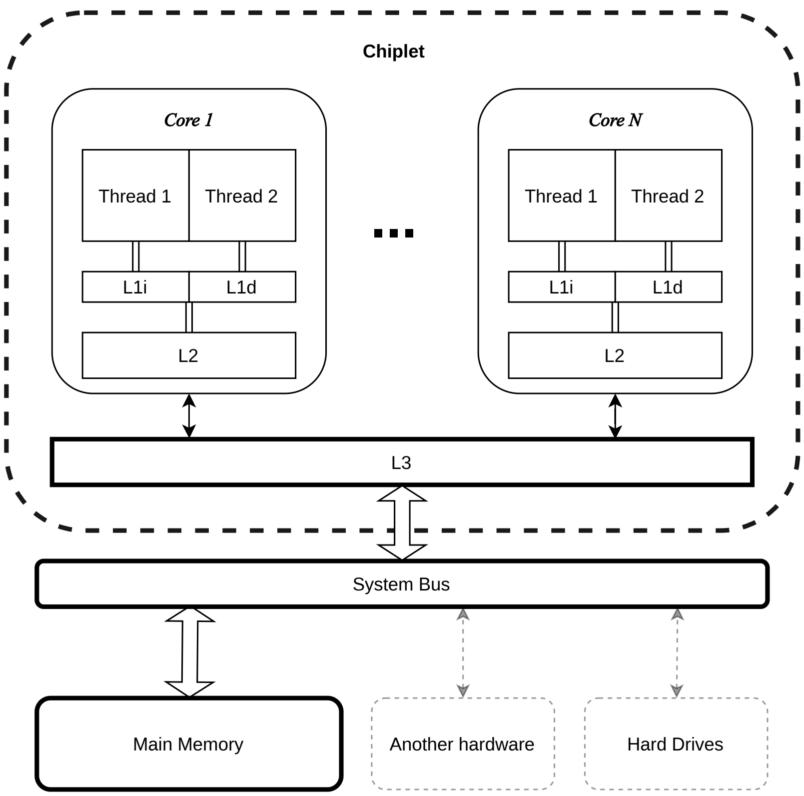


Figure 3.2: Baseline model of a Multi-core Processor Chip.

When analyzing computer systems, it is important to consider those with multi-core processors communicate through a shared memory. This means all cores in the processor, can perform reads (loads) and writes (stores) to all addresses in the

[1] For example, high power consumption, high manufacturing cost, low performance, etc.

memory. A typical system model includes a single chip with multiple cores and off-chip main memory. You can see an illustration of this model in the figure 3.2[2]. Usually, a multi-core processor includes *cache memory*, a special high-speed memory close to the processor that allows fast process access. Caches decrease the average latency when accessing storage structures [70, 80]. In recent times, multi-core chips have adopted a three-tiered cache memory system. Each core has its own private L1 and L2 cache levels, while all the cores share the L3 cache level. The primary purpose of the cache levels L1 and L2 is to provide fast access to data and instructions for the core. Each core uses the first cache level to retrieve required data and execute instructions. Cache L1 is divided into two sub-caches, one for data (L1d) and the other for instructions (L1d). Typically, access to this cache level is faster than access to other levels. The second level of cache is usually more extensive and stores data and instructions about to be executed. Multiple cores share the third cache level and serve as a source for the L2 cache [6, 48].

The *main memory* holds frequently accessed data for the CPU, such as instructions or processing data, and allows faster access than secondary memory. The processor calls the *memory bus* to obtain such data and instructions, which transfers data from the primary memory to the CPU and cache memory. This bus has three parts: the address bus, the control bus, and the data bus. The address bus is used to retrieve information about the location of stored data. On the other hand, the control bus is utilized to transfer control signals from control units to other components of the computer. Finally, the data bus transfers information between the primary memory and the corresponding chipset.

When considering the simplified view of cache and memory architecture, it is important to ensure that shared memory is correct. Incoherence can occur when multiple actors have concurrent access to caches and memory, such as processor cores, external devices, system buses, etc., which may read and/or write to them. The cores will be the main actors, but we must consider the possibility of other actors interacting with caches and memory.

In order to ensure that shared memory is accurate, two important issues must be addressed: *consistency* and *correctness*. Consistency establishes rules for how memory reads (loads) and writes (stores) interact with the memory. These rules must consider the behavior of these operations when multiple threads or even a single thread accesses memory. Consistency models define the proper behaviors for shared memory about loads and stores and do not reference caches or coherence [70]. Memory consistency models (or memory models) specify shared memory correctness. They define the allowed behaviors for multithreaded programs that execute

[2] In the figure 3.2, we omit many features to simplify the reasoning about the hardware

with shared memory. The most intuitive and strongest memory model is *Sequential Consistency* (SC) [57]. Another memory model used by systems *x86* and *SPARC* is *Total Store Order* (TSO) [75, 81, 84]. TSO is driven by the goal of utilizing *first-in-first-out* write buffers to store the outcomes of completed stores prior to writing the results to the caches. Additionally, "relaxed" or "weak" memory models are considered because they show that most memory orderings in strong models are unnecessary [70].

It is important to consider cache coherence protocols when dealing with caching and solving coherence issues. These protocols come into play when multiple cores access multiple copies of data, with at least one being a write access. To ensure that the data accessed is up-to-date and consistent, the distributed set of cores implements a set of rules within a system [70]. Hence, it is essential to consider consistency models and cache coherence protocols to prevent access to stale or incoherent data. The goal of a coherence protocol is to maintain coherence by enforcing the next invariants [70]:

1. *Single Writer, Multiple-Read Invariant* (`SWMR`). At any given (logical) time, only one core may write to memory location A. Other cores may only read the memory location A.

2. *Data-Value Invariant*. The value of the memory location at the beginning of an epoch is the same as the value of the memory location at the end of its last read-write epoch.

To ensure that the `SWMR` and *data value* invariants are always maintained, we use a distributed system consisting of a collection of *coherence controllers*. Such controllers are finite state machines associated with each storage structure (cache and memory). These coherence controllers exchange messages with each other to ensure that invariants are upheld for each structure. The coherence protocol specifies the interaction between these finite-state machines and moving from one state to another based on the conditions of the data and the cache memory [70].

The coherence controllers have several important responsibilities. They handle service requests from two sources: Core and Network. On the "Core side", the cache controller interfaces with the processor core, receiving loads and stores from the core and returning load values to the core. Additionally, the cache controller initiates coherence transactions by issuing a coherence request in the case of a cache miss. This coherence request is sent to one or more coherence controllers across the interconnection network [70]. On the cache controller's "network side", it interfaces with the rest of the system through the interconnection network. The controller

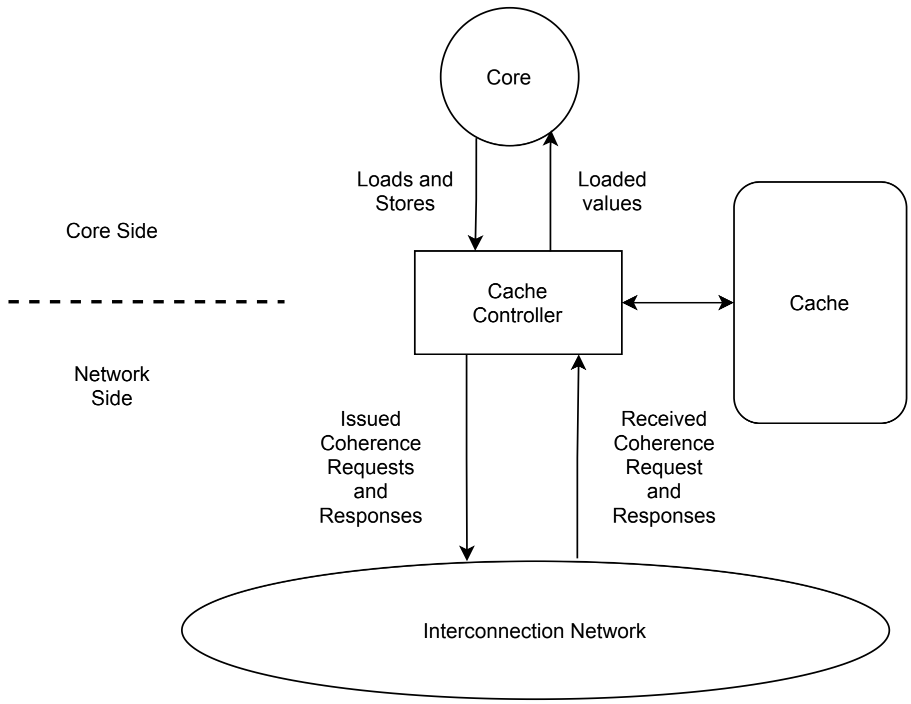


Figure 3.3: Cache Controller

receives requests and coherence responses that it must process [70]. A coherence controller is illustrated in Figure 3.3.

Coherence states are essential for ensuring the smooth operation of a system. These states assist the coherence controller in deciding whether it needs to communicate with other controllers to retrieve new data, update existing data, or continue operating with the current data. The most commonly used coherence states are modified (M), shared (S), and invalid (I). However, AMD has gone a step further with its MOESI protocol [6] and introduced two additional states, owned (O) and exclusive (E), to improve the system's efficiency. On the other hand, Intel has created its extension called MESIF [48] to achieve the same goal. A detailed explanation of how coherence protocols work can be found in the book of Nagarajan et al. [70].

3.3 About Fences And Its Use In Concurrent Algorithms

In many processor architectures, it is common to see cores reordering memory accesses to different addresses to perform efficient computations according to certain rules [70]. These reorders do not affect the execution of a single-thread program. We can consider three possible reorder cases: load-load, store-store, and load-store or store-load. Processors that support the sequential consistency model require each core to preserve the program order in any of those combinations.

However, processors that support the TSO memory model do not guarantee ordering between a store and a subsequent load that comes after it in program order [75, 81]. The reason behind this is that processor cores write to store buffers to hold committed stores until the rest of the memory system can process the stores. Nevertheless, they ensure that the load gets the value of the earlier store.

Compilers can reorder instructions and memory accesses to enhance performance and reduce the cost of certain loads and stores to and from memory. In some cases, if the programmer knows what he is doing, he can use some compilation flags to indicate to the compiler to use instruction reordering, allowing data races of stores in multi-threaded environments[3].

A widely used mechanism to avoid reordering (memory accesses, instructions) is the use of memory fences. A *memory fence* is a special instruction that acts as a barrier that enforces an ordering constraint on memory operations (reads and writes) issued before and after such a memory fence. Memory fences are essential for maintaining memory order consistency in multi-threaded programming environments. This is because most modern CPUs or compilers perform performance optimizations, such as instruction reordering and speculative execution, which can lead to out-of-order execution. Memory fences ensure that memory operations are synchronized and appear to occur in the expected order, preventing potential issues caused by instruction reordering and ensuring the correctness of multi-threaded programs. Typically, these low-level code optimizations, may not significantly impact the behavior of a single-threaded program. However, in concurrent programs where multiple threads are executing simultaneously, these optimizations can lead to unex-

[3]See, for example, the options that control optimizations in the GCC compiler, in specific the flag -Os, which enables all -O3 optimizations, but also turn on the option of `-fallow-store-data-races`: `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`, and the discussion about its lack of clear documentation: `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=97309`.

pected and hard-to-debug issues, such as race conditions and inconsistent data states. Therefore, it is crucial to be mindful of these potential impacts when developing and testing concurrent programs (See Example 3.1).

Example 3.1 (Instruction re-ordering)
Consider the following multi-thread program with two threads, each concurrently running on distinct cores. The first thread executes the code shown in 3.3.1, and the second one executes the code shown in 3.3.2:

```
while (z == 0);
print(y);
```

Code 3.3.1: Code execute by thread 1 on core 1

```
y = 30;
z = 1;
```

Code 3.3.2: Code executed by thread 2 on core 2

In this case, we might expect that the instruction `print(y)` always prints the number 30. Nevertheless, the compiler or the CPU could change the order of the instructions for thread 2, giving, as a result, an execution where the value for `y` is *undefined*, and the instructions could be interleaved as shown in the code 3.3.3:

```
z = 1; // Thread 2
while (z == 0); // Thread 1
print(y); // Thread 1
y = 30; // Thread 2
```

Code 3.3.3: Code reordered by CPU

However, this execution is sequentially consistent but is an out-of-order execution producing an undefined result. With the use of memory barriers, we can ensure that instructions do not be reordered. For example, our code could be rewritten as shown in 3.3.4 and 3.3.5:

```
while (z == 0);
fence();
print(y);
```

Code 3.3.4: Updating code 3.3.1 to use fences.

```
y = 30;
fence();
z = 1;
```

Code 3.3.5: Updating code 3.3.2 to use fences.

Thus, the system cannot print 30 without setting z to 1 before. Using a fence between potentially problematic instructions ensures that the code executes correctly; therefore, the instruction reorders, as shown in Code 3.3.3, cannot occur.

In the case of processors that support the TSO memory model, reordering instructions is not always necessary to produce unpredictable behavior in concurrent programs. This is because the cores of the processors contain store buffers. To better understand this, consider Example 3.2.

Example 3.2 (Dealing with store buffers)
Consider the following multi-thread program with two threads, each concurrently running on distinct cores. Each core has its own store buffer. The first thread executes the code shown in 3.3.6, and the second executes the code shown in 3.3.7. Initially, $x = 0$ and $y = 0$. Is it possible that $(r1, r2) = (0, 0)$ at the end of the execution?

```
    S1: x = FOO;
    L1: r1 = y;
```

Code 3.3.6: Code execute by thread 1 on core 1

```
    S2: y = BAR;
    L2: r2 = x;
```

Code 3.3.7: Code execute by thread 2 on core 2

If we analyze the possible execution of these codes, we can identify four potential outcomes for the values of $r1$ and $r2$. These outcomes are (BAR, FOO), $(BAR, 0)$, $(0, FOO)$, and $(0, 0)$. However, it is important to note that the last outcome is invalid in a Sequential Consistent memory model. In contrast, the

TSO memory model considers the final outcome valid. One might wonder how it is possible to arrive at this value. Consider the following sequence of events:

- Core 1 executes store S1, but FOO is stored in the core's write buffer.
- Similarly, Core 2 executes store S2 and holds BAR in its write buffer.
- Afterward, both cores execute their individual reads, L1 and L2, getting the value 0, which represents the previous value of x and y.
- Finally, both core's write buffers update memory with FOO and BAR.

In the end, the result of the execute the program is $(r1, r2) = (0, 0)$, which is an invalid result in Sequential Consistency. In order to prevent undesirable results, a programmer should similarly use fences as in Example 3.1. Using Code 3.3.6 as a basis, we can add a fence between store S1 and load L1 to ensure that write buffers are emptied into memory and that later loads are not permitted to execute until an earlier fence has been committed. We can use a similar reasoning for Code 3.3.7.

TSO allows for the utilization of a FIFO write buffer, which is beneficial for improving performance by hiding the latency of committed stores. However, a more advanced approach would involve using a non-FIFO write buffer that enables the coalescing of writes. This means that two stores that are not in consecutive program order can be combined and written to the same entry in the write buffer. Nonetheless, employing a non-FIFO coalescing write buffer violates TSO, as TSO mandates that stores must adhere to the program order [70].

TSO behaves similarly to Sequential Consistency, permitting only one type of reordering. Hence, the use of fences is fairly infrequent, and the implementation is not too critical. However, x86 architecture provide three types of fences: `LFENCE`, `SFENCE` and, `MFENCE` [6, 48]:

- `MFENCE`: is a full memory fence that ensures that no later loads or stores are observable globally before any earlier loads or stores. It empties the store buffer before later loads can execute.
- `SFENCE`: only prevents the reordering of writes (it is a store-store barrier), it works well with *non-temporal stores*[4] and other instructions listed as exceptions.

[4]Non-temporal stores means that the data being stored is not going to be read

- LFENCE: is designed to prevent the reordering of reads with subsequent reads and writes, effectively combining load-load and load-store barriers. However, according to x86 specification, load-load, and load-store barriers are always present. As a result, LFENCE by itself is not enough for memory ordering; however, there is a larger discussion about its use far beyond the scope of this thesis.

For consistency models that permit far more reordering, fences are used more frequently, and their implementation can significantly impact performance. In programming languages, memory models are defined to provide a consistent interface for developers to implement concurrent programs, regardless of the different hardware models provided by different processor architectures. We will delve deeper into this topic in the next section.

3.4 After Hardware Foundations, What's Up About Programming Languages?

The preceding sections introduced the foundational hardware concepts needed to comprehend the memory consistency models and cache protocols required to create accurate concurrent programs. The model discussed earlier is situated at the hardware level and low-level software. However, defining or redefining memory models for high-level languages is equally important as it creates a standardized interface between a program and any software or hardware that might modify that program. A memory model also enables us to understand how the program will behave in a multi-core environment, making it easier to reason about its behavior [2, 70].

In recent years, memory models have been specified for two of the most widely used programming languages, C++ [13] and Java [60]. These models describe the expected behavior of language-level threads, locks, atomics, and Read-Modify-Write instructions. The specifications of such memory models outline the anticipated outcomes for high-level language programmers and the capabilities that compilers, runtime systems, and hardware providers can deliver. Figure 3.4 illustrates the difference between (a) high-level and (b) low-level memory models [70].

Java and C++ adopt the relaxed memory model approach of "*Sequential Consistency for Data-Race Freedom (SC for DRF)*" [3]. A data race occurs when two

again soon (i.e., no "temporal locality"). https://sites.utexas.edu/jdm4372/2018/01/01/notes-on-non-temporal-aka-streaming-stores/

memory accesses target the same location simultaneously and are not reads or synchronization operations. The approach of *SC for DRF* guarantees that a program is correctly synchronized if and only if all sequentially consistent executions are free of data races. If a program is correctly synchronized, then all program executions will appear to be sequentially consistent [72].

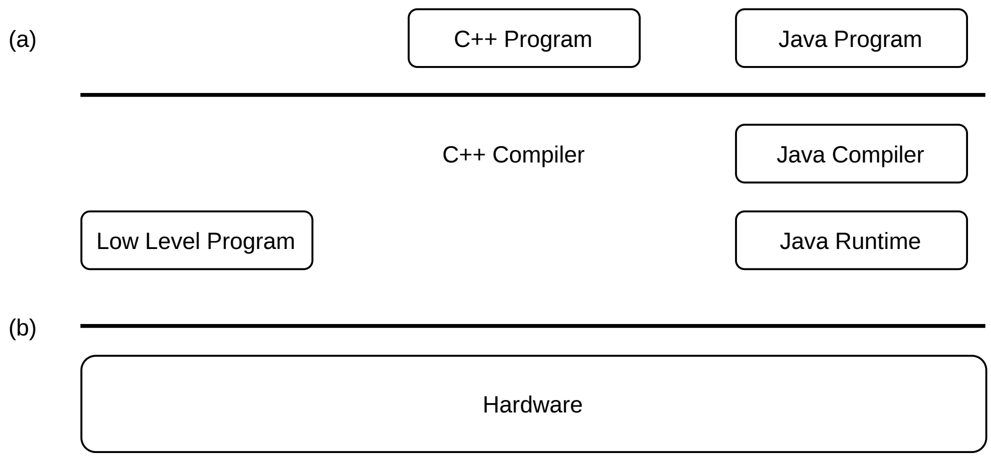


Figure 3.4: (a) High-level and (b) low-level memory models.

The memory model of C++ specifies the order in which memory accesses, including regular, non-atomic memory accesses, should occur around an atomic operation [49]. Without constraints on a multi-core system, when multiple threads simultaneously read and write to multiple variables, one thread may observe the values change in a different order than the order in which another thread wrote them. This can also occur among multiple reader threads [49]. Similar effects can occur even on uniprocessor systems due to compiler transformations allowed by the memory model. By default, all atomic operations provided by the library follow sequentially consistent ordering. However, this default can negatively impact performance. The library's atomic operations can be given an additional memory order argument to specify precise constraints beyond atomicity that the compiler and processor must enforce for a specific operation [49]. This argument specifies the precise constraints the compiler and processor must enforce for a given operation beyond just ensuring atomicity [49]. Six memory orders are defined in the specification, ranging from the weakest order (specified as `std::memory_order_relaxed`) to the strongest one (specified as `std::memory_order_seq_cst`), which is a sequentially-consistent ordering. Table 3.1 shows the description of each memory order as in the C++ specification [49].

Value	Explanation
`memory_order_relaxed`	Relaxed operation: no synchronization or ordering constraints are imposed on other reads or writes; only this operation's atomicity is guaranteed.
`memory_order_consume`	A load operation with this memory order performs a consume operation on the affected memory location: no reads or writes in the current thread dependent on the value currently loaded can be reordered before this load. Writes to data-dependent variables in other threads that release the same atomic variable are visible in the current thread. On most platforms, this only affects compiler optimizations.
`memory_order_acquire`	A load operation with this memory order performs the acquire operation on the affected memory location: no reads or writes in the current thread can be reordered before this load. All writes in other threads that release the same atomic variable are visible in the current thread.
`memory_order_release`	A store operation with this memory order performs the release operation: no reads or writes in the current thread can be reordered after this store. All writes in the current thread are visible in other threads that acquire the same atomic variable (see Release-Acquire ordering below), and writes that carry a dependency into the atomic variable become visible in other threads that consume the same atomic.
`memory_order_acq_rel`	A read-modify-write operation with this memory order is both an acquire operation and a release operation. No memory reads or writes in the current thread can be reordered before the load or after the store. All writes in other threads that release the same atomic variable are visible before the modification, and the modification is visible in other threads that acquire the same atomic variable.

`memory_order_seq_cst`	A load operation with this memory order performs an acquire operation, a store performs a release operation, and read-modify-write performs both an acquire operation and a release operation, plus a single total order exists in which all threads observe all modifications in the same order.

Table 3.1: Memory orders in C++

In the previous section, we discussed memory fences as an essential element of concurrent programming. Both C++ and Java offer memory fences to ensure the correct execution of concurrent programs. In C++, the function `std::atomic_thread_fence` establishes memory synchronization ordering of non-atomic and relaxed atomic accesses as instructed by the memory order, without an associated atomic operation. However, it is worth noting that on x86 systems (x86_64), these functions do not issue any CPU instructions and only affect compile-time code. The exception to this is `std::atomic_thread_fence(std::memory_order::seq_cst)`, which issues the full memory fence instruction `MFENCE` [49].

In the case of Java, for versions less than 9, fences and other low-level operations were restricted to the use of a class named `UNSAFE`[5]. `UNSAFE` was the most powerful tool on the platform because it allowed users to violate established rules and perform otherwise impossible actions. In the latest versions, the Java platform provides the class `VarHandle`[6], which exposes the memory fence methods [73] shown in Table 3.2, additionally to provide access to another low-level operation. Many of these fences try to behave similarly according to the memory orders defined by the specification of C++ [49].

Fence	**Description**
`fullFence`	Ensures that loads and stores before the fence will not be reordered with loads and stores after the fence. This method has memory ordering effects compatible with `atomic_thread_fence(memory_order_seq_cst)`.

[5] sun.misc.UNSAFE
[6] `java.lang.invoke.VarHandle`

`acquireFence`	Ensures that loads before the fence will not be reordered with loads and stores after the fence. This method has memory ordering effects compatible with `atomic_thread_fence(memory_order_acquire)`.
`releaseFence`	Ensures that loads and stores before the fence will not be reordered with stores after the fence. This method has memory ordering effects compatible with `atomic_thread_fence(memory_order_release)`.
`loadLoadFence`	Ensures that loads before the fence will not be reordered with loads after the fence.
`storeStoreFence`	Ensures that stores before the fence will not be reordered with stores after the fence.

Table 3.2: Memory fences provided by Java

It is important to note that the Java Language Specification [72] does not explicitly mention the use of barriers. However, in Java, the usage of barriers may be considered an implementation detail, as its memory model attempts to operate based on *happens-before* semantics. The *happens-before* semantics defines a set of rules about the ordering and visibility guarantees between actions in a program. This helps to show that changes made by one thread become visible to others. As mentioned previously, Java also adopts the relaxed memory model approach of "*Sequential Consistency for Data-Race Freedom*" [3], which is crucial to prevent data-races and ensure the correct behavior of concurrent programs.

Definition 3.3 is a term used in the Java Language Specification to explain what the *happens-before* semantic means. It refers to the main operations that establish this relationship, which include (1) program order, (2) volatile variables, (3) locks, (4) fork-join pattern on threads, (5) thread interruptions, and (6) thread terminations. In certain unforeseen situations not specified in the Java Language Specification, the use of `UNSAFE` or `VarHandle` classes becomes necessary.

> **Definition 3.3 (Happens-before semantics[72])**
> Two actions can be ordered by a *happens-before* relationship. If one action *happens-before* another, then the first is visible to and ordered before the second.
>
> If we have two actions x and y, we write $hb(x, y)$ to indicate that x happens-before y.

> If x and y are actions of the same thread and x comes before y in program order, then $hb(x, y)$.
>
> There is a happens-before edge from the end of an object's constructor to the start of a finalizer for that object.
>
> If an action x synchronizes-with a following action y, then we also have $hb(x, y)$.
>
> If $hb(x, y)$ and $hb(y, z)$, then $hb(x, z)$.

3.5 Experimental Methodology

One of the goals of this thesis is to assess our algorithms' performance. In experimental computer science research and development, benchmarking plays a vital role. Developers perform benchmarking tests on their products under development to evaluate their performance, while researchers use benchmarking to assess the impact on the performance of their novel research ideas [32]. In this thesis, we have followed and adapted the guidelines presented in the work of Forsyth et al.[28], Georges et al. [32] and Lilja [59]. These guidelines provide fundamental techniques for measuring computer performance and strategies for analyzing and interpreting the resulting data. Topics covered include performance metrics, benchmarking programs, and statistical tools. By following these guidelines, we can perform rigorous statistical evaluation, better understand the performance of our algorithm implementations, and compare them against other algorithms.

A standard method of evaluating experimental results is by measuring performance or throughput. But what do these terms mean? *Throughput*, defined by the Cambridge Dictionary, is the amount of work completed in a given period. In contrast, *performance* refers to how well something functions or works. Performance is measured by the amount of useful work a system accomplishes, typically determined by its accuracy, efficiency, and speed of executing instructions. One or more of the following factors might be involved when performance is measured:

1. Short response time for a given piece of work.
2. High throughput.
3. Low utilization of computing resources.
4. High availability of a computing system.

5. High bandwidth.
6. Short data transmission time.

From the work of Lilja [59], some strategies for measurement are:

- **Event driven**: It records the information necessary to calculate the performance metric whenever an event occurs.
- **Tracing**: Similarly to the previous, but instead of recording the event that has occurred, a portion of the system is recorded to identify the event.
- **Sampling**: This strategy records a portion of the system in a fixed time interval.
- **Indirect measurement**: This type occurs when the metric data is not directly accessible, and you must find another metric that can be measured directly.

We can combine those strategies with interval timers to measure the time it takes to execute the program or a section of code, providing a time basis for sampling.

3.5.1 Statistic tools for experiments

As computer science and engineering researchers, we aim to measure and compare the performance of novel and existing algorithms. To evaluate the effectiveness of these algorithms, we require an experimental methodology that enables us to measure their performance and throughput and determine whether they are competitive. This thesis proposes various concrete implementations of the same algorithm for different studies. Our experiments will be categorized into two groups. The first category will measure the performance of different algorithm versions, while the second category will focus on comparing our best algorithm (or the two best) with other algorithms mentioned in the literature. To evaluate the performance of our algorithms, we will measure the time taken to execute a set of operations over a specified time interval. This will help us determine how quickly the program can complete its execution. The technique used to measure the time of an event is the following:

- Read the current time and store it in a variable `start_count`.
- Let the portion of the program execute.
- Read the current time and store it in a variable `stop_count`.

- Take the difference between `start_count` and `stop_count`. This will be the total time required to execute the event.

This technique for measuring the execution time of any portion of a program is known as the *wall clock* time [59]. We will use this technique to measure all the events we want to track. However, remember the measurements from this technique include time spent on other system operations, such as memory paging, thread interleaving, input/output operations, and network communication, if applicable. These external events can introduce uncertainty, errors, or noise into our measurements. To quantify the uncertainty, we need to use probability and statistics tools.

To summarize a collection of measures, we can use indices of central tendency such as the mean (Definition 3.4), median, and mode. The most commonly used index is the sample arithmetic mean or average, which can summarize all the measurements into a single value representing the center of these values' distribution. To quantify the precision of our measurements, we can use a *confidence interval* for the mean value [32, 59]. Other tools we need are the *sample variance* (Definition 3.5), the *standard deviation* (Definition 3.6), and the *coefficient of variation* (Definition 3.7).

Definition 3.4 ((Sample arithmetic) Mean)
Formally, the *(sample arithmetic) mean* is defined to be:

$$\bar{x}_A = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{3.1}$$

Where x_i values are the individual measurements.

Definition 3.5 (Sample Variance)
The *sample variance* represent our calculated estimate of the actual variance. It is defined to be:

$$V = \frac{\sum_{i=1}^{n}(x_i - \bar{x}^2)}{n-1} \tag{3.2}$$

Where the x_i are the n independent measurements and $\bar{x}$ is the corresponding sample mean.

Definition 3.6 (Standard Deviation)
From the Equation 3.2 described in Definition 3.5, the standard deviation is

defined as the positive square root of the variance:

$$s = \sqrt{V} = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x}^2)}{n-1}} \tag{3.3}$$

Definition 3.7 (Coefficient of Variation)
The coefficient of variation (COV) is defined as the standard deviation (Equation 3.3) divided by the mean (Equation 3.1):

$$COV = \frac{s}{\bar{x}} \tag{3.4}$$

Experimental evaluation assumes that errors that occur during an experiment follow a *Gaussian* (a.k.a. normal) distribution. This implies that if multiple measurements are taken of the same value, they will tend to follow a Gaussian distribution centered around the true mean value x [59]. Suppose we assume that the random errors follow a Gaussian distribution. In that case, we can use the properties of the distribution to evaluate the accuracy of our estimate of the true value. Confidence intervals can help us determine a range of values with a high probability of containing the true value. To do so, we need to consider two cases:

1. When the number of measurements is large ($n \geq 30$).
2. When the number of measurements is small ($n < 30$).

The number 30 is typically chosen by convention as the minimum sample size required for the central limit theorem to hold true. This theorem in probability theory explains that as the sample size increases, the distribution of a sample variable should approximate a normal distribution, regardless of the actual shape of the population. As a result, confidence intervals can be used to estimate the overall mean of these averaged values [32, 59].

For the first case, we use the sample mean ($\bar{x}$) as the best approximation of the true value. If the n samples used to calculate $\bar{x}$ are all independents with mean μ y standard deviation s, the central limit theorem then assures us that, for large values of n, the sample mean $\bar{x}$ is approximately Gaussian distributed with mean μ and standard deviation $s/\sqrt{n}$. We can quantify the precision of the measurements by searching two values c_1 and c_2, such that the probability of the mean value being between those two values is $1 - \alpha$. That is $PR[c_1 \leq \bar{x} \leq c_2] = 1 - \alpha$. c_1 and c_2 are chosen to form a symmetric interval around $\bar{x}$ such that $Pr[x < c_1] = Pr[x > c_2] = \frac{\alpha}{2}$.

The interval $[c_1, c_2]$ is called *confidence interval* for $\bar{x}$ and α is called the *significance level* and the value $(1-\alpha)$ is called the *confidence level* [32, 59]. From the central limit theorem, we have:

$$c_1 = \bar{x} - z_{1-\alpha/2}\frac{s}{\sqrt{n}} \tag{3.5}$$

$$c_2 = \bar{x} + z_{1-\alpha/2}\frac{s}{\sqrt{n}} \tag{3.6}$$

where $\bar{x}$ is the sample mean, s is the sample standard deviation, n is the number of measurements and $z_{1-\alpha/2}$ is the value of a standard unit normal distribution with mean $\mu = 0$ and variance s^2, which obeys the following property: $Pr[Z \leq z_{1-\alpha/2}] = 1 - \alpha/2$, where the value $z_{1-\alpha/2}$ is typically obtained from a pre-computed table.

In the second case, for a small number of measurements ($n < 30$), the sample variances s^2 calculated for different groups of measurements can vary significantly. The distribution of the transformed value $z = \frac{\bar{x}-x}{s/\sqrt{n}}$ follows the *Student's t*-distribution with n - 1 degrees of freedom. Then, the confidence interval for $\bar{x}$ when $n < 30$ can be computed as:

$$c_1 = \bar{x} - t_{1-\alpha/2;n-1}\frac{s}{\sqrt{n}} \tag{3.7}$$

$$c_2 = \bar{x} + t_{1-\alpha/2;n-1}\frac{s}{\sqrt{n}} \tag{3.8}$$

where $t_{1-\alpha/2;n-1}$ defined such that a random variable T that follows the *Student's t*-distribution with $n-1$, obeys: $Pr[T < t_{1-\alpha/2;n-1}] = 1 - \alpha/2$, where the value $z_{1-\alpha/2;n-1}$ is typically obtained from a pre-computed table [32, 59].

Confidence intervals are an interesting concept because they provide insight into how much noise there is in measurements. However, when making decisions about the performance of one or more systems, we need to determine whether changes are due to random fluctuations or if they are statistically significant. To do this, we can use the following two techniques [32, 59]:

1. Comparing two alternatives
2. Analysis of variance (ANOVA)

The first technique is simple. The approach to comparing two alternatives is to determine whether the confidence intervals for two groups of measurements overlap. Suppose the intervals of two sets of data do not overlap. In that case, we can conclude that there is no evidence to suggest that there is not a statistically significant difference between them. The wording of this last sentence is important because there is still a probability α that the differences observed in our measurements are due to random effects in our measurements, i.e., we cannot guarantee with absolute certainty that there is a difference between the compared alternatives [32].

Conversely, if the intervals overlap, we cannot confidently conclude that the differences seen in the mean values are not due to random fluctuations. To determine whether there is no statistical difference, we need to calculate the confidence interval for the difference of the means of the two alternatives. First determine the sample mean $\bar{x_1}$ and $\bar{x_2}$ and the sample standard deviation s_1 and s_2. Then, compute the difference of the means as $\bar{x} = \bar{x_1} - \bar{x_2}$. The standard deviation s_x of the difference of the mean values is computed as:

$$s_x = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}} \tag{3.9}$$

Then, the confidence interval for the difference of the means is then given by:

$$c_1 = \bar{x} - z_{1-\alpha/2} s_x \tag{3.10}$$

$$c_2 = \bar{x} + z_{1-\alpha/2} s_x \tag{3.11}$$

The confidence interval calculated before is in the case when the number of measurements is considerable on both systems, i.e., $n_1 \geq 30$ and $n_2 \geq 30$. When the number of measurements on at least one of the systems is smaller than 30, we can no longer assume that the difference between the means is under Gaussian distribution. In the last case, when the number of measurements in both systems is small, i.e., $n_1 < 30$ and $n_2 < 30$, we need to resort to the Student's t distribution by replacing the value $z_{1-\alpha/2}$ with $t_{1-\alpha/2;n_{df}}$, where n_{df} represent the degrees of freedom, which it can approximate by integer number nearest to:

$$n_{df} = \frac{(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2})^2}{\frac{(s_1^2/n_1)^2}{n_1-1} + \frac{(s_2^2/n_2)^2}{n_2-1}} \tag{3.12}$$

In the case of the **Analysis of Variance (ANOVA)**, a general technique for observing the variation in a collection of measurements into meaningful components. To perform this analysis, it is necessary to assume that the errors in the measurements for the distinct alternatives are independent and under normal distribution. The variance for the measurement errors is the same for all alternatives. The variation observed is divided into:

Measurements	1	2	$\dots$	j	$\dots$	k	Overall mean
1	y_{11}	y_{12}	$\dots$	y_{1j}	$\dots$	y_{1k}	
2	y_{21}	y_{22}	$\dots$	y_{2j}	$\dots$	y_{2k}	
$\vdots$	$\vdots$	$\ddots$	$\dots$				
i	y_{i1}	y_{i2}	$\ddots$	y_{ij}	$\vdots$	y_{ik}	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$			
n	y_{n1}	y_{n2}	$\dots$	y_{nj}	$\dots$	y_{nl}	
Column means	$\bar{y}_{.1}$	$\bar{y}_{.2}$	$\dots$	$\bar{y}_{.j}$	$\dots$	$\bar{y}_{.k}$	$\bar{y}_{..}$

Table 3.3: Organizing the n measurements for k alternatives in an ANOVA analysis

1. The variation observed *within* each system is assumed caused by the measurement error.

2. The variation *between* alternatives.

If the variation between the alternatives is larger than the variation within each alternative, then it can be concluded that there is a statistically significant difference between the alternatives. To evaluate ANOVA, we must organize the measurements as shown in the table 3.3: there are $n \cdot k$ measurements for all k alternatives. The column means are defined as:

$$\bar{y}_{.j} = \frac{\sum_{i=1}^{n} y_{ij}}{n} \tag{3.13}$$

The overall mean is defined as:

$$\bar{y}_{..} = \frac{\sum_{j=1}^{k} \sum_{i=1}^{n} y_{ij}}{n \cdot k} \tag{3.14}$$

Compute the sum of squares of the differences between the mean of the measurements for each alternative and the overall mean to find the variation due to the effects of the alternatives (SSA):

$$SSA = n \sum_{j=1}^{k} (\bar{y}_{.j} - \bar{y}_{..})^2 \tag{3.15}$$

The variation within an alternative due to random effects is calculated by summing the differences (or errors) between individual measurements and their respective alternative mean.

$$SSE = \sum_{j=1}^{k} \sum_{i=1}^{n} (y_{ij} - \bar{y}_{.j})^2 \tag{3.16}$$

Finally, the sum-of-squares total, SST, or the sum of squares of the differences between the individual measurements and the overall mean is defined as:

$$SST = \sum_{j=1}^{k} \sum_{i=1}^{n} (y_{ij} - \bar{y}_{..})^2 \tag{3.17}$$

It is possible to split the observed total variation (SST) into a *within* component (SSE) and a *between* component (SSA).

$$SST = SSA + SSE \tag{3.18}$$

ANOVA analysis quantifies whether there is a difference in variation across alternatives (SSA) compared to variation within each alternative (SSE) due to random measurement errors. One way to do this is to compare the fractions $\frac{SSA}{SST}$ and $\frac{SSE}{SST}$.

A more rigorous approach is to use the F-test [59], which tests whether two variances are significantly different. After conducting an ANOVA test, we may determine a significant difference between the alternatives, but the test does not specify which alternatives have a significant difference. Various techniques can be employed to determine whether there is a statistically significant difference between alternatives. We will describe the techniques we will use for each of our specific case studies.

3.5.2 Statistically rigorous methodology

Measuring performance in programming languages like Java is far from trivial due to the many factors that can affect the computation, e.g., the garbage collector and heap size. We use the methodology proposed by Georges, Buytaert, and Eeckhout [32] to obtain statistically rigorous results. The methodology measures the *steady-state* performance, which concerns long-running applications in which start-up is of less interest. An important detail to consider in languages like Java is that JIT compilation (compilation Just In Time) is performed during the start-up of the virtual machine, and the load of the program, steady-state performance suffers less from the variability due to JIT compilation. For compiled languages like C++, this detail is not important.

Two issues must be addressed to quantify steady-state performance. The first is to determine when a steady state is reached. The second is that different evaluations may result in different steady-state performances. Georges et al. proposed a four-step methodology for quantifying steady-state performance. The methodology is as follows for a given experiment:

1. Consider p invocations, each invocation running at most q benchmark iterations. Suppose we want to retain k measurements per invocation.

2. For each invocation i, we must determine the first iteration s_i, where the steady-state performance is reached. This means that the *coefficient of variation* (CoV) [7] of the most recent five executions falls below an established threshold (for example, 0.02). If the CoV never drops below the threshold established for any five consecutive executions, it is considered the five consecutive executions with the lowest CoV.

3. For each invocation, compute the mean $\bar{x}_i$ of the last five executions under steady-state is:
$$\bar{x}_i = \frac{\sum_{j=s_i-k}^{s_i} x_{ij}}{5}$$

4. Compute the confidence interval for a given confidence level across the computed means from the different invocations. For example, computing a 95% confidence interval over the $\bar{x}_i$ measurements.

Since the number of measurements is small, the confidence interval is computed under the assumption that the distribution of the transformed value t corresponds with:
$$t = \frac{\bar{x} - \mu}{s/\sqrt{n}}$$
where s is the sample standard deviation, μ is the population mean and n is the number of measurements; the transformed value follows the *Student's* t-distribution with $n-1$ degrees of freedom. The confidence intervals can be computed as described in Section 3.5.1 and in the work of Georges et al. [32].

[7] CoV is the standard deviation s divided by the mean $\bar{x}$.

CHAPTER 4

Case Study 1: Work-Stealing

In this chapter, we begin with a case study related to the topic of relaxations applied to data structures. Our motivation for this study comes from previous research, which has shown that algorithms for concurrent objects often use expensive and complex synchronization mechanisms that can harm performance (see, for example, [9, 41]). To address this issue, researchers have proposed objects with relaxed semantics that can provide algorithms without the need for such costly mechanisms [65, 68]. We aim to investigate whether there are any non-trivial and useful relaxed objects that can be implemented using only basic synchronization mechanisms without compromising performance. Specifically, we explore how we can provide relaxed data structures that can be applied to the case of work-stealing to solve this problem.

We focus on the First-In-First-Out (FIFO) data structures with single-producer multi-consumer semantics and on two relaxation methods for work-stealing, known as multiplicity and weak-multiplicity. These relaxation methods allow a task to be extracted by more than one Take/Steal operation, but each process can only take the same task at most once. However, this relaxation can only occur in a concurrent environment. The first relaxation's property is directly guaranteed by the definition of set-linearizability. The second relaxation follows from the requirement that solutions must be sequentially exact. We present two read/write, wait-free algorithms for these relaxations that do not require read-after-write synchronization patterns. Additionally, the second algorithm is fence-free with constant step complexity.

Chapter 6 presents an experimental evaluation using three benchmarks related to this case study and the results obtained from the evaluation. Our goal is to determine whether implementing the algorithms presented in this chapter can compete with or even surpass other state-of-the-art work-stealing algorithms.

4.1 Introduction

Work-stealing is a popular technique to implement dynamic *load balancing* for efficient task parallelization of irregular workloads. It has been used in several contexts, e.g., programming languages, parallel-programming frameworks, SAT solvers and state-space exploration in model checking (e.g. [10, 12, 21, 27, 30, 58, 78]).

In work-stealing, each process *owns* a set of tasks that must be executed. The *owner* of the set can put tasks in it and can take tasks from it to execute them. When a process runs out of tasks (i.e., the set is empty), it becomes a *thief* to steal tasks from a *victim*. A work-stealing algorithm provides three high-level operations: Put and Take, which can be invoked only by the owner, and Steal, which a thief can invoke. *Linearizability* is the usual assumed correctness condition, while *lock-freedom* and the stronger *wait-freedom* are the typical progress conditions.

A main target when designing work-stealing algorithms is to have Put and Take operations as simple and efficient as possible, as typically, they are the operations most intensively used by the owner. Unfortunately, it has been formally shown that any work-stealing algorithm in the standard asynchronous shared memory model must use Read-After-Write synchronization patterns or atomic Read-Modify-Write instructions (e.g., Compare&Swap or Test&Set) [8]. Read-After-Write is a useful synchronization pattern based on the *flag principle*, i.e., writing on a shared variable and then reading another variable (see [44]). To correctly implement an algorithm using such a synchronization pattern in real multi-core architectures, a *memory fence* needs to be explicitly added so that the compiler or the architecture does not reorder the Read and Write instructions. It is well-known that fences that avoid reads and writes to be reordered are highly costly, while atomic Read-Modify-Write instructions, with high coordination power (which can be formally measured through the *consensus number* formalism [41]), are in principle slower than the simple Read/Write instructions.[1] Indeed, the known work-stealing algorithms in the literature are based on the flag principle in their Take/Steal operations [22, 31, 36, 37]. Two possible ways to circumvent the impossibility result in [8] are to consider work-stealing with relaxed semantics or to make extra assumptions on the model. As far as we know, [65]

[1] In practice, contention might be the dominant factor, namely, an uncontended Read-Modify-Write instruction can be faster than contended Read/Write instructions.

and [68] are the only works that follow these directions.

Observing that in some contexts, it is ensured that no task is repeated (e.g., by checking first if a task is completed) or the nature of the problem solved tolerates repeatable work (e.g., parallel SAT solvers), Michael, Vechev, and Saraswat propose *idempotent* work-stealing [65], a relaxation allowing a task to be taken *at least once*, instead of *exactly once*. Three idempotent work-stealing algorithms are presented in [65], where tasks are inserted/extracted in different orders. The relaxation allows each of the algorithms to circumvent the impossibility result in [8] in its Put and Take operations as they use only Read/Write instructions and are devoid of Read-After-Write synchronization patterns. However, Steal uses Compare&Swap. Moreover, Put requires that some Write instructions are not reordered, and Steal requires that some Read instructions are not reordered either, and thus fences are required when the algorithms are implemented. However, fences between Read (resp. Write) instructions are usually not too costly. As for progress guarantees, Put and Take are wait-free while Steal is only nonblocking.

Morrison and Afek consider the TSO model [81] and present two work-stealing algorithms in [68] whose Put operation is wait-free and uses only Read/Write instructions, and Take and Steal are either nonblocking and use Compare&Swap, or blocking and use a *lock*. The algorithms are non-trivial adaptations of the well-known Cilk THE and Chase-Lev work-stealing algorithms [22, 31] to the TSO model. Generally speaking, in this model, Write (resp. Read) instructions cannot be reordered; hence, fences among Write (resp. Read) instructions are unnecessary. Additionally, each process has a local buffer where its Write instructions are stored until they are eventually propagated to the main memory (in FIFO order). Reordering some Write (resp. Read), instructions of Morrison and Afek's algorithms compromise correctness. However, TSO prevents this from happening. To avoid Read-After-Write patterns, they assume bounded size Write buffers.

4.2 Work-Stealing with Multiplicity

Work-stealing with *multiplicity* is a relaxation in which, roughly speaking, every task is extracted *at least once*. If several operations extract the task, they must be *concurrent*. In the formal set-sequential specification below (and its variant in the next section), tasks are inserted/extracted in FIFO order. The definition can be easily adapted to encompass other orders (e.g., LIFO). Figure 4.1 depicts an example of a set-sequential execution of work stealing with multiplicity, where concurrent

Take/Steal operations extract the same task.

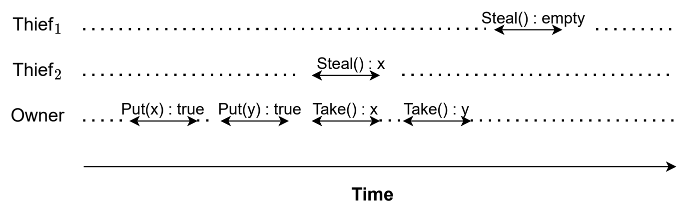


Figure 4.1: A set-sequential execution of work-stealing with multiplicity.

Definition 4.1 (FIFO Work-Stealing with Multiplicity)
The universe of tasks that the owner can put is $\mathbf{N} = \{1, 2, \ldots\}$, and the set of states Q is the infinite set of finite strings $\mathbf{N}^*$. The initial state is the empty string, denoted ϵ. In state q, the first element in q represents the *head* and the last one the *tail*. The transitions are the following:

1. $\forall q \in Q$, $\delta(q, \mathsf{Put}(x)) = (q \cdot x, \langle \mathsf{Put}(x) : \mathsf{true} \rangle)$.

2. $\forall q \in Q, 0 \leq t \leq n-1$, $x \in \mathbf{N}$, $\delta(x \cdot q, \{\mathsf{Take}(), \mathsf{Steal}_1(), \ldots, \mathsf{Steal}_t()\}) = (q,$ $\{\langle \mathsf{Take}() : x \rangle, \langle \mathsf{Steal}_1() : x \rangle, \ldots, \langle \mathsf{Steal}_t() : x \rangle\})$.

3. $\forall q \in Q$, $1 \leq t \leq n-1$, $x \in \mathbf{N}$, $\delta(x \cdot q, \{\mathsf{Steal}_1(), \ldots, \mathsf{Steal}_t()\}) = (q,$ $\{\langle \mathsf{Steal}_1() : x \rangle, \ldots, \langle \mathsf{Steal}_t() : x \rangle\})$.

4. $\delta(\epsilon, \mathsf{Take}()) = (\epsilon, \langle \mathsf{Take}() : \mathsf{empty} \rangle)$.

5. $\delta(\epsilon, \mathsf{Steal}()) = (\epsilon, \langle \mathsf{Steal}() : \mathsf{empty} \rangle)$.

Let $\mathcal{A}$ be a set-linearizable algorithm for work-stealing with multiplicity. Note that items 2 and 3 in Definition 4.1 and the definition of set-linearizability directly imply that in every execution of $\mathcal{A}$, the number of Take/Steal operations that take the same task is, at most, the number of processes in the system, as the operations must be pairwise concurrent to be set-linearized together. Furthermore, every *sequential* execution of $\mathcal{A}$ (i.e., an execution where operations do not overlap in time) is a sequential execution of non-relaxed work-stealing, as every operation is linearized alone, by definition of set-linearizability. Formally, every sequential execution of $\mathcal{A}$ is a sequential execution of (FIFO) work-stealing. We call this property *sequentially-exact*. Thus, without contention, $\mathcal{A}$ provides an exact solution for (FIFO) work-stealing.

Remark 1. Every set-linearizable algorithm for work-stealing with multiplicity is sequentially-exact.

4.2.1 Work-Stealing with Multiplicity from MaxRegister

We show that work-stealing with multiplicity can be reduced to a single instance of a MaxRegister object (defined below). Together with any Read/Write wait-free linearizable algorithms for MaxRegister, our algorithm provides a Read/Write algorithm for work-stealing with multiplicity. We argue that using the MaxRegister algorithm of Aspnes, Attiya, and Censor-Hillel [7], the resulting work-stealing algorithm with multiplicity has logarithmic step complexity, devoid of Read-After-Write synchronization patterns. The algorithm presented in this section seems to have no practical implications. However, it will lead us to our efficient, fully fence-free Read/Write work-stealing algorithm with constant step complexity in all its operations.

Figure 4.2 contains WS-MULT, a set-linearizable algorithm for work-stealing with multiplicity. The algorithm uses a linearizable wait-free MaxRegister object, which provides two operations: MaxRead, which returns the maximum value written so far in the object, and MaxWrite, which writes a new value only if it is greater than the largest value written so far.

In WS-MULT, the tail of the queue is stored in the persistent local variable *tail* of the owner, while the head is stored in the shared MaxRegister $Head$. Persistent means that the local variable retains its value between invocations to operations. When the owner wants to put a new task, it first locally increments *tail* (Line 1) and then stores the task in the corresponding entry of $Tasks$ and marks one more entry with $\perp$ (Line 2); $\perp$ indicates lack of tasks. Recall that the notation in Line 2 (instructions between brackets) denotes that the instructions can be executed in any order. When the owner wants to take a task, it first reads the current head of the queue from $Head$ (Line 4). Then, if there are tasks available (i.e., the head is less or equal to the tail), it reads the task at the head, updates $Head$, and finally returns the task (Lines 6 and 7); if there are no tasks available, the owner returns empty (Lines 9). When a thief wants to steal a task, it first reads the current value of $Head$ (Line 10) and then reads that entry of $Tasks$ (Line 11). If it reads a task (i.e. a non-$\perp$ value), it updates $Head$ and then returns the task (Lines 13 and 14). Otherwise, all tasks have been extracted, and it returns empty (Line 16).

The semantics of MaxWrite guarantees that $Head$ contains the current value of the head at all times, as a "slow" process cannot "move back" the head by writing a smaller value in $Head$ (in Lines 6 or 13). Thus, the MaxRegister $Head$ acts as a sort of barrier in the algorithm. Two Take/Steal operations can return the same task only if concurrent, reading the same value from $Head$.

Shared Variables:
  $Head$: atomic MaxRegister object initialized to 1
  $Tasks[1, 2, \ldots]$: array of atomic Read/Write objects with the first two objects initialized to $\perp$
Local Variables of the Owner:
  $tail \leftarrow 0$

Operation Put(x):
(01) $tail \leftarrow tail + 1$
(02) $\{Tasks[tail]$.Write(x), $Tasks[tail + 2]$.Write($\perp$)$\}$
(03) **return** true
end Put

Operation Take():
(04) $head \leftarrow Head$.MaxRead()
(05) **if** $head \leq tail$ **then**
(06) $\{x \leftarrow Tasks[head]$.Read(), $Head$.MaxWrite($head + 1$)$\}$
(07) **return** x
(08) **end if**
(09) **return** empty
end Take

Operation Steal():
(10) $head \leftarrow Head$.MaxRead()
(11) $x \leftarrow Tasks[head]$.Read()
(12) **if** $x \neq \perp$ **then**
(13) $Head$.MaxWrite($head + 1$)
(14) **return** x
(15) **end if**
(16) **return** empty
end Steal

Figure 4.2: WS-MULT: a MaxRegister-based set-linearizable algorithm for work-stealing with multiplicity.

Note that if only the first object in $Tasks$ is initialized to $\perp$ (and hence Put has modified accordingly), a thief may read a value from $Tasks$ that has not been written by the owner: in execution with a single Put(x) operation, the steps in Line 2 could be executed $Tasks[1]$.Write(x) first and then $Tasks[2]$.Write($\perp$) with a sequence of two Steal operations completing in between, resulting in the second operation reading

$Tasks[2]$, which has not been written yet by the owner, and might contain a value distinct from $\perp$.

Theorem 4.1 WS-MULT (Figure 4.2) is a set-linearizable wait-free algorithm for work-stealing with multiplicity, using Read/Write instructions and a single instance of a linearizable MaxRegister object. Moreover, all operations execute a constant number of Read/Write instructions, invoke a constant number of operations of the MaxRegister object, and Put is Read/Write.

Proof:
The algorithm is wait-free since the MaxRegister object is assumed to be wait-free, and none of the operations executes a loop. Observe that Put uses only Read/Write. Before proving that WS-MULT is set-linearizable, we first observe that at any time, the thieves read the range of $Tasks$ that the owner has already initialized; more specifically, every Steal operation reads from $Tasks$ (in Line 11), a value that was written by the owner, either $\perp$ or a task.

In any given execution, the range $Tasks[Head, Head + 1, \ldots,]$ contains a (possibly empty) sequence of tasks followed by at least one $\perp$ value, considering the entries in index-ascending order. The claim is true initially as the first two entries if $Tasks$ are initialized to $\perp$. Every time the owner stores a new task, it initializes a new entry of $Task$ to $\perp$ (Line 2); hence the claim holds at any time, as $Head$ is incremented only if the owner or a thief reads that $Tasks[Head]$ contains a non-$\perp$ value (Lines 6 or 13). Note that the order of the instructions in Line 2 is irrelevant.

We now prove that WS-MULT is set-linearizable. Consider any finite execution E of it. Since we already argued the algorithm is wait-free, there is a finite extension of E in which all its operations are completed, and no new operations start. Thus, we can assume that there are no pending operations in E.

First, note that the semantics of MaxWrite implies that no pair of non-concurrent Take/Steal operations return the same task: if two operations are not concurrent, then the first one increments the value of $Head$, the second operation cannot read the same tasks from $Tasks$. Thus, we have:

Remark 2. If a task is returned by more than one Take/Steal operation, these operations are pairwise concurrent. Thus, two distinct Take operations cannot return the same task.

The main observation for the set-linearizability proof is that, at any time, the state of the object is represented by the tasks in the range $Tasks[Head, Head+1, \ldots]$, i.e., the sequence of non-$\perp$ values (in index-ascending order) written by the owner in that range. The set-linearization $\mathsf{SetLin}(E)$ of E is obtained as follows:

- Every Put operation is set-linearized *alone* (i.e., in a concurrency class containing only that operation) placed at its step corresponding to $Tasks[tail].\mathsf{Write}(x)$ in E (Line 2).

- For every task returned by at least one $\mathsf{Take}/\mathsf{Steal}$ operation, all operations returning the task are set-linearized in the same concurrency class placed at the first step e of E that corresponds to $Head.\mathsf{MaxWrite}(head + 1)$ (either Line 6 or 13) among the steps of the operations. Note that e occurs between the invocation and response of every operation in the concurrency class. Since the operations return the same task, all of them execute the $\mathsf{MaxRead}$ steps in Lines 4 or 10 before e, and, by definition, e appears in E before any other operation executes its step corresponding to $Head.\mathsf{MaxWrite}(head{+}1)$. Observe that the order in which the instructions in Line 6 are executed is irrelevant.

- Every Take operation returning empty is set-linearized alone, placed at its step in E corresponding to $Head.\mathsf{MaxRead}()$ (Line 4).

- Every Steal operation returning empty is set-linearized alone, placed at its step in E corresponding to $Head.\mathsf{MaxRead}()$ (Line 10).

Every concurrency class of $\mathsf{SetLin}(E)$ is placed at a step of E that lies between the invocation and response of each operation in the concurrency class, which immediately implies that $\mathsf{SetLin}(E)$ respects the partial order $<_E$ of E. Thus, to conclude that $\mathsf{SetLin}(E)$ is a set-linearization of E, we need to show that it is indeed a set-sequential execution of work-stealing with multiplicity.

First, a task can be extracted by a $\mathsf{Take}/\mathsf{Steal}$ operation only if the Put operation that stores the task executes its step corresponding to $Tasks[tail].\mathsf{Write}(x)$ (in Line 2) before the $\mathsf{Take}/\mathsf{Steal}$ operation reads the entry of $Tasks$ where the task is stored. Thus, in $\mathsf{SetLin}(E)$, every task is inserted before it is extracted.

Now, Put stores tasks in $Tasks$ in index-ascending order. Due to the semantics of $\mathsf{MaxRegister}$, $Head$ never "moves back", i.e., it only increases by one at a time, and hence Take and Steal extract tasks in index-ascending order too. Tasks in SetLin(E) are inserted/extracted in FIFO order.

More specifically, for any concurrency class C of SetLin(E) with Take/Steal operations that return the same task x, right before the step e of E where C is set-linearized, we have that x is a task with the smallest index (left-most) in the range $Tasks[Head, Head + 1, \ldots]$, and thus indeed the operations in C get the "oldest" task in the object.

It only remains to be argued that any Take/Steal operation that returns empty, does so correctly, i.e., each of these operations is set-linearized at a step of E at which $Tasks[Head, Head + 1, \ldots]$ is empty, i.e., all its entries initialized by the owner in that range contain $\perp$.

Consider any Take operation in E that returns empty. Observe that this can happen only if the owner sees that $head > tail$, namely, the conditional of Line 5 is not satisfied. This is possible only when no task has been inserted, or all items have been extracted, and hence $Tasks[Head, Head + 1, \ldots]$ is empty. Consider any Steal operation in E that returns empty. This is possible only when the thief reads $\perp$ from $Tasks$ in Line 11, and since we already argued that the owner inserts tasks in ascending order, the sequence $Tasks[Head, Head + 1, \ldots]$ is empty.

We conclude that SetLin(E) is a valid set-sequential execution of work-stealing with multiplicity, and as it respects the partial order $<_E$ of E, we have that it is a set-linearization of E, and therefore WS-MULT is set-linearizable. The theorem follows. ■

If we replace $Head$ with the Read/Write wait-free linearizable MaxRegister algorithm of Aspnes, Attiya, and Censor-Hillel [7], whose step complexity is $O(\log m)$, where $m \geq 1$ is the maximum value that can be stored in the object, the step complexity of WS-MULT is bounded wait-free with logarithmic step complexity too. In the resulting algorithm, at most m tasks can be inserted; the actual value of m is application-dependent. Since the algorithm does not use Read-After-Write synchronization patterns (as explained in the proof of Theorem 4.2), the resulting algorithm does not use those patterns either.

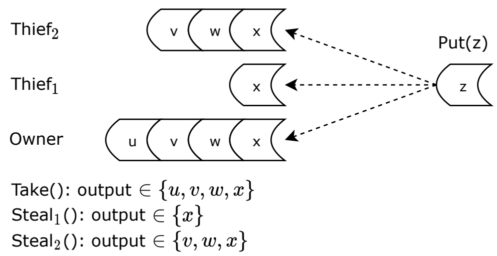
Theorem 4.2 If $Head$ is an instance of the Read/Write wait-free linearizable MaxRegister algorithm of Aspness, Attiya, and Censor-Hillel [7], WS-MULT is set-linearizable, fully Read/Write and Take and Steal have step complexity $O(\log m)$, where m denotes the maximum number of tasks that can be inserted in an execution. Furthermore, no operation uses Read-After-Write synchronization patterns.

Proof:
The algorithm remains set-linearizable by the composability of set-linearizability [15]. While the step complexity of Put is $O(1)$, the step complexity of Take and Steal is $O(\log m)$ as the step complexity MaxRead and MaxWrite of the MaxRegister algorithm [7] is $O(\log m)$.

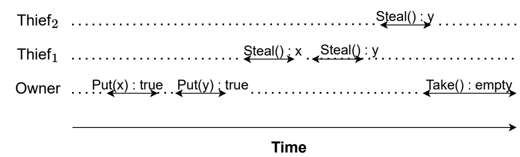
We now argue that Take and Steal do not use Read-After-Write synchronization patterns. The reason is that the MaxRegister algorithm does not use this synchronization mechanism. Roughly speaking, the algorithm consists of a binary tree of height $O(\log m)$ with an atomic bit in each node. When a process wants to perform MaxRead, it reads the bits in the path of the tree from the root to a leaf and then returns a value according to the leaf it reached; the next node in the path the process reads depends on the current node's value. When a process wants to perform MaxWrite, it reads the bits in a path from the root to a leaf, which is a function of the binary representation of the value the process wants to write; then, if the new value is larger than the current one, in a bottom-up manner, it writes 1 in every node in the path with 0 (for the algorithm to be linearizable, the writes should occur in this order. Hence the algorithm is not fence-free). Thus, we have that MaxRead consists of a sequence of reads, and MaxWrite consists of a sequence of reads followed by a (possibly empty) sequence of writes. Therefore, Take/Steal of WS-MULT consists of a sequence of reads followed by a (possibly empty) sequence of writes, and thus the operation does not use Read-After-Write synchronization patterns. ■

4.3 Work-Stealing with Weak Multiplicity

A logarithmic step complexity of the Take operation is prohibitive in practical settings. Ideally, we would like to have constant step complexity in all operations and use simple synchronization mechanisms if possible. In this section, we propose a variant of work-stealing with multiplicity that admits fully Read/Write fence-free algorithms with constant step complexity in all its operations. Intuitively, the variant requires that every task is extracted at least once, but now every process extracts a task *at most once*, hence Take/Steal operations returning the same task *might not* be concurrent. Therefore, the relaxation retains the property that the number of operations that can extract the same task is, at most, the number of processes in the system. We call this relaxation *weak multiplicity*.



(a) A schematic view of work-stealing with weak multiplicity.



(b) A sequential execution of work-stealing with weak multiplicity.

Figure 4.3: Schematic view and sequential execution of work-stealing with weak multiplicity.

Figure 4.3a depicts a schematic view of work stealing with weak multiplicity. Intuitively, each process has its own *virtual* queue of tasks at any state. When the owner inserts a new task, it *atomically* places the task in all virtual queues, as shown in the figure. Therefore, for any pair of processes' virtual queues, one is a suffix of the other. A Take/Steal operation can return any task from its virtual queue that is no "beyond" the first task of the *shortest* virtual queue in the state. In the example depicted in the figure 4.3a, a Steal operation of thief p_1, denoted Steal$_1()$, can return only x, as p_1 has the shortest virtual queue in the state. In contrast, a Take operation can return any task in its virtual queue and remove all preceding tasks. E.g., if w is returned, then u and v are removed from the virtual queue, which leaves the owner's virtual queue containing only x and z. Note that tasks are extracted in FIFO order. This property guarantees that every task is taken at least once. In our algorithms, *all* virtual queues are implemented using a single queue. Figure 4.3b shows an example of sequential execution of work-stealing with weak multiplicity. Note that Take/Steal operations can get the same task, although they are not concurrent.

The next *sequential* specification formally defines work-stealing with weak multiplicity. Without loss of generality, the specification assumes that p_0 is the owner and each invocation/response of thief p_i is subscripted with its index, which belongs to the set $\{1, \ldots, n-1\}$.

Definition 4.2 (FIFO Work-Stealing with Weak Multiplicity)
The universe of tasks that the owner can put is $\mathbf{N} = \{1, 2, \ldots\}$, and the set of states Q is the infinite set of n-vectors of finite strings $\mathbf{N}^* \times \ldots \times \mathbf{N}^*$, with the property that for any two pairs of strings in a vector/state, one of them is a *suffix of the other*. The initial state is the vector with empty strings, $(\epsilon, \ldots, \epsilon)$. The

transitions are the following:

1. $\forall(t_0,\ \ldots,\ t_{n-1}) \in Q,\ \delta((t_0,\ \ldots,\ t_{n-1}), \mathsf{Put}(x)) = ((t_0 \cdot x,\ \ldots,\ t_{n-1} \cdot x), \langle \mathsf{Put}(x) : \mathsf{true} \rangle)$.

2. $\forall(t_0,\ \ldots,\ t_{n-1}) \in Q$ such that $t_0 = x_1 \cdots x_j \cdot q \neq \epsilon$ with $j \geq 1$ and $t_j \cdot q$ being the shortest string in the state (possibly with $x_j \cdot q = \epsilon \cdot \epsilon = \epsilon$), $\delta((t_0,\ \ldots,\ t_{n-1}), \mathsf{Take}()) = \{((\widehat{t_0},\ t_1,\ \ldots,\ t_{n-1}), \langle \mathsf{Take}() : x_k \rangle)\}$, where $k \in \{1,\ \ldots,\ j\}$ and $\widehat{t_0} = x_{k+1} \cdots x_j \cdot q$.

3. $\forall(t_0,\ \ldots,\ t_{n-1}) \in Q$ such that $t_i = x_1 \cdots x_j \cdot q \neq \epsilon$ with $j \geq 1$, $i \in \{1,\ \ldots,\ n-1\}$ and $t_j \cdot q$ being the shortest string in the state (possibly with $x_j \cdot q = \epsilon \cdot \epsilon = \epsilon$), $\delta((t_0,\ \ldots,\ t_{n-1}),\ \mathsf{Steal}_i()) = \{((t_0,\ \ldots,\ t_{i-1},\ \widehat{t_i},\ t_{i+1},\ \ldots,\ t_{n-1}),\ \langle \mathsf{Steal}_i() : x_k \rangle)\}$, where $k \in \{1,\ \ldots,\ j\}$ and $\widehat{t_i} = x_{k+1} \cdots x_j \cdot q$.

Observe that the second and third items in Definition 4.2 correspond to non-deterministic transitions in which a Take/Steal operation can extract any task in $\{x_1, \ldots, x_j\}$ of the invoking process' virtual queue (and removes all preceding tasks); the value returned value can be ϵ when the shortest string in the state is ϵ ($x_j \cdot q = \epsilon \cdot \epsilon = \epsilon$ in the definition). Moreover, note that if $x_1 \cdots x_j \cdot q = \epsilon$, and hence $x_1 \cdots x_j \cdot q$ is the shortest string in the state, then the operation is forced to return ϵ. Furthermore, the definition guarantees that every task is extracted at least once because every Take/Steal operation can only return a task that is not "beyond" the first task in the shortest string of the state.

The specification of work-stealing with concurrent weak multiplicity admits trivial solutions. A simple solution is obtained by replacing *Head* in WS-MULT with one local persistent variable *head* per process (each initialized to 1); Put remains the same, and Take and Steal instead of reading from *Head*, they locally read the current value of *head* and increment it whenever a task is taken. It is not hard to verify that the resulting algorithm is indeed linearizable.

Remark 3. To avoid this kind of simple solution (which can be very inefficient in practice, as every process might execute every task), we restrict our attention to *sequentially-exact* algorithms; namely, every sequential execution of the algorithm is a sequential execution of the specification of (FIFO) work-stealing.[2] It is easy to see

[2]Alternatively, work-stealing with weak multiplicity can be specified using the interval-linearizability formalism in [15], which allows us to specify that a Take/Steal operation can exhibit a non-exact behavior only in the presence of concurrency. Interval-linearizability would imply that any interval-linearizable solution provides an exact solution in sequential executions. Roughly speaking, in interval-linearizability, operations are linearized at intervals that can overlap each

that the algorithm described above does not have this property.

Finally, we stress that, differently from work-stealing with multiplicity, two distinct non-concurrent Take/Steal operations can extract the same task in an execution, which can happen only if *some* operations are concurrent in the execution due to the sequentially-exact requirement. Particularly, in our algorithms, this relaxed behavior can occur when processes are concurrently updating the head of the queue.

4.3.1 Read/Write Fence-Free Work-Stealing with Multiplicity

This subsection presents WS-WMULT, a Read/Write fence-free linearizable algorithm for work-stealing with weak multiplicity. The algorithm is obtained by replacing the linearizable MaxRegister object in WS-MULT, *Head*, with a linearizable RangeMaxRegister object, a relaxation of MaxRegister with a RMaxRead operation that returns a value in a *range* of values that have been written to the register; the range always includes the maximum value written so far.

We present a RangeMaxRegister algorithm that is nearly trivial. However, it allows us to efficiently solve work-stealing with weak multiplicity, with implementations exhibiting good performance in some practical settings, as seen in Section 6.1. As we have mentioned above, to avoid trivial solutions, we focus on sequentially-exact linearizable algorithms for RangeMaxRegister, i.e., each sequential execution of the algorithm is a sequential execution of MaxRegister. [3] When WS-WMULT is combined with this algorithm, it becomes Read/Write, fence-free, linearizable, sequentially-exact, and wait-free with constant step complexity.

Intuitively, in RangeMaxRegister, each process has a *private* MaxRegister, and whenever it invokes RMaxRead, the result lies in the range defined by the value of its private MaxRegister and the maximum among the private MaxRegister values in the state. In the sequential specification of RangeMaxRegister, each invocation/response of process p_i is subscripted with its index i.

Definition 4.3 (RangeMaxRegister)
The set of states Q is the infinite set of n-vectors with natural numbers, with vector $(1, \ldots, 1)$ being the initial state. $\forall\, q = (r_0, \ldots, r_{n-1}) \in Q$ and $i \in \{0, \ldots, n-1\}$, the transitions are the following:

1. **If** $x > r_i$ **then**
   $\delta(q, \mathsf{RMaxWrite}_i(x)) = ((r_0,\ \ldots,\ r_{i-1},\ x,\ r_{i+1},\ \ldots,\ r_{n-1}), \langle \mathsf{RMaxWrite}_i(x) :$

other.

[3] Again, this can be alternatively specified through interval-linearizability.

true$\rangle$),
otherwise
$\delta(q, \mathsf{RMaxWrite}(x)) = (q, \langle \mathsf{RMaxWrite}(x) : \mathsf{true}\rangle)$

2. $\delta(q, \mathsf{RMaxRead}_i()) = \{(q, \langle \mathsf{RMaxRead}_i() : x\rangle)\}$, **where** $x \in \{r_i,\ r_i+1,\ \ldots,\ \max(r_0,\ \ldots,\ r_{n-1})\}$.

As already mentioned, WS-WMULT is the algorithm obtained by replacing $Head$ in WS-MULT with an atomic RangeMaxRegister object initialized to 1 (hence MaxRead and MaxWrite are replaced by RMaxRead and RMaxWrite, respectively).

Theorem 4.3 WS-WMULT is a sequentially-exact linearizable wait-free algorithm for work-stealing with weak multiplicity, using Read/Write instructions and a single sequentially-exact linearizable RangeMaxRegister object. Moreover, all operations execute a constant number of Read/Write instructions and invoke a constant number of operations of the RangeMaxRegister object, and Put is Read/Write.

Proof:
All algorithm operations are wait-free since the RangeMaxRegister object is assumed to be wait-free, and none of the operations executes a loop. Note that Put uses only Read/Write.

As in the proof of Theorem 4.1, it can be argued that thieves read values from $Tasks$ that have been written by the owner.

Consider any finite execution E of the algorithm. Since the algorithm is wait-free, there is a finite extension of E in which all its operations are completed, and no new operations start. Thus, we can assume that there are no pending operations in E.

Proving that E is linearizable is quite straightforward. It is enough to observe that, at any step of E, the state of the object is encoded in the state of $Head$. Let $(r_0, \ldots, r_{n-1})$ be the state of $Head$ at a given step of E. Then, the state of the object is $(t_0, \ldots, t_{n-1})$ with each t_i being the finite sequence of tasks in the range $Tasks[r_i, r_{i+1}, \ldots]$ (i.e. the sequence of non-$\perp$ values written by the owner, in index-ascending order). Thus, in a linearization of E, a $\mathsf{Put}(x)$ operation is linearized at its step $Tasks[tail].\mathsf{Write}(x)$ in Line 2. In contrast, a Take/Steal operation is linearized at its step $Head.\mathsf{RMaxRead}()$ in Line 4/Line 10 (observe that the non-deterministic choice in a transition with a Take/Steal operation is resolved with the outcome of RMaxRead). Therefore, we conclude that every execution of WS-WMULT is linearizable, and thus, the algorithm is linearizable too.

Since $Head$ is assumed to be sequentially-exact, in sequential executions, the algorithm behaves exactly as WS-MULT (exchanging RMaxRead and RMaxWrite with MaxRead and MaxWrite, respectively). Thus, by Remark 1 and since WS-MULT is set-linearizable, the sequential executions of the algorithm are sequential executions of work-stealing. Thus, the algorithm is sequentially-exact. The theorem follows.

■

Figure 4.4 contains a simple sequentially-exact linearizable wait-free algorithm for RangeMaxRegister. All processes share a single Read/Write object R, and each process has a local persistent variable r. The idea is straightforward: each process locally stores in r the maximum value it is aware of; whenever it discovers a new largest value in RMaxWrite, it writes it in r and R, and since R might not have the largest value, it returns the maximum among r and R in RMaxRead.

```
Shared Variables:
        R: atomic Read/Write object initialized to 1
Local Variables of a Process:
        r ← 1

Operation RMaxWrite(x):
        r ← max{r, R.Read()}
        if x > r then
            {r ← x, R.Write(x)}
        end if
        return true
end RMaxWrite

Operation RMaxRead():
        r ← max{r, R.Read()}
        return r
end RMaxRead
```

Figure 4.4: A linearizable wait-free algorithm for RangeMaxRegister.

Theorem 4.4 The algorithm in Figure 4.4 is a sequentially-exact linearizable wait-free and fence-free algorithm for RangeMaxRegister using only Read/Write instructions and with constant step complexity in all its operations.

Proof:
It is clear from its pseudocode that the algorithm is Read/Write, wait-free and fence-free, and each operation has constant step complexity.

Consider any finite execution E of the algorithm. Since the algorithm is wait-free, there is a finite extension of E in which all its operations are completed, and no new operations start. Thus, we can assume that there are no pending operations in E.

To prove linearizability, it is enough to observe that at any step of E, the state of the object is $(r_0, \ldots, r_{n-1})$, where r_i is the value stored in the local persistent variable r of process p_i at that moment. Thus, a RMaxWrite (x) operation with $x > r$ (hence the condition in Line 2 is true) is linearized at its step R.Write(x) in Line 3; if $x \leq r$, the operation is linearized at line 1, i.e., at the beginning of the operation. A RMaxRead () operation is linearized at its step R.Read() in Line 6; note that the operation returns a value between the value in r and the maximum among the r's local variables since that is the maximum value R can store at that time. Thus, the algorithm is linearizable.

Suppose now that E is sequential. By induction of the number of operations, it is easy to show that R always contains the maximum value. Thus, E is a sequential execution of MaxRegister, and therefore the algorithm is sequentially-exact. The theorem follows. ■

We are now able to present the main result of this chapter:

Theorem 4.5 If $Head$ is replaced with an instance of the algorithm in Figure 4.4, WS-WMULT is Read/Write, fence-free, wait-free, sequentially-exact, and linearizable with constant step complexity in all its operations.

```
Shared Variables:
    Head: atomic Read/Write object initialized to 1
    Tasks[1, 2, ...]: array of atomic Read/Write objects
                with the first two objects initialized to ⊥
Local Variables of the Owner:
    head ← 1
    tail ← 0
Local Variables of a Thief:
    head ← 1

Operation Put(x):
    tail ← tail + 1
    {Tasks[tail].Write(x), Tasks[tail + 2].Write(⊥)}
    return true
end Put

Operation Take():
    head ← max{head, Head.Read()}
    if head ≤ tail then
        {x ← Tasks[head].Read(), Head.Write(head + 1)}
        head ← head + 1
        return x
    end if
    return empty
end Take

Operation Steal():
    head ← max{head, Head.Read()}
    x ← Tasks[head].Read()
    if x ≠ ⊥ then
        Head.Write(head + 1)
        head ← head + 1
        return x
    end if
    return empty
end Steal
```

Figure 4.5: WS-WMULT algorithm with the RangeMaxRegister algorithm in Figure 4.4 inlined.

Proof:
By composability of linearizability [45], the algorithm remains linearizable when $Head$ is replaced with an instance of the algorithm in Figure 4.4. The algorithm

is fully Read/Write and wait-free because Put uses only Read/Write instructions and the RangeMaxRegister algorithm in Figure 4.4 is fully Read/Write and wait-free, by Theorem 4.4. The step complexity of Put is $O(1)$. The step complexity of Take and Steal is $O(1)$ because the RangeMaxRegister algorithm in Figure 4.4 has constant step complexity. It is not difficult to verify that the resulting algorithm does not require any specific ordering among its steps beyond what is implied by data dependence. Therefore, it is fully fence-free. The algorithm is sequentially-exact because the algorithm in Figure 4.4 is sequentially-exact. The theorem follows. ■

Figure 4.5 contains an optimized WS-WMULT algorithm with the RangeMaxRegister algorithm in Figure 4.4 inlined. Since Take and Steal first RMaxRead from and then RMaxWrite to $Head$, the algorithm remains sequentially exact when removing Line 1 of RMaxWrite in Figure 4.4. Our experimental evaluation in Section 6.1 tested implementations of this algorithm.

4.4 Bounding the Multiplicity

This section discusses simple variants of our algorithms that bound the number of operations that can extract the same task. We only discuss the case of WS-MULT as the variants for WS-WMULT are similar.

Bounding multiplicity We call this variant B-WS-WMULT. The modification consists of an extra array A of the same length as $Tasks$ array, with its first two entries initialized to true. Steal is modified as follows: after Line 12, a thief performs $A[head]$.Swap(false), and it executes Lines 13 and 14 only if the Swap successfully takes the true value in $A[head]$; otherwise, it goes to the Line 10 to start over. The modified algorithm guarantees no two distinct Steal operations take the same task. However, a Take and a Steal can take the same task. Note that Steal is only nonblocking in the modified algorithm. The new algorithm is a set-linear solution to the work-stealing variant with multiplicity (Definition 4.1), where every concurrency class has at most one Take and one Steal that return the same task. The set-linearizability proof is the same as the difference in the sizes of concurrency classes. Figure 4.6 illustrates the changes made to the algorithm in Figure 4.2 to produce the B-WS-WMULT version.

Removing multiplicity The Take operation of B-WS-MULT can be modified similarly to obtain an algorithm for exact (FIFO) work-stealing, i.e., every task is taken *exactly* once (Definition 4.1 with singleton concurrency classes). The modified Take operation is only nonblocking.

Shared Variables:
$Head$: atomic MaxRegister object initialized to 1
$Tasks[1, 2, \ldots]$: array of atomic Read/Write objects with the first two objects initialized to $\perp$
$A[1, 2, \ldots]$: array of booleans, the first two entries initialized to true.
Local Variables of the Owner:
$tail \leftarrow 0$

Operation Put(x):
(01) $tail \leftarrow tail + 1$
(02) $\{Tasks[tail]$.Write(x), $Tasks[tail + 2]$.Write($\perp$)$\}$
(03) **return** true
end Put

Operation Take():
(04) $head \leftarrow Head$.MaxRead()
(05) **if** $head \leq tail$ **then**
(06) $\{x \leftarrow Tasks[head]$.Read(), $Head$.MaxWrite($head + 1$)$\}$
(07) **return** x
(08) **end if**
(09) **return** empty
end Take

Operation Steal():
(10) $head \leftarrow Head$.MaxRead()
(11) $x \leftarrow Tasks[head]$.Read()
(12) **if** $x \neq \perp$ **and** $A[head]$.Swap($false$) **then**
(13) $Head$.MaxWrite($head + 1$)
(14) **return** x
(15) **else**
(16) **go to line 10**
(17) **end if**
(18) **return** empty
end Steal

Figure 4.6: B-WS-WMULT: algorithm obtained from modify the algorithm WS-MULT as specified in Section 4.4.

Multiplicity on demand Consider a variant of Definition 4.1 in which a task x encodes if several processes can execute it, denoted $\mathsf{mult}(x)$, or it has to be executed by a single process, denoted $\neg\mathsf{mult}(x)$ (in practice this can be done, for example, by stealing a bit from the task representation). Then, WS-MULT can be modified to have multiplicity *on demand*. In the modified Take operation, after executing the instructions in Line 6, the owner tests if $\mathsf{mult}(x)$ holds, and if so, it returns x; otherwise, it performs $Tasks[head].\mathsf{Swap}(\top)$, and then returns x only if the Swap successfully takes the task in $Tasks[head]$ (i.e. if it obtains a value distinct from $\bot$ and $\top$), else it goes to Line 4 and starts over. In the modified Steal operation, after Line 11, a thief checks if $x = \bot$, and if so, it returns empty. Then, it checks if $x \neq \top$ and $\mathsf{mult}(x)$ holds, and if so it returns x. Otherwise, $x \neq \top$ and $\neg\mathsf{mult}(x)$ holds, and hence the thief performs $Tasks[head].\mathsf{Swap}(\top)$ and returns x only if Swap returns a value distinct from $\top$, else it goes to Line 10 and starts over. In the resulting algorithm, if $\mathsf{mult}(x)$ holds, x is taken by one operation. The modified Take and Steal operations are only nonblocking. Again, the set-linearizability proof is the same as the difference in the sizes of concurrency classes.

4.5 Coping with realistic assumptions

Base objects of bounded length We have presented our algorithms assuming all base objects can store values of unbounded length. However, we can assume that base objects can store only 64 bit values. This makes our algorithms *bounded* as at most 2^{64} tasks can be inserted, and the task comes from a set of size $2^{64} - 1$ (or $2^{64} - 2$ if $\top$ is used). Arguably, this number is large enough in any application.

Arrays of finite length We also assumed that tasks are stored in an array of infinite lengths. We now discuss two approaches to remove this assumption; both techniques have been used in previous algorithms (e.g. [1, 4, 36, 65, 90]). In both approaches, only the owner modifies the array; hence, no expensive synchronization mechanisms are needed. We only discuss the case of WS-MULT as the other algorithms can be handled similarly.

In the first approach, $Tasks$ is now a pointer, initially pointing to an array of finite fixed length with its two first entries initialized to $\bot$. Each time the owner detects the array is full in the middle of a Put operation (i.e., when the *tail* is larger than the length of the array $Tasks$ points to), it creates a new array A, whose length doubles the length of the current array. Then, it copies the content to A, initializes the following two entries to $\perp$, updates $Tasks$ to let it point to A (depending on the language, it may be necessary to manually release the memory associated with the old array), and finally, it continues executing the algorithm. Although the modified Put operation remains wait-free, its step complexity is unbounded. The set-linearizable proof of the modified algorithms is essentially the same, with the observation that now Steal operations might read the same tasks from different arrays in Line 11 (because the owner was in the middle of updating $Tasks$), which is not a problem because the Steal operations are concurrent.

Instead of storing tasks in the $Tasks$ array, the second approach involves storing pointers to node objects, each node containing a fixed-length array where tasks are stored. In the beginning, $Tasks$ has only one object in its first entry, and the first two entries of the array associated with the object are initialized to $\perp$. When the owner detects that all entries in the array of the object have been used (in the middle of a Put operation), it creates a new node, initializing the first two entries to $\perp$. The pointer of this new node is stored at the last free position of the dynamic array, and it continues executing the algorithm. An index of $Tasks$ is now a tuple: an array-index to the object and a node-index array. Thus, any pair of nodes can be easily compared (first array-indexes, then node-indexes), and increasing an index can be efficiently performed too (if the node-index is the last one, the array-index moves forward, and the node-index is set to one; otherwise, only the node-index is incremented). The modified Put operation remains wait-free with constant step complexity. The set-linearization proof of the modified algorithm remains the same.

The second approach might bring benefits when memory regions are allocated. In the first approach, each time the algorithm resizes its $Tasks$ array, it is necessary to allocate space that doubles the current one. Allocating large arrays in memory might consume a considerable amount of time. In the second approach, using indirect addressing, separate memory regions of relatively small size simulate a large array; typically, memory regions of small size can be quickly allocated. Additionally, memory management can be improved using a "shrinking after growth" pattern, as in the work of Chase and Lev [22].

Memory management Another issue in practical settings is that of memory management. This issue can be delegated to the garbage collector in programming languages like Java. However, a safe and efficient concurrent memory reclamation protocol should be implemented in programming languages without automatic garbage collection. The main problem arises when a process attempts to reclaim a memory region while another uses it. Hence, a synchronization mechanism is required. Below, we briefly describe some well-known memory management protocols that can be used with our algorithms.

An approach is to let each process announce the objects (memory locations) it plans to access and then register its objects to protect them. When a process needs to reclaim an object, it adds the object to a list containing the objects that have been deleted but not yet freed. When the list grows to a certain size, a process initiates a scan to verify if the object is in use. If it is not, the process can reclaim the object. If it is in use, the object will be kept for future reclamation. Popular protocols for memory reclamation based on this approach are *hazard pointers* [63] and *Pass-the-Buck*, which provides a solution to the Repeat Offender Problem [42]. A different approach involves using reference counting, where every object has a counter that increments when a process uses it and decrements when the object is released. Sundell [85] and Valois [88] have employed this approach. Another known approach is epoch-based reclamation. It uses the concept of epochs, which are global markers that indicate whether a given memory region is safe to be reclaimed. Read-Copy-Update (RCU) schema [61] is based on this approach.

4.6 Idempotent $\neq$ Multiplicity

To finish this chapter, in this section, we explain that idempotent work-stealing algorithms [65] do not implement work-stealing with multiplicity, even the weaker variant. While in our relaxations, every process extracts a task at most once, and hence the number of distinct operations that extract the same task is at most the number of processes in the system, in idempotent work-stealing algorithms, a thief can extract the same task an unbounded number of times. Such executions are arguably "corner cases" but show a theoretical difference between multiplicity and idempotency.

Idempotent work-stealing [65] is defined as: every task is extracted *at least once*, instead of *exactly once* (in some order). The three idempotent work-stealing algorithms of Michael, Vechev, and Saraswat [65] insert/extract tasks in FIFO and LIFO orders and as a double-ended queue (the owner puts in and takes from one side, and the thieves steal from the other).

Figure 4.7 shows the FIFO idempotent work-stealing algorithm [65]. The algorithm stores the tasks in a shared array *tasks*, and shared integers *head*, and *tail* indicate the positions of the head and the tail. For every integer $z > 0$, we describe an execution of the algorithm in which, for every $k \in \{1, \dots, z\}$, there is a task that is extracted by $\Theta(k)$ distinct operations (possibly by the same thief), with only one

```
   Structures:
     Task: task information
     TaskArrayWithSize:
       size: integer
       array: array of Task
     FifoIwsq:
       head: integer;
       tail: integer;
       tasks: TaskArrayWithSize

   constructor FifoIwsq(integer size) {
     head := 0;
     tail := 0;
     tasks := new TaskArrayWithSize(size);
   }

   void put(Task task) {
     Order write at 4 before write at 5
1:   h := head;
2:   t := tail;
3:   if (t = h+tasks.size) {expand(); goto 1;}
4:   tasks.array[t%tasks.size] := task;
5:   tail := t+1;
   }
```

```
   TaskInfo take() {
1:   h := head;
2:   t := tail;
3:   if (h = t) return EMPTY;
4:   task := tasks.array[h%tasks.size];
5:   head := h+1;
6:   return task;
   }

   TaskInfo steal() {
     Order read in 1 before read in 2
     Order read in 1 before read in 4
     Order read in 5 before CAS in 6
1:   h := head;
2:   t := tail;
3:   if (h = t) return EMPTY;
4:   a := tasks;
5:   task := a.array[h%a.size];
6:   if !CAS(head,h,h+1) goto 1;
7:   return task;
   }

   void expand() {
     Order writes in 2 and 4 before write in 5
     Order write in 5 before write in put:5
1:   size := tasks.size;
2:   a := new TaskArrayWithSize(2*size);
3:   for i = head:tail−1,
4:     a.array[i%a.size] := tasks.array[i%tasks.size];
5:   tasks := a;
   }
```

Figure 4.7: Idempotent FIFO work-stealing [65].

of them being concurrent with the others.

1. Let the owner execute alone z times Put. Thus, there are z distinct tasks in $tasks$.

2. Let $r = z$.

3. The owner executes Take and stops before executing Line 5, i.e. it is about to increment $head$.

4. In some order, the thieves sequentially execute r Steal operations; note these Steal operations return the r tasks in $tasks[0, \ldots, r-1]$.

5. We now let the owner increment $head$. If $r > 1$, go to step 3 with r decremented by one; else, end the execution.

Observe that in the execution just described, the task in $tasks[i]$, $i \in \{0, \ldots, z-1\}$, is extracted by a Take operation and by $i+1$ distinct non-concurrent Steal operations (possible by the same thief). Thus, the task is extracted $\Theta(i)$ distinct times. Since z is any positive integer, we conclude there is no bound on the number of times a task can be extracted.

A similar argument works for the other two idempotent work-stealing algorithms. Ultimately, this happens in all algorithms because tasks are not marked as taken in the shared array where they are stored. Thus, when the owner takes a task and experiences a delay before updating the head/tail, all concurrent modifications of the head/tail performed by the thieves are overwritten once the owner completes its operation, leaving all taken tasks ready to be retaken. This situation is avoided in our algorithms by marking the entries of the $Tasks$ array as taken and with the help of MaxRegister and RangeMaxRegister.

CHAPTER 5

Case Study 2: Modular Baskets Queue

In this chapter, we want to take a modular approach to building concurrent queues with multi-producer and multi-consumer semantics. In simple words, we want to think about a queue as a set of parts that can assembled, where each part must satisfy a specification without matters how it is built. The basic design of a modular queue can be thought as two objects to manipulate the *head* and the *tail*, a set of *container* objects to store the items in the queue, and a set of well defined operation to *enqueue* and *dequeue* items. In this way, we can define a set of specifications that these modules must satisfy and design specific algorithms for them.

For the modular concurrent queue, we take up the idea of *baskets* as the containers to store the items. Hoffman, Shalev, and Shavit [46] proposed a variant of the Michael-Scott queue [64] with the objective of reducing the queue's Compare&Swap contention. We can think of each basket as a group of concurrently enqueued items. Items in the same group can be dequeued in any order. This allows that items from different groups can be inserted in parallel. In the work of Ostrovsky and Morrison [74], the concept of the basket was defined explicitly and more rigorously. They proposed an abstract data type for the basket, which allows different basket implementations. Our work goes in this direction, however, our basket specification provides stronger guarantees.

In the case of the objects for manipulating the head and the tail, we propose a novel object we call *load-link/increment-conditional* (LL/IC). This object can be implemented using Read/Write instructions instead of more sophisticated Read-Modify-Write instructions. This design approach can help build more scalable queues using the same interfaces and distinct implementations. The different modules of the queue can be seen as black boxes. In a similar fashion to Chapter 4, Chapter 6 presents an experimental evaluation of the algorithms presented in this chapter and their results.

5.1 Introduction

Concurrent multi-producer/multi-consumer FIFO queues are fundamental shared data structures ubiquitous in all sorts of systems. Several concurrent queue shared-memory implementations have been proposed for over three decades. Despite these efforts, even state-of-the-art concurrent queue algorithms scale poorly; namely, as the number of process grows, the latency of queue operations grows at least linearly on the number of process.

One of the main reasons for the poor scalability is the high contention in the Read-Modify-Write instructions, such as Compare&Swap or Fetch&Increment, that manipulate the head and the tail [24, 25, 46, 55, 56, 64, 66, 67, 74, 90]. The latency of any contended such instruction is linear in the number of contending processes since every instruction acquires exclusive ownership of its location's cache line, and these acquisitions are serialized by the cache coherence protocol [74]. The best-known queue implementations [67, 90] exploit the semantics of the Fetch&Increment instruction, that *do not fail* and hence *always make progress*. In many queue implementations, a queue operation *retries* a failed Compare&Swap until it succeeds [24, 25, 55, 56, 64, 66].

An approach that lies in the middle was proposed by Hoffman, Shalev, and Shavit [46], the *baskets queue*. In this queue, failed Compare&Swap operations that occur during an enqueue operation imply concurrency with other enqueue operations. Therefore, the items of all these operations do not need to be ordered and can be stored in a *basket*. The items in the basket can be dequeued in any order. However, when the Compare&Swap fails, it is retried; it is important to note that the use of standard atomic Read-Modify-Write instructions present a scalability issue due to the coherence protocol that serializes write ownership acquisitions. This serialization results in the average cost of an RMW being highly dependent on the number of cores contending for it. Specifically, when C cores contend a Read-Modify-Write instruction, the average cost is about $\frac{C}{2}$ uncontended cache misses, regardless of whether the Read-Modify-Write is a failed or successful Compare&Swap or another Read-Modify-Write type [74]. To overcome this seemingly inherent bottleneck, it has recently proposed a Compare&Swap implementation from *hardware transactional memory*, that exhibits better performance than the same Compare&Swap implementation in some cases [74].

We observe that Read-Modify-Write instructions are unnecessary to consistently manipulate the head or tail. We believe this observation may open the possibility of concurrent queue implementations with better scalability. Concretely, we present a *modular* baskets queue algorithm based on a novel object that we call *load-link/increment-conditional* (LL/IC) that suffices for manipulating the head and the tail of the queue. LL/IC admits implementations that spread contention and use only simple Read/Write instructions. LL/IC is similar to LL/SC, with the difference that IC, if successful, only increments the current value of the linked register. The modular baskets queue stands for its simplicity, with a simple correctness proof.

5.2 The Modular Basket Queue

The Modular Basket Queue appears in Algorithm 5.1. It is based on two concurrent objects: baskets and LL/IC. Roughly speaking, the baskets hold groups of items that were enqueued concurrently and can be dequeued in any order. Two LL/IC objects store the head and the tail of the queue. For simplicity, the baskets queue algorithm is presented using an infinite shared array[1].

Definition 5.1 (K-Basket)
A *basket of capacity* K or "K*-basket*", is a data structure that can hold up to K items. The state of the basket is represented by a pair (S, C) where S is the set of items that can be added concurrently and C is the number of items in the basket. A K*-basket* is initialized to $(\emptyset, 0)$. The sequential specification of a K*-basket* satisfies the following properties:

1. Put(x). Non-deterministically can return FULL (regardless of the state) or OK. If $C = K$, then return FULL, in another case do $S = S \cup \{x\}$, $C = C+1$ and return OK.

2. Take(). If $S \neq \emptyset$, then do $S = S \setminus \{x\}$ and return x, for some $x \in S$, else if $S == \emptyset$ do $C = K$ and return CLOSED.

[1] There are two common approaches to implement an infinite array. One is to use a circular dynamic array that can expand and shrink as needed. The other is to use a linked list where each node contains a finite-sized array. During execution, the list grows on demand, and each node is appended to the list using the Compare&Swap operation to maintain consistency.

```
Shared Variables:
    A[0, 1, ...] = infinite array of basket objects
    HEAD, TAIL = LL/IC objects initialized to 0

Operation Enqueue(x):
while true do
    tail = TAIL.LL()
    if A[tail].Put(x) == OK then
        TAIL.IC()
        return OK
    endif
    TAIL.IC()
endwhile
end Enqueue

Operation Dequeue():
head = HEAD.LL()
tail = TAIL.LL()
while true do
    if head < tail then
        x = A[head].Take()
        if x ≠ CLOSED then return x endif
        HEAD.IC()
    endif
    head' = HEAD.LL()
    tail' = TAIL.LL()
    if head == head' == tail' == tail then return empty endif
    head = head'
    tail = tail'
endwhile
end Dequeue
```

Figure 5.1: The modular basket queue algorithm.

Definition 5.2 (Load-Linked/Increment-Conditional (LL/IC))
The specification of an object of type LL/IC satisfies the next two properties, where the state of the object is an integer R, initialized to 0, and assuming that any process invokes IC only if it has invoked LL before, then the specification for these operations is the following:

1. LL(): Returns the current value in R.

2. IC(): If R has not been increment since the last LL of the invoking process, then do $R = R + 1$; in any case return OK.

The baskets in the original baskets queue algorithm [46] were defined only *implicitly*. Recently, baskets were explicitly defined in the work of Ostrovsky and Morrison [74]. Our basket specification provides stronger guarantees; the main difference is the following: in the work of Ostrovsky and Morrison [74], a basket_empty operation can return either true or false if the basket is not empty, i.e., it allows false negatives. The Take operation of our specification mixes the functionality of basket_empty and basket_extract as if it returns CLOSED, no item will ever be put or taken from the basket.

Consider the definitions 5.1 and 5.2 about the basket and the LL/IC objects, we state the Theorem 5.1 about the Algorithm 5.1 representing the modular basket queue algorithm.

Theorem 5.1 In the Algorithm 5.1, if the objects in the array A, $HEAD$ and $TAIL$ of type LL/IC are linearizable and wait-free, then the algorithm is a linearizable lock-free implementation of a concurrent queue.

Proof:
Since all shared objects are wait-free, every implementation step is completed. Note that every time a Dequeue/Enqueue operation completes a while loop (hence without returning), an Enqueue (resp. a Dequeue) operation successfully puts (resp. takes) an item in (resp. from) a basket. Thus, in an infinite execution, if a Dequeue/Enqueue operation takes infinitely many steps, infinitely many Dequeue/Enqueue operations terminate. Hence, the implementation is lock-free.

We consider the aspect-oriented linearizability proof framework in [39] to prove that the algorithm is linearizable. Assuming that every item is enqueued at most once, it states that a queue implementation is linearizable if each of its finite executions is *free* of four violations. We enumerate the violations and argue that every algorithm execution is free of them.

1. VFresh: A Dequeue operation returns an item not previously inserted by any Enqueue operation. Dequeue operations return items once put in the baskets, and Enqueue operations put items in the baskets. Thus, each execution is free of VFresh.

2. VRepeat: Two Dequeue operations return the item inserted by the same Enqueue operation. The specification of the basket directly implies that every execution is free of VRepeat.

3. VOrd: Two items are enqueued in a certain order, and a Dequeue returns the later item before any Dequeue of the earlier item starts. LL/IC guarantees that if an Enqueue operation enqueues an item, say x, and then a later Enqueue operation enqueues another item, say y, then x and y are inserted in baskets $A[i]$ and $A[j]$, with $i < j$. Then, x is dequeued first because Dequeue operations scan A in index-ascending order. Thus, every execution is free of VOrd.

4. VWit: A Dequeue operation returning empty even though the queue is never logically empty during the execution of the Dequeue operation. An item is logically in the queue if it is in a basket $A[i]$ and $i < TAIL$. When a Dequeue operation returns empty, there is a point in time where no basket in $A[0, 1, \ldots, TAIL - 1]$ contains an item, and hence the queue is logically empty (it might, however, be the case that $A[TAIL]$ does contain an item at that moment). Hence, every execution is free of VWit.

Therefore, Algorithm 5.1 is a linearizable lock-free implementation of a concurrent queue. ■

The algorithm's scalability depends on the scalability of the concrete implementations of LL/IC and the basket with which it is instantiated. Our proposed solution involves wait-free implementations of each of the objects. In section 5.2.1, we present the implementations of these objects, which include a Compare&Swap-based operation and a Read/Write-based operation.

5.2.1 LL/ IC implementations.

A Compare&Swap-based implementation.

Let p denote a process that invokes an operation on the object. This implementation uses a shared register R initialized to 0. LL first reads R and stores the value in a persistent variable r_p of p, and then returns r_p. IC first reads R and if that value is equal to r_p, then it performs Compare&Swap$(R, r_p, r_p + 1)$; in any it case returns OK. The pseudocode for this implementation is shown in Figure 5.2.

```
Shared Variables:
    R = Atomic Register initialized to 0

Operation LL(r_p):
(01) r_p = R.Read()
(02) return r_p
end LL

Operation IC(r_p):
(03) r = R.Read()
(04) if r == r_p then
(05)    Compare&Swap(R, r_p, r_p + 1)
(06)    endif
(07) return OK
end IC
```

Figure 5.2: Compare&Swap-based LL/IC object

Theorem 5.2 The Compare&Swap-based LL/IC implementation from the algorithm 5.2 is linearizable and wait-free.

Proof:
It is not hard to see that the algorithm is wait-free. For the linearizability proof, consider any finite execution E with no pending operations. We define the following linearization points. The linearization point of an LL operation is when it reads R (Line 1). If an IC operation performs a Compare&Swap, it is linearized at that step (Line 5). Otherwise, it is linearized when it reads R (Line 3). Let S_t be the sequential execution induced by the first t linearization points of E, reading its steps in index-ascending order. By induction on t, it can be shown that S_t is a sequential execution of LL/IC, where T is the number of operations in E. The main observation is that if there is a successful Compare&Swap before the Compare&Swap of an IC operation of a process p, then the contents of R are different from the value p reads in its previous LL operation. ■

A Read/Write implementation.

This implementation uses a shared array M with n entries initialized to 0. LL first reads all entries of M (in some order), stores the maximum value in a persistent variable max_p of p, and then returns max_p. IC first reads all entries of M, and if the maximum among those values is equal to max_p, it performs Write($M[p], max_p + 1$); in any it case returns OK. The pseudocode for this implementation is shown in Figure 5.3.

Shared Variables:
$M = [0, \ldots, 0]$ *n Registers*

Operation LL(max_p):
(01) $max_p = \max(M)$
(02) **return** max_p
end LL

Operation IC(max_p):
(03) $m = \max(M)$
(04) **if** $m == max_p$ **then**
(05) $M[p]$.Write($max_p + 1$)
(06) **endif**
(07) **return OK**
end IC

Figure 5.3: Read/Write-based LL/IC object

Theorem 5.3 The Read/Write-based LL/IC implementation from algorithm 5.3 is linearizable and wait-free.

Proof:
The algorithm is wait-free because the max operation is performed in a finite number of steps, and all other instructions always finish. We next argue that each of its executions is linearizable.

Consider any finite execution of the algorithm with no pending operations. To simplify the argument, suppose that there is a *fictitious* IC operation that atomically writes 0 in all entries of M at the beginning of the execution.

Each IC operation is linearized at its last step. Thus, an IC that writes is linearized at its Write step (Line 5), and an IC that does not write is linearized at its last Read step (Line 3 inside of max operation). Let MAX be the maximum value in the shared array M at the end of the execution. For every $R \in \{0, 1, \ldots, MAX\}$, let IC_R be the IC operation that writes R for the first time in M. We will linearize every LL operation that returns the value $R \in \{0,\ 1,\ \ldots, MAX\ -\ 1\}$ at one of its steps and argue that this step is between IC_R and IC_{R+1}. This will induce a sequential execution that respects the real-time order and is a sequential execution of LL/IC, hence a linearization.

Let op denote any LL that returns $R \in \{0, 1, \ldots, MAX - 1\}$ and let e denote its Read step that reads R for the first time. Observe that IC_R has been linearized when e happens in the execution. We have two cases:

1. If the shared memory M does not contain a value $> R$ when e occurs (hence no $IC_{R'}$ with $R' > R$ has been linearized when e occurs), then op is linearized at e.

2. If the shared memory M does contain a value $> R$ when e occurs, then op is linearized as follows. Let $M[j]$ be the entry read at step e. Note that this case can happen if and only if some entries in the range $M[0, \ldots, j-1]$ contain values $> R$ when e happens (and hence some $IC_{R'}$ with $R' > R$ have been linearized when e occurs). Moreover, it can be shown that the value $R+1$ is written in an entry in the range $M[0, \ldots, j-1]$ at some time between the invocation of op and e. Let $i \in \{0, \ldots, j-1\}$ be the index of the entry where it is written $R+1$ for the first time. Then, op is linearized right before $R+1$ is written in $M[i]$ (and hence before IC_{R+1}).

■

5.2.2 Basket implementations.

The basket implementations appear in Algorithms 5.4 and 5.5. Both implementations have a shared array where the items are put and taken. A Put operation tries to put its item in a location, while a Take operation either take an item from a location or marks it as "canceled". The first implementation follows an approach similar to that of the LCRQ algorithm [67], while the second implementation is reminiscent of locally linearizable generic data structure implementations of [33].

K-Basket

For this implementation, the processes use Fetch&Increment to guarantee that at most two "*opposite*" operations "*compete*" for the same location in the shared array, which can be resolved with a Swap; the idea of this algorithm is similar to the approach in the LCRQ algorithm [67]. We called K-basket to this implementation, shown in the Algorithm 5.4.

Theorem 5.4 Algorithm 5.4 is a wait-free linearizable implementation of a K-basket.

Proof:
It is not hard to see that the algorithm is wait-free. Always that the value of the variables $PUTS$ or $TAKES$ is greater than K or the state of the basket is CLOSED, the algorithm finishes. The total of cycles will always be bounded by K.

For the linearizability proof, given an entry $A[i]$, we will say that a Put operation *successfully puts* its item in $A[i]$ if it gets $\perp$ when it performs Swap on $A[i]$, and that a Take operation *successfully cancels* $A[i]$ if it gets $\perp$ when it performs Swap on $A[i]$, otherwise (i.e., it gets a value distinct from $\perp$), we say that the Take operation *successfully takes* an item from $A[i]$.

From the specification of Fetch&Increment, for every $A[i]$, at most one Put operations tries to put its item in $A[i]$ successfully, and at most one Take operation tries to either successfully cancel $A[i]$ or successfully take an item from $A[i]$. By the specification of Swap, if $A[i]$ is canceled, no Put operation successfully puts an item in it, and no Take operation successfully takes an item from it.

Given any execution of the algorithm, the operations are linearized as follows. A Put operation that successfully puts its item is linearized at its last Fetch&Increment instruction before returning. A Take operation that successfully takes an item from $A[i]$ is linearized right after the Put operation that successfully puts its item in $A[i]$. A Put that returns FULL is linearized at its return step, and similarly, a Take that returns CLOSED is linearized at its return step. Note that, in both cases, at that moment of the execution, every entry of A has been or will be either canceled or a Take operation has or will successfully take an item from it. It can be shown that these linearization points induce a valid linearization of the execution. ■

N-Basket

In the second implementation, each process has a dedicated location in the shared array where it tries to put its item when it invokes Put. When a process invokes Take, it first tries to take an item from its dedicated location. If it does not succeed, it randomly picks a non-previously-picked location, does the same, and repeats until it takes an item or all locations have been canceled. Since several operations might "compete" for the same location, Compare&Swap is needed. This implementation is

```
Shared Variables:
    A[0, 1, ..., K − 1] = [⊥, ⊥, ..., ⊥]
    PUTS, TAKES = 0
    STATE = OPEN

Operation Put(x):
while true do
    state = Read(STATE)
    puts = Read(PUTS)
    if state == CLOSED or puts ≥ K then return FULL
    else
        puts = Fetch&Increment(PUTS)
        if puts ≥ K then return FULL
        else if Swap(A[puts], x) == ⊥ then return OK endif
    endif
endwhile
end Put

Operation Take():
while true do
    state = Read(STATE)
    takes = Read(TAKES)
    if state == CLOSED or takes ≥ K then return CLOSED
    else
        takes = Fetch&Increment(TAKES)
        if takes ≥ K then
            Write(STATE, CLOSED)
            return CLOSED
        else
            x = Swap(A[puts], ⊤)
            if x ≠ ⊥ then return x endif
        endif
    endif
endwhile
end Take
```

Figure 5.4: K-basket from Fetch&Increment and Swap.

reminiscent of *locally linearizable* generic data structure implementations of [33]. We called N-basket to this implementation, shown in the Algorithm 5.5.

Theorem 5.5 Algorithm 5.5 is a wait-free linearizable implementation of an n-basket.

```
Shared Variables:
    A[0, 1, ..., n−1] = [⊥, ⊥, ..., ⊥]
    STATE = OPEN
Persistent Local Variables of p:
    takes_p = {0, 1, ..., n−1}

Operation Put(x):
if Read(STATE) == CLOSED then return FULL
else if Read(A[p]) == ⊥ then
    if Compare&Swap(A[p], ⊥, x) then return OK endif
endif
return FULL
end Put

Function compete(pos):
x = Read(A[pos])
if x == ⊤ then return ⊤
else if Compare&Swap(A[pos], x, ⊤) then return x
else return ⊥ endif
end compete

Operation Take():
while true do
    if Read(STATE) == CLOSED then return CLOSED
    else
        if p ∈ takes_p then pos = p
        else pos = any element of takes_p endif
        takes_p = takes_p \ {pos}
        if takes_p == ∅ then Write(STATE, CLOSED) endif
        x = compete(pos)
        if x ≠ ⊥, ⊤ then return x
        else if x == ⊥ then
            x = compete(pos)
            if x ≠ ⊥, ⊤ then return x endif
        endif
    endif
endwhile
end Take
```

Figure 5.5: N-basket from Compare&Swap. p denote the invoking process.

Proof:
Clearly, Put is wait-free. It is not difficult to see that Take is wait-free because

the number of iterations is limited by the size of the set $takes_p$.

For the linearizability proof, given an entry $A[i]$, we will say that a Put operation of a process p, *successfully puts* its item in $A[p]$ if its Compare&Swap is successful. A Take operation *successfully cancels* $A[i]$ if its Compare&Swap$(A[i], x, \top)$ (in the compete function) is successful, with x being $\perp$; and it *successfully takes* an item from $A[i]$ if its Compare&Swap$(A[i], x, \top)$ (in the compete function) is successful, with x being distinct to $\perp$ and $\top$.

The linearizability proof is similar to the linearizability proof in the previous theorem, with the following main differences. (1) If a Put operation returns FULL, it can be the case that some of the other entries of A will never be canceled or store an item; the response of the Put operation is, however, correct because the sequential specification of n-basket allows Put to return FULL in any state of the object. (2) Several Take operations might try to cancel the same entry $A[i]$ or successfully take an item from it; this is not a problem because the specification of Compare&Swap guarantees that, at most, one succeeds.

Given any execution of the algorithm, the operations are linearized as follows. A Put operation that successfully puts its item is linearized at its (successful) Compare&Swap. A Take operation that successfully takes an item from $A[i]$ is linearized right after the Put operation that successfully puts its item in $A[i]$. A Put that returns FULL is linearized at its return step, and similarly, a Take that returns CLOSED is linearized at its return step. Note that at the execution, a Take that returns CLOSED, every entry of A has been either canceled, or a Take operation has successfully taken an item from it. It can be shown that these linearization points induce a valid linearization of the execution. ■

5.3 Coping with realistic assumptions

The previous construction was suitable for the analysis of properties that we desired in our concurrent queue. However, a realistic implementation will not rely on infinite arrays [2]. As noted in previous chapters, two common approaches for implementing arrays can be considered “infinite”. The first approach involves using circular dynamic arrays, which can expand or shrink as needed. The second approach uses linked lists, where each node contains a finite-sized array that grows on demand during execution. To maintain consistency, each node is appended to the list using a

[2] We refer to the infinite basket array of the queue.

Compare&Swap operation.

The first implementation requires straightforward changes, which rely on a mechanism to double the array size and copy from one array to another. This will be similar to the strategy followed by the Chase-Lev Work-Stealing [22] or the Idempotent Work-Stealing [65] algorithms presented in Chapter 4.

For the second implementation, we need to make more complex changes. Unlike the single-producer multi-consumer queue used for work-stealing in Chapter 4, we are now dealing with a multi-producer multi-consumer environment. This means we must incorporate a mechanism to ensure that node insertion in the queue is executed correctly. Algorithms 5.6 and 5.7 show the necessary changes to convert the Algorithm 5.1 to a queue capable of handling long-run executions. We use the operator $\rightarrow$ to denote the access to elements in a structure.

This update defines a new data structure called `Segment`. It serves as a node that comprises a small segment of the infinite basket array, a pair of objects of type LL/IC that denote the Head and Tail similar to what is stated in Algorithm 5.1 and a pointer to the next node. For simplicity, this node does not perform any circular assignment or deletion over the array as is done in LCRQ queue [67]. This makes it easier to determine when the segment is full or marked as closed for memory management tasks, using only the state of LL/IC objects. Additionally, we have defined two auxiliary functions that help identify when a segment is full or closed.

Besides the new Segment structure, now the shared variables are pointers to nodes of type Segment that represent Head and Tail similar to those used in Michael-Scott's lock-free queue [64]. We will prove that the queue defined in Algorithms 5.6 and 5.7 is lock-free and linearizable.

Theorem 5.6 Algorithms 5.6 and 5.7 are a linearizable lock-free implementation of a concurrent queue.

```
Additional Structures and Operations
   struct Segment :
      items[1, ..., N] array of basket objects of size N
      HEAD, TAIL = LL/IC objects initialized to 0
      next = Pointer to the next segment
   end struct

   Operation isFull(segment*)
      return segment → TAIL.LL() ≥ N
   end Operation

   Operation isClosed(segment*)
      return segment → HEAD.LL() ≥ N
   end Operation

Shared Variables:
   Head, Tail = Pointers to objects of type Segment, initially pointing to a sentinel object

Operation Enqueue(x):
while true do
   lastTail = Tail
   if lastTail ≠ Tail then continue; endif
   lastNext = lastTail → next
   if lastNext ≠ ⊥ then
      Tail.Compare&Swap(lastTail, lastNext); continue;
   endif
   ticket = lastTail → TAIL.LL()
   if isFull(lastTail) then
      newSegment = new Segment()
      newSegment → items[0].put(val);
      newSegment → TAIL.IC()
      if lastTail → next.Compare&Swap(⊥, newSegment) then
         Tail.Compare&Swap(lastTail, newSegment)
         return OK
      endif
   endif
   if lastTail → items[ticket].put(x) == OK
      lastTail → TAIL.IC()
      return OK
   endif
endwhile
end Enqueue
```

Figure 5.6: The modular baskets queue using linked-lists. Enqueue operation.

```
Additional Structures and Operations
    struct Segment :
        items[1, ..., N] array of basket objects of size N
        HEAD, TAIL = LL/IC objects initialized to 0
        next = Pointer to the next segment
    end struct

    Operation isFull(segment*)
        return segment → TAIL.LL() ≥ N
    end Operation

    Operation isClosed(segment*)
        return segment → HEAD.LL() ≥ N
    end Operation

Shared Variables:
    Head, Tail = Pointers to objects of type Segment, initially pointing to a sentinel object

Operation Dequeue():
while true do
    lastHead = Head
    if lastHead ≠ ⊥ then return epty endif
    if lastHead ≠ Head then continue endif
    if isClosed(lastHead) then
        next = lastHead− > next
        Head.CAS(lastHead, next)
        # Memory reclamation can be performed after the previous instruction
    endif
    tailTicket = lastHead → TAIL.LL()
    headTicket = lastHead → HEAD.LL()
    while ¬isClosed(lastHead) do
        if headTicket < tailTicket then
            x = lastHead → items[headTicket].Take()
            if x ≠ closed then return x endif
            lastHead → HEAD.IC()
        endif
        head′ = lastHead → HEAD.LL()
        tail′ = lastHead → TAIL.LL()
        if head == head′ == tail′ == tail and head < N then return epty endif
        headTicket = head′
        tailTicket = tail′
    endwhile
endwhile
end Dequeue
```

Figure 5.7: The modular baskets queue using linked-lists. Dequeue Operation

Proof:
To see that the queue algorithm composed of the Algorithms 5.6 and 5.7 is lock-free, we must analyze the Enqueue function shown in the Algorithm 5.6 and the Dequeue function shown in the Algorithm 5.7 are both lock-free.

We begin analyzing the Enqueue operation. We observe that a Enqueue operation enters a loop when:

- $lastTail$ is not equals to $Tail$ in line 3.
- $lastNext$ is not null in line 5.
- The current segment is full and fails the Compare&Swap at line 13.
- The thread is unable to insert the value in the respective basket at line 18.

Now, we will demonstrate that the Enqueue operation is lock-free by proving that a process only loops beyond a finite number of times if another process completes an Enqueue on the queue.

- At line 3 the condition is satisfied only if the pointer to the Tail has changed; this means that if another process has updated the reference to the segment, then that process must have successfully completed an enqueue operation.
- At line 5 the condition is satisfied only if, while reading the Tail pointer, another process appends a new segment (being succeeded in inserting a new element), and the process still does not update the reference to the Tail pointer. In such a case, we will try to help update the reference to the Tail segment and loop again.
- At line 13 the condition fails only if the Compare&Swap cannot append a new segment; this means that another process appends its segment and succeeds in inserting a new element into the queue.
- At line 18 the condition fails only if the operation cannot insert an element into the basket.

Now, we observe that a Dequeue operation enters a loop under when:

- $lastHead$ is not equals to$Head$ in line 4.

- The inner cycle encompassing lines 11 to 22 has been completed, and it does not return any output.

We will show that the Dequeue operation in Algorithm 5.7 is lock-free by performing a similar analysis to ours for the Enqueue operation. This will involve proving that a process only loops beyond a finite number of times if another process successfully completes a Dequeue operation on the queue.

- At line 4, the condition is satisfied if the pointer to *lastHead* is distinct from the current pointer to *Head*; in such case, we must loop again.

- If the inner cycle from the line 11 to the line 22 does not return anything. In that case, this suggests that one of the following situations could happen: (1) the segment *lastHead* is closed at the beginning of the loop, (2) the element taken from the basket in line 13 in each iteration is equals to the special value CLOSED until detect that the segment is closed, that means other processes are making progress by extract values or return the empty value.

Therefore, both Enqueue and Dequeue operations are lock-free.

To prove linearizability, we will use the same strategy as the one used in Theorem 5.1, using the aspect-oriented linearizability proof framework [39] to prove that the algorithm is linearizable. We assume that LL/IC objects and the baskets objects in the array of each segment are linearizable and wait-free[3]. Assuming that every item is enqueued at most once, it states that a queue implementation is linearizable if each of its finite executions is *free* of four violations. The proof is almost identical to the one shown in the Theorem 5.1. We enumerate the violations and argue that every algorithm execution is free of them.

1. VFresh: A Dequeue operations returns an item not previously inserted by any Enqueue operation. Dequeue operations return items once put in the baskets, and Enqueue operations put items in the baskets. Thus, each execution is free of VFresh.

2. VRepeat: Two Dequeue operations return the item inserted by the same Enqueue operation. The specification of the basket directly implies that every execution is free of VRepeat.

3. VOrd: Two items are enqueued in a certain order, and a Dequeue returns the later item before any Dequeue of the earlier item starts. Now, we are dealing with segments, and each segment has its own LL/IC objects for Head and Tail; we have 2 cases to analyze:

   (a) Inserting elements in the same segment: LL/IC guarantees that if an Enqueue operation enqueued an item, let us say x, and then a later Enqueue operation enqueued another item, let us say y, then x and y are inserted in baskets $items[i]$ and $items[j]$, with $i < j$. Then, x is dequeued first because the Dequeue operation can scan the $items$ array in index-ascending order.

   (b) Inserting elements in distinct segments: Similarly to the prior analysis, when a Enqueue operation inserts an item, let us say w, in a segment, and then a later Enqueue operation enqueues another item, say z, in another distinct segment, the Dequeue operation will first dequeue w. This is because it first checks if the current reference to the Head segment is closed. Since this is still not the case, it extracts w and increments the Head LL/IC object of the segment. When another Dequeue operation is executed, it detects that the pointer to the Head segment is closed and updates the pointer to the next. Now, we can extract z from the new segment.

   Thus, every execution is free of VOrd.

4. VWit: A Dequeue operation returning empty even though the queue is never logically empty during the execution of the Dequeue operation. An item is logically in the queue if it is in a basket $items[i]$; the segment that contains the basket is in the range from the Head pointer to the Tail pointer (both can reference the same segment), and $i < TAIL$, with $TAIL$ being the correspondent LL/IC object in the segment. When a Dequeue operation returns empty, it means that both the Head segment and Tail segment pointers reference the same segment. There is a point in the time where no basket in $items[0, 1, \ldots, TAIL-1]$ range that contains an item, and hence, the queue is logically empty. However, it is possible that $items[TAIL]$ may contain an item at that moment. As a result, every execution is free of VWit.

Therefore, Algorithms 5.6 and 5.7 are a linearizable lock-free implementation of a concurrent queue. ■

Another issue in practical settings is memory management. This issue can be delegated to the garbage collector in languages like Java. However, a safe and efficient concurrent memory reclamation protocol should be implemented in programming languages without automatic memory garbage collection. For example, we implemented all the algorithms and the necessary infrastructure for experimental evaluation of this chapter using C++20. For memory management, we have utilized popular protocols for memory reclamation, like Hazard Pointers [62] and Epoch-based reclamation [29, 61].

[3] We proved that there are linearizable and wait-free LL/IC objects in Theorems 5.2, and 5.3, as well as linearizable and wait-free basket implementations in Theorems 5.4 and 5.5.

CHAPTER 6

Experimental Evaluation and Results

In this chapter, we discuss the results of our experimental evaluation of the algorithms presented in Chapter 4 and Chapter 5. The chapter is divided into two sections. The first section (Section 6.1) is about the work-stealing case study. The second section (Section 6.2) deals with the experimental evaluation of the modular baskets queue.

In the first section, we analyze the performance of the work-stealing algorithms presented in Chapter 4, dividing the experimental evaluation into three benchmarks. The first two have been used before in other articles [30, 65, 68], and the third is an application to a problem that naturally admits parallelization. In the second section, we analyze the performance of the modular baskets queue algorithms presented in Chapter 5. We divide the experimental evaluation into two benchmarks. The first benchmark is designed to evaluate the performance of the modular baskets queue variants, i.e., the combinations of baskets and LL/IC objects. From the result of this first benchmark, we pick up the best performing version, and then we implement the array-based version and the list-of-arrays version as described in Section 5.3. Finally, these variants are compared against many state-of-the-art queues.

For both evaluations, we use the statistically rigorous methodology by Georges et al. [32], and we measure the performance as described in Section 3.5.2. In this experimental evaluation, we provide an in-depth analysis of the data we collected. We have gained valuable insights into our research topic through rigorous testing and analysis. We believe that the results presented in this chapter will contribute to a better understanding of the subject matter and will pave the way for future research in this field. So, let us dive in and explore the outcomes of our study in detail.

6.1

Work-Stealing with Multiplicity

In this section, we discuss the outcome of an experimental evaluation of WS-WMULT and its bounded version, B-WS-WMULT. Using the approaches discussed in Section 4.5, two versions of each algorithm were implemented, one using simple arrays and the other using dynamic arrays. The suffix Lists was added to the list of arrays version. For example, WS-WMULT denotes the version with resizable arrays, and WS-WMULT List denotes the version with a list of arrays. WS-WMULT and B-WS-WMULT were compared to the following algorithms: Cilk THE [30], Chase-Lev [22], and the three idempotent work-stealing algorithms [65]. Three benchmarks were employed to evaluate the performance of the algorithms, following the evaluation methodology by Georges, Buytaert, and Eeckhout [32], as described in the Section 3.5.2.

6.1.1 Experimental Setup

Platforms and Implementation

The experiments were conducted on a machine with an AMD Ryzen Threadripper 3970X processor with 64GB of memory and 32 cores, each capable of executing two hardware threads. The algorithms were implemented in Java 17, which allowed us to ignore tasks such as garbage collection. As mentioned, two versions of WS-WMULT and B-WS-WMULT were implemented. The other tested algorithms, Cilk THE, Chase-Lev, and the three idempotent work-stealing, were implemented following their specification, i.e., inserting fences where required; all these algorithms use resizable arrays to store tasks.

Methodology

To analyze the performance of the algorithms, the experimental evaluation is divided into the following three benchmarks, where the first two have been used before [30, 65, 68] and the third is an application to a problem that naturally admits parallelization: (1) *Zero cost experiments*, (2) *Parallel spanning tree*, and (3) *Parallel SAT*. Below, we briefly describe each benchmark. Performance was measured using the statistically rigorous methodology by Georges et al. [32], described in Section 3.5.2. Repeated work was measured in the second and third benchmarks using a more straightforward methodology.

Zero cost experiments This benchmark shows the performance of a given algorithm in a single core with a single process. Why do we want to evaluate the algorithms in a single core? The experiment results show that the mere presence of heavy synchronization mechanisms in an algorithm slows down the computation, even in sequential executions. We measure the time required for performing a sequence of operations that work-stealing algorithms provide. First, the time needed for Put-Take operations is measured, where a process performs a sequence of Put operations followed by an equal number of Takes. Unlike previous work [65, 68], it also measured the time for Put-Steal operations, all performed by the same process. In both experiments, the number of Put operations is $10,000,000$, followed by the same number of Take or Steal operations; no operation performs any work associated with a task. The algorithms were evaluated considering distinct data structures' initial sizes to test the effect of resizing structures. The considered initial sizes are 256, $1,000,000$, and $10,000,000$. For the case of the implementations based on dynamic arrays, WS-WMULT Lists, and B-WS-WMULT Lists, these sizes correspond to the size of the arrays in each structure node.

Parallel spanning tree As Michael, Vechev, and Saraswat [65], and Morrison and Afek [68], we consider the parallel spanning tree algorithm of Bader and Cong [11]. The algorithm, which uses a form of work-stealing to ensure dynamic load-balancing, was adapted to work with all tested work-stealing implementations. In the algorithm, each process places the tasks it generates (i.e., vertices whose adjacent vertices remain to be explored) in its work-stealing data structure. When it runs out of tasks, it tries to steal tasks from other processes' structures in a round-robin fashion. The algorithms are tested on several types of directed and undirected graphs with $1,000,000$ vertices each:

- *2D Torus.* The vertices are on a 2D mesh, where each vertex connects to its four neighbors in the mesh.
- *2D60 Torus.* The random graph obtained from the previous one, where each edge has a probability of 60% to be present.
- *3D Torus.* The vertices are on a 3D mesh, where each vertex connects to its six neighbors in the mesh.
- *3D40 Torus.* The random graph obtained from the previous one, where each edge has a probability of 40%
- *Random.* Graph with each vertex having six randomly picked neighbors.

The graphs have been considered in previous work [11, 65, 68]. All graphs are represented using the adjacency lists representation. The experiment is executed as follows: The parallel spanning tree algorithm is executed independently on each possible graph, with all combinations of work-stealing algorithms and several available threads. As in the zero-cost experiment, the impact of resizing structures was tested. The experiment is independently executed for each graph, work-stealing algorithm, and several threads with initial structure sizes 250 and $1,000,000$. Hence, the latter requires no resizing structures.

The amount of repeated work in both relaxations of work-stealing was also measured. Computing this value is not straightforward. The parallel spanning tree algorithm may execute repeated work even when no work-stealing tasks exist. The reason is that a vertex can be discovered (i.e., put in a work-stealing structure) concurrently by several distinct processes, and later, each of them can take the vertex from its structure and execute the work associated with the vertex (i.e., explore its neighborhood). One more difficulty is that computing the *exact* number of processes that Take/Steal the same task in *each* work-stealing structures would incur in time and space overheads. Therefore, a simple approach was adopted, where the total number of Puts (i.e., the total number of tasks stored in all work-stealing structures) and the total number of Takes (i.e., the total number of tasks executed) are counted. Each process locally counts its number of Puts/Takes, and the counters are added up at the end of the experiment. These quantities allowed us to estimate the amount of repeated work in the relaxations of work-stealing and its impact. To avoid disturbing performance measurements, the experiments for measuring repeated work were executed independently, following a simpler methodology: each experiment was run five times from which average values were computed.

Parallel SAT We consider parallel SAT solvers (e.g., [14, 26, 35]). It was implemented as a simple single-producer multi-consumer solver with a single instance of one of the tested work-stealing algorithms. The implementation is straightforward: the processes test every possible binary assignment to determine if any of them satisfies a given formula. Each task in the work-stealing structure consists of a range of assignments to be evaluated. The main process (i.e., the main thread in the Java program) is the owner of the work-stealing structure and generates all tasks at the beginning of the experiment. Once all tasks are generated, the main process and the rest of the processes collaborate to identify if there is a satisfying assignment. The input to the experiment is an unsatisfiable formula in conjunctive normal form (CNF). The formula was generated from an SAT formula generator[1]. The experi-

[1]https://cnfgen.readthedocs.io/en/latest/cnfgen.families.pigeonhole.html

ment is executed independently for each number of available threads. Multiple size assignment ranges are tested independently: 50, 100, 250, 500, 1,000, and 2,500.

It was also measured the amount of repeated work in each experiment. In this case, it is easy to compute the exact number as there is a single work-stealing structure. It suffices to compare the total number of Puts and the total number of successful Takes/Steals. These quantities are computed as in the previous benchmark, and, as before, the experiments for measuring repeated work were executed independently so as not to disturb performance measurements. Each experiment was run five times, from which average values were computed.

6.1.2 Experimental Evaluation Results

First, we present a summary of the experimental evaluation results of the case study from Chapter 4:

- Zero cost experiments. In both experiments, Puts-Takes and Puts-Steals, overall, the algorithm with the best performance was WS-WMULT, followed by WS-WMULT Lists and then idempotent FIFO, regardless of the initial array size. This result is expected because WS-WMULT does not use either costly primitives or memory fences. It was also observed that idempotent FIFO performed better than WS-WMULT Lists when no resizing was needed. B-WS-WMULT and B-WS-WMULT Lists performed worst among all algorithms. However, this did not preclude them from exhibiting a competitive performance in the second and third benchmarks.

- Parallel spanning tree. In virtually all experiments, WS-WMULT performed best among all tested algorithms. It outperformed Cilk THE and Chase-Lev, performing better than WS-WMULT Lists and idempotent FIFO by small margins. Idempotent FIFO performed best among the idempotent algorithms. B-WS-WMULT exhibited a competitive performance. Work-stealing algorithms with FIFO policy generally showed low amounts of repeated work. In contrast, algorithms with LIFO of dequeue policies exhibited higher amounts, and, for some graphs, the difference with FIFO is remarkable.

- Parallel SAT. In general, all work-stealing algorithms enhanced performance, and in both relaxations, repeated work was negligible, which resulted in minor performance overhead. The algorithms showed no significant statistical difference in performance for large-size assignments (i.e., fairly complex jobs associated with tasks).

The following subsections explain the results of the experiments in detail.

Zero Cost Experiment

The outcome of the Puts-Takes experiment appears in Figures 6.1a, 6.1b and 6.1c. Overall, the absence of fences in WS-WMULT derived an improvement over idempotent algorithms that range between 9% to 65%. Table 6.1 contains the percentage improvement of WS-WMULT over all algorithms. In all cases, B-WS-WMULT and B-WS-WMULT Lists performed worst. This is arguably attributed to the extra array of boolean flags used for bounding multiplicity. [2] It was also observed that the data structure's initial size impacts performance. When the initial size exceeds the number of operations in the experiment (Figure 6.1c), no resizes are needed, B-WS-WMULT's performance improved considerably. Still, it was affected by the use of two distinct arrays. The outcome of the Puts-Steal experiment is similar, and shown in Figures 6.1d, 6.1e and 6.1f, and Table 6.2. Appendix A.1 contains the detailed measurements of each algorithm in each experiment.

	Chase-Lev	Cilk THE	Idempotent FIFO	Idempotent LIFO	Idempotent DEQUE	WS WMult	B. WS WMult	WS WMult Lists	B. WS WMult Lists
Intial zize 256:	48.00	49.26	51.85	42.90	65.64	0.00	88.75	52.14	71.46
Initial size 1,000,000:	45.48	46.82	48.73	41.77	62.34	0.00	87.83	40.41	65.45
Initial size 10,000,000:	27.91	27.14	14.31	43.81	50.25	0.00	71.22	46.76	75.19

Table 6.1: Percentage improvement of WS-WMULT over all algorithms in the Puts-Takes experiment.

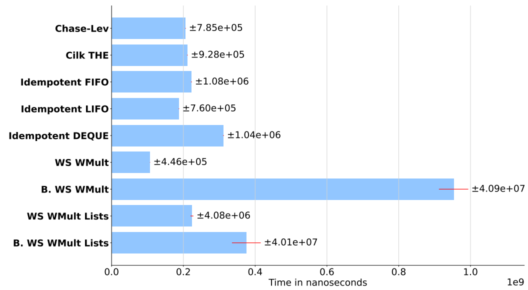
	Chase-Lev	Cilk THE	Idempotent FIFO	Idempotent LIFO	Idempotent DEQUE	WS WMult	B. WS WMult	WS WMult Lists	B. WS WMult Lists
Intial size 256:	46.22	63.68	50.66	51.66	66.60	0.00	89.47	52.60	76.53
Initial size 1,000,000:	43.91	62.91	48.25	51.67	64.03	0.00	88.81	40.90	70.53
Initial size 10,000,000:	24.86	57.88	9.00	53.67	52.61	0.00	75.03	46.32	77.75

Table 6.2: Percentage improvement of WS-WMULT over all algorithms in the Puts-Steals experiment.

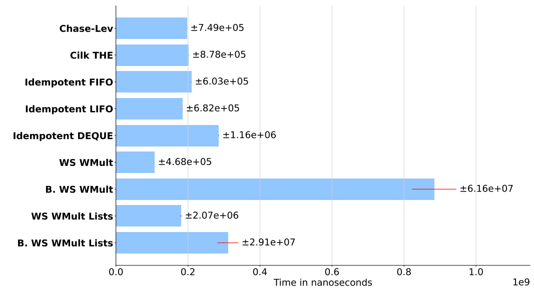
Parallel Spanning Tree

Except for the case of Random graphs, where practically all algorithms performed equally, in general, WS-WMULT outperformed all algorithms. WS-WMULT Lists and idempotent FIFO overall performed second and third best. The improvement of WS-WMULT over WS-WMULT Lists and idempotent FIFO was small, between 0.5% and

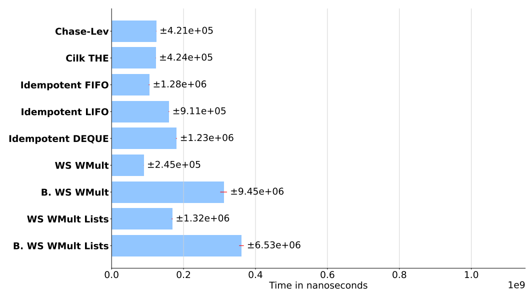
[2] Implementations where the two arrays are consolidated in a single array of objects with two entries, a task, and a flag, performed even worse. Hence, these implementations were discarded.



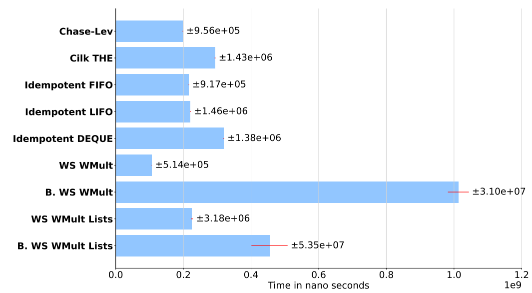
(a) Puts and Takes with an initial size 256



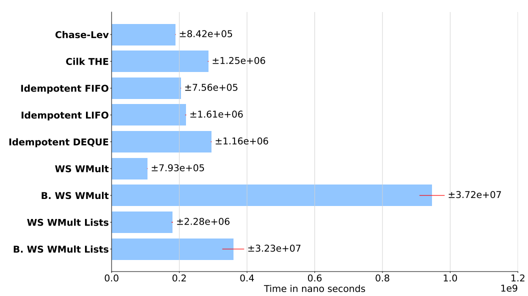
(b) Puts and Takes with an initial size 1,000,000



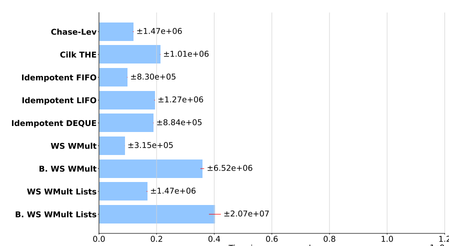
(c) Puts and Takes with an initial size 10,000,000



(d) Puts and Steals with an initial size 256



(e) Puts and Steals with an initial size 1,000,000



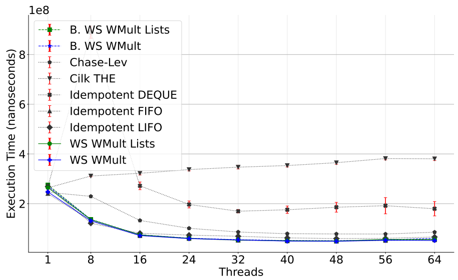
(f) Puts and Steals with an initial size 10,000,000

Figure 6.1: Outcome of the zero cost experiments. Time is in nanoseconds, and red lines over bars show confidence intervals. The results of the Puts-Takes experiment are shown in the first three charts and the results of the Puts-Steals experiment are shown in the remaining charts.

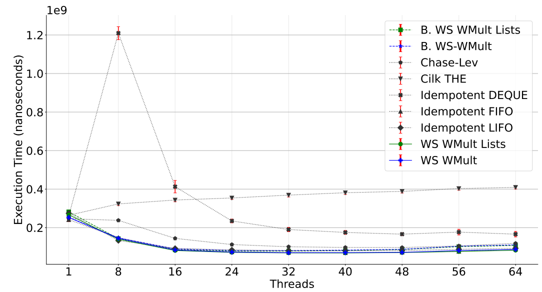
4%, depending on the graph. Thus, the absence of fences in WS-WMULT resulted in a minor improvement over idempotent FIFO. It merits mentioning that B-WS-WMULT and its lists-based version generally showed a competitive performance, in some cases close to the first three algorithms. Usually, Cilk THE and Chase-Lev performed worst, which is expected as they use costly synchronization mechanisms, although this is not the only factor (more on this below). WS-WMULT outperformed Cilk THE by a margin between 1% and 21%, and Chase-Lev by a margin between 0.14% and 32%. The lowest margins occurred in the case of Random graphs, where, as mentioned, all algorithms performed almost equally. Figure 6.2 depicts the result of the experiment in some representative cases. In a few cases (e.g., Directed 2D Torus), Chase-Lev, Cilk THE, and idempotent LIFO performed best with few processes. This seems to be related to the topology of the graph and the algorithm's insert/extract task policy (the owner follows LIFO).

Repeated work was measured indirectly through the total number of Puts (work to be executed), which was compared to the total number of Puts in sequential executions (i.e., $1,000,000$). The difference between these two numbers is called surplus work. Surplus work in all algorithms with FIFO insert/extract policy was generally low, less than 0.7%. All these algorithms implement work-stealing with relaxed semantics. Thus, even if all surplus work was due to relaxation (recall that surplus work can occur even with non-relaxed work-stealing algorithms), it rarely happened, with little impact on performance. In sharp contrast, in all algorithms where the owner follows the LIFO insert/extract policy (Cilk THE, Chase-Lev, idempotent LIFO, and idempotent Deque), surplus work ranged between 1% and 56%. Therefore, neither multiplicity nor idempotency per se increased surplus work considerably, and the dominant factor seems to be the task insert/extract policy combined with the solved problem. Figure 6.3 depicts the surplus work of the experiments in Figure 6.2.

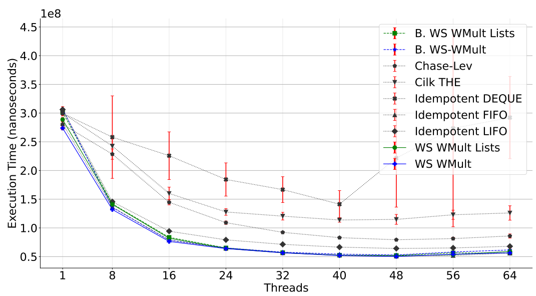
In all algorithms, not all tasks are executed. Processes are constantly checking the distinct number of vertices that have been processed so far, and when this number reaches $1,000,000$, the spanning tree is completed, and the experiment terminates. It can be the case that some vertices remain in one or more work-stealing structures when the tree is finished; not all surplus work is executed. We measured the executed surplus work, i.e., the difference between the total number of Takes (actual work executed) and the total number of Takes in sequential executions (i.e., $1,000,000$). Executed surplus work in Cilk THE, Chase-Lev, idempotent LIFO, and idempotent Deque ranged between 1% and 49%. Figure 6.4 shows the executed surplus work of the experiments in Figure 6.2. Finally, in some experiments (e.g., Random graphs), WS-WMULT executed more surplus work than the algorithms with LIFO insert/extract policy, but still, it performed slightly better. We attribute this to the fact that in the FIFO policy, Takes are more likely to read tasks from cache



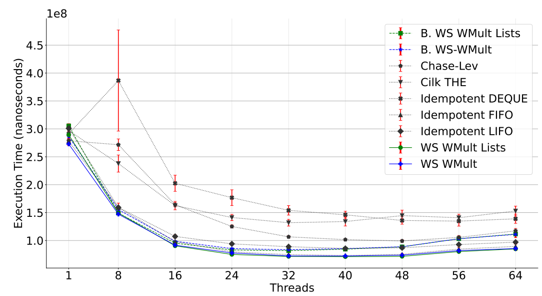
(a) Graph: Directed Torus 2D. Initial size of 256 items



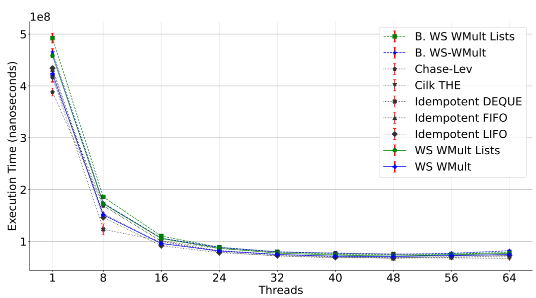
(b) Graph: Directed Torus 2D. Initial size of 1,000,000 items



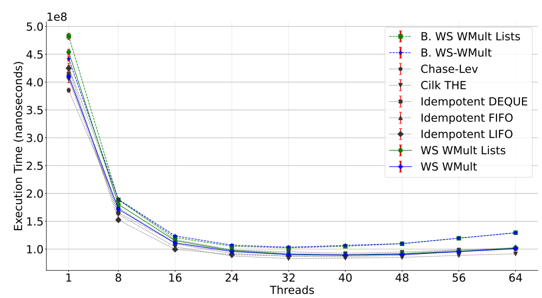
(c) Graph: Directed Torus 3D. Initial size of 256 items



(d) Graph: Directed Torus 3D. Initial size of 1,000,000 items



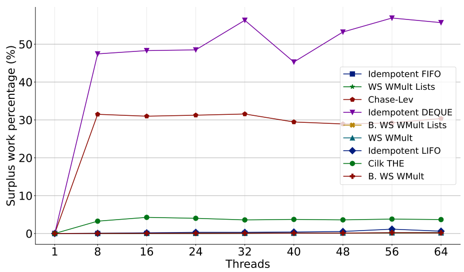
(e) Graph: Directed Random. Initial size of 256 entries



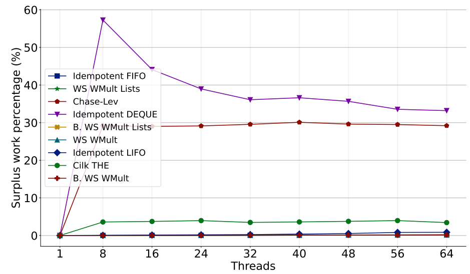
(f) Graph: Directed Random: Initial size of 1,000,000 entries

Figure 6.2: Mean times reported for executing the graph application benchmark.

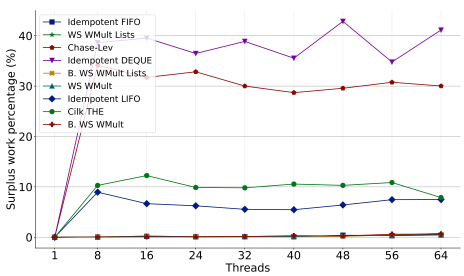
memory, whereas in LIFO, Takes are more likely to read from main memory, which is costly. Appendix A.2 contains all results of the benchmark.



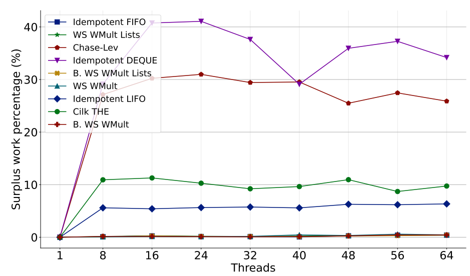
(a) Surplus work: Directed Torus 2D. Initial size of 256 items.



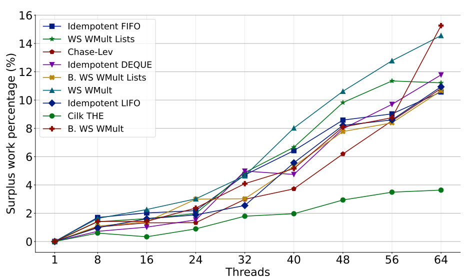
(b) Surplus work: Directed Torus 2D. Initial size of 1,000,000 items.



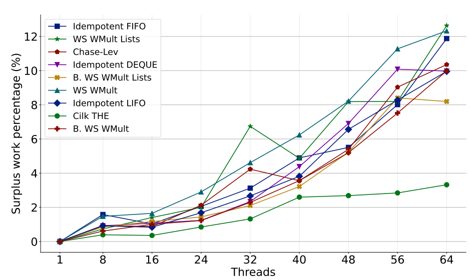
(c) Surplus work: Directed Torus 3D. Initial size of 256 items.



(d) Surplus work: Directed Torus 3D. Initial size of 1,000,000 items.

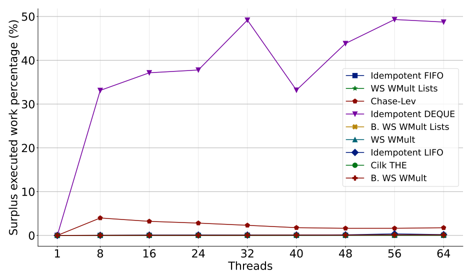


(e) Surplus work: Directed Random. Initial size of 256 items.

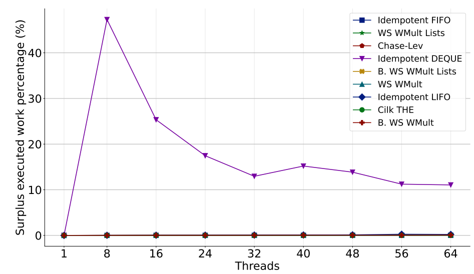


(f) Surplus work: Directed Random: Initial size of 1,000,000 items.

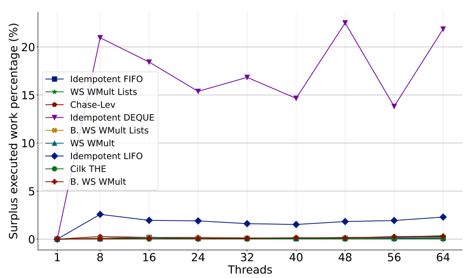
Figure 6.3: Surplus work (percentage) of the experiments. Surplus work: the difference between the total number of Puts and the number of puts in sequential executions (i.e., $1,000,000$).



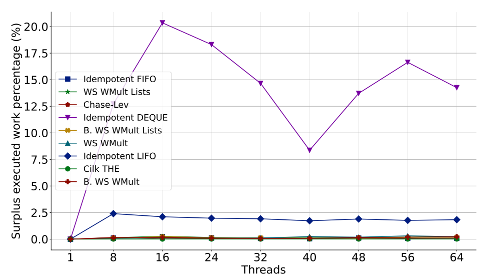
(a) Executed surplus work: Directed Torus 2D. Initial size of 256 items.



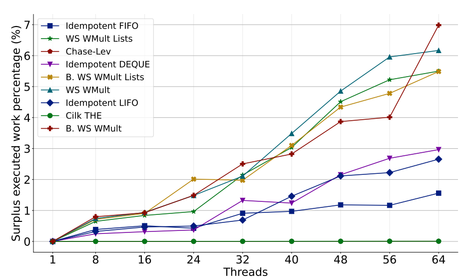
(b) Executed surplus work: Directed Torus 2D. Initial size of 1,000,000 items.



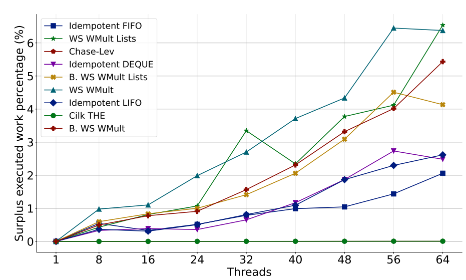
(c) Executed surplus work: Directed Torus 3D. Initial size of 256 items.



(d) Executed surplus work: Directed Torus 3D. Initial size of 1,000,000 items.

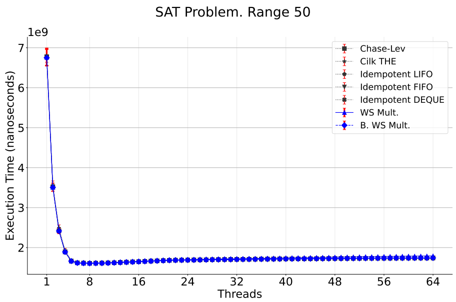


(e) Executed surplus work: Directed Random. Initial size of 256 items.

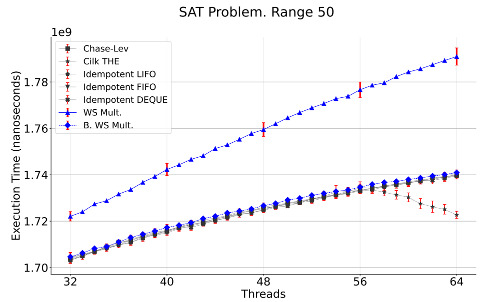


(f) Executed surplus work: Directed Random: Initial size of 1,000,000 items.

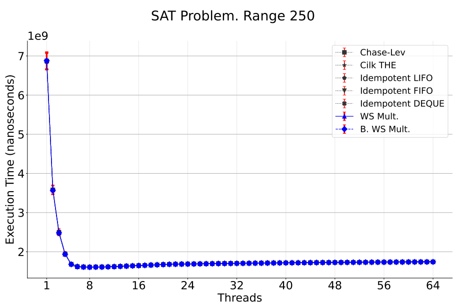
Figure 6.4: Executed surplus work (percentage) of the experiments. Surplus work: the difference between the total number of Takes and the number of takes in sequential executions (i.e., $1,000,000$).



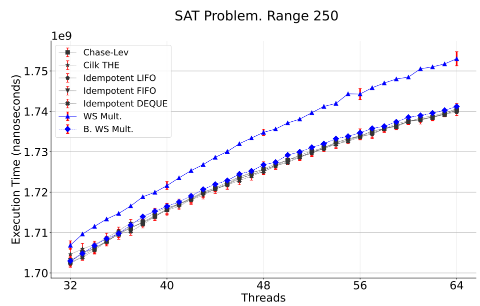
(a) Range assignment size 50.



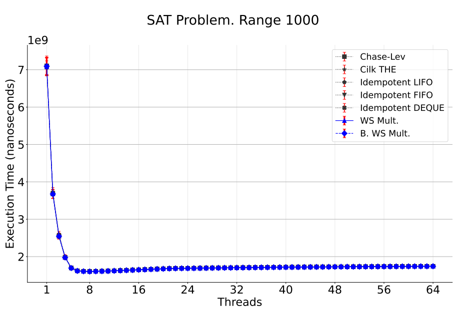
(b) Range assignment size 50. Zoom in to the number of processes 32 to 64.



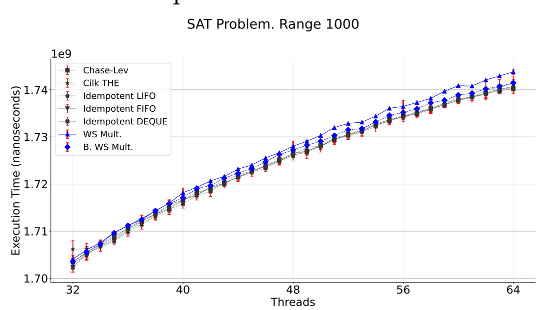
(c) Range assignment size 250.



(d) Range assignment size 250. Zoom in to the number of processes 32 to 64.



(e) Range assignment size 1,000.



(f) Range assignment size 1,000. Zoom in to several processes 32 to 64.

Figure 6.5: Mean times of the Parallel SAT benchmark for range assignment sizes 50, 250, and 1,000.

Parallel SAT

The outcomes for range assignment sizes 50, 250, and 1,000 are depicted in Figures 6.5a, 6.5c, and 6.5e, respectively. All algorithms speeded up sequential computation by 70%, and generally, all performed very similarly. However, repeated work (the difference between the number of Puts and the number of successful Takes/Steals) slightly impacted the performance of WS-WMULT. Contrary to previous benchmarks, whose tasks are simple, tasks in this benchmark require more computation; hence, repeated work is costly. In the experiment with range size 50, WS-WMULT's repeated work was larger than other algorithms, and this tendency became more pronounced as the number of processes increased. This happens because (1) a small range size increases the possibilities of concurrent Puts/Takes and (2) interleavings of Puts/Takes of WS-WMULT, where multiplicity arises, are arguably not too complex. However, repeated work was always low, less than 1%. Still, the small amount of repeated work had some minor impact on WS-WMULT's performance (see Figure 6.5b). For larger range sizes, 250 and 1,000, the amount of repeated work of WS-WMULT decreased to almost zero (as concurrent Puts/Takes are less likely to happen), and hence its impact became negligible (see Figures 6.5d and 6.5f). In contrast, idempotent algorithms had low amounts of repeated work in all cases (always close to zero), which arguably happened because the interleavings where the relaxation occurs are less likely to occur. All algorithms exhibited the same performance when the range sizes were more significant, with ranges sizes of 250 and 1,000. It is worth stressing that insert/extract policies did not affect performance, as all tasks were generated at the beginning of the experiment; hence, basically, every Take/Steal had to read from main memory at all times.

The outcomes of the rest of the experiments, for range assignment sizes 100, 500, and 2,500, are similar. Appendix A.3 contains all results of the benchmark.

6.2 Modular Basket Queues

In this section, we discuss the outcome of an experimental evaluation of the modular baskets queue algorithms presented in Chapter 5. To evaluate the modular queue's performance, we have designed a set of experiments that allow us to determine whether it is competitive with queues in the state-of-the-art literature. We divide our experiments into two classes: *inner experiments* and *outer experiments*.

Inner experiments evaluate the performance of the modular queue using distinct implementations of LL/IC objects and baskets. These experiments allow us to consider which implementations (combinations of LL/IC objects and baskets to build the modular queue) perform best. Also, this allows us to know if, using more relaxed objects, the queue's performance can compete with the classical synchronization objects. Once the inner experiments have been evaluated and the best combination to build the queue is chosen, we asses the selected queue against state-of-the-art queues (*outer experiments*) to assess its performance and throughput. The list of queue algorithms against which we evaluate our chosen queue are:

- Wait-Free queue by Yang and Mellor-Crummey [90].
- Lock-Free LCRQ queue by Morrison and Afek [67].
- Lock-Free queue by Michael and Scott [64].
- Lock-Free queue by Ramalhete [77], which was strongly inspired by the obstruction-free queue shown in the work of Yang and Mellor-Crummey [90]
- Lock-Free queue by Ostrovsky and Morrison [74][3].

6.2.1 Experimental Setup

Platforms and Implementation

The experiments were conducted on a machine with an AMD Ryzen Threadripper 3970X processor with 64GB of memory and 32 cores, each capable of executing two hardware threads. This gives a total of 64 hardware threads for the evaluation. We have developed the algorithms and infrastructure to carry out experimental evaluation using the C++20 programming language. This allows us to benefit from the new concurrency and parallelism features integrated with this version, including updates to atomics and synchronization facilities.

For the *inner experiments* of the modular queue variants, we do not use any advanced memory reclamation protocol. Instead, we added basic memory reclamation after each evaluation. During these experiments, we only followed the specifications of the queue, LL/IC objects, and *baskets* mentioned in Section 5.2 to implement the distinct variants. The main objective of this experiment was to understand how different implementations of the same object (LL/IC, *baskets*) can affect the performance of the modular queue. Below, we have provided a detailed explanation of how the experiment will be conducted.

For our *outer experiments*, we use Hazard Pointers as memory reclamation for the following queues: the lock-free queue by Michael-Scott [64], the LCRQ queue [67], and the lock-free queue by Ramalhete-Correia [77], as well as the list-of-arrays version of the modular queue. In the case of the queue by Ostrovsky-Morrison [74] and

[3]We implemented only the version that use simple Compare&Swap

the queue by Yang-Mellor Crummey [90], we used their respective memory reclamation algorithms, such as epoch-based memory reclamation [29, 61]. We followed the specifications for dynamic arrays and the list of the arrays described in Section 5.3 to implement the modular queue for these experiments. We compared the implementations of the modular queue to the state-of-the-art queues. A detailed explanation of how the experiments are conducted is provided below.

Methodology

To evaluate the performance of the queue and its components, we conduct an experimental evaluation divided into two benchmarks. The first benchmark analyzes the LL/IC objects described in section 5.2.1 and the baskets described in section 5.2.2. The second benchmark evaluates the performance of the modular queue using the best LL/IC object and basket compared to queues in the state-of-the-art literature. We use the statistically rigorous methodology by Georges et al. [32], as described in section 3.5.2, to measure performance in both benchmarks. Each software thread is pinned to a specific hardware thread in both cases.

Inner Experiments

To evaluate the performance of our distinct variants for the LL/IC objects and the baskets, which are fundamental for the construction of the modular queue, we perform the following experiments:

1. For the case of the LL/IC object implementations, we conducted a test to measure the time required for executing $5,000,000$ interspersed LL- IC calls to the same object by multiple threads. Each thread performs a random amount of work between LL and IC calls to avoid artificial long-run scenarios (see, for example, [90]). This random work consists of spinning a small amount of time (approximately six μs) in an empty loop. We took the *false sharing problem* [50] into account in the array-based LL/IC implementations (i.e., Read/Write implementation). We tested the following versions:

    - Compare&Swap-based implementation.
    - Read/Write-based implementation with distinct padding sizes for each array entry (0, 16, 32, 64 bytes of padding).

    We also tested the Fetch&Increment instruction in a similar setting to provide a comparison point for the LL/IC objects. Instead of making two calls with random work in between, we carried out Fetch&Increment with random work after its execution.

2. For the case of the combinations of LL/IC objects and baskets to build the modular queue, we did a similar evaluation to the previous, but testing interspersed calls to enqueue and dequeue in a shared queue by all threads. We measure the time for executing $5 \cdot 10^6$ interspersed calls to enqueue and dequeue, and similar to the previous test, we perform some artificial work to avoid artificial long-run scenarios, using the same technique described previously. The value K selected for the K-basket was $\sqrt{N}$, with N the number of processes in the experiment. We tested the following combinations of LL/IC objects with the respective basket:

    - LL/IC Read/Write (with padding) with N-basket
    - LL/IC Compare&Swap with N-basket
    - LL/IC Read/Write (with padding) with K-basket
    - LL/IC Compare&Swap with K-basket

Outer Experiments

The previous experiment's results displayed the performance of different modular queue variations. Based on that, we selected the best LL/IC object combination with the basket for the modular queue. In this experiment, we tested three versions of that modular queue against the following queues: Yang and Mellor-Crummey's Wait-Free queue [90], Morrison and Afek's Lock-Free LCRQ queue [67], Michael and Scott's Lock-Free queue [64], Ramalhete and Correia's Lock-Free queue [77], which was inspired by Yang and Mellor-Crummey's obstruction-free queue-Crummey [90], and Ostrovsky and Morrison's Lock-Free queue [74].

There are three versions of the modular queue: the *classic version* specified in Section 5.2, which has a fixed size[4] and does not have the option to resize[5]; the *dynamic array* version, which has an initial size of 1024 baskets and doubles its size whenever the array becomes full; and the *list of arrays* version, which use Hazard Pointers[59] for memory management and utilizes nodes with basket arrays of size 1024. The last two versions are specified in Section 5.3.

To evaluate all the queues, we adopt a benchmark similar to that used by Ostrovsky and Morrison [74]. This benchmark consists of three workloads: producer-only,

[4] 1,000,000 of baskets

[5] It is the same used for the inner experiments

consumer-only, and a mixed producer/consumer workload. During the experimental evaluation, each process can have the role of either a producer, who can call the Enqueue function, or a consumer, who can call the Dequeue function. Similar to the experiments performed in the section 6.2.1, we measure the time it takes until all threads complete $1,000,000$ operations. We use the statistically rigorous methodology described in the section 3.5.2 that follows the methodology of Georges et al. [32] for the experimental evaluation.

6.2.2 Experimental Evaluation Results

First, we present a summary of the experimental evaluation results from the Case Of Study presented in Chapter 5.

Inner experiments (LL/IC evaluation) In this experiment, we observed that all objects had similar behavior. As expected, the Fetch&Increment instruction was faster than the LL/IC objects in virtually all cases. An interesting observation is that as the number of contending processes increases, the behavior of the LL/ICCompare&Swap version is pretty similar to the Fetch&Increment evaluation. The Read/Write version of LL/IC objects behaved similarly but with lower performance than the Fetch&Increment and Compare&SwapLL/IC objects. The versions using 16 and 64 bytes of padding were found to be the best.

Inner experiments (Modular queue variants evaluation) In this experiment, we observed that queues based on the K-basket are the best among all the modular queue variants. In particular, the version based on the Compare&SwapLL/IC object performed the best. However, when the number of contending processes increases, it scales slightly and performs similarly. The queue using Read/Write LL/IC and the same type of basket does not scale, as shown in Figure 6.7. On the other hand, queues based on N-basket will not even be considered for more research in the future. However, it is interesting to see how the same algorithm can perform very differently using only distinct types of modules for its main parts.

Outer experiments Based on the test results, the Yang-Mellor Crummey queue is the most efficient compared to other queues. The LCRQ queue and Fetch&Increment queue are slightly less efficient than the Yang-Mellor Crummey queue. The performance of the list-of-arrays version and the classic version of the modular queue are less efficient than that of the previous three queues. However, the list-of-array version of the modular queue performs better than the original version. The Michael-Scott queue is less efficient than the previous queues. The Ostrovsky-Morrison queue performs poorly for a few processes, but its performance improves when the number of processes increases. The dynamic array version of the modular queue performs poorly, possibly due to contention when the array resizes when it becomes full.

Inner Experiments - LL/IC Performance

The outcome of the LL/IC performance experiments for 64 processes appears in Figure 6.6, with their respective percentage improvement shown in Table 6.3. In these experiments, the only real competitor for the Fetch&Increment instruction in terms of performance was the LL/IC object Compare&Swap-based implementation, where, in some moments, the range of improvement was over 0.46% to 3.49%, taking as reference the execution using the same number of processes. Nonetheless, when the number of threads was low, it showed no improvement. The Read/Write versions show a negative improvement, ranging from -3.54% to -47.57%. This result is expected due to additional instructions needed to execute the LL/IC operations. For further information about the results, refer to Appendix B.1 for a more detailed insight. Considering these results, we decided to test the modular queue's performance using the Compare&Swap-based version and the Read/Write-based version with 16 and 64 bytes of padding for each entry in the next subsection.

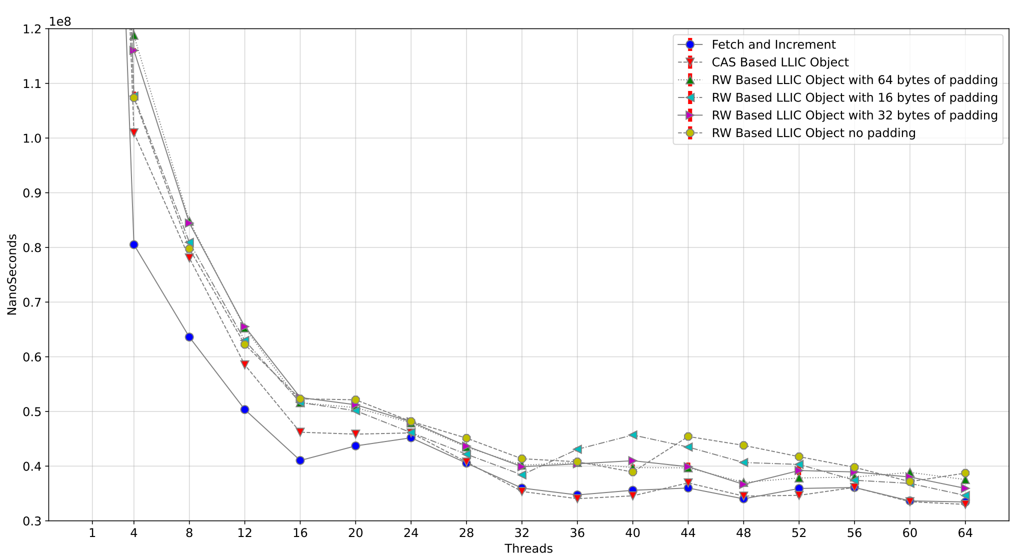


Figure 6.6: Mean times for LL/IC experiment. 1,000,000 interspersed calls to Take and Put for 64 threads

	Fetch and Increment	CAS LL/IC	RW LL/IC 64 padding	RW LL/IC 16 padding	RW LL/IC 32 padding	RW LL/IC no padding
1	0.00	-5.81	-9.76	-10.70	-12.23	-10.91
8	0.00	-22.76	-33.34	-27.24	-32.77	-25.41
16	0.00	-12.59	-25.75	-25.80	-28.05	-27.46
24	0.00	-1.96	-6.00	-2.10	-6.47	-6.61
32	0.00	1.73	-11.61	-6.71	-10.84	-14.91
40	0.00	2.86	-11.66	-28.41	-15.19	-9.38
48	0.00	-1.48	-8.85	-19.53	-7.78	-28.75
56	0.00	-0.05	-5.30	-3.68	-7.88	-10.23
64	0.00	1.33	-12.25	-3.54	-7.44	-15.80

Table 6.3: Percentage improvement of LL/IC objects respect to Fetch&Increment from 1 to 64 threads of execution.

Inner Experiments - Modular Queue Variants

The Enqueue - Dequeue Outer Experiment outcome for 64 processes appears in Figure 6.7, and their respective percentage improvements are shown in Table 6.4. In these experiments, we observe that our best version of the modular queue is the combination of the Compare&Swap-based LL/IC object and the K-basket. In particular, all the queue versions tested based on the N-basket performed worse than those based on the K-basket. For example, taking the best version of the N-basket, which is the one that uses LL/IC object Compare&Swap-based, they have a lousy performance concerning the version conformed by LL/IC object Compare&Swap-based and the K-basket ranging from -1.74% using one thread to -1229.5% using 64 threads. The queue based on the N-basket does not scale well. We observe similar behavior in the queue based on N-basket with Read/WriteLL/IC objects (16 and 64 bytes of padding). They also range from -0.89% to -1356.19% of lousy performance with respect to the performance of the queue that uses K-basket and Compare&Swap-based LL/IC objects.

The queue with K-basket and Read/Write-based LL/IC objects also performed worse, but not so severely. Its performance ranges from -0.79% to -281.49% concerning using K-basket and Compare&Swap-based LL/IC objects. Based on the results obtained, we have decided to test the modular queue that employs K-basket and Compare&Swap-based LL/IC objects against the state-of-the-art queues mentioned in Section 6.2.1. This queue version will be referred to as the Castañeda-Piña queue in the following section.

Outer Experiments

The outcome of the Enqueue - Dequeue Outer Experiment for 64 processes appears in Figure 6.8, and their respective percentage improvement is shown in Table 6.5. In

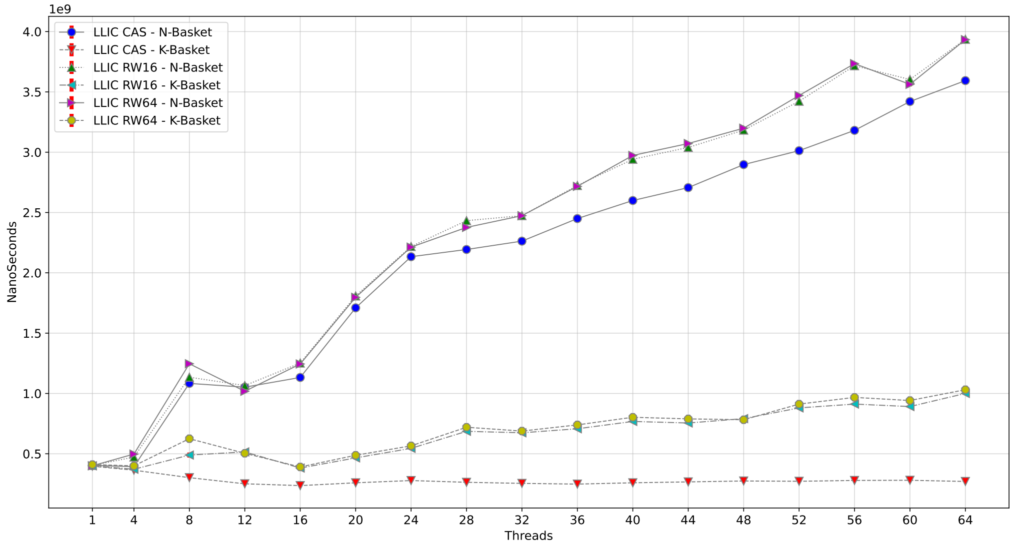


Figure 6.7: Mean times for Enqueue - Dequeue in inner experiments. 1,000,000 interspersed calls to Enqueue and Dequeue for 64 threads

	LLIC CAS - N-Basket	LLIC CAS - K-Basket	LLIC RW16 - N-Basket	LLIC RW16 - K-Basket	LLIC RW64 - N-Basket	LLIC RW64 - K-Basket
1	-1.74	0.00	-1.62	-0.79	-0.89	-3.39
8	-258.89	0.00	-275.82	-62.07	-312.78	-106.99
16	-379.75	0.00	-430.37	-61.63	-427.12	-65.46
24	-667.53	0.00	-697.90	-96.02	-696.10	-103.47
32	-789.13	0.00	-872.14	-164.84	-872.05	-170.39
40	-902.46	0.00	-1034.40	-196.16	-1046.65	-209.51
48	-956.29	0.00	-1060.54	-188.44	-1066.28	-185.11
56	-1040.85	0.00	-1233.05	-227.03	-1239.22	-246.94
64	-1229.50	0.00	-1356.19	-270.04	-1355.10	-281.49

Table 6.4: Percentage improvement of Enqueue - Dequeue respect to LL/IC Compare&Swap & K-Basket from 1 to 64 threads of execution.

these experiments, we observe that Yang-Mellor Crummey's queue performed best in almost every execution, followed closely by Ramalhete's Fetch&Increment queue (Fetch&Increment queue) and the LCRQ queue.

Their graphs look very similar; however, for executions using few cores occasionally, LCRQ and the Fetch&Increment queues have some improvements over the performance of the Yang-Mellor Crummey queue. The Fetch&Increment queue in some executions has an improvement ranging from 0.87% to 6.92%, but after 16 threads, its improvements begin to descend, ranging from -5.59% to -50.75%. LCRQ's negative improvement ranged from -6.96% to -193.12%. In some moments, its improvement grows up to 5.61%.

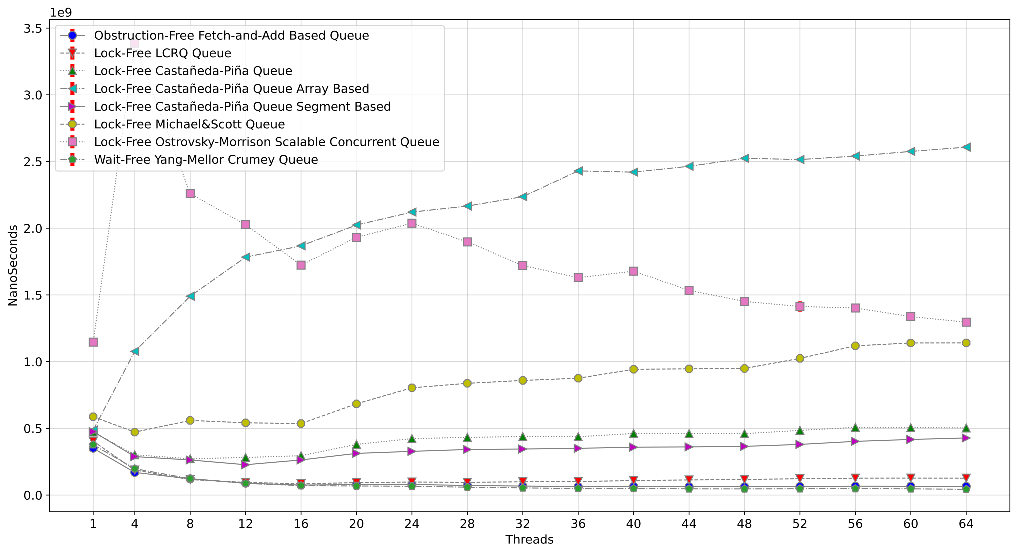


Figure 6.8: Mean times for Enqueue - Dequeue in outer experiments. 1,000,000 interspersed calls to Enqueue and Dequeue for 64 threads

According to performance analysis, the Castañeda-Piña list of arrays and its classic versions are the following queues that perform better among all queues. Although their overall performance is similar, the list-of-arrays version outperforms the classic version. The classic version shows a negative improvement ranging from -25.25% to -1063.06%, whereas the list-of-array version ranges from -26.03% to -890.09%, both of which are inferior to Yang-Mellor Crummey's queue. However, they perform better than other reported queues, as illustrated in Figure 6.8. A reason the list of arrays performed better than the classic is due to the allocation/deallocation of small pieces of memory instead of big chunks, like in the classic version.

	Fetch-and-Add	LCRQ	Castañeda-Piña	Castañeda-Piña Array	Castañeda-Piña Segments	Michael and Scott	Ostrovsky-Morrison	YMC
1	6.92	-6.96	-25.25	-29.12	-26.03	-55.65	-203.81	0.00
8	0.87	5.61	-121.32	-1112.60	-114.00	-355.21	-1738.11	0.00
16	-5.59	-19.50	-316.54	-2537.73	-271.74	-656.75	-2333.73	0.00
24	-20.50	-48.03	-540.67	-3112.91	-396.35	-1117.93	-2986.94	0.00
32	-25.66	-85.23	-717.10	-4067.25	-542.98	-1500.82	-3106.55	0.00
40	-33.56	-122.33	-843.99	-4849.80	-633.71	-1827.70	-3332.12	0.00
48	-34.40	-152.73	-895.45	-5358.40	-689.08	-1951.75	-3039.06	0.00
56	-38.72	-164.22	-957.90	-5195.85	-739.01	-2231.83	-2822.57	0.00
64	-50.75	-193.12	-1063.09	-5931.08	-890.09	-2538.04	-2897.45	0.00

Table 6.5: Percentage improvement of Enqueue - Dequeue respect to Yang and Mellor-Crummey Queue from 1 to 64 threads of execution.

It has been observed that the Michael-Scott and Ostrovsky-Morrison queues underperformed compared to the previous queues. In the case of Michael-Scott's queue, we noticed a decline in performance ranging from -55.65% to -2,538.04% compared to Yang-Mellor Crummey's queue. However, the decline in performance is consistent with the increase in the number of threads.

Similarly, for Ostrovsky-Morrison's queue, the performance deteriorates as the number of threads increases. For instance, the performance for one thread was found to be close to -55% as compared to Yang-Mellor Crummey's queue using one thread as well. However, when the number of threads increased from 4 to 32, we observed a further decline in performance ranging from -1738% to -3106%. After this number of threads, the performance improvement began to reduce until it reached a value close to -2807% compared to the performance of Yang-Mellor Crummey's queue for the same number of threads.

The Castañeda-Piña queue using dynamic arrays had the worst overall performance, exhibiting a non-scalable behavior that only increases in time as the number of threads increases, ranging from -29.12% to -5931.08%. A possible reason for the bad performance is the time it takes to double its size and the contention while this operation is performed.

CHAPTER 7

Discussion and Conclussions

This thesis delves into the evolution of concurrent computing and the shift from traditional to more flexible approaches when programming concurrent algorithms. The primary objective of this study was to determine whether it is possible to implement meaningful and useful objects using only synchronization mechanisms among the simplest ones without compromising performance in practical settings.

In Chapter 4, the problem of work-stealing was addressed, and the limits of the standard asynchronous Read/Write wait-free, shared memory model were explored. In Chapter 5, the focus shifted towards building objects from a modular perspective while keeping in mind the use of simple synchronization mechanisms. Specifically, a modular queue was built, where some components can be implemented using only Read/Write operations.

In Chapter 6, we present an experimental evaluation to measure the performance of the algorithms presented in Chapters 4 and 5. For work-stealing, the study reveals that the use of simple mechanisms can compete and even, in some cases, outperform state-of-the-art algorithms. In the case of the modular queue, the study reveals that the queue cannot compete directly against the fastest state-of-the-art queues. However, its performance is good enough, and the performance lies in particular implementations of its modules.

Case Study: Work-Stealing In Chapter 4, we studied the use of multiplicity applied to work-stealing. We studied two relaxations for work-stealing, called multiplicity and weak multiplicity. Both of them allow a task to be extracted by more than one Take/Steal operation, but each process can take the same task at most once; however, the relaxation can arise only in concurrency. For the first relaxation, this property is directly guaranteed by the definition of set-linearizability. The second relaxation follows from the fact that solutions must be sequentially-exact. We presented two Read/Write, wait-free algorithms for the relaxations, both devoid of Read-After-Write synchronization patterns. Moreover, the second algorithm is fence-free with constant step complexity. To our knowledge, these are the first algorithms for relaxations of work-stealing having all these properties, evading the known impossibility result [9] in all their high-level operations. From the theoretical perspective of the consensus number hierarchy [41], we have thus shown that work-stealing with multiplicity and weak multiplicity lay at the lowest level with objects whose consensus number is one. We also argued that the idempotent work-stealing [65] does not solve either work-stealing with multiplicity or weak multiplicity. Therefore, the relaxations and algorithms proposed here provide stronger guarantees. An experimental evaluation showed that the benefits in the performance of work-stealing with relaxed semantics depend on the type of application and the complexity of the work associated with a task. Therefore, it cannot be guaranteed that relaxations of work-stealing will always lead to improvements.

Viewed collectively, our results show that the simplest synchronization mechanisms suffice to solve non-trivial coordination problems without compromising performance in some practical applications.

Case Study: Modular Baskets Queue In Chapter 5, we adopted a modular approach to building concurrent objects using simple synchronization mechanisms. We designed a modular concurrent queue with multi-producer and multi-consumer semantics. We proposed two concurrent objects that act as modules for the modular queue: baskets and LL/IC objects. The baskets contain groups of items that were enqueued concurrently and can be dequeued in any order. The LL/IC objects store the head and tail of the queue and allow concurrent manipulation of the enqueues and dequeues.

We introduced a general modular basket queue algorithm that utilizes an infinite array of basket objects along with two LL/IC objects to store the head and the tail. Two different LL/IC implementations were presented, one that relies solely on Read/Write operations and another that utilizes the Compare&Swap instruction. In addition, we presented two distinct basket implementations. The first implementation follows an approach similar to the LCRQ algorithm by Morrison and Afek [67]. In contrast, the second is reminiscent of locally generic data structure implementations based on the work of Henzinger et al. [33]. However, since the first approach of this modular queue was designed using infinite arrays, we presented a second approach that considers the problem of a realistic implementation not relying on infinite arrays. Therefore, two queue variants were presented: a dynamic array version and a list-of-arrays version.

The results of an experimental evaluation revealed that the most efficient approach for implementing the modular queue was using the Compare&Swap-based LL/IC object with the K-basket. With respect to the version of Read/Write, this one was slightly less performant. Comparing the modular queue against state-of-the-art queues shows that the queue cannot directly compete with the fastest queues. However, its performance is still good enough, showing that the performance of the modular queue lies in particular implementations of its modules. This last comes from evaluating three distinct implementations of the modular queue. The first was implementing the first approach with minimal changes, and the other two were based on the variants mentioned at the end of the previous paragraph. Results showed that the first approach and the list-of-arrays version had better performance, with the latter being the best performant. The version based on dynamic arrays does not scale as well as expected.

Viewed collectively, our results show that modular and concurrent algorithms can be built whose performance depends only on the performance of the algorithm's modules. They also show that a simple synchronization mechanism can still be used to develop such algorithms.

Future Research The study of the simplest synchronization mechanisms to solve concurrency problems is an ongoing field of research. Attiya et al.'s work [9] has shown that it is impossible to eliminate expensive synchronization in classic and ubiquitous specifications, which raises the question of whether it is possible to bypass this impossibility result in any way. We have considered two possible ways to circumvent this result: (1) by considering relaxed semantics and (2) by making additional assumptions about the model.

In the case of Work-Stealing, we have considered relaxed semantics like Multiplicity [16, 20]. For future research, we are interested in designing algorithms for work-stealing with multiplicity and weak-multiplicity that insert/extract tasks in orders different from FIFO. Also, it is interesting to explore if the techniques in the algorithms from Chapter 4 can be applied to solve relaxed versions of other concurrent objects efficiently. For FIFO queues with multi-producer multi-consumer semantics, we are interested in exploring if the modular design combined with techniques like those mentioned in Chapter 4 can be used to obtain FIFO queues with multiplicity or weak-multiplicity by manipulating the head and the tail through objects of type MaxRead or RMaxRead.

In the case of the modular basket queue and the techniques developed in Chapter 5, we are interested in investigating if there are more efficient ways to implement baskets, for example, using concurrent sets instead of arrays to store the concurrent inputs. We are also interested in testing whether the use of baskets in algorithms like LCRQ and Mellor-Crummey's queue can improve their performance and help with some problems like latency, as pointed out by Ramalhete [77], which are out of the scope of this work.

Bibliography

[1] Adas, D., and Friedman, R. Brief announcement: Jiffy: A fast, memory efficient, wait-free multi-producers single-consumer queue. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference* (2020), pp. 50:1–50:3.

[2] Adve, S. V., and Boehm, H. Memory models: a case for rethinking parallel languages and hardware. *Commun. ACM 53*, 8 (2010), 90–101.

[3] Adve, S. V., and Hill, M. D. Weak ordering - A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture, Seattle, WA, USA, June 1990* (1990), pp. 2–14.

[4] Afek, Y., Korland, G., and Yanovsky, E. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings* (2010), pp. 395–410.

[5] Alistarh, D., Brown, T., Kopinsky, J., Li, J. Z., and Nadiradze, G. Distributionally linearizable data structures. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018* (2018), pp. 133–142.

[6] AMD. Amd64 architecture programmer's manual volume 2: System programming. *2023 2* (2023).

[7] Aspnes, J., Attiya, H., and Censor-Hillel, K. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM 59*, 1 (2012), 2:1–2:24.

[8] Attiya, H., Guerraoui, R., Hendler, D., and Kuznetsov, P. The complexity of obstruction-free implementations. *J. ACM 56*, 4 (2009), 24:1–24:33.

[9] Attiya, H., Guerraoui, R., Hendler, D., Kuznetsov, P., Michael, M. M., and Vechev, M. T. 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), T. Ball and M. Sagiv, Eds., ACM, pp. 487–498.

[10] Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J. P., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., and Zhang, G. The design of openmp tasks. *IEEE Trans. Parallel Distributed Syst. 20*, 3 (2009), 404–418.

[11] Bader, D. A., and Cong, G. A fast, parallel spanning tree algorithm for symmetric multiprocessors. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* (April 2004), pp. 38–.

[12] Blumofe, R. D., Joerg, C. F., Kuszmaul, B. C., Leiserson, C. E., Randall, K. H., and Zhou, Y. Cilk: An efficient multithreaded runtime system. *J. Parallel Distributed Comput. 37*, 1 (1996), 55–69.

[13] Boehm, H., and Adve, S. V. Foundations of the C++ concurrency memory model. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008* (2008), pp. 68–78.

[14] Böhm, M., and Speckenmeyer, E. A fast parallel sat-solver - efficient workload balancing. *Ann. Math. Artif. Intell. 17*, 3-4 (1996), 381–400.

[15] Castañeda, A., Rajsbaum, S., and Raynal, M. Unifying concurrent objects and distributed tasks: Interval-linearizability. *J. ACM 65*, 6 (2018), 45:1–45:42.

[16] Castañeda, A., Rajsbaum, S., and Raynal, M. Set-linearizable implementations from read/write operations: Sets, fetch &increment, stacks and queues with multiplicity. *Distributed Comput. 36*, 2 (2023), 89–106.

[17] Castañeda, A., and Piña, M. Fully read/write fence-free work-stealing with multiplicity. In *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)* (2021), S. Gilbert, Ed., vol. 209 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 16:1–16:20.

[18] Castañeda, A., and Piña, M. Modular baskets queue. ArXiv.org, 2022. `https://arxiv.org/abs/2205.06323`.

[19] Castañeda, A., and Piña, M. Read/write fence-free work-stealing with multiplicity. *J. Parallel Distributed Comput. 186* (2024), 104816.

[20] Castañeda, A., Rajsbaum, S., and Raynal, M. Relaxed queues and stacks from read/write operations. In *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France (Virtual Conference)* (2020), Q. Bramas, R. Oshman, and P. Romano, Eds., vol. 184 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 13:1–13:19.

[21] Charles, P., Grothoff, C., Saraswat, V. A., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA* (2005), pp. 519–538.

[22] Chase, D., and Lev, Y. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2005), SPAA '05, Association for Computing Machinery, p. 21–28.

[23] Ellen, F., Hendler, D., and Shavit, N. On the inherent sequentiality of concurrent objects. *SIAM J. Comput. 41*, 3 (2012), 519–536.

[24] Fatourou, P., and Kallimanis, N. D. A highly-efficient wait-free universal construction. In *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011)* (2011), pp. 325–334.

[25] Fatourou, P., and Kallimanis, N. D. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012* (2012), pp. 257–266.

[26] Feldman, Y., Dershowitz, N., and Hanna, Z. Parallel multithreaded satisfiability solver: Design and implementation. In *Proceedings of the 3rd International Workshop on Parallel and Distributed Methods in Verification, PDMC 2004, London, UK, September 4, 2004* (2004), pp. 75–90.

[27] Flood, C. H., Detlefs, D., Shavit, N., and Zhang, X. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA* (2001).

[28] Forsyth, D. *Probability and statistics for computer science*. Springer, 2018.

[29] Fraser, K. *Practical lock-freedom*. PhD thesis, University of Cambridge, UK, 2004.

[30] Frigo, M., Leiserson, C. E., and Randall, K. H. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998* (1998), pp. 212–223.

[31] Frigo, M., Leiserson, C. E., and Randall, K. H. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998* (1998), pp. 212–223.

[32] Georges, A., Buytaert, D., and Eeckhout, L. Statistically rigorous java performance evaluation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada* (2007), R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., Eds., ACM, pp. 57–76.

[33] Haas, A., Henzinger, T. A., Holzer, A., Kirsch, C. M., Lippautz, M., Payer, H., Sezgin, A., Sokolova, A., and Veith, H. Local linearizability for concurrent container-type data structures. In *27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada* (2016), pp. 6:1–6:15.

[34] Haas, A., Lippautz, M., Henzinger, T. A., Payer, H., Sokolova, A., Kirsch, C. M., and Sezgin, A. Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation. In *Computing Frontiers Conference, CF'13, Ischia, Italy, May 14 - 16, 2013* (2013), pp. 17:1–17:9.

[35] Hamadi, Y., and Sais, L., Eds. *Handbook of Parallel Constraint Reasoning*. Springer, 2018.

[36] Hendler, D., Lev, Y., Moir, M., and Shavit, N. A dynamic-sized non-blocking work stealing deque. *Distributed Comput. 18*, 3 (2006), 189–207.

[37] Hendler, D., and Shavit, N. Non-blocking steal-half work queues. In *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing* (New York, NY, USA, 2002), PODC '02, Association for Computing Machinery, p. 280–289.

[38] Henzinger, T. A., Kirsch, C. M., Payer, H., Sezgin, A., and Sokolova, A. Quantitative relaxation of concurrent data structures. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013* (2013), pp. 317–328.

[39] Henzinger, T. A., Sezgin, A., and Vafeiadis, V. Aspect-oriented linearizability proofs. In *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings* (2013), pp. 242–256.

[40] Herlihy, M. Impossibility results for asynchronous PRAM (extended abstract). In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '91, Hilton Head, South Carolina, USA, July 21-24, 1991* (1991), pp. 327–336.

[41] Herlihy, M. Wait-free synchronization. *ACM Trans. Program. Lang. Syst. 13*, 1 (1991), 124–149.

[42] Herlihy, M., Luchangco, V., and Moir, M. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Distributed Computing, 16th International Conference, DISC 2002, Toulouse, France, October 28-30, 2002 Proceedings* (2002), pp. 339–353.

[43] Herlihy, M., Luchangco, V., and Moir, M. Obstruction-free synchronization: Double-ended queues as an example. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003), 19-22 May 2003, Providence, RI, USA* (2003), pp. 522–529.

[44] Herlihy, M., and Shavit, N. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

[45] Herlihy, M., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (1990), 463–492.

[46] Hoffman, M., Shalev, O., and Shavit, N. The baskets queue. In *Proceedings of the 11th International Conference on Principles of Distributed Systems* (Berlin, Heidelberg, 2007), OPODIS'07, Springer-Verlag, p. 401–414.

[47] IBM. IBM100 - Power 4 : The First Multi-Core, 1GHz Processor — ibm.com. `https://www.ibm.com/ibm/history/ibm100/us/en/icons/power4/#:~:text=In%202001%2C%20IBM%20introduced%20the,more%20than%20170%20million%20transistors.` [Accessed 02-Jan-2023].

[48] Intel. Intel® 64 and ia-32 architectures software developer's manual. *2023 2*, 11 (2023), 0–40.

[49] ISO International Standard, I. std::memory_order, 2020.

[50] J., W. B., and Scott, M. L. False sharing and its effect on shared memory performance. In *USENIX SEDMS 1993* (USA, 1993), USENIX Association, p. 3.

[51] Jayanti, P., Tan, K., and Toueg, S. Time and space lower bounds for nonblocking implementations. *SIAM J. Comput. 30*, 2 (2000), 438–456.

[52] Johnen, C., Khattabi, A., and Milani, A. Efficient wait-free queue algorithms with multiple enqueuers and multiple dequeuers. In *26th International Conference on Principles of Distributed Systems, OPODIS 2022, December 13-15, 2022, Brussels, Belgium* (2022), pp. 4:1–4:19.

[53] Kirsch, C. M., Lippautz, M., and Payer, H. Fast and scalable, lock-free k-fifo queues. In *Parallel Computing Technologies - 12th International Conference, PaCT 2013, St. Petersburg, Russia, September 30 - October 4, 2013. Proceedings* (2013), pp. 208–223.

[54] Kirsch, C. M., Payer, H., Röck, H., and Sokolova, A. Performance, scalability, and semantics of concurrent FIFO queues. In *Algorithms and Architectures for Parallel Processing - 12th International Conference, ICA3PP 2012, Fukuoka, Japan, September 4-7, 2012, Proceedings, Part I* (2012), pp. 273–287.

[55] Kogan, A., and Petrank, E. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011* (2011), pp. 223–234.

[56] Ladan-Mozes, E., and Shavit, N. An optimistic approach to lock-free FIFO queues. *Distributed Comput. 20*, 5 (2008), 323–341.

[57] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers C-28 9* (September 1979), 690–691.

[58] Lea, D. A java fork/join framework. In *Proceedings of the ACM 2000 Java Grande Conference, San Francisco, CA, USA, June 3-5, 2000* (2000), pp. 36–43.

[59] Lilja, D. J. *Measuring Computer Performance.* Cambridge University Press, Jul 2000.

[60] Manson, J., Pugh, W. W., and Adve, S. V. The java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005* (2005), pp. 378–391.

[61] McKenney, P. E., Appavoo, J., Kleen, A., Krieger, O., Russell, R., Sarma, D., and Soni, M. Read-copy update. In *AUUG Conference Proceedings* (2001), AUUG, Inc., p. 175.

[62] Michael, M. M. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. 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. 21–30.

[63] Michael, M. M. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distributed Syst. 15*, 6 (2004), 491–504.

[64] Michael, M. M., and Scott, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996* (1996), J. E. Burns and Y. Moses, Eds., ACM, pp. 267–275.

[65] Michael, M. M., Vechev, M. T., and Saraswat, V. A. 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), D. A. Reed and V. Sarkar, Eds., ACM, pp. 45–54.

[66] Milman-Sela, G., Kogan, A., Lev, Y., Luchangco, V., and Petrank, E. BQ: A lock-free queue with batching. *ACM Trans. Parallel Comput. 9*, 1 (2022), 5:1–5:49.

[67] Morrison, A., and Afek, Y. Fast concurrent queues for x86 processors. *SIGPLAN Not. 48*, 8 (Feb. 2013), 103–112.

[68] Morrison, A., and Afek, Y. Fence-free work stealing on bounded tso processors. *SIGPLAN Not. 49*, 4 (Feb. 2014), 413–426.

[69] Morrison, A., and Afek, Y. Fence-free work stealing on bounded tso processors. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2014), ASPLOS '14, Association for Computing Machinery, p. 413–426.

[70] Nagarajan, V., Sorin, D. J., Hill, M. D., and Wood, D. A. *A Primer on Memory Consistency and Cache Coherence, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2020.

[71] Neiger, G. Set-linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994* (1994), p. 396.

[72] Oracle. Java language specification: Chapter 17. threads and locks, 2017.

[73] Oracle. Varhandle: Java api, 2017.

[74] Ostrovsky, O., and Morrison, A. Scaling concurrent queues by using htm to profit from failed atomic operations. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2020), PPoPP '20, Association for Computing Machinery, p. 89–101.

[75] Owens, S., Sarkar, S., and Sewell, P. A better x86 memory model: x86-tso. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings* (2009), pp. 391–407.

[76] Papadimitriou, C. H. The serializability of concurrent database updates. *J. ACM 26*, 4 (1979), 631–653.

[77] Ramalhete, P., and Correia, A. Fetch-and-add array queue - mpmc lock-free queue, Nov 2016.

[78] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G. R., and Kozyrakis, C. Evaluating mapreduce for multi-core and multiprocessor systems. In *13st International Conference on High-Performance Computer Architecture (HPCA-13 2007), 10-14 February 2007, Phoenix, Arizona, USA* (2007), pp. 13–24.

[79] Rihani, H., Sanders, P., and Dementiev, R. Brief announcement: Multiqueues: Simple relaxed concurrent priority queues. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015* (2015), pp. 80–82.

[80] Scott, M. L. *Shared-Memory Synchronization*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[81] Sewell, P., Sarkar, S., Owens, S., Nardelli, F. Z., and Myreen, M. O. x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM 53*, 7 (2010), 89–97.

[82] Shavit, N. Data structures in the multicore age. *Commun. ACM 54*, 3 (2011), 76–84.

[83] Shavit, N., and Taubenfeld, G. The computability of relaxed data structures: queues and stacks as examples. *Distributed Comput. 29*, 5 (2016), 395–407.

[84] SPARC International, Inc, C. *The SPARC architecture manual: version 8*. Prentice-Hall, Inc., 1992.

[85] Sundell, H. Wait-free reference counting and memory management. In *19th International Parallel and Distributed Processing Symposium (IPDPS 2005), CD-ROM / Abstracts Proceedings, 4-8 April 2005, Denver, CO, USA* (2005), IEEE Computer Society.

[86] Talmage, E., and Welch, J. L. Relaxed data types as consistency conditions. *Algorithms 11*, 5 (2018), 61.

[87] Turing, A. M. On computable numbers, with an application to the entscheidungsproblem. *Proc. London Math. Soc. s2-42*, 1 (1937), 230–265.

[88] Valois, J. D. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995* (1995), J. H. Anderson, Ed., ACM, pp. 214–222.

[89] von Neumann, J. First draft of a report on the EDVAC. *IEEE Ann. Hist. Comput. 15*, 4 (1993), 27–75.

[90] Yang, C., and Mellor-Crummey, J. M. A wait-free queue as fast as fetch-and-add. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016* (2016), R. Asenjo and T. Harris, Eds., ACM, pp. 16:1–16:13.

APPENDIX A

Work-Stealing Results

A.1 Results of Zero Cost Experiments

This appendix shows the results obtained by executing the zero-cost experiments following the methodology suggested by Georges, Buytaert, and Eeckout [32]. The data is divided in the case of the experiment of Puts-Takes and the case of Puts-Steals.

	1	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64
Lock-Free LCRQ Queue	403572408.43	187820883.50	116026264.60	94770156.63	84622830.50	93050854.37	97720927.97	95509834.43	99412632.57	101217631.23	108716588.03	113385554.47	116865624.87	122543525.30	126754099.80	127109938.03	126730354.90
Obstruction-Free Fetch-and-Add Based Queue	351200027.63	169692112.77	121854879.07	88994036.27	74776242.40	78627962.80	79543299.43	71817559.87	67443713.50	63240889.47	65311551.40	63473601.93	62148500.10	64238033.60	66547657.57	65885871.70	65178512.30
Lock-Free Castañeda-Piña Queue Array Based	487165396.30	1077150109.93	1490503277.57	1783321075.57	1867961323.20	2023925634.47	2120916067.10	2166062984.03	2236554396.10	2429715567.90	2420424604.93	2464267895.70	2523987255.27	2514987713.47	2540588509.17	2575744556.40	2607573629.43
Lock-Free Michael&Scott Queue	587248865.70	471544279.67	559538345.73	541606321.37	535908982.53	684008030.47	803985551.10	837462016.30	859154816.23	875829430.07	942633485.97	946151498.10	948736421.87	1024468571.63	1118652395.07	1140158428.23	1140572453.10
Lock-Free Castañeda-Piña Queue	472555550.90	301829643.73	272038021.70	281703838.03	294978630.73	379995043.80	422919854.07	432414808.13	438538137.47	437121875.30	461605337.17	460308301.33	460302515.57	484756656.03	507507288.80	505315540.43	502870762.67
Wait-Free Yang-Mellor Crumey Queue	377297171.80	197394522.23	122918448.70	87396589.83	70816962.53	67941366.37	66012273.80	59065987.40	53669770.97	48970421.20	48899447.00	46628124.97	46240439.33	47095151.57	47973167.30	46690629.67	43235588.63
Lock-Free Castañeda-Piña Queue Segment Based	475511479.40	288435581.67	263046870.43	227342770.57	263256036.93	312408909.57	327654824.50	341570968.90	345087614.40	350204025.40	358781500.57	360854102.40	364872124.93	379651287.00	402499664.00	416756883.57	428069875.03
Lock-Free Ostrovsky-Morrison Scalable Concurrent Queue	1146251708.13	3387292159.17	2259382162.50	2026162066.43	1723493401.93	1932146663.93	2037756243.03	1897402494.37	1720945612.37	1629208308.67	1678287663.47	1534113476.07	1451514990.13	1413332353.37	1402048064.73	1337866333.20	1295963945.30

Table A.1: Mean times for Enqueue - Dequeue experiment for 64 threads

A.1.1 Puts-Takes

A.1.2 Puts-Steals

	Chase-Lev	Cilk THE	Idempotent FIFO	Idempotent LIFO	Idempotent DEQUE	WS WMult	B. WS WMult	WS WMult Lists	B. WS WMult Lists
Mean	206105723.49	211217590.84	222570807.61	187686110.36	311857192.28	107167632.11	952902162.60	223914128.86	375494492.90
Low Limit	205713311.48	210753568.00	222029102.93	187305885.73	311337719.28	106944566.24	932470976.30	221872884.15	355430610.14
High Limit	206498135.50	211681613.68	223112512.29	188066334.99	312376665.28	107390697.98	973333348.90	225955373.57	395558375.66
Confidence Interval	784824.01	928045.67	1083409.37	760449.26	1038946.01	446131.74	40862372.61	4082489.43	40127765.52

Table A.2: The values shown in the table were calculated under the methodology suggested in [32]. These values in nanoseconds, are the mean time, the confidence interval limits (high and low) and the size region of the confidence interval. The zero cost experiment for puts and takes was performed with an initial structure size of 256 items for each worker. The amount of operations to perform was of 10000000 operations.

	Chase-Lev	Cilk THE	Idempotent FIFO	Idempotent LIFO	Idempotent DEQUE	WS WMult	B. WS WMult	WS WMult Lists	B. WS WMult Lists
Mean	197232290.46	202217219.83	209743459.03	184687931.23	285572059.50	107538022.02	883850090.93	180476312.87	311215062.70
Low Limit	196858011.16	201778117.71	209441782.36	184346868.43	284992355.03	107304017.73	853055195.43	179441832.20	296671676.50
High Limit	197606569.76	202656321.95	210045135.70	185028994.03	286151763.97	107772026.31	914644986.43	181510793.54	325758448.90
Confidence Interval	748558.61	878204.23	603353.34	682125.60	1159408.95	468008.59	61589791.00	2068961.35	29086772.40

Table A.3: The values shown in the table were calculated under the methodology suggested in [32]. These values in nanoseconds, are the mean time, the confidence interval limits (high and low) and the size region of the confidence interval. The zero cost experiment for puts and takes was performed with an initial structure size of 1000000 items for each worker. The amount of operations to perform was of 10000000 operations.

	Chase-Lev	Cilk THE	Idempotent FIFO	Idempotent LIFO	Idempotent DEQUE	WS WMult	B. WS WMult	WS WMult Lists	B. WS WMult Lists
Mean	124295472.28	122986153.27	104572052.73	159480162.96	180129261.80	89607898.79	311403582.39	168305627.62	361147626.67
Low Limit	124085001.89	122774139.69	103932958.18	159024771.73	179516019.76	89485513.68	306677492.57	167645846.38	357881600.91
High Limit	124505942.67	123198166.85	105211147.28	159935554.19	180742503.84	89730283.90	316129672.21	168965408.86	364413652.43
Confidence Interval	420940.77	424027.16	1278189.10	910782.45	1226484.07	244770.22	9452179.63	1319562.48	6532051.52

Table A.4: The values shown in the table were calculated under the methodology suggested in [32]. These values in nanoseconds, are the mean time, the confidence interval limits (high and low) and the size region of the confidence interval. The zero cost experiment for puts and takes was performed with an initial structure size of 10000000 items for each worker. The amount of operations to perform was of 10000000 operations.

	Chase-Lev	Cilk THE	Idempotent FIFO	Idempotent LIFO	Idempotent DEQUE	WS WMult	B. WS WMult	WS WMult Lists	B. WS WMult Lists
Mean	198416960.62	293803996.52	216277326.98	220720524.13	319510678.37	106704209.84	1012954992.34	225121073.52	454634203.16
Low Limit	197939052.88	293090297.22	215818699.72	219989525.44	318822843.61	106447024.18	997435581.59	223531813.39	427898503.65
High Limit	198894868.36	294517695.82	216735954.24	221451522.82	320198513.13	106961395.50	1028474403.09	226710333.65	481369902.67
Confidence Interval	955815.48	1427398.60	917254.52	1461997.39	1375669.52	514371.33	31038821.50	3178520.26	53471399.03

Table A.5: The values shown in the table were calculated under the methodology suggested in [32]. These values in nanoseconds, are the mean time, the confidence interval limits (high and low) and the size region of the confidence interval. The zero cost experiment for puts and steals was performed with an initial structure size of 256 items for each worker. The amount of operations to perform was of 10000000 operations.

	Chase-Lev	Cilk THE	Idempotent FIFO	Idempotent LIFO	Idempotent DEQUE	WS WMult	B. WS WMult	WS WMult Lists	B. WS WMult Lists
Mean	188742815.65	285389986.59	204578776.35	219054343.94	294326647.21	105862001.14	946421899.22	179108873.00	359239862.76
Low Limit	188321766.63	284765322.59	204200829.72	218248707.65	293747378.70	105465468.28	927800797.17	177966882.72	343069305.85
High Limit	189163864.67	286014650.59	204956722.98	219859980.23	294905915.72	106258534.00	965043001.27	180250863.28	375410419.67
Confidence Interval	842098.03	1249328.01	755893.26	1611272.57	1158537.02	793065.71	37242204.11	2283980.57	32341113.83

Table A.6: The values shown in the table were calculated under the methodology suggested in [32]. These values in nanoseconds, are the mean time, the confidence interval limits (high and low) and the size region of the confidence interval. The zero cost experiment for puts and steals was performed with an initial structure size of 1000000 items for each worker. The amount of operations to perform was of 10000000 operations.

	Chase-Lev	Cilk THE	Idempotent FIFO	Idempotent LIFO	Idempotent DEQUE	WS WMult	B. WS WMult	WS WMult Lists	B. WS WMult Lists
Mean	119167926.10	212595892.45	98402983.58	193260103.58	188956689.55	89542389.85	358625879.40	166818125.62	402391945.69
Low Limit	118432187.01	212092094.09	97987902.47	192623917.53	188514507.55	89384931.83	355364057.87	166082636.81	392057077.47
High Limit	119903665.19	213099690.81	98818064.69	193896289.63	189398871.55	89699847.87	361887700.93	167553614.43	412726813.91
Confidence Interval	1471478.18	1007596.71	830162.22	1272372.10	884364.00	314916.03	6523643.06	1470977.62	20669736.45

Table A.7: The values shown in the table were calculated under the methodology suggested in [32]. These values in nanoseconds, are the mean time, the confidence interval limits (high and low) and the size region of the confidence interval. The zero cost experiment for puts and steals was performed with an initial structure size of 10000000 items for each worker. The amount of operations to perform was of 10000000 operations.

A.2

Results of Parallel Spanning Tree experiments

A.2.1 Time measurements of Parallel Spanning Tree experiment

In this section, we report the measurements of the executions of the spanning tree experiment, which were carried out using rigorous statistical methodology and thus have reliable values. The executions are shown for the following graphs: Torus 2D, Torus 2D 60%, Torus 3D, Torus 3D 40%, and Random. Each one has a million vertices, and evaluations are made on its directed and undirected versions. In the experiments, for all instances of the work-stealing algorithms, an initial size of 256 and 1,000,000 elements to store was established. The order of these figures and tables is as follows:

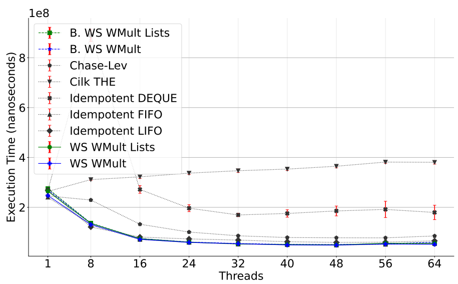
1. Directed and undirected Torus 2D with 256 and 1,000,000 of items for the starting size for work-stealing structures. Figure A.1, and tables A.8, A.9, A.10 and A.11.

2. Directed and undirected Torus 2D 60% with 256 and 1,0000,000 of items for the starting size for work-stealing structures. Figure A.2, and tables A.12, A.13, A.14, and A.15.

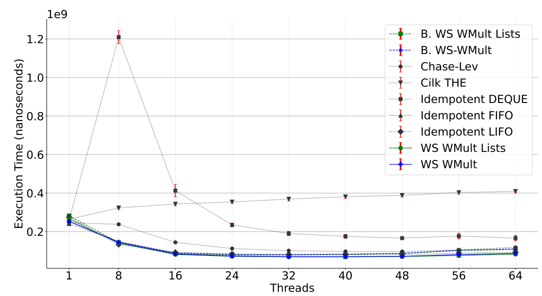
3. Directed and undirected Torus 3D with 256 and 1,000,000 items for the starting size for work-stealing structures. Figure A.3, and tables A.16, A.17, A.18, and A.19.

4. Directed and undirected Torus 3D 40% with 256 and 1,000,000 items for the starting size for work-stealing structures. Figure A.4, and tables A.20, A.21, A.22, and A.23.

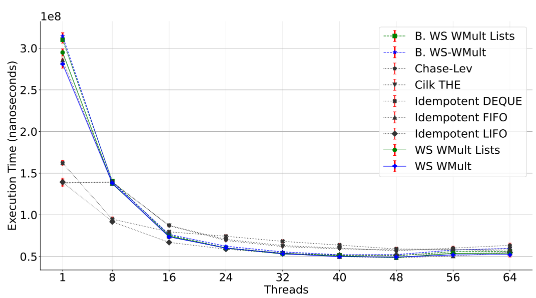
5. Directed and undirected Random Graph with 256 and 1,000,000 items for the starting size for work-stealing structures. Figure A.5, and tables A.24, A.25, A.26, and A.27.



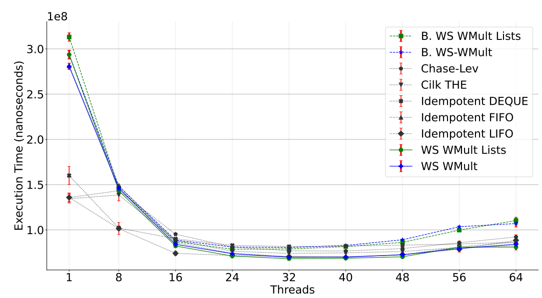
(a) Mean times for the graph application benchmark. These are the results for the 2D Torus Directed graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of 256 entries.



(b) Mean times for the graph application benchmark. These are the results for the 2D Torus Directed graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of 1,000,000 entries.



(c) Mean times for the graph application benchmark. These are the results for the 2D Torus Undirected graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of 256 entries.



(d) Mean times for the graph application benchmark. These are the results for the 2D Torus Undirected graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of 1,000,000 entries.

Figure A.1: 2D Torus Directed and Undirected Graph with 256 and 1,000,000 initial sizes respectively.

	1	8	16	24	32	40	48	56	64
B. WS WMult Lists	275477463.00	136788862.74	74324477.77	60730736.43	53505251.35	50743232.51	50292847.11	54089550.83	56234022.28
B. WS WMult	270377949.90	135945596.35	74881098.84	61011775.05	55033500.39	51617389.18	50911385.25	54141726.30	60230204.70
Chase-Lev	245820345.21	229356624.44	132371426.13	100919202.73	85931100.37	79043201.15	78056439.48	77733696.92	85279360.99
Cilk THE	263677364.18	311281988.58	322321281.34	337196064.37	346876936.91	353384223.55	364851039.81	381583332.07	380634053.09
Idempotent DEQUE	268351478.04	890041843.06	271731282.68	196719701.28	169542956.14	175498130.08	185603584.54	191766131.46	179564368.52
Idempotent FIFO	240555867.99	130412115.53	71135466.76	59469052.26	52425282.13	50053083.46	48699445.66	51530675.32	54082623.12
Idempotent LIFO	268557944.01	121240023.29	80842396.35	73393176.30	68756656.08	62265055.27	59531339.13	60499769.67	65800153.04
WS WMult Lists	264386117.47	135442458.17	74359990.60	60075063.34	53356451.31	51045223.01	49362511.61	56191157.64	55810986.14
WS WMult	247514986.11	130992091.43	71628084.25	59275666.19	54259775.64	49197376.25	48457494.80	52845091.43	51699244.96

Table A.8: Mean times for the graph application benchmark. These are the results for the 2D Torus Directed graph. Each algorithm begin its execution with an initial size of 256 items.

	1	8	16	24	32	40	48	56	64
B. WS WMult Lists	282032101.13	143069494.72	85534815.94	78504149.89	78699282.75	81141869.21	86677012.58	101372180.23	107427536.39
B. W WMult	254029656.93	147059256.57	89875139.77	81711438.58	81440512.00	83371685.71	87706342.83	103781818.41	110569292.18
Chase-Lev	246442791.95	237923122.57	144322166.44	112347776.21	100841004.28	97312919.02	96845768.48	105140083.88	118527697.40
Cilk THE	264300653.34	323239004.94	343356947.48	354030678.14	368922285.60	381228576.03	388191674.43	402911552.15	408918000.97
Idempotent DEQUE	260815108.20	1209548909.30	412614106.02	234745065.18	190273084.36	175218721.12	166271433.10	176938712.66	166152657.38
Idempotent FIFO	241691715.81	147779277.84	86794723.48	74580951.70	71319113.57	69953403.69	72951557.67	76481707.91	88300511.33
Idempotent LIFO	274669968.74	132876867.01	92359758.62	84527174.06	80000747.41	79918156.55	81057426.41	87742754.73	93122905.36
WS WMult Lists	266732948.81	139727360.22	81166332.81	71537631.71	68838660.20	68622394.33	70694380.57	76278304.11	82762680.50
WS WMult	253102197.42	144215742.72	83755232.58	73098841.65	69905539.84	70307884.91	71467092.10	81326080.28	86772858.87

Table A.9: Mean times for the graph application benchmark. These are the results for the 2D Torus Directed graph. Each algorithm begins its execution with an initial size of 1000000 items.

	1	8	16	24	32	40	48	56	64
B. WS WMult Lists	310159873.86	140439779.78	75432726.40	60406182.71	53911161.76	51430777.91	50753159.38	55951600.18	56137962.96
B. WS WMult	314452637.35	139249013.63	76542543.20	62311010.87	55528149.56	52097619.34	52246241.13	57724836.67	59747465.26
Chase-Lev	138828051.42	138897518.79	87292876.34	70522950.03	63105704.93	59978683.77	56908800.75	60120171.07	63348521.30
Cilk THE	138028046.12	139574907.75	86992872.27	69182659.71	61861841.16	59163947.75	58000268.78	57943903.04	59164952.68
Idempotent DEQUE	162061744.20	94795698.28	79606910.26	74293432.40	68175669.64	63633505.52	59008790.10	58410327.72	56516387.48
Idempotent FIFO	285722944.41	137771932.41	74313817.07	59775880.18	54199431.72	49858050.44	48473814.48	50603478.05	54748949.06
Idempotent LIFO	139329399.29	91652023.21	66936992.03	58831954.84	53465545.93	51570670.85	51610279.19	53622297.77	54754410.32
WS WMult Lists	294863903.34	137331873.11	74299658.78	60022488.79	53029369.50	50656621.07	48692765.58	53784545.47	53296961.05
WS WMult	281148635.39	137928573.11	73505852.13	60207428.55	53334767.43	49825802.09	49308366.02	51896274.03	52172465.28

Table A.10: Mean times for the graph application benchmark. These are the results for the 2D Torus Undirected graph. Each algorithm begins its execution with an initial size of 256 items.

	1	8	16	24	32	40	48	56	64
B. WS WMult Lists	313137477.28	146059161.29	87094776.55	78679883.57	78976895.23	81553430.32	86244666.42	99783595.98	110398812.46
B. WS WMult	293784827.34	147980221.95	88731594.96	81121304.79	80579591.88	82813668.25	89143510.85	103539594.14	107036269.51
Chase-Lev	135739209.18	143310608.07	95323139.46	81052618.70	76940451.27	76699127.81	79395479.61	85791529.95	92214170.11
Cilk THE	134426266.75	138584211.69	88507523.61	76405888.32	74084595.82	74632761.47	76161617.37	80595336.35	82538974.04
Idempotent DEQUE	160055984.00	102061319.64	90198138.78	82973638.50	81645834.10	82150883.36	83227444.86	84420556.12	86499396.64
Idempotent FIFO	281355092.48	149030688.81	85883852.60	74176611.42	70542525.74	70451746.00	72308567.54	81236910.85	86493594.23
Idempotent LIFO	135852779.09	101491242.31	74122390.31	71756053.52	70549755.43	69617711.32	72795329.85	78634423.92	88549939.13
WS WMult Lists	293649514.77	143351585.30	82231696.72	71153322.35	68221248.95	68690573.69	70317039.66	80917247.72	81000767.24
WS WMult	280609111.62	146014648.66	84057327.09	73737753.72	69754078.34	70027563.45	72554542.24	79464648.48	84181768.65

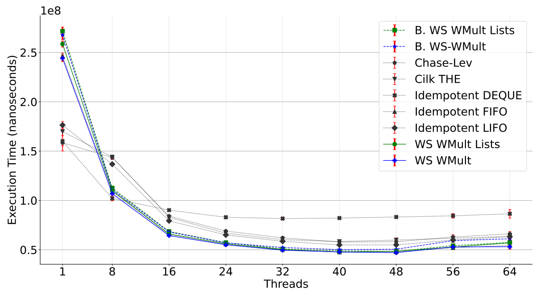
Table A.11: Mean times for the graph application benchmark. These are the results for the 2D Torus Undirected graph. Each algorithm begins its execution with an initial size of 1000000 items.

	1	8	16	24	32	40	48	56	64
B. WS WMult Lists	271606510.37	112596991.25	68569676.02	57301051.36	50764719.52	48888215.29	48355334.41	54134929.14	57394951.90
B. WS WMult	267846777.75	110997123.47	68040897.63	57020102.54	52278828.53	50189180.68	50565768.85	59297832.00	61153079.83
Chase-Lev	158506506.33	143484647.03	84147418.00	68976751.32	62140583.83	57845418.33	58101207.31	62954762.83	66131499.15
Cilk THE	169847040.68	144015298.42	82866364.81	66790422.97	60273306.47	58391753.66	59823679.89	61730076.35	63568056.79
Idempotent DEQUE	160055984.00	102061319.64	90198138.78	82973638.50	81645834.10	82150883.36	83227444.86	84420556.12	86499396.64
Idempotent FIFO	246549072.59	108976897.48	65641256.85	55840030.43	50411383.53	47746138.13	47814235.24	52140361.59	54387574.39
Idempotent LIFO	176418657.30	136781388.73	79376036.88	65015427.06	58639994.94	54894622.52	54955449.08	59872541.30	63279878.32
WS WMult Lists	258595656.45	109902199.41	66031946.41	56068001.08	50130919.94	47661233.40	48538151.53	52661930.18	57048192.31
WS WMult	244164548.83	106927739.38	64424631.37	55088020.64	49565809.86	47789569.96	47079594.83	52927191.06	53077060.64

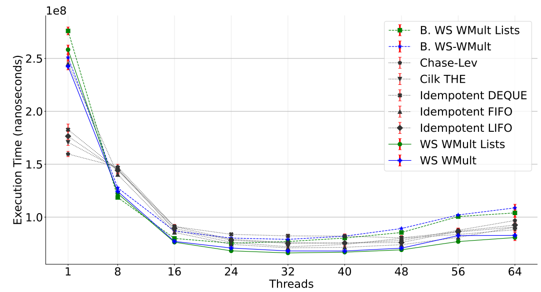
Table A.12: Mean times for the graph application benchmark. These are the results for the 2D Torus 60% Directed graph. Each algorithm begins its execution with an initial size of 256 items.

	1	8	16	24	32	40	48	56	64
B. WS WMult Lists	276019680.35	118608819.28	79879863.83	75368321.91	76921962.09	80232098.52	85462153.81	100530174.03	104015378.66
B. WS WMult	250827613.38	127704204.84	86842336.67	80203663.41	78982977.31	81987558.65	89238292.37	102066852.80	108750912.15
Chase-Lev	159671850.22	147428688.35	91156625.45	79026716.82	75869090.23	75450430.54	77846602.34	87578990.75	96827608.33
Cilk THE	170751689.40	145787527.82	87660021.64	76220044.95	71997062.28	74177705.44	79934972.24	83618370.09	88110479.39
Idempotent DEQUE	182658961.84	144968223.58	91149682.18	83740629.10	82285138.84	82047462.08	80435457.82	85897624.22	90689999.84
Idempotent FIFO	244991185.08	140245626.99	85598340.14	74238515.55	70912264.52	71469675.60	73935154.93	80825298.37	90538667.68
Idempotent LIFO	176481668.10	144272589.64	89256520.53	78199274.43	75257101.72	75557372.58	75999435.33	86619269.45	92537025.58
WS WMult Lists	258286193.60	122279814.84	76291996.09	68133590.83	66153163.63	66865654.60	69070482.60	76900557.87	80688747.87
WS WMult	242540777.49	124338868.84	77031753.27	70801264.40	68030449.61	67980027.38	70670605.67	82555005.58	82774264.49

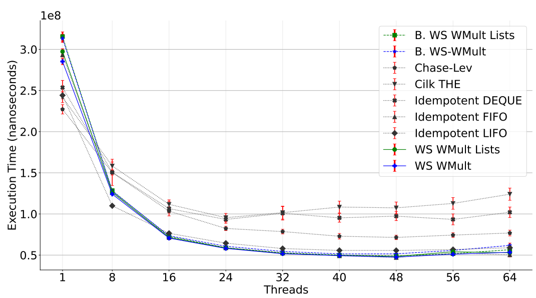
Table A.13: Mean times for the graph application benchmark. These are the results for the 2D Torus 60% Directed graph. Each algorithm begins its execution with an initial size of 1000000 items.



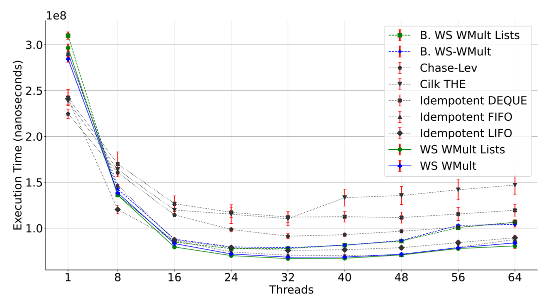
(a) Mean times for the graph application benchmark. These are the results of the 2D Torus 60% Directed graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of 256 entries.



(b) Mean times for the graph application benchmark. These are the results of the 2D Torus 60% Directed graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of 1,000,000 entries.



(c) Mean times for the graph application benchmark. These are the results of the 2D Torus 60% Undirected graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of 256 entries.



(d) Mean times for the graph application benchmark. These are the results of the 2D Torus 60% Undirected graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of 1,000,000 entries.

Figure A.2: 2D Torus 60% Directed and Undirected Graph with 256 and 1,000,000 initial sizes respectively.

	1	8	16	24	32	40	48	56	64
B. WS WMult Lists	315547343.65	128360940.38	72648802.92	58927358.92	52792408.11	50039481.98	48804691.21	51732128.93	56555628.45
B. WS WMult	313326495.26	127723398.79	73483161.75	60717597.04	54443097.02	51433412.83	51552878.37	55703893.17	61765110.29
Chase-Lev	227181330.71	150290622.08	103416065.16	82380750.85	78577017.18	72905889.61	71494636.33	74295370.35	76825437.30
Cilk THE	241733376.13	158313912.20	111680955.02	95495931.87	101569649.50	108395508.88	107417568.44	112791812.70	123995328.19
Idempotent DEQUE	253631054.34	150531220.36	106522134.60	93301475.32	100813833.08	95285547.84	97328945.64	93391967.26	102039301.48
Idempotent FIFO	293454305.80	126736415.05	71307930.79	58829639.90	52462841.41	49072818.39	47060387.78	52425106.09	50348606.01
Idempotent LIFO	244193909.45	109910536.53	76506588.60	64495660.08	57956406.18	55706367.83	55631734.30	56756671.07	58952951.58
WS WMult Lists	297318338.24	126209394.42	71354059.76	58646037.82	51841644.46	49599705.40	48392678.56	54069070.60	52808540.08
WS WMult	285278176.72	124107241.78	70627418.10	58085150.05	51669687.98	49369980.00	47603702.40	51037809.72	53382542.26

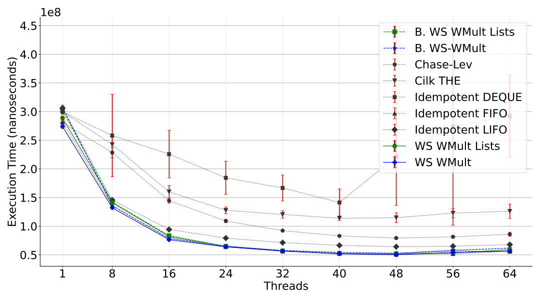
Table A.14: Mean times for the graph application benchmark. These are the results for the 2D Torus 60% Undirected graph. Each algorithm begins its execution with an initial size of 256 items.

	1	8	16	24	32	40	48	56	64
B. WS WMult Lists	309856315.98	136303438.88	84801109.48	77676595.01	77760793.85	81481981.85	85896394.19	100627631.63	106309709.93
B. WS WMult	290354542.52	142736854.03	87950406.31	79587545.45	78588242.76	81475207.56	86517784.67	103123341.20	103949351.04
Chase-Lev	224603931.97	160504055.83	114446930.51	98575701.24	91205053.43	92858685.79	96642196.09	100917291.75	106538505.87
Cilk THE	238559704.41	163907735.92	119766251.42	115236994.47	110200387.50	133250227.29	135537288.52	141820918.53	146968210.28
Idempotent DEQUE	242414652.28	169993061.78	126641918.18	117239442.28	112027599.84	112470998.08	111536200.72	115373385.14	119482468.00
Idempotent FIFO	290306090.80	146658908.67	86560565.57	74055672.27	70429908.94	69865238.79	71211933.45	79175134.19	87616728.40
Idempotent LIFO	241003755.68	120340684.91	86919528.18	78779290.43	75678791.80	76471373.39	78683534.58	84196068.70	89505827.99
WS WMult Lists	296513168.96	135982366.57	79425785.45	70076106.00	66821235.21	67149530.26	70754231.29	77729508.08	80592631.25
WS WMult	284133861.51	138538767.64	83190684.63	71858766.90	68338658.79	68446034.77	71509952.60	78648218.58	83889876.34

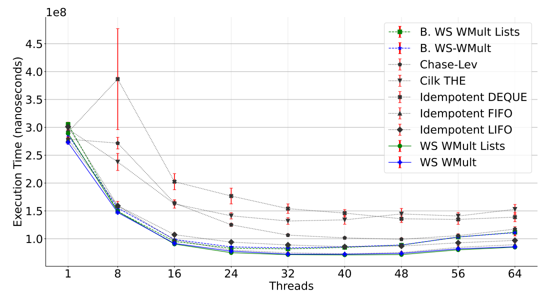
Table A.15: Mean times for the graph application benchmark. These are the results for the 2D Torus 60% Undirected graph. Each algorithm begins its execution with an initial size of 1000000 items.

	1	8	16	24	32	40	48	56	64
B. WS WMult Lists	301624917.75	141866549.86	83541479.95	65138808.25	56914186.43	52423454.12	51989751.02	56527527.81	56014742.33
B. WS WMult	307030538.52	135703652.72	78782482.95	65127326.79	57715887.45	54287146.47	52865866.01	57975637.71	61597515.65
Chase-Lev	280392485.15	228490657.84	144625861.02	109098309.70	92276276.37	83024023.87	79330745.40	81379045.57	85906640.40
Cilk THE	299818699.91	242544084.79	160037649.11	127730079.75	120478873.41	113797574.00	114802833.54	122864155.78	126030085.89
Idempotent DEQUE	299736006.28	258097839.40	225742191.98	184364046.70	166686299.08	141299789.18	221591062.68	273390848.52	292214859.04
Idempotent FIFO	281603607.82	136152638.28	78509517.98	64302364.58	56427898.85	52667607.33	50843691.96	51910053.16	56209351.97
Idempotent LIFO	306122801.62	145650799.53	94241295.91	79193284.39	71350221.69	66482956.94	64179470.43	65091183.51	67905411.43
WS WMult Lists	288758130.76	141026789.51	82019900.84	64081744.46	56705351.91	52060495.93	51302462.90	53337902.56	59025199.96
WS WMult	273827102.97	132030977.87	76413364.21	63917969.81	56562794.21	51825154.68	50271924.64	54089613.78	56571351.07

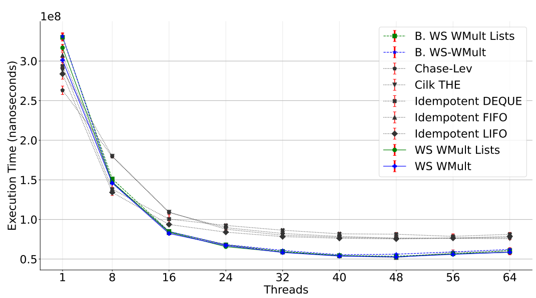
Table A.16: Mean times for the graph application benchmark. These are the results for the 3D Torus Directed graph. Each algorithm begins its execution with an initial size of 256 items.



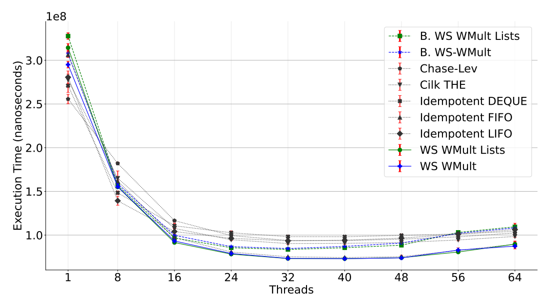
(a) Mean times for the graph application benchmark. These are the results of the 3D Torus Directed graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of 256 entries.



(b) Mean times for the graph application benchmark. These are the results of the 3D Torus Directed graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of 1,000,00 entries.



(c) Mean times for the graph application benchmark. These are the results of the 3D Torus Undirected graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of 256 entries.



(d) Mean times for the graph application benchmark. These are the results of the 3D Torus Undirected graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of 1,000,000 entries.

Figure A.3: 3D Torus Directed and Undirected Graph with 256 and 1,000,000 initial sizes respectively.

	1	8	16	24	32	40	48	56	64
B. WS WMult Lists	305386171.91	149046365.13	95194976.05	83089189.21	81849733.73	84456797.58	88612485.27	103127683.01	111643788.60
B. WS WMult	286170202.17	155416147.14	98692484.33	85463352.85	83512427.37	85456104.05	88874130.18	103205452.01	111004273.95
Chase-Lev	278736420.94	271617495.59	164154925.33	125110061.65	106566596.33	101604051.41	99166808.03	105596223.23	117346843.22
Cilk THE	296880199.62	237861235.97	162585819.44	141211156.01	131884833.85	133994568.66	144624731.92	140846305.32	152922059.48
Idempotent DEQUE	291164788.92	386607331.02	202531346.52	176731787.64	154063302.46	146056409.62	135880453.30	134783560.36	138854522.50
Idempotent FIFO	278180866.51	159103860.27	98048277.27	79703731.70	74716915.26	73100122.13	75161749.36	84572371.02	89422809.81
Idempotent LIFO	301046988.65	159255115.41	107324698.34	94007081.70	88918399.30	85772169.76	86925836.93	92797785.35	97037925.58
WS WMult Lists	289429455.36	149254306.51	90492036.98	75025116.54	71479061.67	71015053.44	71581342.57	80150110.38	84911758.39
WS WMult	272901359.61	147192902.24	91605903.29	77732968.05	72367652.95	72288630.73	73772772.47	81914543.31	85625911.00

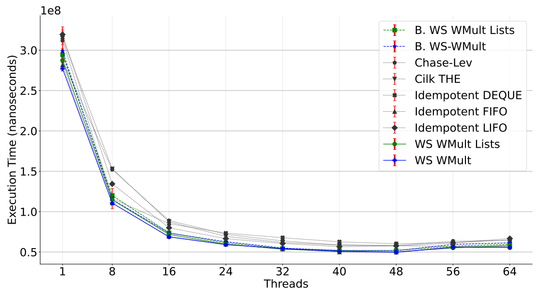
Table A.17: Mean times for the graph application benchmark. These are the results for the 3D Torus Directed graph. Each algorithm begins its execution with an initial size of 1000000 items.

	1	8	16	24	32	40	48	56	64
B. WS WMult Lists	329963181.86	151237154.92	84869838.17	67958542.51	59713831.95	54468184.21	53567371.26	56654627.98	60827191.22
B. WS WMult	331185252.48	146485279.46	84701527.73	68272626.35	60861442.11	55445870.97	56134319.33	59038658.18	62044348.75
Chase-Lev	263083757.40	179485868.93	108749102.80	89621581.64	82340911.59	78506473.93	76517833.65	76532206.85	76983181.99
Cilk THE	288413877.71	179880361.54	109264333.28	87696126.26	80130481.87	77436833.45	75972286.49	76154694.41	75715301.52
Idempotent DEQUE	293601547.68	138258919.26	100495723.16	92474474.20	86429098.06	81863930.54	81369974.02	78652635.16	81262219.14
Idempotent FIFO	307245700.44	147417337.93	83189889.36	67974190.23	58801237.69	53994442.80	51875146.32	58401317.40	59453077.65
Idempotent LIFO	283722458.29	134208645.80	93649945.13	83971242.41	78331924.14	76256331.63	75417609.21	76412158.70	78572650.04
WS WMult Lists	316728772.49	147223826.60	83985379.30	65631752.33	58718689.24	54261373.97	52678773.73	56547641.85	58570046.69
WS WMult	301154107.56	145915148.41	82284655.05	66963616.84	58307819.83	53742415.50	52712760.46	55935847.51	58660468.71

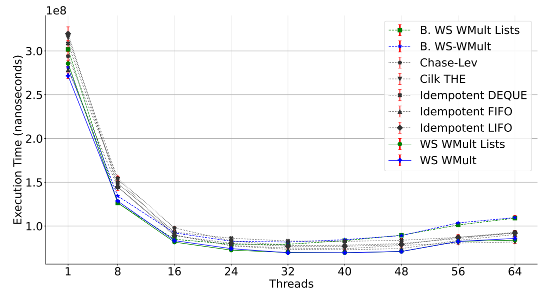
Table A.18: Mean times for the graph application benchmark. These are the results for the 3D Torus Undirected graph. Each algorithm begins its execution with an initial size of 256 items.

	1	8	16	24	32	40	48	56	64
B. WS WMult Lists	327707616.44	156712246.64	96753448.36	85516751.68	83516731.27	85541958.28	88430855.09	103074767.38	109510162.32
B. WS WMult	307644871.81	158759098.92	99763923.60	86807733.13	84602962.87	87048309.48	90879164.43	102032203.69	108190237.33
Chase-Lev	255662963.74	182008226.94	116576204.10	99491054.93	93867107.54	93663588.23	95350740.85	98024792.96	102423692.66
Cilk THE	277586748.65	164994070.46	107628297.54	94664444.99	90367736.38	90586973.10	91502227.94	94433445.56	98434700.61
Idempotent DEQUE	270934340.74	148410630.10	110685130.22	102702609.18	98250872.80	98132701.26	99675589.20	101358720.78	104287965.50
Idempotent FIFO	306233859.67	161649052.10	97163877.14	80651172.49	75172044.83	74370372.54	75250964.27	82449397.40	89552225.15
Idempotent LIFO	280588358.87	139279241.39	104076830.09	96289342.68	93469834.15	94179853.38	96572950.01	101111485.48	106344477.16
WS WMult Lists	314355211.34	155150671.36	91510527.90	78208152.20	73170127.05	73027834.68	74104343.61	80636998.43	89670216.57
WS WMult	295011609.38	155384336.91	93245752.33	78940618.11	73342410.83	73015917.47	73924488.32	82844202.73	87369619.10

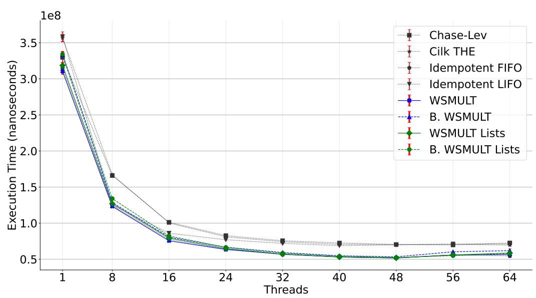
Table A.19: Mean times for the graph application benchmark. These are the results for the 3D Torus Undirected graph. Each algorithm begins its execution with an initial size of 1000000 items.



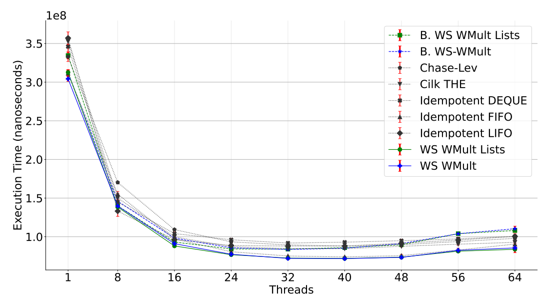
(a) Mean times for the graph application benchmark. These are the results of the 3D Torus 40% Directed graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of 256 entries.



(b) Mean times for the graph application benchmark. These are the results of the 3D Torus 40% Directed graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of the benchmark. Each algorithm begins execution with an initial size of 1,000,000 entries.



(c) Mean times for the graph application benchmark. These are the results of the 3D Torus 40% Undirected graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of the benchmark. Each algorithm begins execution with an initial size of 256 entries.



(d) Mean times for the graph application benchmark. These are the results of the 3D Torus 40% Undirected graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of the benchmark. Each algorithm begins execution with an initial size of 1,000,000 entries.

Figure A.4: 3D Torus 40% Directed and Undirected Graph with 256 and 1,000,000 initial sizes respectively.

	1	8	16	24	32	40	48	56	64
B. WS WMult Lists	294432517.37	120021425.55	73823501.16	61390038.43	54354181.95	51833900.64	52023575.33	57131362.05	59648865.02
B. WS WMult	298663678.64	115054783.47	73453047.11	62747462.27	54920691.10	52001289.43	51681136.53	59499980.10	61377477.24
Chase-Lev	287240600.88	152115465.89	89012227.38	72023688.99	63761750.39	58658047.86	58004155.86	63342223.89	64548293.28
Cilk THE	314871019.02	153049093.83	87438645.82	69353428.09	61776800.67	58530855.79	57804129.29	56974132.59	56592253.73
Idempotent DEQUE	312830298.62	116142320.22	85066483.24	73484481.82	67703602.98	62678789.22	60261928.04	61803224.08	64834634.28
Idempotent FIFO	281784801.76	110651097.78	69253386.17	59583683.11	53682754.00	49991330.61	49891748.90	56450615.25	56486785.18
Idempotent LIFO	319435336.99	134334876.65	80042328.23	66562292.68	60604520.16	56610182.19	57532942.70	61246247.57	66680269.13
WS WMult Lists	287387470.69	114685026.00	71794869.08	59806522.26	53452614.01	51173585.38	50070658.75	55112170.91	58297658.11
WS WMult	277155350.79	110607845.40	68713507.66	59139515.09	54219708.89	50765966.96	49719089.51	55809517.26	55806848.41

Table A.20: Mean times for the graph application benchmark. These are the results for the 3D Torus 40% Directed graph. Each algorithm begins its execution with an initial size of 256 items.

	1	8	16	24	32	40	48	56	64
B. WS WMult Lists	301763957.74	126292839.15	84887857.49	79272185.13	79154954.71	83335257.72	89510324.32	101238037.04	109048106.48
B. WS WMult	281186009.17	134302281.87	92202552.27	82636217.31	81748787.41	84392048.40	89105290.52	103621912.52	109658658.31
Chase-Lev	293916625.21	154585238.00	97856281.89	83025587.50	77807388.15	78126159.57	79948552.74	85725727.08	92310399.45
Cilk THE	315191868.24	153424659.25	89517444.61	77694402.78	74875845.38	74071535.76	77838585.82	80591541.79	81821539.23
Idempotent DEQUE	308537573.96	149211106.90	93096120.06	86063667.52	82913118.38	82531620.54	83710627.36	87052170.00	92779273.54
Idempotent FIFO	278383663.89	144757144.09	89903067.87	77336892.60	73423810.85	72991479.36	74251359.12	83638031.97	90377859.79
Idempotent LIFO	319638620.35	144624689.59	88959163.63	80275708.29	76970703.88	76881719.75	78838337.38	87078149.69	91934674.19
WS WMult Lists	285576627.83	126550275.57	81652075.25	72512300.43	69779523.11	69421029.67	70961813.29	82500795.31	83848751.96
WS WMult	271697926.58	128486529.15	83368234.66	74191411.95	69519514.10	69521643.56	70943495.75	82417635.97	85894721.48

Table A.21: Mean times for the graph application benchmark. These are the results for the 3D Torus 40% Directed graph. Each algorithm begins its execution with an initial size of 1000000 items.

	1	8	16	24	32	40	48	56	64
Chase-Lev	329816027.99	166018668.00	101194304.00	82693813.55	75637331.26	72548609.15	70421774.27	69807147.69	72606679.43
Cilk THE	356916317.95	167055350.86	100233817.43	81074173.36	74222943.72	70749147.75	70338848.18	70562136.49	69687049.08
Idempotent FIFO	314791538.27	124518727.11	77680336.73	63919327.65	56394746.44	53229205.73	51781643.96	55674948.38	54641531.71
Idempotent LIFO	358165273.94	125105689.07	86102533.77	77056873.00	71960543.09	68881040.65	69877008.87	71220489.02	71125362.10
WS WMULT	310482623.44	123277866.27	75627540.46	63491941.48	57072664.51	53150879.35	51860315.85	55971221.94	56381071.13
B. WS WMULT	333148110.98	128544299.77	80629725.80	66742869.48	59245808.63	54831212.23	53350917.70	60339368.99	61904545.78
WS WMULT Lists	318598312.42	126984605.65	79327818.59	64873208.49	57030534.60	52973456.00	51696004.73	55958368.16	58612461.28
B. WS WMULT Lists	334232009.92	134058201.59	82287300.73	66418704.09	58477222.64	54223431.95	52443567.62	55004602.31	58013627.62

Table A.22: Mean times for the graph application benchmark. These are the results for the 3D Torus 40% Undirected graph. Each algorithm begins its execution with an initial size of 256 items.

	1	8	16	24	32	40	48	56	64
B. WS WMult Lists	334464593.22	139869441.62	93118738.82	84202596.98	83980353.63	85865325.35	89570045.81	103955321.31	108082353.61
B. WS WMult	311579693.68	145881522.20	98130498.61	85629026.30	83509625.99	85149457.30	91105762.40	103885268.05	110346559.14
Chase-Lev	332908360.24	170224669.95	109220232.75	93343788.98	89389033.42	87979103.86	89744702.88	93698982.84	97875180.09
Cilk THE	352562123.61	150060951.66	99760520.74	87126086.44	84656500.65	84533702.71	87602278.89	90333960.91	92750637.97
Idempotent DEQUE	345947619.04	144885937.88	104036420.18	96097964.76	91840290.82	92917394.80	94867142.72	97202251.38	100523216.90
Idempotent FIFO	311553075.87	155166420.27	97058448.94	80591951.38	75066314.78	73895334.04	75520212.03	82801873.16	90442958.12
Idempotent LIFO	357016153.83	133154299.38	96322449.36	88612359.68	87952279.63	88014320.73	91586087.90	95690280.32	100364216.97
WS WMult Lists	312648264.91	138383488.50	88072075.50	76709536.59	72390868.09	71909451.21	73524758.08	81289669.58	83682658.43
WS WMult	304293026.00	139635085.90	91022803.16	77301315.95	71943494.11	71677604.62	73267315.81	82367515.38	85539771.68

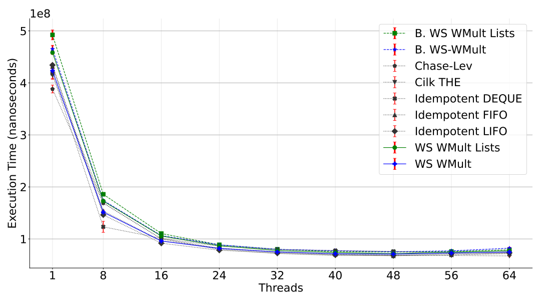
Table A.23: Mean times for the graph application benchmark. These are the results for the 3D Torus 40% Undirected graph. Each algorithm begins its execution with an initial size of 1000000 items.

	1	8	16	24	32	40	48	56	64
B. WS WMult Lists	492263536.51	185833977.72	110426588.40	88710895.63	80108599.36	75622130.99	74978359.76	75920034.33	79101590.74
B. WS WMult	464920786.36	174035456.11	106386422.68	88570436.72	79898215.19	77502739.17	75394325.17	77090345.63	82158396.11
Chase-Lev	388237323.37	168767464.73	100412992.35	81817160.73	74211385.42	70332957.26	69280621.13	69033972.39	73297305.92
Cilk THE	414885668.90	169415076.62	100124493.84	79805125.83	72108213.75	68273252.47	68369272.70	68145787.43	67390061.84
Idempotent DEQUE	417685137.16	123250084.16	101264250.42	86142952.92	79690817.06	77644288.80	76110697.96	72222956.16	75078709.50
Idempotent FIFO	431714806.10	153648343.56	96384410.52	82515552.81	75213108.75	70371870.20	67562026.30	69506128.03	71745378.98
Idempotent LIFO	434271558.65	146245687.74	91764425.56	78334009.50	72611394.64	69217120.27	67916615.78	71863500.28	73252446.32
WS WMult Lists	458274872.50	172752713.06	105669957.09	86870765.39	77646018.05	73957764.17	71841786.10	74602240.90	77254487.10
WS WMult	423361253.32	151414860.14	95985749.42	81696821.86	74922903.38	71679742.34	70944246.68	73027424.41	74832776.27

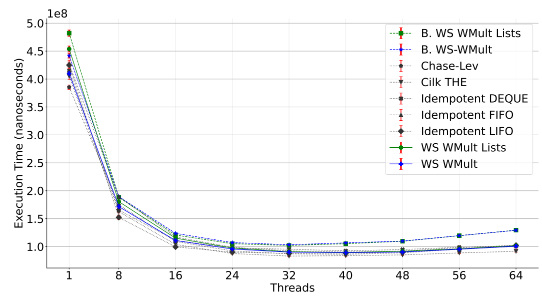
Table A.24: Mean times for the graph application benchmark. These are the results for the Random Directed graph. Each algorithm begins its execution with an initial size of 256 items.

	1	8	16	24	32	40	48	56	64
B. WS WMult Lists	482160068.48	188336095.42	120700826.52	105196857.53	102113641.09	105162929.44	109584170.31	119487423.69	129451320.56
B. WS WMult	441884607.61	189459191.22	123728462.81	107166435.55	103353938.65	106676865.57	109881300.60	119661581.70	129178737.27
Chase-Lev	385340577.40	174664858.65	109232027.31	92202519.86	87016638.92	86185560.86	88905638.86	95454735.65	100458140.05
Cilk THE	405144969.95	162520672.80	103929415.16	87591486.16	83002578.54	83794041.85	85119207.59	88656720.05	91558486.63
Idempotent DEQUE	414733560.90	165342618.80	111167218.22	97768702.72	95273218.10	92395813.22	94765056.90	98724645.98	101022772.80
Idempotent FIFO	426842811.31	189062631.20	117185450.64	98759795.45	92265610.82	90328848.01	92077270.18	98034222.72	102493646.11
Idempotent LIFO	425017719.73	152344725.60	99461217.11	90006238.62	87681136.21	88206820.36	90283254.07	95488801.00	102316035.46
WS WMult Lists	453825844.09	180552914.55	115121584.18	97398795.85	90786582.21	89502129.75	91607446.68	96029389.94	101519435.08
WS WMult	409944339.41	171618034.92	110920773.91	95664656.10	90049503.96	89225431.26	90197459.73	95304067.23	100696398.66

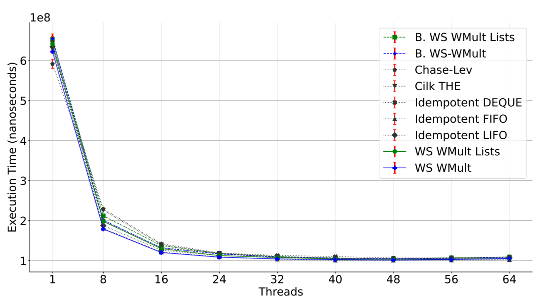
Table A.25: Mean times for the graph application benchmark. These are the results for the Random Directed graph. Each algorithm begins its execution with an initial size of 1000000 items.



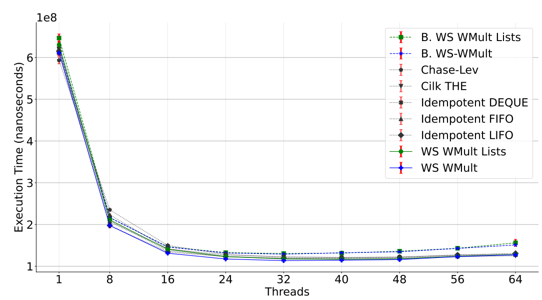
(a) Mean times for the graph application benchmark. These are the results of the Random Directed graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of the benchmark. Each algorithm begins execution with an initial size of 256 entries.



(b) Mean times for the graph application benchmark. These are the results of the Random Directed graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of the benchmark. Each algorithm begins execution with an initial size of 256 entries.



(c) Mean times for the graph application benchmark. These are the results of the Random Directed graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of the benchmark. Each algorithm begins execution with an initial size of 256 entries.



(d) Mean times for the graph application benchmark. These are the results of the Random Undirected graph. For each work-stealing algorithm's data structure, it begins its execution with an initial size of the benchmark. Each algorithm begins execution with an initial size of 1,000,000 entries.

Figure A.5: Random Directed and Undirected Graph with 256 and 1,000,000 initial sizes respectively.

	1	8	16	24	32	40	48	56	64
B. WS WMult Lists	652599333.99	212017984.57	137665783.14	117554752.46	110222245.73	106698088.43	104742436.78	107187587.60	109281820.92
B. WS WMult	654074473.51	201163088.41	131849720.51	116574987.43	108857132.28	106226178.52	104760944.68	105417615.98	108921032.66
Chase-Lev	591680426.65	229336231.17	142109070.85	119283686.27	110387662.00	105982878.56	105258463.21	104349248.85	105506659.24
Cilk THE	628957009.63	226012672.44	139882962.01	118098897.42	109379482.56	105188663.35	103293630.32	104095251.74	104713698.81
Idempotent DEQUE	646148838.78	199370245.96	132641626.64	118901933.56	113004891.32	110191577.76	106606170.40	107039317.74	109154609.76
Idempotent FIFO	638027662.94	180713367.95	121445354.47	109490247.72	103656997.95	101075641.68	101095735.40	101566154.53	102251728.23
Idempotent LIFO	634327156.00	187644988.26	126163770.06	112724494.01	107319262.19	104627987.26	104663456.64	106040748.03	107394458.73
WS WMult Lists	641436399.23	198037184.55	129737047.19	112688727.80	107659841.75	103953948.28	103050511.01	103532647.86	105339241.71
WS WMult	622307814.91	178968825.24	120324249.12	108599936.53	104330814.77	102432961.97	101552823.38	102759165.67	105269567.75

Table A.26: Mean times for the graph application benchmark. These are the results for the Random Undirected graph. Each algorithm begins its execution with an initial size of 256 items.

	1	8	16	24	32	40	48	56	64
B. WS WMult Lists	646541284.19	216263265.20	146759280.43	132541176.53	130512193.85	131523019.65	135918877.69	142989655.30	156049348.80
B. WS WMult	622923092.34	216837946.90	146021906.28	131696898.21	128922852.74	132160589.40	134643408.57	142655863.73	150954630.75
Chase-Lev	593236904.85	235082293.49	149973065.80	128300559.06	121496061.31	120640561.49	121437035.74	126943494.15	130407001.13
Cilk THE	620863347.51	209168448.86	137197146.35	122570308.33	117931034.32	116320400.08	118205988.97	122554842.13	125001132.55
Idempotent DEQUE	620533697.72	206167259.40	141569448.10	126905708.14	122788765.72	121143353.94	122235287.98	126617596.40	127807686.04
Idempotent FIFO	611933226.01	223162610.33	141521019.59	123592991.13	118780262.20	118437149.38	119279944.51	124372070.49	127962219.33
Idempotent LIFO	615205002.42	197693607.26	134245575.36	123088544.61	119117560.12	119495674.49	121411201.90	126847418.11	130739705.03
WS WMult Lists	630514406.43	211180985.60	140475330.14	123174961.72	117468510.92	116733496.47	117762755.18	123778079.83	128006874.12
WS WMult	610832074.69	197574834.99	131195696.62	117332821.11	113455313.17	114436181.30	115915780.99	123082517.02	127229090.01

Table A.27: Mean times for the graph application benchmark. These are the results for the Random Undirected graph. Each algorithm begins its execution with an initial size of 1000000 items.

A.2.2 Puts and takes performed in the Paralled Spanning Tree experiment

This section reports the number of puts and takes performed during the execution of the parallel spanning tree. This evaluation was performed for each graph and all work-stealing algorithms. Additionally, the difference between the total number of puts and the total number of takes is calculated. Finally, the total surplus work is calculated as the difference between the total put and the total available work (number of vertices). For purposes of visualizing the amount of surplus work, this is displayed as a graph in terms of the percentage of total available work.

Directed Torus 2D. Initial size of 256 items.

Algorithm	Chase-Lev					Cilk THE					Idempotent LIFO				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1459383.80	1041490.20	28.63	31.48	3.98	1033699.60	1000020.40	3.26	3.26	0.00	1000583.40	1000241.00	0.03	0.06	0.02
16	1448842.40	1033211.00	28.69	30.98	3.21	1044556.60	1000118.60	4.25	4.27	0.01	1001805.40	1000745.80	0.11	0.18	0.07
24	1454352.20	1028947.20	29.25	31.24	2.81	1041856.20	1000150.40	4.00	4.02	0.02	1003160.00	1000912.80	0.22	0.32	0.09
28	1433539.00	1022538.80	28.67	30.24	2.20	1041198.80	1000140.00	3.94	3.96	0.01	1002856.20	1000665.40	0.22	0.28	0.07
32	1461140.80	1023658.80	29.94	31.56	2.31	1037250.60	1000150.00	3.58	3.59	0.01	1003144.40	1000718.80	0.24	0.31	0.07
40	1417516.60	1018013.40	28.18	29.45	1.77	1038668.60	1000147.00	3.71	3.72	0.01	1004017.00	1000873.80	0.31	0.40	0.09
48	1407082.60	1016505.40	27.76	28.93	1.62	1037384.00	1000138.00	3.59	3.60	0.01	1005458.80	1001351.00	0.41	0.54	0.13
56	1412557.20	1016545.60	28.04	29.21	1.63	1039636.00	1000179.00	3.80	3.81	0.02	1011643.20	1003597.60	0.80	1.15	0.36
64	1436173.00	1017850.20	29.13	30.37	1.75	1038216.20	1000176.20	3.66	3.68	0.02	1006204.40	1001809.40	0.44	0.62	0.18

Table A.28: The number of puts and takes performed during the spanning tree experiment on a Torus 2D directed graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, and Idempotent LIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1902357.20	1495215.60	21.40	47.43	33.12	1000080.80	1000036.40	0.00	0.01	0.00	1000153.00	1000088.80	0.01	0.02	0.01
16	1934233.20	1590823.80	17.75	48.30	37.14	1000267.60	1000117.20	0.02	0.03	0.01	1000292.00	1000190.20	0.01	0.03	0.02
24	1941439.00	1607217.20	17.22	48.49	37.78	1000312.40	1000107.60	0.02	0.03	0.01	1000390.20	1000246.40	0.01	0.04	0.02
28	2224206.20	1940711.00	12.75	55.04	48.47	1000467.00	1000141.60	0.03	0.05	0.01	1000552.80	1000332.20	0.02	0.06	0.03
32	2288846.00	1967366.40	14.05	56.31	49.17	1000690.40	1000225.40	0.05	0.07	0.02	1000609.20	1000351.60	0.03	0.06	0.04
40	1827589.80	1496496.80	18.12	45.28	33.18	1000863.60	1000198.60	0.07	0.09	0.02	1000914.60	1000481.80	0.04	0.09	0.05
48	2137192.80	1780388.20	16.70	53.21	43.83	1001107.80	1000279.20	0.08	0.11	0.03	1001438.80	1000844.20	0.06	0.14	0.08
56	2320612.60	1972625.40	15.00	56.91	49.31	1001801.20	1000437.60	0.14	0.18	0.04	1001633.40	1000978.40	0.07	0.16	0.10
64	2256633.40	1950514.60	13.57	55.69	48.73	1001345.40	1000333.40	0.10	0.13	0.03	1002296.20	1001442.80	0.09	0.23	0.14

Table A.29: The number of puts and takes performed during the spanning tree experiment on a Torus 2D directed graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000328.80	1000182.20	0.01	0.03	0.02	1000185.80	1000133.80	0.01	0.02	0.01	1000192.80	1000146.20	0.00	0.02	0.01
16	1000481.60	1000293.20	0.02	0.05	0.03	1000379.60	1000247.20	0.01	0.04	0.02	1000332.80	1000236.60	0.01	0.03	0.02
24	1000553.60	1000303.60	0.02	0.06	0.03	1000465.00	1000297.00	0.02	0.05	0.03	1000512.60	1000345.40	0.02	0.05	0.03
28	1000707.00	1000384.00	0.03	0.07	0.04	1000572.80	1000357.80	0.02	0.06	0.04	1000539.80	1000365.60	0.02	0.05	0.04
32	1000720.00	1000389.60	0.03	0.07	0.04	1000599.80	1000375.20	0.02	0.06	0.04	1000835.40	1000524.80	0.03	0.08	0.05
40	1001108.40	1000652.20	0.05	0.11	0.07	1001085.40	1000721.80	0.04	0.11	0.07	1001161.20	1000725.60	0.04	0.12	0.07
48	1001236.00	1000693.40	0.05	0.12	0.07	1001267.80	1000791.20	0.05	0.13	0.08	1001099.80	1000695.80	0.04	0.11	0.07
56	1002056.60	1001285.20	0.08	0.21	0.13	1001683.20	1001116.60	0.06	0.17	0.11	1001527.80	1000939.60	0.06	0.15	0.09
64	1001891.20	1001111.80	0.08	0.19	0.11	1001977.00	1001245.60	0.07	0.20	0.12	1001780.80	1001181.40	0.06	0.18	0.12

Table A.30: The number of puts and takes performed during the spanning tree experiment on a Torus 2D directed graph with an initial size of 256 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Directed Torus 2D. Initial size of 1,000,000 items.

Algorithm	Chase-Lev					Cilk THE					Idempotent LIFO				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1412180.00	1000301.80	29.17	29.19	0.03	1037307.20	999999.00	3.60	3.60	-0.00	1000616.60	1000271.00	0.03	0.06	0.03
16	1408743.20	1000469.20	28.98	29.01	0.05	1038837.80	1000006.80	3.74	3.74	0.00	1001289.60	1000452.60	0.08	0.13	0.05
24	1411153.20	1000544.80	29.10	29.14	0.05	1041228.60	1000011.20	3.96	3.96	0.00	1001875.00	1000549.40	0.13	0.19	0.05
28	1422954.60	1000678.80	29.68	29.72	0.07	1040132.20	1000009.60	3.86	3.86	0.00	1002199.40	1000833.60	0.14	0.22	0.08
32	1419592.60	1000673.00	29.51	29.56	0.07	1036052.20	1000013.00	3.48	3.48	0.00	1002494.00	1000762.00	0.17	0.25	0.08
40	1430669.20	1000874.40	30.04	30.10	0.09	1037476.20	1000008.40	3.61	3.61	0.00	1003855.80	1000973.60	0.29	0.38	0.10
48	1420825.60	1001096.40	29.54	29.62	0.11	1039141.40	1000043.80	3.76	3.77	0.00	1005325.20	1001255.60	0.40	0.53	0.13
56	1418509.40	1000987.20	29.43	29.50	0.10	1041312.20	1000033.00	3.96	3.97	0.00	1008324.60	1002621.20	0.57	0.83	0.26
64	1412314.60	1001042.00	29.12	29.19	0.10	1035602.80	1000062.20	3.43	3.44	0.01	1008637.20	1002190.80	0.64	0.86	0.22

Table A.31: The number of puts and takes performed during the spanning tree experiment on a Torus 2D directed graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, and Idempotent LIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	2341456.60	1897128.00	18.98	57.29	47.29	1000082.00	1000044.60	0.00	0.01	0.00	1000169.20	1000124.60	0.00	0.02	0.01
16	1789294.20	1339375.40	25.15	44.11	25.34	1000253.80	1000162.80	0.01	0.03	0.02	1000321.80	1000218.60	0.01	0.03	0.02
24	1638609.20	1211498.80	26.07	38.97	17.46	1000384.00	1000157.40	0.02	0.04	0.02	1000442.00	1000233.00	0.02	0.04	0.02
28	1569116.60	1172962.00	25.25	36.27	14.75	1000476.80	1000166.20	0.03	0.05	0.02	1000510.80	1000243.40	0.03	0.05	0.02
32	1564781.20	1148789.40	26.58	36.09	12.95	1000658.40	1000208.60	0.04	0.07	0.02	1000618.20	1000290.00	0.03	0.06	0.03
40	1577941.40	1179196.40	25.27	36.63	15.20	1000783.20	1000220.40	0.06	0.08	0.02	1001053.60	1000444.80	0.06	0.11	0.04
48	1554558.20	1160841.00	25.33	35.67	13.86	1001201.80	1000313.20	0.09	0.12	0.03	1001365.80	1000574.80	0.08	0.14	0.06
56	1504966.80	1126507.20	25.15	33.55	11.23	1001396.40	1000343.00	0.11	0.14	0.03	1001310.20	1000518.00	0.08	0.13	0.05
64	1496983.60	1124354.40	24.89	33.20	11.06	1001243.60	1000327.40	0.09	0.12	0.03	1001638.80	1000641.80	0.10	0.16	0.06

Table A.32: The number of puts and takes performed during the spanning tree experiment on a Torus 2D directed graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000149.80	1000112.40	0.00	0.01	0.01	1000143.00	1000085.40	0.01	0.01	0.01	1000203.20	1000143.60	0.01	0.02	0.01
16	1000341.20	1000244.80	0.01	0.03	0.02	1000347.60	1000207.20	0.01	0.03	0.02	1000356.40	1000288.40	0.01	0.04	0.03
24	1000392.60	1000236.00	0.02	0.04	0.02	1000578.40	1000302.20	0.03	0.06	0.03	1000447.80	1000269.20	0.02	0.04	0.03
28	1000588.00	1000342.20	0.02	0.06	0.03	1000689.60	1000352.00	0.03	0.07	0.04	1000719.40	1000439.60	0.03	0.07	0.04
32	1000625.60	1000350.20	0.03	0.06	0.04	1000822.40	1000378.20	0.04	0.08	0.04	1000652.80	1000336.60	0.03	0.07	0.03
40	1000919.80	1000499.60	0.04	0.09	0.05	1001063.60	1000457.20	0.06	0.11	0.05	1001033.40	1000550.80	0.05	0.10	0.06
48	1001096.00	1000495.40	0.06	0.11	0.05	1001533.60	1000634.40	0.09	0.15	0.06	1001381.00	1000670.00	0.07	0.14	0.07
56	1001013.60	1000438.00	0.06	0.10	0.04	1001744.80	1000803.00	0.09	0.17	0.08	1001672.60	1000910.40	0.08	0.17	0.09
64	1001565.00	1000766.60	0.08	0.16	0.08	1001901.20	1000743.60	0.12	0.19	0.07	1002036.20	1001157.40	0.09	0.20	0.12

Table A.33: The number of puts and takes performed during the spanning tree experiment on a Torus 2D directed graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Undirected Torus 2D. Initial size of 256 items.

Algorithm	Chase-Lev					Cilk THE					Idempotent LIFO				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1005802.20	1000299.00	0.55	0.58	0.03	1005609.00	1000267.00	0.53	0.56	0.03	1000642.20	1000589.00	0.01	0.06	0.06
16	1006274.20	1000635.60	0.56	0.62	0.06	1012449.40	1000720.60	1.16	1.23	0.07	1001783.60	1001413.80	0.04	0.18	0.14
24	1014993.80	1001002.80	1.38	1.48	0.10	1017575.40	1001520.40	1.58	1.73	0.15	1002208.20	1001296.80	0.09	0.22	0.13
28	1026536.20	1001682.40	2.42	2.59	0.17	1016091.60	1000982.00	1.49	1.58	0.10	1003306.80	1001993.00	0.13	0.33	0.20
32	1015987.60	1001424.40	1.43	1.57	0.14	1018616.00	1000944.80	1.73	1.83	0.09	1002711.40	1001281.80	0.14	0.27	0.13
40	1030795.60	1001357.40	2.86	2.99	0.14	1017967.00	1001503.80	1.62	1.76	0.15	1003790.40	1001714.80	0.21	0.38	0.17
48	1041349.40	1002717.40	3.71	3.97	0.27	1021047.60	1001816.20	1.88	2.06	0.18	1008262.00	1003388.40	0.48	0.82	0.34
56	1034852.20	1002225.60	3.15	3.37	0.22	1031250.40	1001911.80	2.84	3.03	0.19	1005978.00	1002870.00	0.31	0.59	0.29
64	1052953.60	1002509.60	4.79	5.03	0.25	1023687.20	1001614.20	2.16	2.31	0.16	1010531.40	1004579.80	0.59	1.04	0.46

Table A.34: The number of puts and takes performed during the spanning tree experiment on a Torus 2D undirected graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, and Idempotent LIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1007652.80	1002225.20	0.54	0.76	0.22	1000038.80	1000023.00	0.00	0.00	0.00	1000034.80	1000020.60	0.00	0.00	0.00
16	1010227.60	1002944.60	0.72	1.01	0.29	1000082.20	1000054.80	0.00	0.01	0.01	1000097.60	1000062.20	0.00	0.01	0.01
24	1022708.40	1006915.20	1.54	2.22	0.69	1000149.60	1000073.80	0.01	0.01	0.01	1000207.00	1000128.00	0.01	0.02	0.01
28	1021754.40	1005416.60	1.60	2.13	0.54	1000209.00	1000099.00	0.01	0.02	0.01	1000265.40	1000154.60	0.01	0.03	0.02
32	1025247.20	1005598.00	1.92	2.46	0.56	1000252.40	1000108.40	0.01	0.03	0.01	1000343.60	1000204.40	0.01	0.03	0.02
40	1035406.20	1008212.00	2.63	3.42	0.81	1000515.20	1000242.20	0.03	0.05	0.02	1000560.20	1000304.20	0.03	0.06	0.03
48	1045111.20	1010315.80	3.33	4.32	1.02	1000569.40	1000246.20	0.03	0.06	0.02	1000794.80	1000443.20	0.04	0.08	0.04
56	1037480.40	1007844.80	2.86	3.61	0.78	1001077.60	1000399.80	0.07	0.11	0.04	1001056.00	1000618.80	0.04	0.11	0.06
64	1053497.60	1012477.80	3.89	5.08	1.23	1000975.00	1000367.00	0.06	0.10	0.04	1001043.80	1000604.00	0.04	0.10	0.06

Table A.35: The number of puts and takes performed during the spanning tree experiment on a Torus 2D undirected graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000052.40	1000032.00	0.00	0.01	0.00	1000052.40	1000036.20	0.00	0.01	0.00	1000116.80	1000103.00	0.00	0.01	0.01
16	1000131.20	1000080.60	0.01	0.01	0.01	1000099.40	1000068.20	0.00	0.01	0.01	1000102.60	1000072.20	0.00	0.01	0.01
24	1000195.60	1000128.60	0.01	0.02	0.01	1000179.80	1000121.80	0.01	0.02	0.01	1000193.80	1000136.60	0.01	0.02	0.01
28	1000310.20	1000186.60	0.01	0.03	0.02	1000956.20	1000859.20	0.01	0.10	0.09	1000262.20	1000179.80	0.01	0.03	0.02
32	1000383.20	1000258.00	0.01	0.04	0.03	1000291.20	1000163.20	0.01	0.03	0.02	1000302.60	1000197.60	0.01	0.03	0.02
40	1000646.60	1000430.80	0.02	0.06	0.04	1000492.60	1000315.20	0.02	0.05	0.03	1000500.60	1000297.20	0.02	0.05	0.03
48	1000905.80	1000577.60	0.03	0.09	0.06	1000823.00	1000502.00	0.03	0.08	0.05	1000801.00	1000522.20	0.03	0.08	0.05
56	1000981.60	1000609.40	0.04	0.10	0.06	1001089.20	1000632.80	0.05	0.11	0.06	1001093.60	1000638.40	0.05	0.11	0.06
64	1001431.00	1000862.40	0.06	0.14	0.09	1001267.20	1000791.00	0.05	0.13	0.08	1001504.80	1001047.80	0.05	0.15	0.10

Table A.36: The number of puts and takes performed during the spanning tree experiment on a Torus 2D undirected graph with an initial size of 256 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Undirected Torus 2D. Initial size of 1,000,000 items.

Algorithm	Chase-Lev					Cilk THE					Idempotent LIFO				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1004333.80	1000319.80	0.40	0.43	0.03	1005854.60	1000312.00	0.55	0.58	0.03	1000644.80	1000575.00	0.01	0.06	0.06
16	1013695.40	1000731.40	1.28	1.35	0.07	1010464.20	1000652.40	0.97	1.04	0.07	1001762.40	1001352.00	0.04	0.18	0.14
24	1020985.00	1001456.20	1.91	2.06	0.15	1009914.40	1000805.40	0.90	0.98	0.08	1002361.60	1001356.60	0.10	0.24	0.14
28	1020140.20	1001042.40	1.87	1.97	0.10	1016056.80	1000913.00	1.49	1.58	0.09	1002467.20	1001390.40	0.11	0.25	0.14
32	1016086.20	1000828.20	1.50	1.58	0.08	1017728.80	1001498.20	1.59	1.74	0.15	1002958.80	1001593.60	0.14	0.30	0.16
40	1024334.60	1001527.00	2.23	2.38	0.15	1018041.20	1001206.80	1.65	1.77	0.12	1004015.20	1001884.80	0.21	0.40	0.19
48	1036484.40	1001674.60	3.36	3.52	0.17	1030817.40	1001003.00	2.89	2.99	0.10	1005163.80	1002497.80	0.27	0.51	0.25
56	1044977.40	1002189.80	4.09	4.30	0.22	1021632.40	1001473.40	1.97	2.12	0.15	1006869.40	1003188.80	0.37	0.68	0.32
64	1044545.80	1001234.00	4.15	4.26	0.12	1026773.00	1001678.60	2.44	2.61	0.17	1007726.60	1004101.40	0.36	0.77	0.41

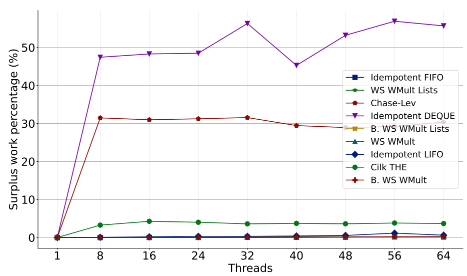
Table A.37: The number of puts and takes performed during the spanning tree experiment on a Torus 2D undirected graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, and Idempotent LIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1005282.80	1001761.80	0.35	0.53	0.18	1000039.40	1000025.20	0.00	0.00	0.00	1000044.60	1000031.20	0.00	0.00	0.00
16	1008602.80	1002598.00	0.60	0.85	0.26	1000094.80	1000064.00	0.00	0.01	0.01	1000101.20	1000070.40	0.00	0.01	0.01
24	1027203.80	1007382.00	1.93	2.65	0.73	1000152.20	1000071.60	0.01	0.02	0.01	1000175.40	1000098.00	0.01	0.02	0.01
28	1016077.40	1004284.60	1.16	1.58	0.43	1000189.20	1000094.20	0.01	0.02	0.01	1000269.40	1000163.80	0.01	0.03	0.02
32	1030261.40	1010193.40	1.95	2.94	1.01	1000287.60	1000118.60	0.02	0.03	0.01	1000270.20	1000147.40	0.01	0.03	0.01
40	1036930.00	1007912.60	2.80	3.56	0.79	1000369.60	1000148.80	0.02	0.04	0.01	1000432.80	1000234.00	0.02	0.04	0.02
48	1047548.20	1009787.80	3.60	4.54	0.97	1000608.20	1000260.80	0.03	0.06	0.03	1000730.20	1000380.20	0.03	0.07	0.04
56	1034493.40	1008468.00	2.52	3.33	0.84	1000716.60	1000274.60	0.04	0.07	0.03	1000850.00	1000486.60	0.04	0.08	0.05
64	1048158.40	1011917.60	3.46	4.59	1.18	1000914.00	1000384.00	0.05	0.09	0.04	1001207.00	1000764.20	0.04	0.12	0.08

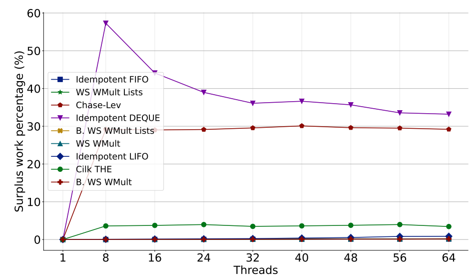
Table A.38: The number of puts and takes performed during the spanning tree experiment on a Torus 2D undirected graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000045.80	1000033.60	0.00	0.00	0.00	1000047.80	1000036.00	0.00	0.00	0.00	1000091.00	1000077.80	0.00	0.01	0.01
16	1000125.20	1000092.80	0.00	0.01	0.01	1000113.20	1000078.60	0.00	0.01	0.01	1000114.40	1000082.40	0.00	0.01	0.01
24	1000198.20	1000128.00	0.01	0.02	0.01	1000177.20	1000108.00	0.01	0.02	0.01	1000180.20	1000125.40	0.01	0.02	0.01
28	1000215.40	1000141.20	0.01	0.02	0.01	1000228.80	1000133.40	0.01	0.02	0.01	1000259.00	1000164.60	0.01	0.03	0.02
32	1000327.00	1000220.60	0.01	0.03	0.02	1000277.40	1000153.20	0.01	0.03	0.02	1000328.60	1000221.40	0.01	0.03	0.02
40	1000481.80	1000291.80	0.02	0.05	0.03	1000507.40	1000281.00	0.02	0.05	0.03	1000488.00	1000337.00	0.02	0.05	0.03
48	1000721.60	1000449.20	0.03	0.07	0.04	1000788.20	1000505.60	0.03	0.08	0.05	1000717.40	1000417.60	0.03	0.07	0.04
56	1000751.80	1000474.00	0.03	0.08	0.05	1001097.00	1000550.20	0.05	0.11	0.05	1000849.00	1000481.60	0.04	0.08	0.05
64	1000844.40	1000448.80	0.04	0.08	0.04	1001115.40	1000564.20	0.06	0.11	0.06	1001020.00	1000581.00	0.04	0.10	0.06

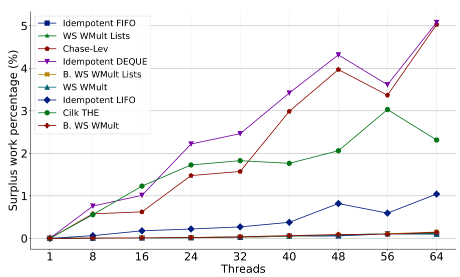
Table A.39: The number of puts and takes performed during the spanning tree experiment on a Torus 2D undirected graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.



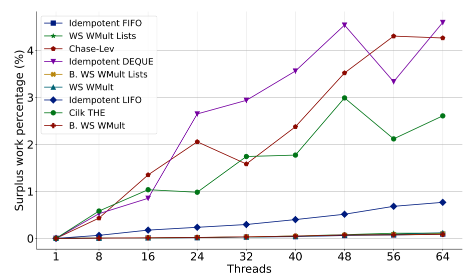
(a) Surplus work: Directed Torus 2D. Initial size of 256 items



(b) Surplus work: Directed Torus 2D. Initial size of 1,000,000 items

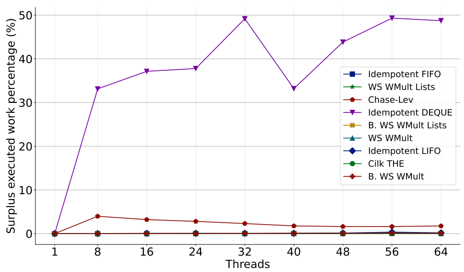


(c) Surplus work: Undirected Torus 2D. Initial size of 256 items

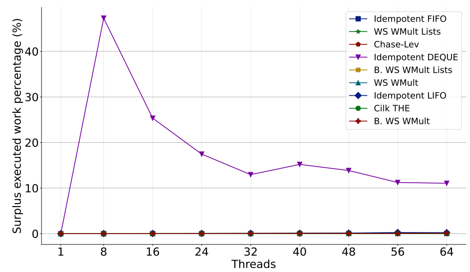


(d) Surplus work: Undirected Torus 2D. Initial size of 1,000,000 items

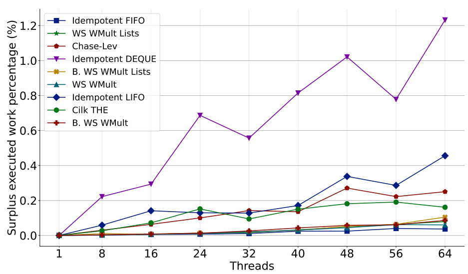
Figure A.6: Surplus work (percentage) of the experiments. Surplus work: the difference between the total number of Puts and the number of puts in sequential executions (i.e., $1,000,000$).



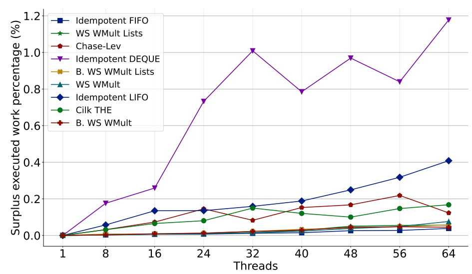
(a) Executed surplus work: Directed Torus 2D. Initial size of 256 items



(b) Executed surplus work: Directed Torus 2D. Initial size of 1,000,000 items



(c) Executed surplus work: Undirected Torus 2D. Initial size of 256 items



(d) Executed surplus work: Undirected Torus 2D. Initial size of 1,000,000 items

Figure A.7: Executed surplus work (percentage) of the experiments. Surplus work: the difference between the total number of Takes and the number of takes in sequential executions (i.e., $1,000,000$).

Directed Torus 2D 60%. Initial size of 256 items.

Algorithm	Chase-Lev					Cilk THE					Idempotent LIFO				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1020806.00	1000143.60	2.02	2.04	0.01	1023550.20	1000084.60	2.29	2.30	0.01	1027143.60	1008311.80	1.83	2.64	0.82
16	1037467.40	1000165.40	3.60	3.61	0.02	1042807.00	1000167.00	4.09	4.10	0.02	1050478.40	1012666.40	3.60	4.81	1.25
24	1064950.60	1000357.80	6.07	6.10	0.04	1050968.80	1000198.80	4.83	4.85	0.02	1060968.00	1010891.40	4.72	5.75	1.08
28	1068789.20	1000359.00	6.40	6.44	0.04	1059995.60	1000244.20	5.64	5.66	0.02	1073664.80	1010931.40	5.84	6.86	1.08
32	1073080.40	1000402.40	6.77	6.81	0.04	1073922.60	1000298.60	6.86	6.88	0.03	1069279.20	1015430.60	5.04	6.48	1.52
40	1097050.60	1000536.00	8.80	8.85	0.05	1060958.00	1000283.80	5.72	5.75	0.03	1092589.80	1012147.40	7.36	8.47	1.20
48	1117179.80	1000760.00	10.42	10.49	0.08	1079921.60	1000413.00	7.36	7.40	0.04	1105993.60	1014937.40	8.23	9.58	1.47
56	1119717.40	1000970.80	10.61	10.69	0.10	1070216.20	1000368.00	6.53	6.56	0.04	1100948.60	1014673.40	7.84	9.17	1.45
64	1117871.00	1000870.80	10.47	10.54	0.09	1077195.00	1000351.00	7.13	7.17	0.04	1098635.00	1013103.20	7.79	8.98	1.29

Table A.40: The number of puts and takes performed during the spanning tree experiment on a Torus 2D 60 directed graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, and Idempotent LIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1055143.40	1015515.40	3.76	5.23	1.53	1000134.60	1000038.40	0.01	0.01	0.00	1000215.20	1000105.60	0.01	0.02	0.01
16	1041253.00	1008765.60	3.12	3.96	0.87	1000561.80	1000128.00	0.04	0.06	0.01	1000843.20	1000328.20	0.05	0.08	0.03
24	1059020.40	1010059.40	4.62	5.57	1.00	1001149.40	1000203.00	0.09	0.11	0.02	1000893.80	1000267.00	0.06	0.09	0.03
28	1071630.20	1012417.60	5.53	6.68	1.23	1000978.00	1000192.00	0.08	0.10	0.02	1000972.60	1000295.60	0.07	0.10	0.03
32	1092837.20	1016400.60	6.99	8.50	1.61	1001236.40	1000249.00	0.10	0.12	0.02	1001161.20	1000368.80	0.08	0.12	0.04
40	1098742.40	1018963.40	7.26	8.99	1.86	1001562.80	1000264.80	0.13	0.16	0.03	1001804.60	1000528.80	0.13	0.18	0.05
48	1111491.40	1019119.20	8.31	10.03	1.88	1002043.40	1000303.80	0.17	0.20	0.03	1002361.20	1000712.20	0.16	0.24	0.07
56	1134282.80	1022720.40	9.84	11.84	2.22	1001972.00	1000315.60	0.17	0.20	0.03	1002196.60	1000640.40	0.16	0.22	0.06
64	1142434.60	1024968.60	10.28	12.47	2.44	1002755.00	1000401.40	0.23	0.27	0.04	1003306.60	1001347.80	0.20	0.33	0.13

Table A.41: The number of puts and takes performed during the spanning tree experiment on a Torus 2D 60 directed graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000316.80	1000157.80	0.02	0.03	0.02	1000259.60	1000127.80	0.01	0.03	0.01	1000272.00	1000170.80	0.01	0.03	0.02
16	1000800.20	1000386.80	0.04	0.08	0.04	1000788.60	1000396.20	0.04	0.08	0.04	1000540.80	1000279.40	0.03	0.05	0.03
24	1000965.00	1000379.00	0.06	0.10	0.04	1001005.40	1000443.00	0.06	0.10	0.04	1000848.60	1000386.60	0.05	0.08	0.04
28	1001255.60	1000588.60	0.07	0.13	0.06	1001240.40	1000485.40	0.08	0.12	0.05	1001522.40	1000718.00	0.08	0.15	0.07
32	1001309.80	1000466.00	0.08	0.13	0.05	1001597.60	1000711.40	0.09	0.16	0.07	1001534.00	1000711.40	0.08	0.15	0.07
40	1001733.20	1000668.60	0.11	0.17	0.07	1002056.00	1000908.20	0.11	0.21	0.09	1001854.40	1000821.00	0.10	0.19	0.08
48	1002386.80	1000831.00	0.16	0.24	0.08	1002334.00	1000961.80	0.14	0.23	0.10	1002596.20	1001169.20	0.14	0.26	0.12
56	1002905.80	1001200.40	0.17	0.29	0.12	1003004.60	1001336.40	0.17	0.30	0.13	1002587.40	1001248.00	0.13	0.26	0.12
64	1002694.80	1000990.00	0.17	0.27	0.10	1002972.80	1001159.80	0.18	0.30	0.12	1003686.00	1001759.00	0.19	0.37	0.18

Table A.42: The number of puts and takes performed during the spanning tree experiment on a Torus 2D 60 directed graph with an initial size of 256 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Directed Torus 2D 60%. Initial size of 1,000,000 items.

Algorithm	Chase-Lev					Cilk THE					Idempotent LIFO				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1025175.60	1000138.80	2.44	2.46	0.01	1024845.60	1000102.40	2.41	2.42	0.01	1026934.20	1007803.20	1.86	2.62	0.77
16	1035440.00	1000209.40	3.40	3.42	0.02	1030896.00	1000106.80	2.99	3.00	0.01	1039888.60	1010166.80	2.86	3.84	1.01
24	1048179.80	1000201.00	4.58	4.60	0.02	1049445.80	1000216.20	4.69	4.71	0.02	1054200.40	1008389.60	4.35	5.14	0.83
28	1063140.80	1000313.60	5.91	5.94	0.03	1054725.80	1000222.80	5.17	5.19	0.02	1057040.00	1008122.20	4.63	5.40	0.81
32	1071357.80	1000335.60	6.63	6.66	0.03	1068250.60	1000250.00	6.37	6.39	0.02	1059531.00	1007431.60	4.92	5.62	0.74
40	1095432.60	1000462.00	8.67	8.71	0.05	1089231.20	1000408.00	8.15	8.19	0.04	1076395.00	1009218.60	6.24	7.10	0.91
48	1104878.20	1000583.20	9.44	9.49	0.06	1082717.20	1000373.40	7.61	7.64	0.04	1086233.00	1010473.60	6.97	7.94	1.04
56	1113319.60	1000719.00	10.11	10.18	0.07	1079389.80	1000353.40	7.32	7.36	0.04	1090696.60	1010237.40	7.38	8.32	1.01
64	1104270.40	1000584.80	9.39	9.44	0.06	1077770.20	1000360.40	7.18	7.22	0.04	1089794.00	1011644.40	7.17	8.24	1.15

Table A.43: The number of puts and takes performed during the spanning tree experiment on a Torus 2D 60 directed graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, and Idempotent LIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1025842.00	1005123.20	2.02	2.52	0.51	1000123.00	1000039.00	0.01	0.01	0.00	1000301.40	1000149.60	0.02	0.03	0.01
16	1046681.40	1010157.60	3.49	4.46	1.01	1000450.80	1000107.60	0.03	0.05	0.01	1000686.20	1000285.40	0.04	0.07	0.03
24	1058272.40	1009314.40	4.63	5.51	0.92	1001161.20	1000237.00	0.09	0.12	0.02	1000730.80	1000249.80	0.05	0.07	0.02
28	1073096.00	1014384.00	5.47	6.81	1.42	1001102.80	1000202.00	0.09	0.11	0.02	1001206.00	1000391.20	0.08	0.12	0.04
32	1078527.80	1012209.60	6.15	7.28	1.21	1001080.00	1000165.80	0.09	0.11	0.02	1001670.00	1000495.80	0.12	0.17	0.05
40	1100400.40	1018060.60	7.48	9.12	1.77	1001490.60	1000242.40	0.12	0.15	0.02	1001779.40	1000504.00	0.13	0.18	0.05
48	1122020.80	1018072.80	9.26	10.88	1.78	1002240.80	1000358.80	0.19	0.22	0.04	1002334.00	1000691.00	0.16	0.23	0.07
56	1135747.60	1023634.80	9.87	11.95	2.31	1002077.40	1000340.60	0.17	0.21	0.03	1002878.00	1000802.40	0.21	0.29	0.08
64	1143298.40	1018276.20	10.94	12.53	1.79	1002330.80	1000346.40	0.20	0.23	0.03	1002949.40	1001015.80	0.19	0.29	0.10

Table A.44: The number of puts and takes performed during the spanning tree experiment on a Torus 2D 60 directed graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000270.80	1000170.40	0.01	0.03	0.02	1000284.60	1000121.20	0.02	0.03	0.01	1000336.40	1000184.60	0.02	0.03	0.02
16	1000653.60	1000361.00	0.03	0.07	0.04	1000498.80	1000211.00	0.03	0.05	0.02	1000598.20	1000300.60	0.03	0.06	0.03
24	1001416.60	1000766.40	0.06	0.14	0.08	1001630.00	1001148.80	0.05	0.16	0.11	1000971.00	1000429.20	0.05	0.10	0.04
28	1001197.00	1000499.20	0.07	0.12	0.05	1001101.20	1000328.00	0.08	0.11	0.03	1001178.20	1000492.60	0.07	0.12	0.05
32	1001631.60	1000773.40	0.09	0.16	0.08	1001289.60	1000404.60	0.09	0.13	0.04	1001458.20	1000694.20	0.08	0.15	0.07
40	1001923.40	1000808.40	0.11	0.19	0.08	1001757.40	1000538.20	0.12	0.18	0.05	1001854.40	1000776.80	0.11	0.19	0.08
48	1002054.40	1000666.60	0.14	0.21	0.07	1002171.60	1000781.40	0.14	0.22	0.08	1002693.60	1001078.80	0.16	0.27	0.11
56	1002239.40	1000873.40	0.14	0.22	0.09	1002582.40	1000726.20	0.19	0.26	0.07	1002769.00	1001239.80	0.15	0.28	0.12
64	1002380.00	1000918.80	0.15	0.24	0.09	1002292.00	1000762.40	0.15	0.23	0.08	1002695.20	1001221.80	0.15	0.27	0.12

Table A.45: The number of puts and takes performed during the spanning tree experiment on a Torus 2D 60 directed graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Undirected Torus 2D 60%. Initial size of 256 items.

Algorithm	Chase-Lev					Cilk THE					Idempotent LIFO				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1082651.80	1000242.40	7.61	7.63	0.02	1063099.40	1000081.80	5.93	5.94	0.01	1015008.20	1006389.00	0.85	1.48	0.63
16	1190434.80	1001568.20	15.87	16.00	0.16	1092441.80	1000196.40	8.44	8.46	0.02	1016436.40	1007097.00	0.92	1.62	0.70
24	1232420.20	1001869.00	18.71	18.86	0.19	1111028.00	1000172.40	9.98	9.99	0.02	1016585.20	1005068.60	1.13	1.63	0.50
28	1273800.40	1003398.60	21.23	21.49	0.34	1090600.00	1000155.80	8.29	8.31	0.02	1019334.80	1006844.20	1.23	1.90	0.68
32	1313017.20	1003624.60	23.56	23.84	0.36	1108379.00	1000206.60	9.76	9.78	0.02	1013934.80	1004031.60	0.98	1.37	0.40
40	1268782.40	1003778.80	20.89	21.18	0.38	1118167.20	1000243.40	10.55	10.57	0.02	1025509.00	1007804.20	1.73	2.49	0.77
48	1301853.80	1004452.00	22.84	23.19	0.44	1078986.40	1000199.20	7.30	7.32	0.02	1027885.20	1007215.40	2.01	2.71	0.72
56	1337093.00	1004828.60	24.85	25.21	0.48	1095201.20	1000176.80	8.68	8.69	0.02	1037988.40	1010988.40	2.60	3.66	1.09
64	1378929.80	1005921.80	27.05	27.48	0.59	1078281.00	1000188.60	7.24	7.26	0.02	1038693.20	1010601.20	2.70	3.73	1.05

Table A.46: The number of puts and takes performed during the spanning tree experiment on a Torus 2D 60 undirected graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, and Idempotent LIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1120475.40	1056299.20	5.73	10.75	5.33	1000032.80	1000022.40	0.00	0.00	0.00	1000033.20	1000021.80	0.00	0.00	0.00
16	1136410.60	1045044.80	8.04	12.00	4.31	1000076.60	1000049.80	0.00	0.01	0.00	1000106.80	1000068.60	0.00	0.01	0.01
24	1146217.40	1040549.60	9.22	12.76	3.90	1000157.20	1000075.80	0.01	0.02	0.01	1000179.40	1000103.60	0.01	0.02	0.01
28	1268897.80	1081951.60	14.73	21.19	7.57	1000211.40	1000092.40	0.01	0.02	0.01	1000230.00	1000132.40	0.01	0.02	0.01
32	1267577.20	1066799.40	15.84	21.11	6.26	1000312.00	1000149.80	0.02	0.03	0.01	1000282.60	1000158.40	0.01	0.03	0.02
40	1355016.40	1084539.80	19.96	26.20	7.79	1000457.40	1000148.00	0.03	0.05	0.01	1000600.00	1000294.80	0.03	0.06	0.03
48	1325575.60	1067129.20	19.50	24.56	6.29	1000677.20	1000292.60	0.04	0.07	0.03	1000770.20	1000385.20	0.04	0.08	0.04
56	1405648.40	1087529.60	22.63	28.86	8.05	1000916.20	1000367.80	0.05	0.09	0.04	1000938.40	1000455.60	0.05	0.09	0.05
64	1383867.00	1083713.80	21.69	27.74	7.72	1000917.40	1000327.00	0.06	0.09	0.03	1001373.20	1000823.80	0.05	0.14	0.08

Table A.47: The number of puts and takes performed during the spanning tree experiment on a Torus 2D 60 undirected graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000066.80	1000029.80	0.00	0.01	0.00	1000039.80	1000028.00	0.00	0.00	0.00	1000047.00	1000034.80	0.00	0.00	0.00
16	1000115.40	1000074.00	0.00	0.01	0.01	1000095.40	1000066.20	0.00	0.01	0.01	1000106.20	1000075.80	0.00	0.01	0.01
24	1000189.60	1000115.00	0.01	0.02	0.01	1000179.40	1000113.20	0.01	0.02	0.01	1000176.20	1000118.00	0.01	0.02	0.01
28	1000286.80	1000161.60	0.01	0.03	0.02	1000230.00	1000146.00	0.01	0.02	0.01	1000251.40	1000174.20	0.01	0.03	0.02
32	1000334.20	1000186.60	0.01	0.03	0.02	1000289.00	1000165.60	0.01	0.03	0.02	1000283.40	1000175.20	0.01	0.03	0.02
40	1000536.60	1000305.20	0.02	0.05	0.03	1000497.40	1000282.00	0.02	0.05	0.03	1000471.00	1000296.00	0.02	0.05	0.03
48	1000764.00	1000489.00	0.03	0.08	0.05	1000716.40	1000428.00	0.03	0.07	0.04	1000636.40	1000390.60	0.02	0.06	0.04
56	1001162.60	1000686.40	0.05	0.12	0.07	1000981.80	1000585.60	0.04	0.10	0.06	1000915.60	1000593.20	0.03	0.09	0.06
64	1000952.80	1000557.60	0.04	0.10	0.06	1001459.40	1000924.40	0.05	0.15	0.09	1001155.80	1000685.60	0.05	0.12	0.07

Table A.48: The number of puts and takes performed during the spanning tree experiment on a Torus 2D 60 undirected graph with an initial size of 256 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Undirected Torus 2D 60%. Initial size of 1,000,000 items.

Algorithm	Chase-Lev					Cilk THE					Idempotent LIFO				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1112582.60	1000231.00	10.10	10.12	0.02	1067912.60	1000053.20	6.35	6.36	0.01	1012152.40	1005565.40	0.65	1.20	0.55
16	1173239.20	1001615.60	14.63	14.77	0.16	1072525.20	1000120.40	6.75	6.76	0.01	1017683.20	1006878.60	1.06	1.74	0.68
24	1251781.80	1002965.60	19.88	20.11	0.30	1087262.20	1000188.00	8.01	8.03	0.02	1026943.80	1010741.00	1.58	2.62	1.06
28	1320410.80	1003479.40	24.00	24.27	0.35	1075167.00	1000119.40	6.98	6.99	0.01	1028850.40	1009456.40	1.89	2.80	0.94
32	1385489.00	1005180.60	27.45	27.82	0.52	1089971.20	1000149.00	8.24	8.25	0.01	1026137.00	1007941.00	1.77	2.55	0.79
40	1321151.20	1004953.60	23.93	24.31	0.49	1111391.20	1000184.20	10.01	10.02	0.02	1032877.60	1009386.40	2.27	3.18	0.93
48	1386393.80	1005561.40	27.47	27.87	0.55	1094168.40	1000152.00	8.59	8.61	0.02	1019789.20	1005502.40	1.40	1.94	0.55
56	1371865.60	1005792.40	26.68	27.11	0.58	1095965.40	1000166.20	8.74	8.76	0.02	1033268.60	1007121.40	2.53	3.22	0.71
64	1420248.60	1008221.40	29.01	29.59	0.82	1092770.00	1000175.20	8.47	8.49	0.02	1056644.80	1012537.00	4.17	5.36	1.24

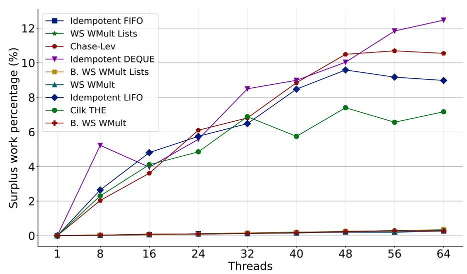
Table A.49: The number of puts and takes performed during the spanning tree experiment on a Torus 2D 60 undirected graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, and Idempotent LIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1122378.40	1048091.20	6.62	10.90	4.59	1000031.80	1000020.40	0.00	0.00	0.00	1000039.20	1000028.20	0.00	0.00	0.00
16	1264133.60	1092552.00	13.57	20.89	8.47	1000082.60	1000057.00	0.00	0.01	0.01	1000096.00	1000064.60	0.00	0.01	0.01
24	1331263.80	1092263.20	17.95	24.88	8.45	1000148.40	1000074.00	0.01	0.01	0.01	1000210.00	1000142.80	0.01	0.02	0.01
28	1322992.00	1078581.40	18.47	24.41	7.29	1000196.60	1000093.40	0.01	0.02	0.01	1000215.60	1000132.40	0.01	0.02	0.01
32	1376831.40	1094660.40	20.49	27.37	8.65	1000296.20	1000139.20	0.02	0.03	0.01	1000306.20	1000169.00	0.01	0.03	0.02
40	1393376.60	1091114.20	21.69	28.23	8.35	1000454.60	1000162.40	0.03	0.05	0.02	1000418.20	1000203.20	0.02	0.04	0.02
48	1302068.20	1066873.00	18.06	23.20	6.27	1000485.00	1000202.20	0.03	0.05	0.02	1000669.20	1000335.00	0.03	0.07	0.03
56	1371225.80	1068227.80	22.10	27.07	6.39	1000779.00	1000293.60	0.05	0.08	0.03	1001079.20	1000576.40	0.05	0.11	0.06
64	1408743.20	1079272.40	23.39	29.01	7.34	1000845.80	1000302.60	0.05	0.08	0.03	1001079.40	1000588.40	0.05	0.11	0.06

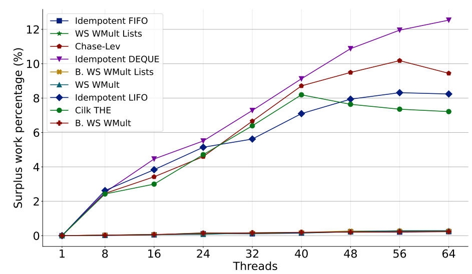
Table A.50: The number of puts and takes performed during the spanning tree experiment on a Torus 2D 60 undirected graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000076.60	1000052.00	0.00	0.01	0.01	1000044.80	1000032.80	0.00	0.00	0.00	1000131.00	1000121.40	0.00	0.01	0.01
16	1000102.80	1000075.60	0.00	0.01	0.01	1000104.80	1000075.20	0.00	0.01	0.01	1000093.40	1000067.80	0.00	0.01	0.01
24	1000166.80	1000106.40	0.01	0.02	0.01	1000172.60	1000103.20	0.01	0.02	0.01	1000242.40	1000178.40	0.01	0.02	0.02
28	1000217.60	1000138.40	0.01	0.02	0.01	1000225.40	1000126.40	0.01	0.02	0.01	1000251.00	1000179.00	0.01	0.03	0.02
32	1000388.40	1000257.40	0.01	0.04	0.03	1000311.80	1000203.20	0.01	0.03	0.02	1000368.20	1000268.60	0.01	0.04	0.03
40	1000494.40	1000288.40	0.02	0.05	0.03	1000457.60	1000237.20	0.02	0.05	0.02	1000458.00	1000279.60	0.02	0.05	0.03
48	1000700.60	1000446.80	0.03	0.07	0.04	1000655.40	1000335.20	0.03	0.07	0.03	1000841.00	1000484.40	0.04	0.08	0.05
56	1000793.40	1000432.80	0.04	0.08	0.04	1000984.60	1000607.00	0.04	0.10	0.06	1000735.40	1000403.40	0.03	0.07	0.04
64	1001009.80	1000573.80	0.04	0.10	0.06	1001190.80	1000569.20	0.06	0.12	0.06	1000992.80	1000564.00	0.04	0.10	0.06

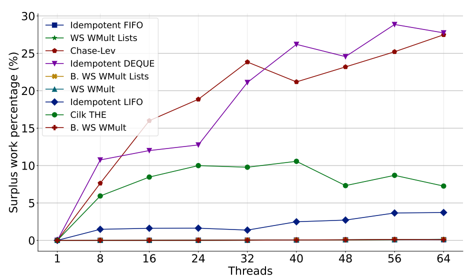
Table A.51: The number of puts and takes performed during the spanning tree experiment on a Torus 2D 60 undirected graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.



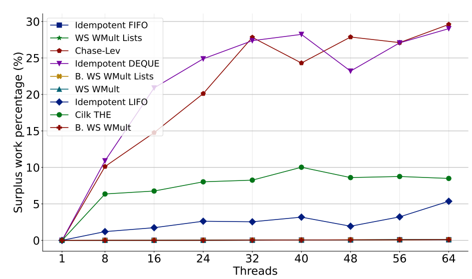
(a) Surplus work: Directed Torus 2D 60%. Initial size of 256 items



(b) Surplus work: Directed Torus 2D 60%. Initial size of 1,000,000 items

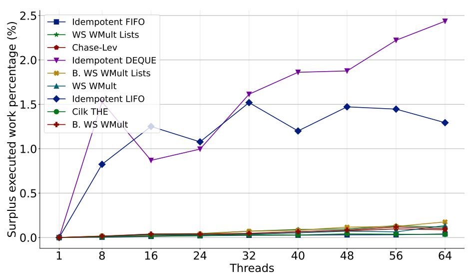


(c) Surplus work: Undirected Torus 2D 60%. Initial size of 256 items

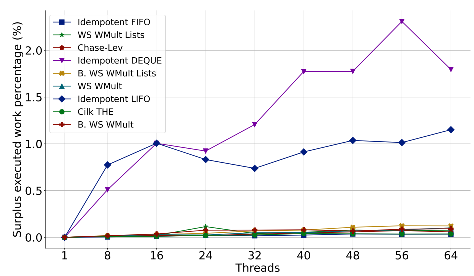


(d) Surplus work: Undirected Torus 2D 60%. Initial size of 1,000,000 items

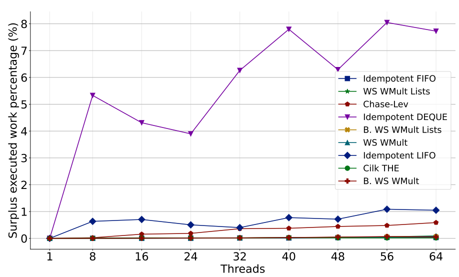
Figure A.8: Surplus work (percentage) of the experiments. Surplus work: the difference between the total number of Puts and the number of puts in sequential executions (i.e., $1,000,000$).



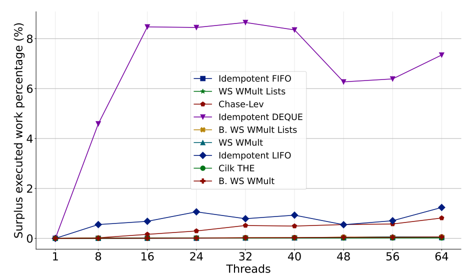
(a) Executed surplus work: Directed Torus 2D 60%. Initial size of 256 items



(b) Executed surplus work: Directed Torus 2D 60%. Initial size of 1,000,000 items



(c) Executed Surplus work: Undirected Torus 2D 60%. Initial size of 256 items



(d) Executed surplus work: Undirected Torus 2D 60%. Initial size of 1,000,000 items

Figure A.9: Executed surplus work (percentage) of the experiments. Surplus work: the difference between the total number of Takes and the number of takes in sequential executions (i.e., $1,000,000$).

Directed Torus 3D. Initial size of 256 items.

Algorithm	Chase-Lev					Cilk THE					Idempotent LIFO				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1518382.80	1002729.80	33.96	34.14	0.27	1114649.40	1000045.60	10.28	10.29	0.00	1098251.40	1026479.20	6.54	8.95	2.58
16	1464633.60	1001836.80	31.60	31.72	0.18	1139491.40	1000081.80	12.23	12.24	0.01	1071382.80	1019950.80	4.80	6.66	1.96
24	1488988.60	1001817.80	32.72	32.84	0.18	1109569.60	1000095.40	9.87	9.87	0.01	1066547.60	1019358.00	4.42	6.24	1.90
28	1401148.20	1001090.00	28.55	28.63	0.11	1118098.60	1000104.80	10.55	10.56	0.01	1060865.60	1017061.20	4.13	5.74	1.68
32	1428766.60	1001228.60	29.92	30.01	0.12	1108755.80	1000092.20	9.80	9.81	0.01	1058670.80	1016389.00	3.99	5.54	1.61
40	1402662.80	1000773.60	28.65	28.71	0.08	1118007.40	1000136.80	10.54	10.56	0.01	1057805.80	1015497.60	4.00	5.46	1.53
48	1419858.00	1001236.40	29.48	29.57	0.12	1114775.00	1000101.40	10.29	10.30	0.01	1068424.80	1018635.00	4.66	6.40	1.83
56	1444332.80	1001655.20	30.65	30.76	0.17	1121852.40	1000118.20	10.85	10.86	0.01	1080560.40	1019777.80	5.63	7.46	1.94
64	1428939.60	1002321.20	29.86	30.02	0.23	1085480.00	1000122.40	7.86	7.87	0.01	1080866.60	1023482.20	5.31	7.48	2.29

Table A.52: The number of puts and takes performed during the spanning tree experiment on a Torus 3D directed graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, and Idempotent LIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1628867.80	1265167.80	22.33	38.61	20.96	1000414.20	1000398.60	0.00	0.04	0.04	1000310.60	1000286.60	0.00	0.03	0.03
16	1653954.80	1225914.00	25.88	39.54	18.43	1001764.20	1001724.80	0.00	0.18	0.17	1002291.40	1001867.80	0.04	0.23	0.19
24	1574299.20	1181708.80	24.94	36.48	15.38	1000944.60	1000684.60	0.03	0.09	0.07	1001082.80	1000976.80	0.01	0.11	0.10
28	1623697.60	1185717.00	26.97	38.41	15.66	1001322.40	1000952.00	0.04	0.13	0.10	1001658.40	1000991.80	0.07	0.17	0.10
32	1636044.00	1202424.20	26.50	38.88	16.83	1001061.40	1000741.60	0.03	0.11	0.07	1001306.60	1000690.20	0.06	0.13	0.07
40	1551282.60	1171691.40	24.47	35.54	14.65	1000989.80	1000481.40	0.05	0.10	0.05	1001691.40	1000946.00	0.07	0.17	0.09
48	1750071.00	1290436.00	26.26	42.86	22.51	1004021.80	1000727.80	0.33	0.40	0.07	1003146.40	1001578.40	0.16	0.31	0.16
56	1533304.40	1160251.40	24.33	34.78	13.81	1003050.00	1000705.60	0.23	0.30	0.07	1003658.00	1001941.00	0.17	0.36	0.19
64	1698806.60	1279848.40	24.66	41.14	21.87	1004193.20	1000838.80	0.33	0.42	0.08	1005172.80	1002514.80	0.26	0.51	0.25

Table A.53: The number of puts and takes performed during the spanning tree experiment on a Torus 3D directed graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000512.20	1000367.40	0.01	0.05	0.04	1000903.80	1000891.00	0.00	0.09	0.09	1000700.40	1000686.60	0.00	0.07	0.07
16	1000832.80	1000705.00	0.01	0.08	0.07	1002111.00	1002073.40	0.00	0.21	0.21	1001943.60	1001889.60	0.01	0.19	0.19
24	1000730.80	1000478.40	0.03	0.07	0.05	1001042.40	1000933.20	0.01	0.10	0.09	1001613.60	1001425.80	0.02	0.16	0.14
28	1002505.80	1001286.60	0.12	0.25	0.13	1001728.20	1001212.20	0.05	0.17	0.12	1001119.80	1000932.20	0.02	0.11	0.09
32	1001376.00	1000711.20	0.07	0.14	0.07	1001091.00	1000958.20	0.01	0.11	0.10	1001096.80	1000720.60	0.04	0.11	0.07
40	1003103.00	1001514.20	0.16	0.31	0.15	1002010.40	1001200.20	0.08	0.20	0.12	1001492.80	1001108.40	0.04	0.15	0.11
48	1002573.80	1001210.40	0.14	0.26	0.12	1002963.20	1001781.00	0.12	0.30	0.18	1001636.40	1000914.20	0.07	0.16	0.09
56	1006062.00	1002658.60	0.34	0.60	0.27	1003712.80	1002003.20	0.17	0.37	0.20	1004573.40	1001954.40	0.26	0.46	0.20
64	1006329.00	1002971.80	0.33	0.63	0.30	1007954.20	1003634.60	0.43	0.79	0.36	1004020.40	1001714.00	0.23	0.40	0.17

Table A.54: The number of puts and takes performed during the spanning tree experiment on a Torus 3D directed graph with an initial size of 256 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Directed Torus 3D. Initial size of 1,000,000 items.

Algorithm	Chase-Lev					Cilk THE					Idempotent LIFO				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1372395.20	1001635.40	27.02	27.13	0.16	1122614.60	1000088.20	10.91	10.92	0.01	1059276.80	1024567.20	3.28	5.60	2.40
16	1433214.60	1000323.40	30.20	30.23	0.03	1127121.80	1000090.60	11.27	11.28	0.01	1057170.20	1021495.00	3.37	5.41	2.10
24	1448832.80	1000533.80	30.94	30.98	0.05	1114541.40	1000115.80	10.27	10.28	0.01	1059802.20	1020125.00	3.74	5.64	1.97
28	1385503.60	1001000.20	27.75	27.82	0.10	1111023.60	1000073.40	9.99	9.99	0.01	1064373.40	1019316.40	4.23	6.05	1.90
32	1416874.80	1000771.80	29.37	29.42	0.08	1101505.00	1000085.40	9.21	9.22	0.01	1060945.20	1019519.00	3.90	5.74	1.91
40	1419027.00	1000730.20	29.48	29.53	0.07	1106658.20	1000147.20	9.62	9.64	0.01	1059128.00	1017581.40	3.92	5.58	1.73
48	1342022.00	1000699.40	25.43	25.49	0.07	1122954.40	1000110.60	10.94	10.95	0.01	1066871.60	1019310.20	4.46	6.27	1.89
56	1378406.80	1000663.00	27.40	27.45	0.07	1095288.80	1000113.60	8.69	8.70	0.01	1065900.60	1017963.20	4.50	6.18	1.76
64	1349161.60	1001019.60	25.80	25.88	0.10	1107860.00	1000147.20	9.72	9.74	0.01	1067699.60	1018627.60	4.60	6.34	1.83

Table A.55: The number of puts and takes performed during the spanning tree experiment on a Torus 3D directed graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, and Idempotent LIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1411172.60	1145740.00	18.81	29.14	12.72	1000993.40	1000972.00	0.00	0.10	0.10	1000764.60	1000752.00	0.00	0.08	0.08
16	1687686.20	1255514.00	25.61	40.75	20.35	1002500.00	1002471.40	0.00	0.25	0.25	1001408.00	1001336.40	0.01	0.14	0.13
24	1696765.80	1224102.60	27.86	41.06	18.31	1001302.40	1001198.80	0.01	0.13	0.12	1001260.00	1001009.60	0.03	0.13	0.10
28	1641905.20	1184651.60	27.85	39.10	15.59	1001690.80	1001322.80	0.04	0.17	0.13	1002090.00	1001566.00	0.05	0.21	0.16
32	1603087.60	1171833.40	26.90	37.62	14.66	1001271.80	1000695.20	0.06	0.13	0.07	1001681.60	1001116.40	0.06	0.17	0.11
40	1410310.80	1091270.80	22.62	29.09	8.36	1000720.40	1000444.60	0.03	0.07	0.04	1004530.20	1002401.60	0.21	0.45	0.24
48	1560914.00	1159006.60	25.75	35.93	13.72	1002612.60	1001084.40	0.15	0.26	0.11	1003340.60	1001872.80	0.15	0.33	0.19
56	1593918.80	1199605.40	24.74	37.26	16.64	1003855.60	1000798.40	0.30	0.38	0.08	1005926.20	1003121.40	0.28	0.59	0.31
64	1519171.60	1166243.60	23.23	34.17	14.25	1003976.00	1000806.60	0.32	0.40	0.08	1004574.40	1002374.00	0.22	0.46	0.24

Table A.56: The number of puts and takes performed during the spanning tree experiment on a Torus 3D directed graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1001618.60	1001247.60	0.04	0.16	0.12	1001379.00	1001365.00	0.00	0.14	0.14	1000828.60	1000821.00	0.00	0.08	0.08
16	1001677.60	1001638.40	0.00	0.17	0.16	1002739.40	1002661.80	0.01	0.27	0.27	1002436.40	1002410.60	0.00	0.24	0.24
24	1001148.20	1000935.80	0.02	0.11	0.09	1001889.40	1001561.80	0.03	0.19	0.16	1001531.20	1001391.40	0.01	0.15	0.14
28	1001631.20	1001572.00	0.01	0.16	0.16	1002189.20	1001866.80	0.03	0.22	0.19	1001543.80	1001294.40	0.02	0.15	0.13
32	1000783.60	1000616.60	0.02	0.08	0.06	1001376.80	1001014.80	0.04	0.14	0.10	1001528.60	1001371.20	0.02	0.15	0.14
40	1001286.80	1000786.60	0.05	0.13	0.08	1002312.60	1001193.40	0.11	0.23	0.12	1001227.20	1000858.40	0.04	0.12	0.09
48	1002554.80	1001447.00	0.11	0.25	0.14	1003279.00	1001647.80	0.16	0.33	0.16	1002338.00	1001087.40	0.12	0.23	0.11
56	1004276.80	1001896.40	0.24	0.43	0.19	1003606.00	1001852.00	0.17	0.36	0.18	1002938.80	1001639.80	0.13	0.29	0.16
64	1004331.00	1002208.80	0.21	0.43	0.22	1003899.20	1002418.40	0.15	0.39	0.24	1004068.00	1001807.20	0.23	0.41	0.18

Table A.57: The number of puts and takes performed during the spanning tree experiment on a Torus 3D directed graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Undirected Torus 3D. Initial size of 256 items.

Algorithm	Chase-Lev					Cilk THE					Idempotent LIFO				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1012480.80	1000075.60	1.23	1.23	0.01	1001764.60	1000048.60	0.17	0.18	0.00	1006449.60	1001777.80	0.46	0.64	0.18
16	1011540.40	1000119.80	1.13	1.14	0.01	1003511.00	1000097.60	0.34	0.35	0.01	1009970.00	1002425.20	0.75	0.99	0.24
24	1032579.00	1000137.20	3.14	3.16	0.01	1014026.80	1000154.40	1.37	1.38	0.02	1024091.80	1004787.40	1.89	2.35	0.48
28	1024871.20	1000147.00	2.41	2.43	0.01	1018743.00	1000119.60	1.83	1.84	0.01	1054275.20	1010015.00	4.20	5.15	0.99
32	1038443.40	1000195.40	3.68	3.70	0.02	1017599.60	1000141.40	1.72	1.73	0.01	1042960.20	1007669.60	3.38	4.12	0.76
40	1068735.80	1000163.40	6.42	6.43	0.02	1030333.60	1000163.20	2.93	2.94	0.02	1084445.80	1015863.00	6.32	7.79	1.56
48	1090499.80	1000194.40	8.28	8.30	0.02	1040533.80	1000186.00	3.88	3.90	0.02	1099221.20	1016691.80	7.51	9.03	1.64
56	1149163.00	1000275.20	12.96	12.98	0.03	1038258.40	1000210.00	3.66	3.68	0.02	1131282.40	1021065.00	9.74	11.60	2.06
64	1127615.20	1000292.20	11.29	11.32	0.03	1035878.80	1000245.20	3.44	3.46	0.02	1161668.60	1025824.80	11.69	13.92	2.52

Table A.58: The number of puts and takes performed during the spanning tree experiment on a Torus 3D undirected graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, and Idempotent LIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1008935.40	1002324.00	0.66	0.89	0.23	1000076.00	1000067.20	0.00	0.01	0.01	1000072.40	1000062.00	0.00	0.01	0.01
16	1005664.00	1000783.80	0.49	0.56	0.08	1000249.00	1000229.80	0.00	0.02	0.02	1000228.00	1000205.00	0.00	0.02	0.02
24	1019054.20	1004002.20	1.48	1.87	0.40	1000277.20	1000252.60	0.00	0.03	0.03	1000227.60	1000196.40	0.00	0.02	0.02
28	1040647.80	1009364.40	3.01	3.91	0.93	1000310.40	1000268.00	0.00	0.03	0.03	1000314.80	1000267.60	0.00	0.03	0.03
32	1034936.40	1006775.80	2.72	3.38	0.67	1000363.20	1000318.20	0.00	0.04	0.03	1000268.80	1000211.80	0.01	0.03	0.02
40	1091603.40	1020057.40	6.55	8.39	1.97	1000370.40	1000300.20	0.01	0.04	0.03	1000308.40	1000237.40	0.01	0.03	0.02
48	1096793.00	1017072.00	7.27	8.83	1.68	1000376.60	1000284.40	0.01	0.04	0.03	1000644.20	1000483.60	0.02	0.06	0.05
56	1139456.20	1028849.40	9.71	12.24	2.80	1000434.20	1000282.80	0.02	0.04	0.03	1000847.20	1000609.40	0.02	0.08	0.06
64	1170783.80	1031857.80	11.87	14.59	3.09	1000582.20	1000272.60	0.03	0.06	0.03	1002944.60	1002011.20	0.09	0.29	0.20

Table A.59: The number of puts and takes performed during the spanning tree experiment on a Torus 3D undirected graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000093.60	1000056.20	0.00	0.01	0.01	1000098.20	1000092.40	0.00	0.01	0.01	1000071.20	1000066.20	0.00	0.01	0.01
16	1000127.20	1000089.80	0.00	0.01	0.01	1000255.40	1000238.60	0.00	0.03	0.02	1000243.80	1000227.80	0.00	0.02	0.02
24	1000152.60	1000119.20	0.00	0.02	0.01	1000236.80	1000204.00	0.00	0.02	0.02	1000215.40	1000181.00	0.00	0.02	0.02
28	1000263.00	1000153.00	0.01	0.03	0.02	1000269.40	1000216.40	0.01	0.03	0.02	1000284.60	1000248.60	0.00	0.03	0.02
32	1000399.20	1000189.60	0.02	0.04	0.02	1000340.40	1000286.20	0.01	0.03	0.03	1000289.60	1000247.80	0.00	0.03	0.02
40	1000306.40	1000234.00	0.01	0.03	0.02	1000324.40	1000260.40	0.01	0.03	0.03	1000350.40	1000281.20	0.01	0.04	0.03
48	1000529.40	1000335.40	0.02	0.05	0.03	1000423.40	1000322.60	0.01	0.04	0.03	1000376.60	1000296.60	0.01	0.04	0.03
56	1001127.00	1000344.40	0.08	0.11	0.03	1001306.60	1001000.80	0.03	0.13	0.10	1000621.40	1000389.80	0.02	0.06	0.04
64	1001016.60	1000428.80	0.06	0.10	0.04	1005599.20	1003886.20	0.17	0.56	0.39	1002100.20	1000660.00	0.14	0.21	0.07

Table A.60: The number of puts and takes performed during the spanning tree experiment on a Torus 3D undirected graph with an initial size of 256 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Undirected Torus 3D. Initial size of 1,000,000 items.

Algorithm	Chase-Lev					Cilk THE					Idempotent LIFO				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1007042.80	1000047.20	0.69	0.70	0.00	1002218.40	1000051.20	0.22	0.22	0.01	1004675.40	1001251.40	0.34	0.47	0.12
16	1004167.80	1000118.60	0.40	0.42	0.01	1001520.40	1000105.20	0.14	0.15	0.01	1004673.20	1001481.20	0.32	0.47	0.15
24	1035109.00	1000155.00	3.38	3.39	0.02	1008261.00	1000124.40	0.81	0.82	0.01	1026903.20	1006133.40	2.02	2.62	0.61
28	1031260.40	1000129.40	3.02	3.03	0.01	1015662.00	1000117.40	1.53	1.54	0.01	1027101.20	1006384.60	2.02	2.64	0.63
32	1059586.40	1000160.20	5.61	5.62	0.02	1025879.80	1000162.80	2.51	2.52	0.02	1032898.80	1007668.40	2.44	3.19	0.76
40	1073014.20	1000190.40	6.79	6.80	0.02	1024220.60	1000202.20	2.35	2.36	0.02	1036169.20	1007405.40	2.78	3.49	0.74
48	1083288.40	1000192.80	7.67	7.69	0.02	1025092.00	1000183.00	2.43	2.45	0.02	1087383.00	1019444.60	6.25	8.04	1.91
56	1117937.00	1000232.20	10.53	10.55	0.02	1034225.00	1000243.00	3.29	3.31	0.02	1082462.20	1017152.00	6.03	7.62	1.69
64	1119506.60	1000232.60	10.65	10.67	0.02	1032297.00	1000199.00	3.11	3.13	0.02	1104746.20	1019559.20	7.71	9.48	1.92

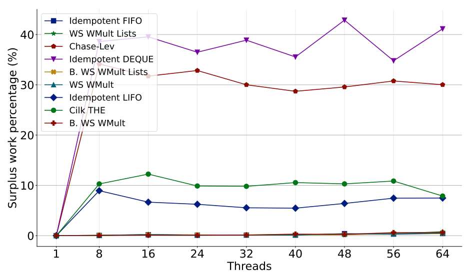
Table A.61: The number of puts and takes performed during the spanning tree experiment on a Torus 3D undirected graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, and Idempotent LIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1008787.00	1002766.40	0.60	0.87	0.28	1000103.60	1000096.20	0.00	0.01	0.01	1000100.20	1000092.40	0.00	0.01	0.01
16	1012180.00	1004830.20	0.73	1.20	0.48	1000322.80	1000309.80	0.00	0.03	0.03	1000266.20	1000242.20	0.00	0.03	0.02
24	1020304.60	1004372.20	1.56	1.99	0.44	1000377.20	1000352.40	0.00	0.04	0.04	1000371.20	1000340.00	0.00	0.04	0.03
28	1033808.80	1007483.60	2.55	3.27	0.74	1000281.60	1000243.20	0.00	0.03	0.02	1000384.20	1000334.20	0.00	0.04	0.03
32	1053407.60	1012625.60	3.87	5.07	1.25	1000302.80	1000246.40	0.01	0.03	0.02	1000283.40	1000226.00	0.01	0.03	0.02
40	1046909.20	1010809.80	3.45	4.48	1.07	1000287.80	1000200.40	0.01	0.03	0.02	1000327.00	1000246.00	0.01	0.03	0.02
48	1094059.80	1019987.60	6.77	8.60	1.96	1000421.00	1000217.80	0.02	0.04	0.02	1000417.20	1000328.60	0.01	0.04	0.03
56	1111778.80	1023329.00	7.96	10.05	2.28	1000677.60	1000316.00	0.04	0.07	0.03	1000755.00	1000551.80	0.02	0.08	0.06
64	1116098.60	1020284.40	8.58	10.40	1.99	1000990.60	1000259.60	0.07	0.10	0.03	1003775.40	1002759.80	0.10	0.38	0.28

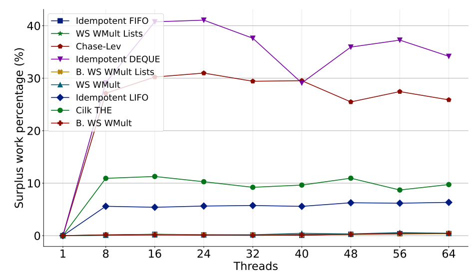
Table A.62: The number of puts and takes performed during the spanning tree experiment on a Torus 3D undirected graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000105.00	1000093.80	0.00	0.01	0.01	1000130.60	1000123.40	0.00	0.01	0.01	1000091.40	1000086.00	0.00	0.01	0.01
16	1000231.80	1000216.20	0.00	0.02	0.02	1000285.00	1000265.60	0.00	0.03	0.03	1000295.60	1000282.80	0.00	0.03	0.03
24	1000340.40	1000310.80	0.00	0.03	0.03	1000307.80	1000274.00	0.00	0.03	0.03	1000298.00	1000272.60	0.00	0.03	0.03
28	1000418.00	1000380.60	0.00	0.04	0.04	1000280.60	1000230.00	0.01	0.03	0.02	1000382.60	1000347.00	0.00	0.04	0.03
32	1000320.40	1000277.40	0.00	0.03	0.03	1000320.40	1000263.80	0.01	0.03	0.03	1000417.40	1000370.00	0.00	0.04	0.04
40	1000385.60	1000296.40	0.01	0.04	0.03	1000376.40	1000312.60	0.01	0.04	0.03	1000320.40	1000261.80	0.01	0.03	0.03
48	1000414.60	1000336.80	0.01	0.04	0.03	1001021.20	1000777.60	0.02	0.10	0.08	1000518.80	1000430.60	0.01	0.05	0.04
56	1000976.80	1000364.00	0.06	0.10	0.04	1002241.60	1001664.60	0.06	0.22	0.17	1001148.20	1000391.00	0.08	0.11	0.04
64	1000859.80	1000354.20	0.05	0.09	0.04	1003762.40	1002766.20	0.10	0.37	0.28	1000451.40	1000343.00	0.01	0.05	0.03

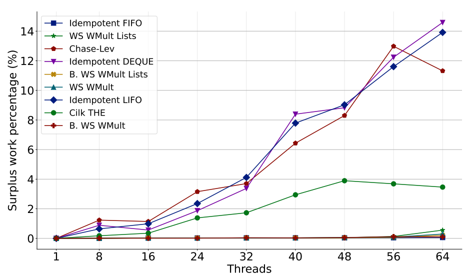
Table A.63: The number of puts and takes performed during the spanning tree experiment on a Torus 3D undirected graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.



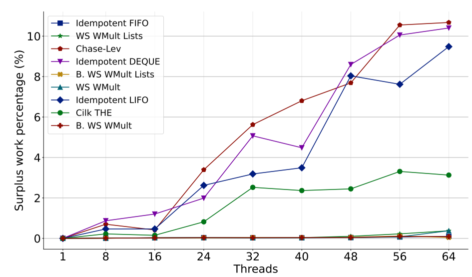
(a) Surplus work: Directed Torus 3D. Initial size of 256 items



(b) Surplus work: Directed Torus 3D. Initial size of 1,000,000 items

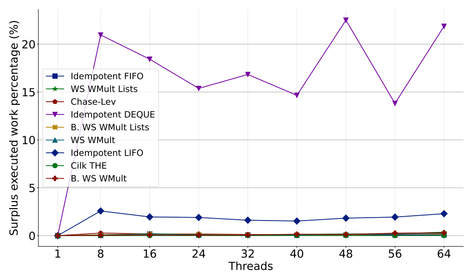


(c) Surplus work: Undirected Torus 3D. Initial size of 256 items

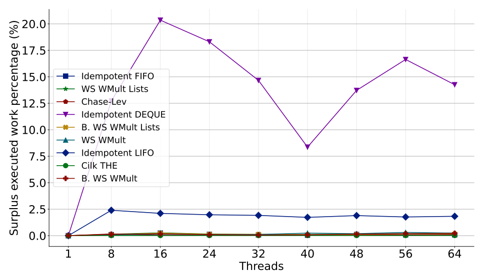


(d) Surplus work: Undirected Torus 3D. Initial size of 1,000,000 items

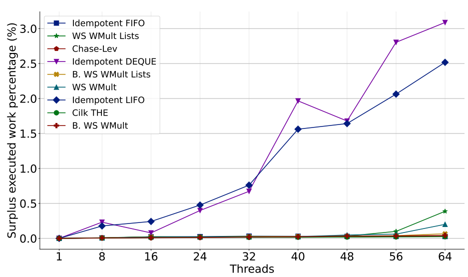
Figure A.10: Surplus work (percentage) of the experiments. Surplus work: the difference between the total number of Puts and the number of puts in sequential executions (i.e., $1,000,000$).



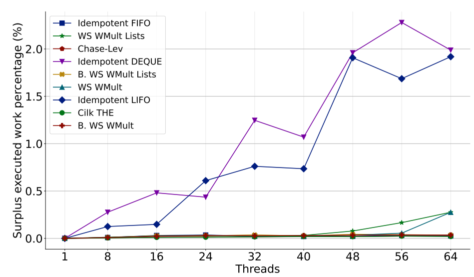
(a) Executed surplus work: Directed Torus 3D. Initial size of 256 items



(b) Executed surplus work: Directed Torus 3D. Initial size of 1,000,000 items



(c) Executed surplus work: Undirected Torus 3D. Initial size of 256 items



(d) Executed surplus work: Undirected Torus 3D. Initial size of 1,000,000 items

Figure A.11: Executed surplus work (percentage) of the experiments. Surplus work: the difference between the total number of Takes and the number of takes in sequential executions (i.e., $1,000,000$).

Directed Torus 3D 40%. Initial size of 256 items.

Algorithm	Chase-Lev					Cilk THE					Idempotent LIFO				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1005449.60	1000025.40	0.54	0.54	0.00	1003531.00	1000011.60	0.35	0.35	0.00	1007831.60	1003007.60	0.48	0.78	0.30
16	1007845.00	1000040.80	0.77	0.78	0.00	1003157.80	1000021.20	0.31	0.31	0.00	1007907.00	1002678.20	0.52	0.78	0.27
24	1010827.80	1000051.00	1.07	1.07	0.01	1014180.00	1000052.40	1.39	1.40	0.01	1012551.00	1003577.60	0.89	1.24	0.36
28	1018479.60	1000072.80	1.81	1.81	0.01	1009475.60	1000041.40	0.93	0.94	0.00	1025882.20	1006902.20	1.85	2.52	0.69
32	1020975.20	1000083.40	2.05	2.05	0.01	1018990.80	1000064.00	1.86	1.86	0.01	1031138.00	1009279.60	2.12	3.02	0.92
40	1034711.20	1000118.00	3.34	3.35	0.01	1020566.60	1000070.60	2.01	2.02	0.01	1050068.40	1012530.20	3.57	4.77	1.24
48	1057946.40	1000196.00	5.46	5.48	0.02	1033972.80	1000102.40	3.28	3.29	0.01	1068848.60	1016887.80	4.86	6.44	1.66
56	1068222.80	1000269.20	6.36	6.39	0.03	1026236.60	1000101.20	2.55	2.56	0.01	1072725.20	1018435.00	5.06	6.78	1.81
64	1081135.20	1000354.40	7.47	7.50	0.04	1034986.40	1000129.00	3.37	3.38	0.01	1084878.80	1019539.00	6.02	7.82	1.92

Table A.64: The number of puts and takes performed during the spanning tree experiment on a Torus 3D 40 directed graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, and Idempotent LIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1008481.80	1003449.40	0.50	0.84	0.34	1000128.80	1000055.00	0.01	0.01	0.01	1000526.80	1000219.80	0.03	0.05	0.02
16	1005470.20	1001761.00	0.37	0.54	0.18	1000658.00	1000200.00	0.05	0.07	0.02	1000935.00	1000436.80	0.05	0.09	0.04
24	1009159.80	1002624.60	0.65	0.91	0.26	1000920.00	1000149.80	0.08	0.09	0.01	1003026.40	1001321.00	0.17	0.30	0.13
28	1015154.40	1003866.60	1.11	1.49	0.39	1001958.60	1000274.20	0.17	0.20	0.03	1002323.40	1000898.80	0.14	0.23	0.09
32	1017233.80	1004198.20	1.28	1.69	0.42	1001449.80	1000200.40	0.12	0.14	0.02	1002768.00	1001051.00	0.17	0.28	0.10
40	1046653.80	1011064.20	3.40	4.46	1.09	1003686.00	1000461.20	0.32	0.37	0.05	1004101.00	1001486.00	0.26	0.41	0.15
48	1068845.40	1016652.60	4.88	6.44	1.64	1005327.00	1000561.80	0.47	0.53	0.06	1007847.20	1003027.40	0.48	0.78	0.30
56	1070162.00	1013574.00	5.29	6.56	1.34	1008234.00	1000919.00	0.73	0.82	0.09	1009025.80	1002972.00	0.60	0.89	0.30
64	1105482.20	1023885.60	7.38	9.54	2.33	1006061.80	1000680.80	0.53	0.60	0.07	1010481.40	1003566.20	0.68	1.04	0.36

Table A.65: The number of puts and takes performed during the spanning tree experiment on a Torus 3D 40 directed graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1001496.00	1000859.80	0.06	0.15	0.09	1000215.20	1000154.00	0.01	0.02	0.02	1000227.20	1000153.60	0.01	0.02	0.02
16	1001202.00	1000555.00	0.06	0.12	0.06	1000642.80	1000386.60	0.03	0.06	0.04	1000830.00	1000488.00	0.03	0.08	0.05
24	1001979.40	1000864.40	0.11	0.20	0.09	1001499.00	1000901.80	0.06	0.15	0.09	1001272.00	1000702.20	0.06	0.13	0.07
28	1001512.20	1000549.40	0.10	0.15	0.05	1001234.40	1000620.80	0.06	0.12	0.06	1001945.00	1000943.60	0.10	0.19	0.09
32	1003173.60	1001363.80	0.18	0.32	0.14	1003907.80	1001866.60	0.20	0.39	0.19	1002539.80	1001121.60	0.14	0.25	0.11
40	1003953.20	1001584.80	0.24	0.39	0.16	1003717.00	1001515.40	0.22	0.37	0.15	1003422.80	1001369.80	0.20	0.34	0.14
48	1004153.40	1001451.40	0.27	0.41	0.14	1008118.80	1003261.20	0.48	0.81	0.33	1003330.80	1001519.80	0.18	0.33	0.15
56	1004527.40	1001453.00	0.31	0.45	0.15	1008654.00	1003760.60	0.49	0.86	0.37	1005827.80	1002499.80	0.33	0.58	0.25
64	1008753.40	1003547.40	0.52	0.87	0.35	1008999.20	1003677.20	0.53	0.89	0.37	1007851.60	1002852.00	0.50	0.78	0.28

Table A.66: The number of puts and takes performed during the spanning tree experiment on a Torus 3D 40 directed graph with an initial size of 256 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Directed Torus 3D 40%. Initial size of 1,000,000 items.

Algorithm	Chase-Lev					Cilk THE					Idempotent LIFO				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1004781.60	1000020.20	0.47	0.48	0.00	1003283.60	1000008.40	0.33	0.33	0.00	1006380.40	1002846.20	0.35	0.63	0.28
16	1006512.20	1000034.60	0.64	0.65	0.00	1002174.40	1000019.80	0.21	0.22	0.00	1005317.20	1002013.00	0.33	0.53	0.20
24	1014385.20	1000063.80	1.41	1.42	0.01	1005093.20	1000034.20	0.50	0.51	0.00	1010438.20	1002924.40	0.74	1.03	0.29
28	1022011.60	1000084.60	2.15	2.15	0.01	1004570.20	1000035.40	0.45	0.45	0.00	1015315.40	1004004.00	1.11	1.51	0.40
32	1021742.80	1000087.60	2.12	2.13	0.01	1010249.40	1000051.00	1.01	1.01	0.01	1021117.00	1005840.20	1.50	2.07	0.58
40	1037049.80	1000134.80	3.56	3.57	0.01	1020277.40	1000074.20	1.98	1.99	0.01	1041185.00	1010045.20	2.99	3.96	0.99
48	1051062.80	1000184.20	4.84	4.86	0.02	1022430.20	1000086.60	2.19	2.19	0.01	1057283.00	1012886.80	4.20	5.42	1.27
56	1074435.40	1000252.80	6.90	6.93	0.03	1030442.80	1000108.00	2.94	2.95	0.01	1068879.60	1018493.40	4.71	6.44	1.82
64	1056956.00	1000211.00	5.37	5.39	0.02	1029732.00	1000106.40	2.88	2.89	0.01	1084300.60	1019628.00	5.96	7.77	1.93

Table A.67: The number of puts and takes performed during the spanning tree experiment on a Torus 3D 40 directed graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, and Idempotent LIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1005001.40	1002014.20	0.30	0.50	0.20	1000122.00	1000075.60	0.00	0.01	0.01	1000217.00	1000138.80	0.01	0.02	0.01
16	1004703.20	1001704.00	0.30	0.47	0.17	1000770.20	1000267.00	0.05	0.08	0.03	1000395.20	1000321.20	0.01	0.04	0.03
24	1009500.80	1002804.60	0.66	0.94	0.28	1000797.00	1000188.00	0.06	0.08	0.02	1002385.40	1001090.60	0.13	0.24	0.11
28	1019746.40	1004967.60	1.45	1.94	0.49	1002046.40	1000323.60	0.17	0.20	0.03	1002226.00	1001122.20	0.11	0.22	0.11
32	1022973.00	1006644.20	1.60	2.25	0.66	1002253.60	1000343.20	0.19	0.22	0.03	1004358.40	1002140.60	0.22	0.43	0.21
40	1037510.20	1007760.60	2.87	3.62	0.77	1002563.40	1000389.40	0.22	0.26	0.04	1004019.20	1001589.80	0.24	0.40	0.16
48	1064999.20	1015797.00	4.62	6.10	1.56	1004420.80	1000568.80	0.38	0.44	0.06	1004806.60	1001815.60	0.30	0.48	0.18
56	1061730.80	1012102.60	4.67	5.81	1.20	1006483.40	1000712.60	0.57	0.64	0.07	1006010.40	1002515.40	0.35	0.60	0.25
64	1066844.20	1014133.80	4.94	6.27	1.39	1005115.40	1000629.20	0.45	0.51	0.06	1007441.40	1002877.60	0.45	0.74	0.29

Table A.68: The number of puts and takes performed during the spanning tree experiment on a Torus 3D 40 directed graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000210.60	1000149.60	0.01	0.02	0.01	1000323.80	1000266.00	0.01	0.03	0.03	1000296.00	1000256.40	0.00	0.03	0.03
16	1000692.80	1000428.00	0.03	0.07	0.04	1000723.00	1000457.40	0.03	0.07	0.05	1000523.20	1000335.40	0.02	0.05	0.03
24	1000685.80	1000359.00	0.03	0.07	0.04	1001289.80	1000686.20	0.06	0.13	0.07	1001071.80	1000511.80	0.06	0.11	0.05
28	1001066.80	1000540.60	0.05	0.11	0.05	1001539.40	1000735.00	0.08	0.15	0.07	1001075.00	1000473.00	0.06	0.11	0.05
32	1002309.00	1000954.80	0.14	0.23	0.10	1002734.80	1001197.00	0.15	0.27	0.12	1003033.60	1001503.40	0.15	0.30	0.15
40	1002858.20	1001243.00	0.16	0.29	0.12	1003612.00	1001454.60	0.21	0.36	0.15	1002216.20	1000841.80	0.14	0.22	0.08
48	1002879.00	1001330.40	0.15	0.29	0.13	1005406.20	1002005.60	0.34	0.54	0.20	1004107.80	1001592.60	0.25	0.41	0.16
56	1005787.80	1002161.40	0.36	0.58	0.22	1008493.40	1003380.40	0.51	0.84	0.34	1006130.40	1002085.20	0.40	0.61	0.21
64	1005020.00	1001893.40	0.31	0.50	0.19	1009661.60	1003703.00	0.59	0.96	0.37	1006580.60	1002327.00	0.42	0.65	0.23

Table A.69: The number of puts and takes performed during the spanning tree experiment on a Torus 3D 40 directed graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Undirected Torus 3D 40%. Initial size of 256 items.

Algorithm	Chase-Lev					Cilk THE					Idempotent LIFO				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1017995.80	1000015.00	1.77	1.77	0.00	1003827.00	1000010.20	0.38	0.38	0.00	1014102.20	1003530.20	1.04	1.39	0.35
16	1017650.00	1000027.40	1.73	1.73	0.00	1006828.60	1000026.60	0.68	0.68	0.00	1025272.40	1004920.80	1.98	2.46	0.49
24	1037891.20	1000048.20	3.65	3.65	0.00	1012529.80	1000042.40	1.23	1.24	0.00	1031634.60	1006082.80	2.48	3.07	0.60
28	1042682.60	1000046.20	4.09	4.09	0.00	1013805.20	1000034.00	1.36	1.36	0.00	1035589.20	1006250.20	2.83	3.44	0.62
32	1063334.80	1000060.20	5.95	5.96	0.01	1017024.40	1000060.40	1.67	1.67	0.01	1062079.60	1012646.00	4.65	5.85	1.25
40	1065346.20	1000072.40	6.13	6.13	0.01	1022461.80	1000062.20	2.19	2.20	0.01	1093480.20	1018313.60	6.87	8.55	1.80
48	1091432.20	1000092.40	8.37	8.38	0.01	1039476.20	1000079.80	3.79	3.80	0.01	1076567.60	1013001.80	5.90	7.11	1.28
56	1139723.40	1000176.20	12.24	12.26	0.02	1038957.20	1000095.40	3.74	3.75	0.01	1121967.00	1024412.20	8.69	10.87	2.38
64	1144039.40	1000169.80	12.58	12.59	0.02	1042307.80	1000097.80	4.05	4.06	0.01	1111321.60	1017669.40	8.43	10.02	1.74

Table A.70: The number of puts and takes performed during the spanning tree experiment on a Torus 3D 40 undirected graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, and Idempotent LIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1012364.60	1002952.80	0.93	1.22	0.29	1000045.00	1000037.80	0.00	0.00	0.00	1000040.80	1000030.40	0.00	0.00	0.00
16	1015285.00	1004607.00	1.05	1.51	0.46	1000103.60	1000087.00	0.00	0.01	0.01	1000111.80	1000086.80	0.00	0.01	0.01
24	1024174.00	1006047.20	1.77	2.36	0.60	1000125.60	1000098.20	0.00	0.01	0.01	1000163.60	1000122.60	0.00	0.02	0.01
28	1043091.00	1009550.40	3.22	4.13	0.95	1000159.00	1000109.40	0.00	0.02	0.01	1000191.80	1000138.60	0.01	0.02	0.01
32	1045208.20	1009458.40	3.42	4.33	0.94	1000175.60	1000112.40	0.01	0.02	0.01	1000183.00	1000123.80	0.01	0.02	0.01
40	1093916.80	1019826.00	6.77	8.59	1.94	1000204.40	1000129.60	0.01	0.02	0.01	1000251.80	1000163.20	0.01	0.03	0.02
48	1090897.60	1018033.80	6.68	8.33	1.77	1000246.00	1000137.60	0.01	0.02	0.01	1001355.40	1000927.00	0.04	0.14	0.09
56	1126565.20	1026626.60	8.87	11.23	2.59	1001315.60	1000141.60	0.12	0.13	0.01	1000574.40	1000364.40	0.02	0.06	0.04
64	1128948.20	1023499.80	9.34	11.42	2.30	1001102.40	1000172.40	0.09	0.11	0.02	1003930.20	1002669.60	0.13	0.39	0.27

Table A.71: The number of puts and takes performed during the spanning tree experiment on a Torus 3D 40 undirected graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000097.80	1000032.60	0.01	0.01	0.00	1000037.20	1000029.20	0.00	0.00	0.00	1000042.60	1000034.60	0.00	0.00	0.00
16	1000144.00	1000082.60	0.01	0.01	0.01	1000110.60	1000090.20	0.00	0.01	0.01	1000113.40	1000094.00	0.00	0.01	0.01
24	1000142.00	1000089.80	0.01	0.01	0.01	1000150.60	1000121.60	0.00	0.02	0.01	1000146.00	1000114.60	0.00	0.01	0.01
28	1000202.20	1000113.80	0.01	0.02	0.01	1000171.80	1000117.00	0.01	0.02	0.01	1000172.60	1000124.60	0.00	0.02	0.01
32	1000230.40	1000144.00	0.01	0.02	0.01	1000185.80	1000126.80	0.01	0.02	0.01	1000216.60	1000162.60	0.01	0.02	0.02
40	1000306.60	1000193.60	0.01	0.03	0.02	1000218.40	1000149.00	0.01	0.02	0.01	1000234.60	1000168.40	0.01	0.02	0.02
48	1000651.80	1000237.80	0.04	0.07	0.02	1000293.40	1000201.80	0.01	0.03	0.02	1000295.60	1000211.20	0.01	0.03	0.02
56	1000916.80	1000281.00	0.06	0.09	0.03	1002471.20	1001770.60	0.07	0.25	0.18	1000659.20	1000273.00	0.04	0.07	0.03
64	1001691.80	1000395.80	0.13	0.17	0.04	1003185.40	1002393.20	0.08	0.32	0.24	1001787.80	1000506.40	0.13	0.18	0.05

Table A.72: The number of puts and takes performed during the spanning tree experiment on a Torus 3D 40 undirected graph with an initial size of 256 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Undirected Torus 3D 40%. Initial size of 1,000,000 items.

Algorithm	Chase-Lev				Cilk THE				Idempotent LIFO				Idempotent DEQUE				Idempotent FIFO			
Operation	Puts	Takes	Difference (%)	Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)
Processes																				
1	1000000.00	1000000.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00
8	1010972.00	1000014.20	1.08	1.09	1003617.80	1000009.20	0.36	0.36	1009566.40	1002906.80	0.66	0.95	1013092.00	1003388.60	0.96	1.29	1000047.20	1000039.20	0.00	0.00
16	1009884.40	1000030.40	0.98	0.98	1004682.40	1000028.80	0.46	0.47	1007973.80	1002418.80	0.55	0.79	1011237.40	1003333.40	0.78	1.11	1000109.00	1000090.60	0.00	0.01
24	1039995.00	1000052.00	3.84	3.85	1011143.80	1000056.00	1.10	1.10	1036845.60	1010304.40	2.56	3.55	1026754.80	1006000.00	2.02	2.61	1000125.40	1000091.00	0.00	0.01
28	1044321.40	1000043.40	4.24	4.24	1014743.20	1000049.00	1.45	1.45	1033028.60	1009013.40	2.32	3.20	1036250.40	1008419.00	2.69	3.50	1000163.60	1000115.40	0.00	0.02
32	1056481.80	1000054.80	5.34	5.35	1019167.00	1000057.20	1.88	1.88	1049470.80	1012449.80	3.53	4.71	1052996.60	1012604.80	3.84	5.03	1000177.80	1000119.20	0.01	0.02
40	1069169.40	1000073.60	6.46	6.47	1021383.60	1000066.80	2.09	2.09	1054736.60	1012277.00	4.03	5.19	1062180.80	1014550.00	4.48	5.85	1000191.40	1000114.80	0.01	0.02
48	1081696.80	1000078.00	7.55	7.55	1020685.80	1000083.60	2.02	2.03	1091483.80	1020718.40	6.48	8.38	1085018.00	1017446.00	6.23	7.84	1000244.00	1000151.00	0.01	0.02
56	1115203.20	1000117.20	10.32	10.33	1033933.60	1000101.20	3.27	3.28	1109457.80	1023300.40	7.77	9.87	1127022.00	1026420.80	8.93	11.27	1001294.40	1000152.00	0.11	0.13
64	1125126.20	1000147.20	11.11	11.12	1032473.00	1000106.00	3.13	3.15	1124086.80	1031452.20	8.24	11.04	1133354.60	1025677.80	9.50	11.77	1001170.60	1000225.00	0.09	0.12

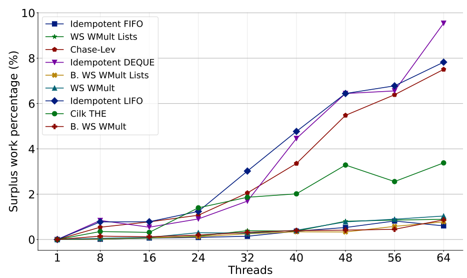
Table A.73: The number of puts and takes performed during the spanning tree experiment on a Torus 3D 40 undirected graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, Idempotent LIFO, Idempotent DEQUE, and Idempotent FIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the scheduled tasks and the total work avalaible (total of vertices).

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1122378.40	1048091.20	6.62	10.90	4.59	1000031.80	1000020.40	0.00	0.00	0.00	1000039.20	1000028.20	0.00	0.00	0.00
16	1264133.60	1092552.00	13.57	20.89	8.47	1000082.60	1000057.00	0.00	0.01	0.01	1000096.00	1000064.60	0.00	0.01	0.01
24	1331263.80	1092263.20	17.95	24.88	8.45	1000148.40	1000074.00	0.01	0.01	0.01	1000210.00	1000142.80	0.01	0.02	0.01
28	1322992.00	1078581.40	18.47	24.41	7.29	1000196.60	1000093.40	0.01	0.02	0.01	1000215.60	1000132.40	0.01	0.02	0.01
32	1376831.40	1094660.40	20.49	27.37	8.65	1000296.20	1000139.20	0.02	0.03	0.01	1000306.20	1000169.00	0.01	0.03	0.02
40	1393376.60	1091114.20	21.69	28.23	8.35	1000454.60	1000162.40	0.03	0.05	0.02	1000418.20	1000203.20	0.02	0.04	0.02
48	1302068.20	1066873.00	18.06	23.20	6.27	1000485.00	1000202.20	0.03	0.05	0.02	1000669.20	1000335.00	0.03	0.07	0.03
56	1371225.80	1068227.80	22.10	27.07	6.39	1000779.00	1000293.60	0.05	0.08	0.03	1001079.20	1000576.40	0.05	0.11	0.06
64	1408743.20	1079272.40	23.39	29.01	7.34	1000845.80	1000302.60	0.05	0.08	0.03	1001079.40	1000588.40	0.05	0.11	0.06

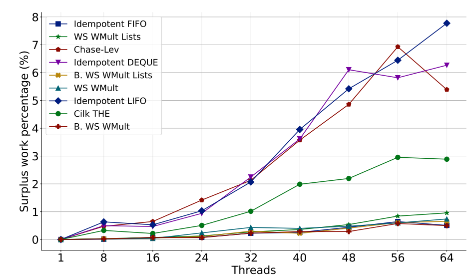
Table A.74: The number of puts and takes performed during the spanning tree experiment on a Torus 3D 40 undirected graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000076.60	1000052.00	0.00	0.01	0.01	1000044.80	1000032.80	0.00	0.00	0.00	1000131.00	1000121.40	0.00	0.01	0.01
16	1000102.80	1000075.60	0.00	0.01	0.01	1000104.80	1000075.20	0.00	0.01	0.01	1000093.40	1000067.80	0.00	0.01	0.01
24	1000166.80	1000106.40	0.01	0.02	0.01	1000172.60	1000103.20	0.01	0.02	0.01	1000242.40	1000178.40	0.01	0.02	0.02
28	1000217.60	1000138.40	0.01	0.02	0.01	1000225.40	1000126.40	0.01	0.02	0.01	1000251.00	1000179.00	0.01	0.03	0.02
32	1000388.40	1000257.40	0.01	0.04	0.03	1000311.80	1000203.20	0.01	0.03	0.02	1000368.20	1000268.60	0.01	0.04	0.03
40	1000494.40	1000288.40	0.02	0.05	0.03	1000457.60	1000237.20	0.02	0.05	0.02	1000458.00	1000279.60	0.02	0.05	0.03
48	1000700.60	1000446.80	0.03	0.07	0.04	1000655.40	1000335.20	0.03	0.07	0.03	1000841.00	1000484.40	0.04	0.08	0.05
56	1000793.40	1000432.80	0.04	0.08	0.04	1000984.60	1000607.00	0.04	0.10	0.06	1000735.40	1000403.40	0.03	0.07	0.04
64	1001009.80	1000573.80	0.04	0.10	0.06	1001190.80	1000569.20	0.06	0.12	0.06	1000992.80	1000564.00	0.04	0.10	0.06

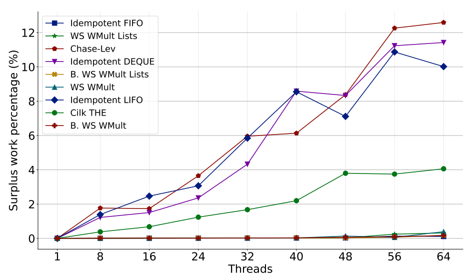
Table A.75: The number of puts and takes performed during the spanning tree experiment on a Torus 3D 40 undirected graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.



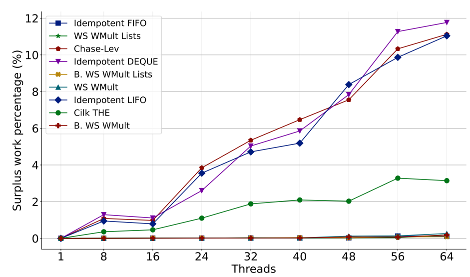
(a) Surplus work: Directed Torus 3D 40%. Initial size of 256 items



(b) Surplus work: Directed Torus 2D 60%. Initial size of 1,000,000 items

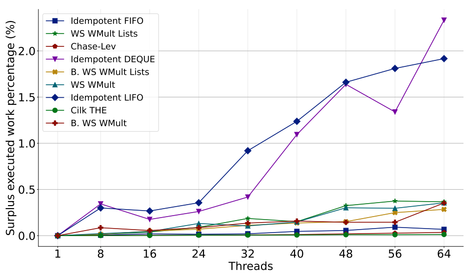


(c) Surplus work: Undirected Torus 3D 40%. Initial size of 256 items

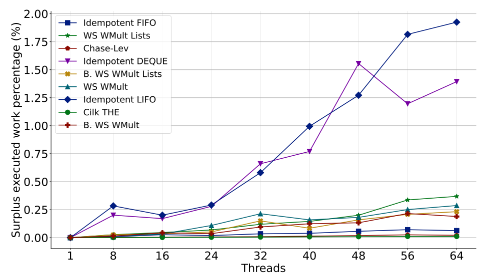


(d) Surplus work: Undirected Torus 3D 40%. Initial size of 1,000,000 items

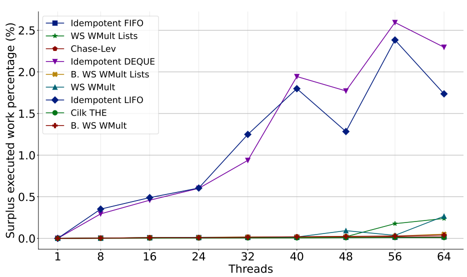
Figure A.12: Surplus work (percentage) of the experiments. Surplus work: the difference between the total number of Puts and the number of puts in sequential executions (i.e., $1,000,000$).



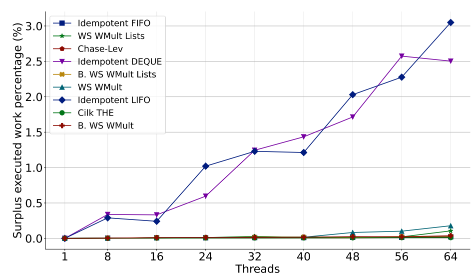
(a) Executed surplus work: Directed Torus 3D 40%. Initial size of 256 items



(b) Executed surplus work: Directed Torus 2D 60%. Initial size of 1,000,000 items



(c) Executed surplus work: Undirected Torus 3D 40%. Initial size of 256 items



(d) Executed surplus work: Undirected Torus 3D 40%. Initial size of 1,000,000 items

Figure A.13: Executed surplus work (percentage) of the experiments. Surplus work: the difference between the total number of Takes and the number of takes in sequential executions (i.e., $1,000,000$).

Directed Random. Initial size of 256 items.

Algorithm	Chase-Lev					Cilk THE					Idempotent LIFO				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1010376.60	1000004.60	1.03	1.03	0.00	1006004.60	1000007.00	0.60	0.60	0.00	1009841.20	1003227.40	0.65	0.97	0.32
16	1013402.20	1000012.60	1.32	1.32	0.00	1003361.20	1000015.00	0.33	0.33	0.00	1016342.40	1004681.00	1.15	1.61	0.47
24	1013500.80	1000022.60	1.33	1.33	0.00	1009018.20	1000024.40	0.89	0.89	0.00	1019164.80	1005003.80	1.39	1.88	0.50
28	1029575.80	1000028.20	2.87	2.87	0.00	1010834.80	1000029.60	1.07	1.07	0.00	1043575.20	1011793.80	3.05	4.18	1.17
32	1030755.60	1000028.40	2.98	2.98	0.00	1018235.40	1000036.20	1.79	1.79	0.00	1026127.60	1006960.80	1.87	2.55	0.69
40	1038680.20	1000039.40	3.72	3.72	0.00	1020048.60	1000042.40	1.96	1.97	0.00	1058921.20	1014816.40	4.17	5.56	1.46
48	1065990.80	1000049.20	6.19	6.19	0.00	1030244.00	1000048.80	2.93	2.94	0.00	1089436.60	1021604.20	6.23	8.21	2.11
56	1093318.80	1000055.80	8.53	8.54	0.01	1036149.60	1000058.60	3.48	3.49	0.01	1094043.80	1022734.20	6.52	8.60	2.22
64	1121298.40	1000065.20	10.81	10.82	0.01	1037715.40	1000064.80	3.63	3.63	0.01	1122813.60	1027289.40	8.51	10.94	2.66

Table A.76: The number of puts and takes performed during the spanning tree experiment on a Random undirected graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, and Idempotent LIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1007270.40	1002469.20	0.48	0.72	0.25	1017301.80	1003871.60	1.32	1.70	0.39	1016594.00	1007466.80	0.90	1.63	0.74
16	1010265.20	1003156.80	0.70	1.02	0.31	1020638.60	1005089.60	1.52	2.02	0.51	1023145.60	1009440.60	1.34	2.26	0.94
24	1015514.60	1003715.20	1.16	1.53	0.37	1022334.40	1004281.20	1.77	2.18	0.43	1031111.00	1015066.60	1.56	3.02	1.48
28	1016363.20	1004100.00	1.21	1.61	0.41	1027849.80	1004393.20	2.28	2.71	0.44	1064811.20	1026901.40	3.56	6.09	2.62
32	1052398.60	1013426.00	3.70	4.98	1.32	1049329.80	1009179.80	3.83	4.70	0.91	1048733.00	1021542.60	2.59	4.65	2.11
40	1049780.40	1012542.40	3.55	4.74	1.24	1068649.60	1009808.20	5.51	6.42	0.97	1087231.20	1036120.80	4.70	8.02	3.49
48	1086571.00	1022005.00	5.94	7.97	2.15	1093902.20	1011951.80	7.49	8.58	1.18	1118815.00	1051030.60	6.06	10.62	4.86
56	1107383.00	1027598.00	7.20	9.70	2.69	1099169.80	1011793.00	7.95	9.02	1.17	1146516.00	1063355.60	7.25	12.78	5.96
64	1133525.00	1030556.20	9.08	11.78	2.97	1118209.20	1015841.20	9.15	10.57	1.56	1170370.40	1065732.40	8.94	14.56	6.17

Table A.77: The number of puts and takes performed during the spanning tree experiment on a Random undirected graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1014371.60	1008044.20	0.62	1.42	0.80	1014159.80	1006525.80	0.75	1.40	0.65	1011005.20	1007182.00	0.38	1.09	0.71
16	1014463.00	1009346.20	0.50	1.43	0.93	1016470.80	1008468.60	0.79	1.62	0.84	1014230.00	1009122.00	0.50	1.40	0.90
24	1024233.20	1015111.60	0.89	2.37	1.49	1020183.80	1009737.60	1.02	1.98	0.96	1030887.20	1020534.80	1.00	3.00	2.01
28	1032665.80	1019667.00	1.26	3.16	1.93	1027046.00	1012317.80	1.43	2.63	1.22	1024738.00	1016588.00	0.80	2.41	1.63
32	1042648.80	1025679.40	1.63	4.09	2.50	1051207.40	1021926.80	2.79	4.87	2.15	1031164.60	1020148.80	1.07	3.02	1.98
40	1054389.60	1029050.00	2.40	5.16	2.82	1071398.00	1031291.00	3.74	6.66	3.03	1056291.40	1032001.60	2.30	5.33	3.10
48	1088046.40	1040259.40	4.39	8.09	3.87	1108918.40	1047254.00	5.56	9.82	4.51	1084315.60	1045354.40	3.59	7.78	4.34
56	1095902.00	1041798.00	4.94	8.75	4.01	1128059.60	1055091.80	6.47	11.35	5.22	1091565.20	1050216.40	3.79	8.39	4.78
64	1180099.80	1075089.40	8.90	15.26	6.98	1126343.60	1058193.80	6.05	11.22	5.50	1119308.00	1058057.60	5.47	10.66	5.49

Table A.78: The number of puts and takes performed during the spanning tree experiment on a Random undirected graph with an initial size of 256 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Directed Random. Initial size of 1,000,000 items.

Algorithm	Chase-Lev					Cilk THE					Idempotent LIFO				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000018.20	1000004.80	0.00	0.00	0.00	1000018.40	1000001.80	0.00	0.00	0.00	1000018.80	1000006.80	0.00	0.00	0.00
16	1000039.40	1000009.20	0.00	0.00	0.00	1000041.20	1000013.20	0.00	0.00	0.00	1000038.80	1000015.60	0.00	0.00	0.00
24	1000510.60	1000016.40	0.05	0.05	0.00	1000261.60	1000013.80	0.02	0.03	0.00	1000133.40	1000033.00	0.01	0.01	0.00
28	1004930.80	1000016.20	0.49	0.49	0.00	1003804.80	1000021.60	0.38	0.38	0.00	1004964.60	1000489.80	0.45	0.49	0.05
32	1005807.00	1000016.20	0.58	0.58	0.00	1008753.80	1000022.40	0.87	0.87	0.00	1005759.20	1000494.80	0.52	0.57	0.05
40	1021093.80	1000029.80	2.06	2.07	0.00	1018513.20	1000031.80	1.81	1.82	0.00	1016913.20	1001447.60	1.52	1.66	0.14
48	1015945.00	1000030.40	1.57	1.57	0.00	1043088.80	1000030.00	4.13	4.13	0.00	1025256.60	1001997.20	2.27	2.46	0.20
56	1060381.80	1000043.20	5.69	5.69	0.00	1041672.40	1000043.20	4.00	4.00	0.00	1064961.60	1005538.20	5.58	6.10	0.55
64	1088914.60	1000042.40	8.16	8.17	0.00	1049740.00	1000050.40	4.73	4.74	0.01	1074591.40	1005933.40	6.39	6.94	0.59

Table A.79: The number of puts and takes performed during the spanning tree experiment on a Random undirected graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, and Idempotent LIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000020.20	1000007.80	0.00	0.00	0.00	1000016.60	1000004.00	0.00	0.00	0.00	1000239.60	1000098.80	0.01	0.02	0.01
16	1000046.60	1000019.40	0.00	0.00	0.00	1000027.40	1000008.40	0.00	0.00	0.00	1000847.40	1000314.20	0.05	0.08	0.03
24	1000326.40	1000030.80	0.03	0.03	0.00	1001083.00	1000074.80	0.10	0.11	0.01	1000951.00	1000240.40	0.07	0.10	0.02
28	1004643.20	1000264.80	0.44	0.46	0.03	1004168.60	1000301.00	0.39	0.42	0.03	1005449.60	1001284.20	0.41	0.54	0.13
32	1006258.40	1000273.60	0.59	0.62	0.03	1004946.60	1000354.20	0.46	0.49	0.04	1015911.60	1003828.00	1.19	1.57	0.38
40	1005879.80	1000392.60	0.55	0.58	0.04	1012025.00	1000774.20	1.11	1.19	0.08	1043118.00	1009668.00	3.21	4.13	0.96
48	1035772.40	1003179.60	3.15	3.45	0.32	1054191.20	1004105.20	4.75	5.14	0.41	1022970.40	1005817.80	1.68	2.25	0.58
56	1072659.20	1005832.60	6.23	6.77	0.58	1056553.60	1003397.80	5.03	5.35	0.34	1082621.80	1019145.80	5.86	7.63	1.88
64	1065156.80	1004712.00	5.67	6.12	0.47	1119496.00	1008768.80	9.89	10.67	0.87	1095691.60	1025332.60	6.42	8.73	2.47

Table A.80: The number of puts and takes performed during the spanning tree experiment on a Random undirected graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000013.60	1000006.00	0.00	0.00	0.00	1000014.00	1000007.20	0.00	0.00	0.00	1000013.40	1000006.40	0.00	0.00	0.00
16	1000022.60	1000015.60	0.00	0.00	0.00	1000026.20	1000012.00	0.00	0.00	0.00	1000026.00	1000014.80	0.00	0.00	0.00
24	1000244.20	1000152.20	0.01	0.02	0.02	1000830.40	1000105.00	0.07	0.08	0.01	1000042.40	1000020.80	0.00	0.00	0.00
28	1000964.20	1000503.40	0.05	0.10	0.05	1003826.40	1000623.40	0.32	0.38	0.06	1003124.00	1001179.60	0.19	0.31	0.12
32	1005632.20	1002692.80	0.29	0.56	0.27	1003712.40	1000352.40	0.33	0.37	0.04	1001447.40	1000502.40	0.09	0.14	0.05
40	1018807.80	1007936.60	1.07	1.85	0.79	1017450.80	1002580.40	1.46	1.72	0.26	1018311.00	1006735.00	1.14	1.80	0.67
48	1053572.40	1018307.40	3.35	5.08	1.80	1037995.20	1006740.60	3.01	3.66	0.67	1033461.40	1009894.40	2.28	3.24	0.98
56	1035705.00	1012473.60	2.24	3.45	1.23	1077590.60	1013753.00	5.92	7.20	1.36	1039675.60	1011814.60	2.68	3.82	1.17
64	1084255.60	1027541.80	5.23	7.77	2.68	1076885.20	1013112.00	5.92	7.14	1.29	1076659.60	1022538.00	5.03	7.12	2.20

Table A.81: The number of puts and takes performed during the spanning tree experiment on a Random undirected graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Undirected Random. Initial size of 256 items.

Algorithm	Chase-Lev					Cilk THE					Idempotent LIFO				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000018.80	1000006.00	0.00	0.00	0.00	1000017.40	1000004.20	0.00	0.00	0.00	1000020.60	1000009.20	0.00	0.00	0.00
16	1000476.00	1000013.00	0.05	0.05	0.00	1000260.20	1000010.20	0.02	0.03	0.00	1000037.20	1000017.60	0.00	0.00	0.00
24	1000366.20	1000013.80	0.04	0.04	0.00	1001121.80	1000011.40	0.11	0.11	0.00	1003924.00	1000330.40	0.36	0.39	0.03
28	1013758.40	1000017.00	1.36	1.36	0.00	1001877.80	1000011.60	0.19	0.19	0.00	1002989.00	1000233.80	0.27	0.30	0.02
32	1007197.20	1000018.60	0.71	0.71	0.00	1005250.00	1000021.40	0.52	0.52	0.00	1002835.00	1000186.60	0.26	0.28	0.02
40	1020710.20	1000030.40	2.03	2.03	0.00	1020256.80	1000026.40	1.98	1.99	0.00	1019875.00	1001324.00	1.82	1.95	0.13
48	1042370.60	1000027.40	4.06	4.06	0.00	1046451.00	1000038.00	4.44	4.44	0.00	1069346.40	1005433.40	5.98	6.48	0.54
56	1116503.20	1000043.80	10.43	10.43	0.00	1053478.00	1000043.80	5.07	5.08	0.00	1080183.40	1005619.40	6.90	7.42	0.56
64	1113287.80	1000048.40	10.17	10.18	0.00	1055949.80	1000039.80	5.29	5.30	0.00	1154413.60	1012866.80	12.26	13.38	1.27

Table A.82: The number of puts and takes performed during the spanning tree experiment on a Random undirected graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, and Idempotent LIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000023.00	1000010.60	0.00	0.00	0.00	1000015.40	1000006.80	0.00	0.00	0.00	1000015.80	1000010.00	0.00	0.00	0.00
16	1000046.40	1000015.80	0.00	0.00	0.00	1000027.00	1000011.40	0.00	0.00	0.00	1000027.40	1000012.80	0.00	0.00	0.00
24	1000339.80	1000033.80	0.03	0.03	0.00	1000596.00	1000040.20	0.06	0.06	0.00	1002022.60	1000278.20	0.17	0.20	0.03
28	1000315.00	1000030.80	0.03	0.03	0.00	1003849.00	1000154.60	0.37	0.38	0.02	1010325.00	1001209.60	0.90	1.02	0.12
32	1014743.60	1001100.40	1.34	1.45	0.11	1015855.80	1000799.00	1.48	1.56	0.08	1028760.80	1003780.20	2.43	2.80	0.38
40	1027781.00	1002153.00	2.49	2.70	0.21	1036343.40	1001090.60	3.40	3.51	0.11	1029849.60	1004925.40	2.42	2.90	0.49
48	1052371.40	1004724.00	4.53	4.98	0.47	1072843.80	1002135.00	6.59	6.79	0.21	1052163.80	1007264.60	4.27	4.96	0.72
56	1101924.60	1009423.20	8.39	9.25	0.93	1089803.00	1002223.00	8.04	8.24	0.22	1167712.00	1043092.40	10.67	14.36	4.13
64	1125583.40	1011713.80	10.12	11.16	1.16	1151217.60	1004626.40	12.73	13.14	0.46	1161370.00	1035727.00	10.82	13.89	3.45

Table A.83: The number of puts and takes performed during the spanning tree experiment on a Random undirected graph with an initial size of 256 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000012.00	1000006.40	0.00	0.00	0.00	1000013.60	1000007.60	0.00	0.00	0.00	1000017.60	1000009.80	0.00	0.00	0.00
16	1000029.60	1000016.20	0.00	0.00	0.00	1000027.00	1000015.20	0.00	0.00	0.00	1000028.60	1000015.20	0.00	0.00	0.00
24	1000135.20	1000059.20	0.01	0.01	0.01	1000616.60	1000096.80	0.05	0.06	0.01	1000302.20	1000142.80	0.02	0.03	0.01
28	1003249.60	1001104.00	0.21	0.32	0.11	1006872.80	1001182.20	0.57	0.68	0.12	1003740.80	1001998.80	0.17	0.37	0.20
32	1005261.20	1001713.20	0.35	0.52	0.17	1011898.00	1001469.80	1.03	1.18	0.15	1009561.00	1003831.60	0.57	0.95	0.38
40	1022896.60	1006463.60	1.61	2.24	0.64	1026421.20	1004840.00	2.10	2.57	0.48	1024850.20	1007092.00	1.73	2.42	0.70
48	1049618.40	1012988.20	3.49	4.73	1.28	1053696.80	1008819.20	4.26	5.10	0.87	1032677.00	1009754.60	2.22	3.16	0.97
56	1112410.40	1027408.40	7.64	10.11	2.67	1130562.40	1032846.60	8.64	11.55	3.18	1075503.40	1020879.40	5.08	7.02	2.05
64	1151473.80	1040264.20	9.66	13.15	3.87	1172624.40	1048693.60	10.57	14.72	4.64	1109762.60	1031496.40	7.05	9.89	3.05

Table A.84: The number of puts and takes performed during the spanning tree experiment on a Random undirected graph with an initial size of 256 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Undirected Random. Initial size of 1,000,000 items.

Algorithm	Chase-Lev				Cilk THE				Idempotent LIFO				Idempotent DEQUE				Idempotent FIFO			
Operation	Puts	Takes	Difference (%)	Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)
Processes																				
1	1000000.00	1000000.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00
8	1000018.20	1000004.80	0.00	0.00	1000018.40	1000001.80	0.00	0.00	1000018.80	1000006.80	0.00	0.00	1000020.20	1000007.80	0.00	0.00	1000016.60	1000004.00	0.00	0.00
16	1000039.40	1000009.20	0.00	0.00	1000041.20	1000013.20	0.00	0.00	1000038.80	1000015.60	0.00	0.00	1000046.60	1000019.40	0.00	0.00	1000027.40	1000008.40	0.00	0.00
24	1000510.60	1000016.40	0.05	0.05	1000261.60	1000013.80	0.02	0.03	1000133.40	1000033.00	0.01	0.01	1000326.40	1000030.80	0.03	0.03	1001083.00	1000074.80	0.10	0.11
28	1004930.80	1000016.20	0.49	0.49	1003804.80	1000021.60	0.38	0.38	1004964.60	1000489.80	0.45	0.49	1004643.20	1000264.80	0.44	0.46	1004168.60	1000301.00	0.39	0.42
32	1005807.00	1000016.20	0.58	0.58	1008753.80	1000022.40	0.87	0.87	1005759.20	1000494.80	0.52	0.57	1006258.40	1000273.60	0.59	0.62	1004946.60	1000354.20	0.46	0.49
40	1021093.80	1000029.80	2.06	2.07	1018513.20	1000031.80	1.81	1.82	1016913.20	1001447.60	1.52	1.66	1005879.80	1000392.60	0.55	0.58	1012025.00	1000774.20	1.11	1.19
48	1015945.00	1000030.40	1.57	1.57	1043088.80	1000030.00	4.13	4.13	1025256.60	1001997.20	2.27	2.46	1035772.40	1003179.60	3.15	3.45	1054191.20	1004105.20	4.75	5.14
56	1060381.80	1000043.20	5.69	5.69	1041672.40	1000043.20	4.00	4.00	1064961.60	1005538.20	5.58	6.10	1072659.20	1005832.60	6.23	6.77	1056553.60	1003397.80	5.03	5.35
64	1088914.60	1000042.40	8.16	8.17	1049740.00	1000050.40	4.73	4.74	1074591.40	1005933.40	6.39	6.94	1065156.80	1004712.00	5.67	6.12	1119496.00	1008768.80	9.89	10.67

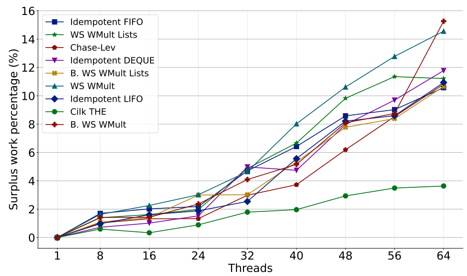
Table A.85: The number of puts and takes performed during the spanning tree experiment on a Random undirected graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Chase-Lev, Cilk THE, Idempotent LIFO, Idempotent DEQUE, and Idempotent FIFO. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the scheduled tasks and the total work avalaible (total of vertices).

Algorithm	Idempotent DEQUE					Idempotent FIFO					WS WMult				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000020.20	1000007.80	0.00	0.00	0.00	1000016.60	1000004.00	0.00	0.00	0.00	1000239.60	1000098.80	0.01	0.02	0.01
16	1000046.60	1000019.40	0.00	0.00	0.00	1000027.40	1000008.40	0.00	0.00	0.00	1000847.40	1000314.20	0.05	0.08	0.03
24	1000326.40	1000030.80	0.03	0.03	0.00	1001083.00	1000074.80	0.10	0.11	0.01	1000951.00	1000240.40	0.07	0.10	0.02
28	1004643.20	1000264.80	0.44	0.46	0.03	1004168.60	1000301.00	0.39	0.42	0.03	1005449.60	1001284.20	0.41	0.54	0.13
32	1006258.40	1000273.60	0.59	0.62	0.03	1004946.60	1000354.20	0.46	0.49	0.04	1015911.60	1003828.00	1.19	1.57	0.38
40	1005879.80	1000392.60	0.55	0.58	0.04	1012025.00	1000774.20	1.11	1.19	0.08	1043118.00	1009668.00	3.21	4.13	0.96
48	1035772.40	1003179.60	3.15	3.45	0.32	1054191.20	1004105.20	4.75	5.14	0.41	1022970.40	1005817.80	1.68	2.25	0.58
56	1072659.20	1005832.60	6.23	6.77	0.58	1056553.60	1003397.80	5.03	5.35	0.34	1082621.80	1019145.80	5.86	7.63	1.88
64	1065156.80	1004712.00	5.67	6.12	0.47	1119496.00	1008768.80	9.89	10.67	0.87	1095691.60	1025332.60	6.42	8.73	2.47

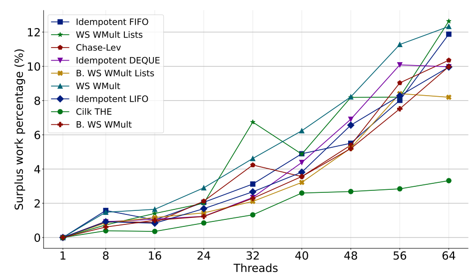
Table A.86: The number of puts and takes performed during the spanning tree experiment on a Random undirected graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: Idempotent DEQUE, Idempotent FIFO, and WS WMult. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.

Algorithm	B. WS WMult					WS WMult Lists					B. WS WMult Lists				
Operation	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)	Puts	Takes	Difference (%)	Surplus (%)	Executed Surplus (%)
Processes															
1	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00	1000000.00	1000000.00	0.00	0.00	0.00
8	1000013.60	1000006.00	0.00	0.00	0.00	1000014.00	1000007.20	0.00	0.00	0.00	1000013.40	1000006.40	0.00	0.00	0.00
16	1000022.60	1000015.60	0.00	0.00	0.00	1000026.20	1000012.00	0.00	0.00	0.00	1000026.00	1000014.80	0.00	0.00	0.00
24	1000244.20	1000152.20	0.01	0.02	0.02	1000830.40	1000105.00	0.07	0.08	0.01	1000042.40	1000020.80	0.00	0.00	0.00
28	1000964.20	1000503.40	0.05	0.10	0.05	1003826.40	1000623.40	0.32	0.38	0.06	1003124.00	1001179.60	0.19	0.31	0.12
32	1005632.20	1002692.80	0.29	0.56	0.27	1003712.40	1000352.40	0.33	0.37	0.04	1001447.40	1000502.40	0.09	0.14	0.05
40	1018807.80	1007936.60	1.07	1.85	0.79	1017450.80	1002580.40	1.46	1.72	0.26	1018311.00	1006735.00	1.14	1.80	0.67
48	1053572.40	1018307.40	3.35	5.08	1.80	1037995.20	1006740.60	3.01	3.66	0.67	1033461.40	1009894.40	2.28	3.24	0.98
56	1035705.00	1012473.60	2.24	3.45	1.23	1077590.60	1013753.00	5.92	7.20	1.36	1039675.60	1011814.60	2.68	3.82	1.17
64	1084255.60	1027541.80	5.23	7.77	2.68	1076885.20	1013112.00	5.92	7.14	1.29	1076659.60	1022538.00	5.03	7.12	2.20

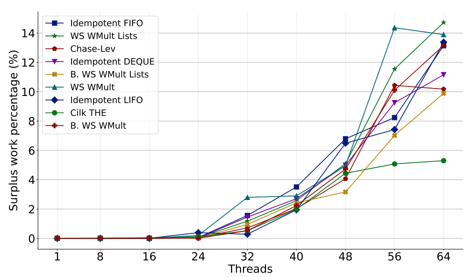
Table A.87: The number of puts and takes performed during the spanning tree experiment on a Random undirected graph with an initial size of 1000000 items is provided. The table presents data on the following algorithms: B. WS WMult, WS WMult Lists, and B. WS WMult Lists. Furthermore, we present the percentage difference between the number of puts and takes for each available thread, relative to the total number of puts. Finally, also we show the "surplus" work, which is the difference of the total number of Puts (Work to be scheduled) and the total number of Puts in sequential executions (i.e., 1,000,000), and the "executed surplus work", which is the difference between the total number of Takes (actual work executed) and the total of Takes in sequential executions.



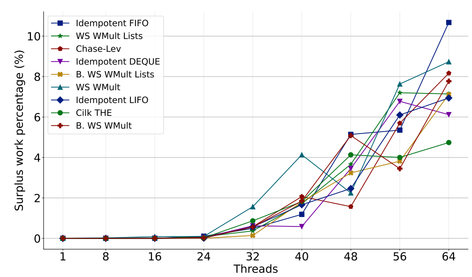
(a) Surplus work: Directed Random Graph. Initial size of 256 items



(b) Surplus work: Directed Random Graph. Initial size of 1,000,000 items

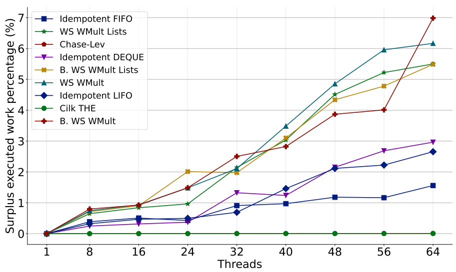


(c) Surplus work: Undirected Random Graph. Initial size of 256 items

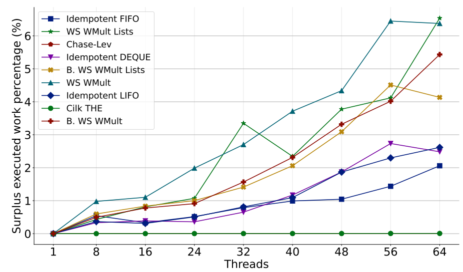


(d) Surplus work: Undirected Random Graph. Initial size of 1,000,000 items

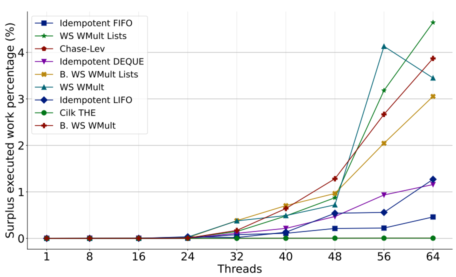
Figure A.14: Surplus work (percentage) of the experiments. Surplus work: the difference between the total number of Puts and the number of puts in sequential executions (i.e., $1,000,000$).



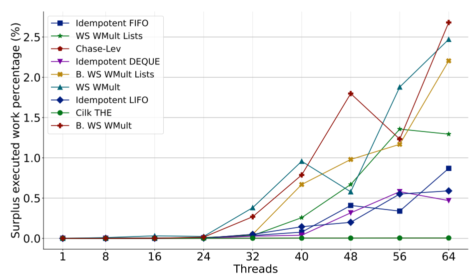
(a) Executed surplus work: Directed Random Graph. Initial size of 256 items



(b) Executed surplus work: Directed Random Graph. Initial size of 1,000,000 items



(c) Executed surplus work: Undirected Random Graph. Initial size of 256 items

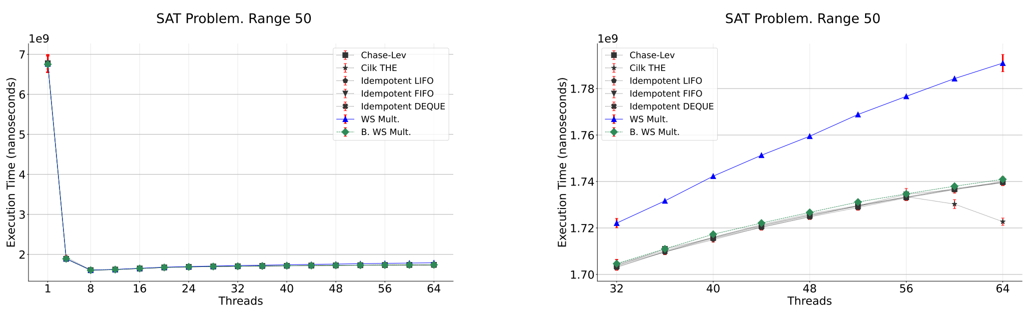


(d) Executed surplus work: Undirected Random Graph. Initial size of 1,000,000 items

Figure A.15: Executed surplus work (percentage) of the experiments. Surplus work: the difference between the total number of Takes and the number of takes in sequential executions (i.e., $1,000,000$).

A.3 Results of SAT experiment

This section presents the measurements from the parallel SAT experiment. Shows the results for the different ranges with which the parallel job was tested. Measurements were made using rigorous statistical methodology. This evaluation was performed for all work-stealing algorithms. Additionally, the percentage of repeated work is shown as the number of takes plus steals made. This is easily measurable because there is only one work-stealing structure with a single producer and multiple consumers. The owner of the structure calls the take method every time it is going to process a task, and the other workers call the steal method. Therefore, the number of puts is always fixed, while the sum of takes and puts is at least the total amount of work that was inserted via the put method.



(a) Range assigment size 50.

(b) Ranges assignment size 50. Zoom into the number of processes 32 to 64.

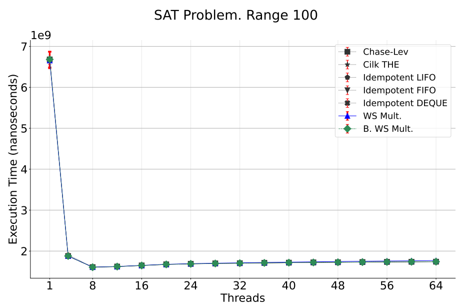
Figure A.16: Mean times of the Parallel SAT benchmark for range assignment 50.

	1	4	8	12	16	20
Chase-Lev	6771106493.51	1889235456.63	1604639874.37	1619823677.67	1646573644.81	1673070572.69
Cilk THE	6802197168.26	1933130275.93	1610054131.24	1625700961.91	1652168564.44	1678643423.36
Idempotent LIFO	6765613525.70	1889322149.43	1604852066.19	1620383537.76	1646698759.04	1672839289.99
Idempotent FIFO	6777176155.51	1889535666.67	1606530877.24	1619900006.93	1646961802.84	1672795260.39
Idempotent DEQUE	6754222054.47	1887515675.74	1607441806.56	1620652684.31	1646659093.64	1672990966.89
WS Mult.	6759188962.90	1895287204.46	1608931490.97	1626576516.29	1653947113.14	1681567490.83
B. WS Mult.	6749674804.90	1891140517.30	1606570003.57	1621611512.86	1648643759.01	1674782575.49

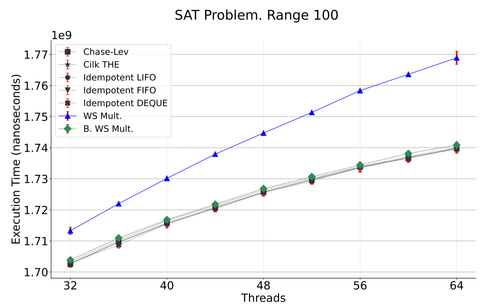
	24	28	32	36	40	44
Chase-Lev	1687294862.61	1696749620.63	1703680039.41	1710985761.03	1715687626.59	1720808375.81
Cilk THE	1693219776.67	1700315677.49	1704476100.71	1710217427.63	1715966400.77	1721428221.41
Idempotent LIFO	1687124110.16	1696275853.74	1704046387.47	1709655019.76	1716006135.71	1721069733.01
Idempotent FIFO	1686690773.01	1696541623.33	1703370023.54	1709696507.80	1714958659.79	1720376563.60
Idempotent DEQUE	1687002947.97	1696362167.43	1703012345.57	1709754367.74	1715467579.27	1720232884.76
WS Mult.	1697659943.97	1710112808.26	1722093689.80	1731664198.24	1742294997.76	1751311084.83
B. WS Mult.	1688764361.97	1697863713.51	1704635152.71	1710999825.43	1717298452.77	1722122072.69

	48	52	56	60	64
Chase-Lev	1725459846.29	1729465873.47	1733202358.64	1736669234.59	1739748595.84
Cilk THE	1726061148.66	1729668882.90	1733466762.36	1730280495.31	1722706049.09
Idempotent LIFO	1725560718.04	1729361763.40	1733293132.63	1736993631.74	1739532414.24
Idempotent FIFO	1724996570.13	1729562393.91	1734432732.00	1736538220.60	1740059007.49
Idempotent DEQUE	1724737751.76	1728808479.63	1732979869.21	1736495267.69	1739516302.51
WS Mult.	1759492255.69	1768839583.80	1776646590.17	1784288295.49	1790959219.97
B. WS Mult.	1726707115.19	1731157469.19	1734667658.91	1737970629.40	1740941783.40

Table A.88: Resulting mean times for the SAT benchmark. These are the results for tasks with 50 assignments.



(a) Range assignment size 100.

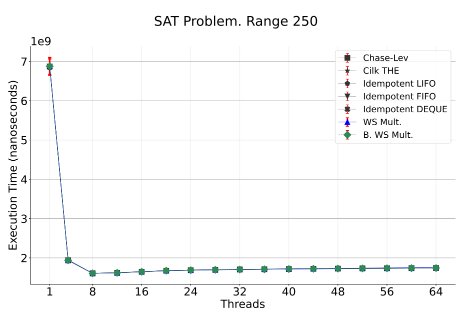


(b) Ranges assignment size 100. Zoom into the number of processes 32 to 64.

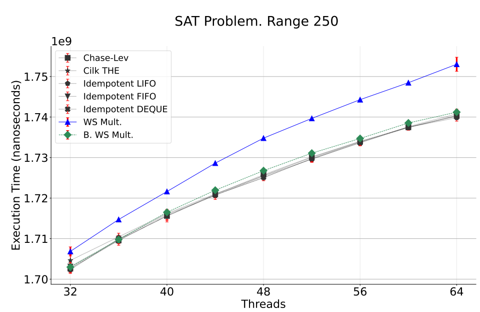
Figure A.17: Mean times of the Parallel SAT benchmark for range assignment 100.

	1	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64
Chase-Lev	6681894196.79	1878532981.09	1607198238.76	1619373794.26	1646161727.76	1672418279.90	1686050945.86	1695660527.96	1702542263.80	1709612160.87	1715821031.64	1720880142.11	1725753779.47	1729752224.94	1733896976.53	1736565443.89	1739731314.63
Cilk THE	6713144570.01	1907821954.13	1608959290.14	1625924986.24	1650636064.94	1677254077.36	1691318036.30	1700170199.61	1703303768.40	1710257851.40	1716483961.57	1721373546.30	1726313143.23	1730158320.93	1733852571.80	1737180435.81	1740086113.23
Idempotent LIFO	6683734211.79	1879640341.39	1605755065.69	1619557166.63	1646749050.20	1672605855.71	1686356001.69	1695418620.26	1702768108.47	1709845572.26	1715641938.47	1720572054.74	1725680760.97	1729833982.57	1733560046.70	1736864400.26	1739675502.83
Idempotent FIFO	6663575563.97	1874581080.89	1605273186.81	1619821900.27	1646469758.39	1672356605.59	1686321505.14	1696049308.93	1702713437.63	1709494167.00	1715453752.30	1720722648.44	1725673225.36	1729568516.90	1733430752.01	1736873945.57	1739588215.41
Idempotent DEQUE	6677808307.77	1878565281.23	1605869783.66	1619938274.40	1646046886.26	1672922371.53	1686246470.23	1696015949.89	1702965954.56	1708681178.37	1715512242.60	1720354719.36	1725544754.73	1729265270.43	1733664563.41	1736726526.67	1739877762.29
WS Mult.	6655863065.41	1881147196.97	1607518117.34	1623255189.83	1650666167.06	1677309219.59	1692459616.23	1703860952.49	1713270314.14	1721989747.53	1730125346.56	1737926826.01	1744710003.79	1751353566.93	1758374969.06	1763599554.51	1768917861.41
B. WS Mult.	6681927524.90	1880138570.61	1605398337.66	1620824357.39	1647505376.86	1673775816.50	1687458197.30	1696578580.30	1703858003.49	1710946785.63	1716817210.01	1721798460.24	1726811390.09	1730674356.66	1734405481.01	1738132915.11	1740855354.73

Table A.89: Resulting mean times for the SAT benchmark. These are the results for tasks with 100 assignments.



(a) Range assignment size 250.

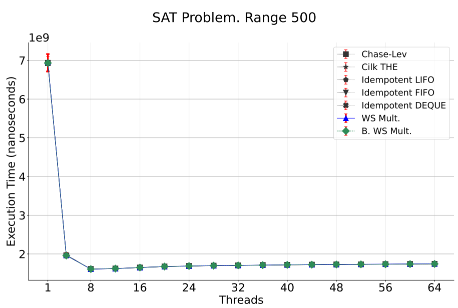


(b) Ranges assignment size 250. Zoom into the number of processes 32 to 64.

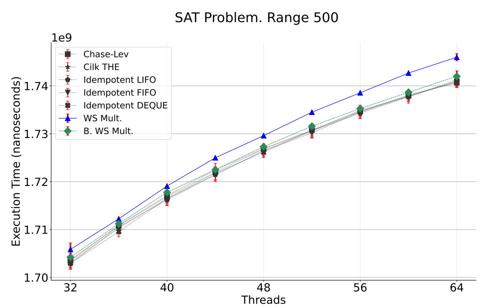
Figure A.18: Mean times of the Parallel SAT benchmark for range assignment 250.

	1	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64
Chase-Lev	6873445079.24	1937653863.57	1606972292.80	1619205417.23	1645883862.89	1672565765.04	1686108943.24	1695520730.29	1702611001.40	1709559896.36	1715602240.96	1720963682.66	1725769561.09	1730172454.43	1734027355.60	1737472991.40	1740254440.54
Cilk THE	6875028459.24	1959745947.46	1610486322.41	1625744051.71	1651538819.10	1677231463.31	1690639363.76	1700513211.83	1704628510.50	1710445482.24	1716174303.07	1721021251.53	1725477089.57	1729576812.63	1733729589.64	1737404823.44	1740422259.99
Idempotent LIFO	6878090215.96	1938415675.89	1605878572.03	1619793415.77	1646596740.19	1672372114.36	1686318373.71	1695300861.13	1702312300.86	1709987913.47	1715945378.74	1720697647.57	1725524531.09	1730048238.43	1733892540.46	1737481118.31	1739844707.90
Idempotent FIFO	6866640892.36	1937023457.49	1606928729.37	1619938301.06	1646275164.26	1672816511.77	1685763283.73	1695156940.99	1703113213.94	1709763706.10	1715538749.19	1720783151.70	1724994339.86	1729769693.43	1733913361.59	1737526489.19	1740200339.11
Idempotent DEQUE	6852495208.80	1939055893.97	1607330597.00	1620249924.66	1646729687.16	1672855387.61	1686035950.37	1695371945.71	1702707149.84	1709651887.43	1715675990.53	1720745589.03	1725140243.09	1729696482.51	1733620328.11	1737650032.50	1740754516.83
WS Mult.	6875483244.49	1938377399.24	1605994613.47	1621346740.47	1648266404.39	1674968318.34	1689142418.30	1698799487.89	1706850697.47	1714728342.01	1721622302.51	1728628212.80	1734798784.49	1739671471.44	1744291932.59	1748478873.30	1753029265.56
B. WS Mult.	6877635301.10	1937149383.50	1606471154.43	1620287894.74	1646879481.73	1673840790.31	1687088965.14	1696199528.83	1702975030.71	1709772465.10	1716497656.74	1721945035.64	1726768953.61	1731093324.67	1734714056.29	1738543663.74	1741215004.37

Table A.90: Resulting mean times for the SAT benchmark. These are the results for tasks with 250 assignments.



(a) Range assignment size 500



(b) Ranges assignment size 500. Zoom into the number of processes 32 to 64.

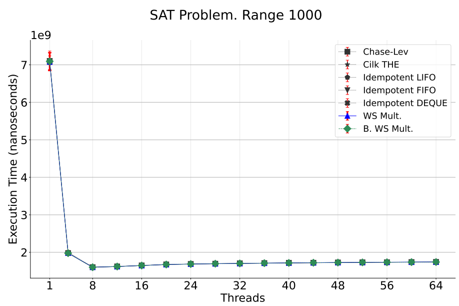
Figure A.19: Mean times of the Parallel SAT benchmark for range assignment 500.

	1	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64
Chase-Lev	6929564176.53	1962356250.31	1608736481.26	1621003229.70	1646975570.73	1673756820.14	1687049713.60	1696213900.90	1703519780.76	1710788786.86	1716857220.11	1721959767.91	1726835855.14	1730820623.54	1734719331.21	1737961013.69	1740716874.67
Cilk THE	6912324334.66	1983392407.26	1609250455.46	1626045321.24	1652635422.01	1677718770.91	1690872478.11	1699507757.77	1703629868.87	1710899930.43	1717087114.60	1722474996.91	1726891605.96	1730633955.79	1734895478.30	1737824620.39	1741031516.86
Idempotent LIFO	6938817046.87	1959947659.90	1608214583.84	1621135063.17	1647173738.36	1673759542.47	1686974442.50	1696001392.71	1702875049.84	1710578061.11	1716184071.44	1721464503.40	1726484625.76	1730612104.41	1734288717.36	1737732367.14	1741274700.61
Idempotent FIFO	6937608406.37	1961545509.50	1608892969.94	1621345265.97	1647598444.21	1673953863.27	1687037866.73	1695866624.00	1703210949.86	1710302436.50	1716479260.90	1721441409.83	1726447990.73	1730166027.84	1734605294.54	1737752347.86	1740908508.87
Idempotent DEQUE	6937359409.17	1963321353.93	1609532412.24	1621723264.37	1647570981.86	1673659653.29	1687096931.34	1696037366.76	1702930264.50	1709570495.61	1716455285.23	1721803123.63	1726156240.33	1730675206.10	1734512775.30	1738113467.40	1740642007.23
WS Mult.	6953642294.01	1964653727.61	1606483278.41	1622011677.60	1648963492.74	1674968452.13	1688713696.41	1697905891.57	1705870308.14	1712248981.64	1719082468.23	1724993413.71	1729608788.74	1734514136.53	1738559813.89	1742685289.41	1745993579.69
B. WS Mult.	6931443097.93	1958245557.29	1609304687.60	1621863245.30	1648301547.71	1674573921.93	1687755581.83	1696767401.76	1704237470.97	1711207442.14	1717762871.64	1722499639.66	1727297672.37	1731586865.09	1735262792.63	1738669147.23	1741995072.59

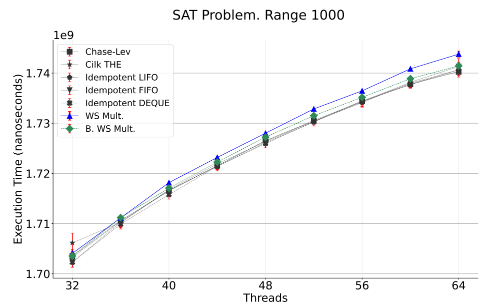
Table A.91: Resulting mean times for the SAT benchmark. These are the results for tasks with 500 assignments.

	1	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64
Chase-Lev	7101992087.37	1979707451.03	1603211263.53	1620289634.73	1647023202.16	1673572231.80	1686733853.03	1695924649.54	1703527137.94	1710596493.31	1716588173.61	1721543922.10	1726540782.64	1730570729.13	1734372572.73	1737754017.34	1740262548.61
Cilk THE	7055770308.69	2002027974.31	1608170999.31	1625709276.99	1653000016.10	1677573034.31	1690493929.46	1699667909.93	1706170527.49	1710216326.49	1716748556.57	1721918943.34	1726306039.14	1730437249.33	1734115871.04	1738019415.39	1740646870.79
Idempotent LIFO	7082494441.09	1977223018.27	1602157958.69	1620240059.81	1647322201.07	1673871567.47	1686803374.93	1695808593.16	1703267437.07	1710497021.80	1716501637.69	1721518076.11	1726494522.09	1730428162.84	1734349021.71	1737682760.06	1740655073.59
Idempotent FIFO	7097390663.23	1975364539.59	1601702015.56	1620517777.10	1646972275.27	1673468734.17	1686646937.86	1695440109.91	1702314201.47	1710286992.43	1716648944.34	1721414521.77	1726138531.57	1730267460.99	1734238133.24	1738207421.77	1741318541.17
Idempotent DEQUE	7081257196.74	1979273899.74	1602560385.40	1620681555.97	1647436979.11	1673836696.39	1686589863.60	1695591262.99	1702299850.11	1709902463.20	1715781418.39	1721410731.40	1725957918.24	1730296964.81	1734484793.96	1737931770.73	1740310999.67
WS Mult.	7077855329.04	1979473352.23	1602619356.23	1621144550.61	1647485600.06	1674717420.54	1687502415.11	1696528650.60	1704057410.81	1711138151.66	1718161857.90	1723183779.64	1728024080.63	1732856909.03	1736463606.03	1740855797.61	1743786633.84
B. WS Mult.	7093155803.06	1977400243.77	1602281797.34	1620955203.83	1647594863.51	1674349645.91	1687481618.97	1696851535.23	1703524216.44	1711186223.33	1717068924.64	1722277463.64	1727228341.41	1731510251.93	1735178870.09	1738877526.50	1741442959.59

Table A.92: Resulting mean times for the SAT benchmark. These are the results for tasks with 1000 assignments.

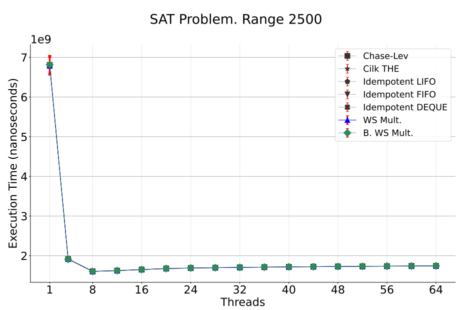


(a) Range assignment size 1000.

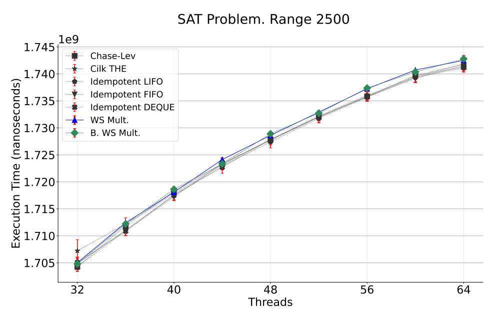


(b) Ranges assignment size 1000. Zoom into the number of processes 32 to 64.

Figure A.20: Mean times of the Parallel SAT benchmark for range assignment 1000.



(a) Range assignment size 2500.

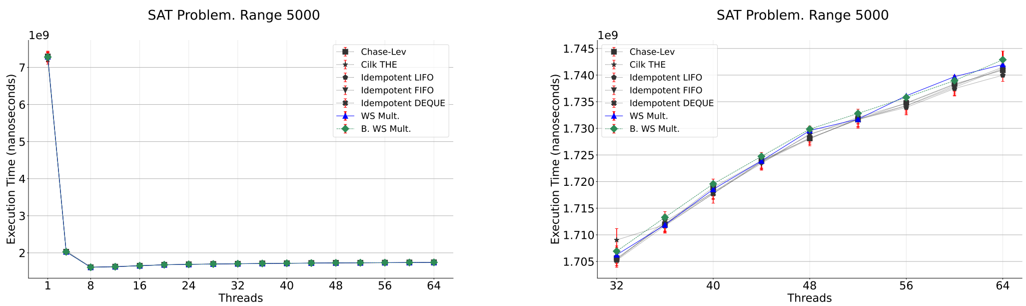


(b) Ranges assignment size 2500. Zoom into the number of processes 32 to 64.

Figure A.21: Mean times of the Parallel SAT benchmark for range assignment 2500.

	1	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64
Chase-Lev	6807896107.89	1914236833.24	1603361961.70	1622162661.17	1649249364.19	1675510309.77	1688450827.23	1697078640.41	1704259265.10	1711891852.91	1718234309.80	1723285770.54	1727774562.54	1732035230.83	1735832380.43	1739661272.07	1741298534.37
Cilk THE	6816506253.61	1932739779.44	1611365446.93	1628545371.31	1655263073.23	1679975184.97	1692563919.00	1701139295.01	1707224169.89	1712210277.49	1717784417.57	1723180177.33	1727869542.70	1732205092.44	1736024048.11	1739360323.06	1741844965.31
Idempotent LIFO	6795883410.40	1914543477.76	1603142288.10	1622895473.20	1649876811.87	1676189619.76	1687969233.69	1696979093.07	1704837211.54	1711074544.46	1717509408.44	1722935965.70	1727425424.61	1731790119.49	1735652704.74	1739413664.16	1741105015.00
Idempotent FIFO	6810312046.70	1913149015.50	1603686063.44	1622888939.60	1649595480.87	1676361758.97	1688785496.49	1696830178.04	1704291373.21	1710928138.03	1717216537.53	1723406657.81	1727716454.86	1732037552.71	1735788625.06	1739158050.40	1741537995.69
Idempotent DEQUE	6795867933.59	1909316403.64	1603308552.04	1622970973.50	1649619396.29	1675952278.53	1688318822.86	1696818099.57	1705028493.33	1710875171.26	1717544596.67	1722731932.97	1727864071.40	1732024811.96	1735806591.34	1739699164.11	1741808369.77
WS Mult.	6784463078.43	1916227074.50	1604103323.94	1623598975.61	1649743692.73	1676147521.41	1689076484.73	1697984582.23	1705022964.66	1712433782.90	1718085727.01	1724124612.69	1728615057.01	1732908764.11	1737249566.67	1740731898.13	1742501919.73
B. WS Mult.	6816171226.89	1910416234.89	1604261338.93	1623642904.71	1649990344.76	1676795907.16	1689289061.96	1697714414.36	1704790061.79	1712202655.50	1718618778.06	1723342785.13	1728904778.16	1732726504.89	1737372085.27	1740367275.94	1742789852.70

Table A.93: Resulting mean times for the SAT benchmark. These are the results for tasks with 2500 assignments.



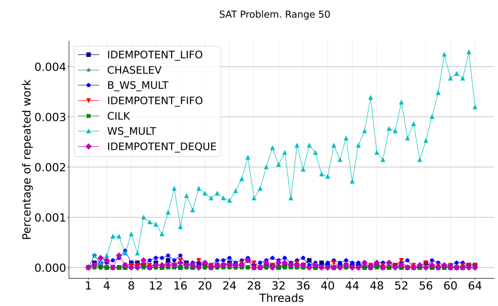
(a) Range assignment size 5000.

(b) Ranges assignment size 5000. Zoom into the number of processes 32 to 64.

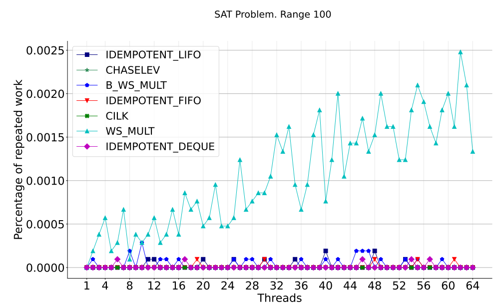
Figure A.22: Mean times of the Parallel SAT benchmark for range assignment 5000.

	1	4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64
Chase-Lev	7285444348.56	2029120544.70	1610840931.86	1625864632.37	1651924121.30	1676669914.19	1689242705.07	1697761499.00	1705625180.01	1711979785.74	1718904839.54	1723943987.14	1728117052.69	1731646894.47	1734762955.20	1738339069.49	1740921260.93
Cilk THE	7211213456.17	2055101188.23	1619246521.59	1631604785.83	1657556098.74	1680742572.53	1693503452.83	1702467072.90	1709034288.04	1711895178.07	1717737552.56	1723647133.99	1728024879.33	1731808821.01	1734251872.60	1737982613.44	1741501563.60
Idempotent LIFO	7280892297.93	2030058334.57	1611255684.93	1625648710.04	1652762654.09	1677250803.73	1689223900.27	1697778168.57	1705417895.04	1712077598.86	1718013767.90	1723794875.74	1728199281.56	1731677208.97	1733925082.49	1737467449.27	1739973111.63
Idempotent FIFO	7292628631.07	2029210917.37	1611248818.73	1626563681.34	1651834408.63	1677416077.41	1689620847.89	1697647235.00	1705353469.41	1712485674.60	1717932209.66	1723407431.76	1728950916.26	1731629093.40	1734217745.26	1737699384.86	1741246940.77
Idempotent DEQUE	7281505676.71	2029738420.67	1611572491.56	1626080625.09	1651964482.79	1677499923.70	1689722028.29	1698187601.06	1705212003.09	1711708188.13	1717743140.26	1723714608.24	1728802667.06	1732071276.20	1734686919.41	1738214847.00	1741009595.29
WS Mult.	7284741026.76	2027702744.71	1611911847.44	1626653605.89	1652396510.61	1677979382.30	1689506748.50	1698444679.17	1706250329.36	1711870566.90	1718549688.06	1723897100.57	1729581382.76	1731725029.69	1736112183.91	1739701400.29	1741991081.84
B. WS Mult.	7279765424.16	2027206495.57	1611890981.34	1625662595.94	1652782239.31	1678883118.04	1690393847.14	1698697144.41	1706923686.73	1713277524.11	1719563027.36	1724737935.97	1729858738.33	1732800003.49	1735807533.34	1738960635.77	1742884074.94

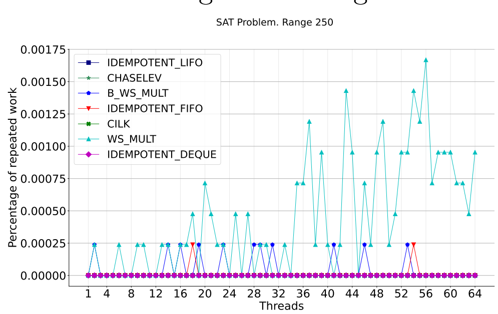
Table A.94: Resulting mean times for the SAT benchmark. These are the results for tasks with 5000 assignments.



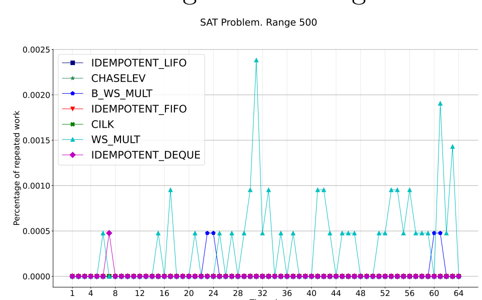
(a) Percentage of repeated work by algorithm.
Tasks with a range of 50 assignments.



(b) Percentage of repeated work by algorithm.
Tasks with a range of 100 assignments.

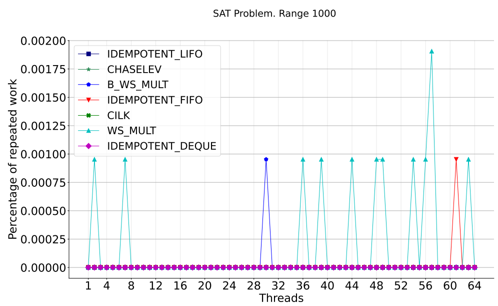


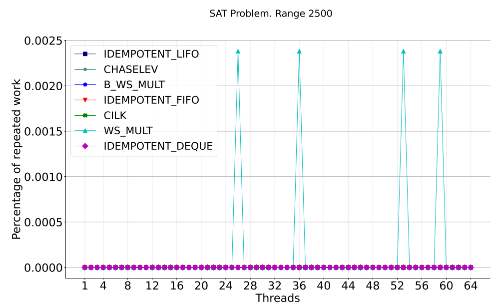
(c) Percentage of repeated work by algorithm.
Tasks with a range of 250 assignments.



(d) Percentage of repeated work by algorithm.
Tasks with a range of 500 assignments.

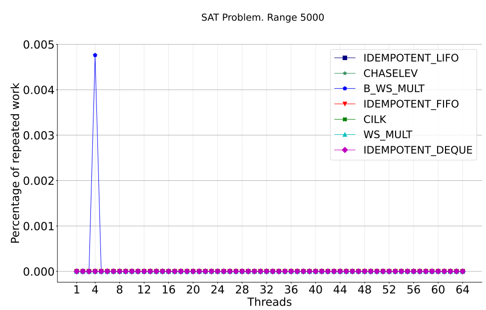
Figure A.23: percentage of repeated work performed by each algorithm when the range of assignments varies. This percentage is the number of repeated tasks concerning the total of tasks. Tested ranges of (50, 100, 250, 500).





(a) Percentage of repeated work by algorithm.
Tasks with a range of 1000 assignments.

(b) Percentage of repeated work by algorithm.
Tasks with a range of 2500 assignments.



(c) Percentage of repeated work by algorithm.
Tasks with a range of 5000 assignments.

Figure A.24: percentage of repeated work performed by each algorithm when the range of assignments varies. This percentage is the number of repeated tasks concerning the total of tasks. Tested ranges of (1000, 2500, 5000).

APPENDIX B

Queue evaluation Results

B.1 Results of Inner Experiments (LL/IC Evaluation)

This appendix shows the results obtained by executing the Inner Experiments for the evaluation of the LL/IC objects, following the methodology suggested by Georges, Buytaert, and Eeckout [32].

	Fetch and Increment	CAS LL/IC	RW LL/IC 64 padding	RW LL/IC 16 padding	RW LL/IC 32 padding	RW LL/IC no padding
1	288860972.63	305638864.47	317065162.60	319755230.40	324187867.90	320378736.50
4	80506440.00	100958610.23	118799737.40	107684065.93	116027730.73	107367825.10
8	63599707.47	78075928.57	84805029.10	80927260.60	84444217.30	79758733.07
12	50347568.87	58526625.20	65273976.33	62985744.70	65516543.10	62251471.77
16	41037828.93	46203364.33	51603157.93	51627402.33	52549893.80	52305099.07
20	43698354.07	45852623.00	50705047.53	50136623.90	51232438.97	52131764.40
24	45187224.67	46074348.43	47896947.17	46134113.00	48109941.77	48176109.00
28	40547444.73	40755668.37	43482972.33	42171177.13	43660332.43	45119529.33
32	35988846.27	35367188.00	40166244.37	38405117.57	39891455.73	41353746.77
36	34734531.67	34048104.30	40661526.40	43080326.90	40424838.03	40796560.43
40	35586323.23	34568989.23	39735940.17	45696269.77	40990485.80	38923483.30
44	36000699.17	36938976.17	39689813.13	43486986.73	39893678.63	45420477.57
48	34018945.07	34521533.77	37031091.27	40663404.70	36665244.53	43799081.63
52	35914989.23	34660215.90	37839803.70	40314233.27	39197357.90	41730669.73
56	36094405.17	36113970.37	38005800.77	37420952.30	38938837.97	39787863.63
60	33648086.47	33491633.80	38858203.27	36847570.77	38009665.63	37164143.80
64	33447182.50	33003608.20	37543791.23	34631263.33	35936880.00	38732530.37

Table B.1: Mean times for LL/IC experiemnt

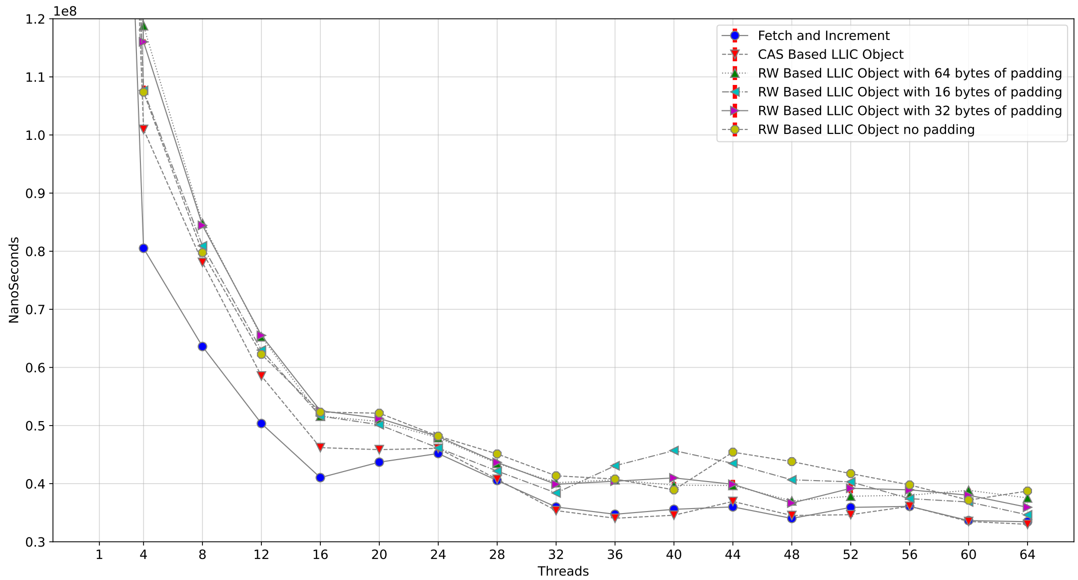


Figure B.1: 1,000,000 Interspersed Takes and Puts (CAS vs FAI) for 64 threads.

	Fetch and Increment	CAS LL/IC	RW LL/IC 64 padding	RW LL/IC 16 padding	RW LL/IC 32 padding	RW LL/IC no padding
1	0.00	-5.81	-9.76	-10.70	-12.23	-10.91
4	0.00	-25.40	-47.57	-33.76	-44.12	-33.37
8	0.00	-22.76	-33.34	-27.24	-32.77	-25.41
12	0.00	-16.25	-29.65	-25.10	-30.13	-23.64
16	0.00	-12.59	-25.75	-25.80	-28.05	-27.46
20	0.00	-4.93	-16.03	-14.73	-17.24	-19.30
24	0.00	-1.96	-6.00	-2.10	-6.47	-6.61
28	0.00	-0.51	-7.24	-4.00	-7.68	-11.28
32	0.00	1.73	-11.61	-6.71	-10.84	-14.91
36	0.00	1.98	-17.06	-24.03	-16.38	-17.45
40	0.00	2.86	-11.66	-28.41	-15.19	-9.38
44	0.00	-2.61	-10.25	-20.79	-10.81	-26.17
48	0.00	-1.48	-8.85	-19.53	-7.78	-28.75
52	0.00	3.49	-5.36	-12.25	-9.14	-16.19
56	0.00	-0.05	-5.30	-3.68	-7.88	-10.23
60	0.00	0.46	-15.48	-9.51	-12.96	-10.45
64	0.00	1.33	-12.25	-3.54	-7.44	-15.80

Table B.2: Percentage improvement of LL/IC objects respect to Fetch&Increment from 1 to 64 threads of execution.

B.2

Results of Inner Experiments (Module Queue Variants)

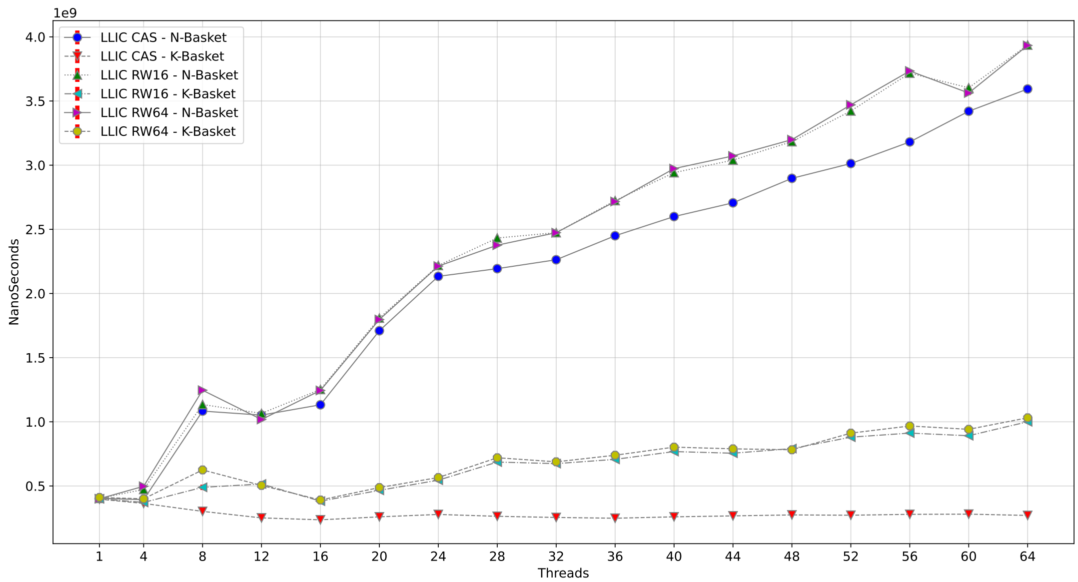


Figure B.2: 1,000,000 interspersed enqueue - dequeue calls for 64 threads.

	LLIC CAS - N-Basket	LLIC CAS - K-Basket	LLIC RW16 - N-Basket	LLIC RW16 - K-Basket	LLIC RW64 - N-Basket	LLIC RW64 - K-Basket
1	403837572.97	396923559.23	403364522.07	400055812.90	400458163.90	410361616.03
4	393072415.97	363825211.27	472695382.10	371714856.90	497516571.87	399706830.60
8	1083497864.53	301903826.67	1134626840.47	489293420.43	1246191018.60	624905787.60
12	1052047521.10	251061393.53	1065479541.67	515315431.30	1018033354.13	503958613.03
16	1132563585.47	236075970.90	1252081224.87	381578408.67	1244407677.97	390607216.17
20	1709842518.17	259198051.17	1808325283.23	465887426.90	1796996372.03	487635858.33
24	2133731012.70	278000991.57	2218159617.33	544932910.07	2213154726.97	565660003.90
28	2193455107.57	263495757.53	2432517335.17	686078560.50	2376492044.60	719843684.83
32	2262827345.93	254498662.33	2474083071.13	674008201.03	2473843065.13	688144066.47
36	2449877745.93	248545543.60	2723899346.60	707239057.03	2716683811.97	739506175.80
40	2599424591.43	259304569.17	2941545544.63	767947461.03	2973306680.03	802582429.10
44	2706567513.60	266755495.50	3039584219.03	755145375.97	3072020329.00	789260312.00
48	2897334261.40	274293059.03	3183274684.97	791166367.20	3199031929.60	782029672.97
52	3012856935.07	271947171.90	3422409173.57	880173001.43	3470237815.97	911715385.53
56	3181102862.03	278835375.93	3717005695.57	911886436.97	3734223566.20	967380566.13
60	3420201175.47	280311296.37	3602981476.13	891396736.43	3562427501.20	941420017.77
64	3593712103.47	270304602.33	3936145887.47	1000239685.37	3933207413.67	1031183359.33

Table B.3: Mean times for Enqueue - Dequeue inner experiment for 64 threads.

	LLIC CAS - N-Basket	LLIC CAS - K-Basket	LLIC RW16 - N-Basket	LLIC RW16 - K-Basket	LLIC RW64 - N-Basket	LLIC RW64 - K-Basket
1	-1.74	0.00	-1.62	-0.79	-0.89	-3.39
4	-8.04	0.00	-29.92	-2.17	-36.75	-9.86
8	-258.89	0.00	-275.82	-62.07	-312.78	-106.99
12	-319.04	0.00	-324.39	-105.25	-305.49	-100.73
16	-379.75	0.00	-430.37	-61.63	-427.12	-65.46
20	-559.67	0.00	-597.66	-79.74	-593.29	-88.13
24	-667.53	0.00	-697.90	-96.02	-696.10	-103.47
28	-732.44	0.00	-823.17	-160.38	-801.91	-173.19
32	-789.13	0.00	-872.14	-164.84	-872.05	-170.39
36	-885.69	0.00	-995.94	-184.55	-993.03	-197.53
40	-902.46	0.00	-1034.40	-196.16	-1046.65	-209.51
44	-914.62	0.00	-1039.46	-183.09	-1051.62	-195.87
48	-956.29	0.00	-1060.54	-188.44	-1066.28	-185.11
52	-1007.88	0.00	-1158.48	-223.66	-1176.07	-235.25
56	-1040.85	0.00	-1233.05	-227.03	-1239.22	-246.94
60	-1120.14	0.00	-1185.35	-218.00	-1170.88	-235.85
64	-1229.50	0.00	-1356.19	-270.04	-1355.10	-281.49

Table B.4: Percentage improvement of Enqueue - Dequeue respect to LL/IC Compare&Swap & K-Basket from 1 to 64 threads of execution.

B.3

Results of Outer Experiments

	Fetch-and-Add	LCRQ	Castañeda-Piña	Castañeda-Piña Array	Castañeda-Piña Segments	Michael and Scott	Ostrovsky-Morrison	YMC
1	351200027.63	403572408.43	472555550.90	487165396.30	475511479.40	587248865.70	1146251708.13	377297171.80
4	169692112.77	187820883.50	301829643.73	1077150109.93	288435581.67	471544279.67	3387292159.17	197394522.23
8	121854879.07	116026264.60	272038021.70	1490503277.57	263046870.43	559538345.73	2259382162.50	122918448.70
12	88994036.27	94770156.63	281703838.03	1783321075.57	227342770.57	541606321.37	2026162066.43	87396589.83
16	74776242.40	84622830.50	294978630.73	1867961323.20	263256036.93	535908982.53	1723493401.93	70816962.53
20	78627962.80	93050854.37	379995043.80	2023925634.47	312408909.57	684008030.47	1932146663.93	67941366.37
24	79543299.43	97720927.97	422919854.07	2120916067.10	327654824.50	803985551.10	2037756243.03	66012273.80
28	71817559.87	95509834.43	432414808.13	2166062984.03	341570968.90	837462016.30	1897402494.37	59065987.40
32	67443713.50	99412632.57	438538137.47	2236554396.10	345087614.40	859154816.23	1720945612.37	53669770.97
36	63240889.47	101217631.23	437121875.30	2429715567.90	350204025.40	875829430.07	1629208308.67	48970421.20
40	65311551.40	108716588.03	461605337.17	2420424604.93	358781500.57	942633485.97	1678287663.47	48899447.00
44	63473601.93	113385554.47	460308301.33	2464267895.70	360854102.40	946151498.10	1534113476.07	46628124.97
48	62148500.10	116865624.87	460302515.57	2523987255.27	364872124.93	948736421.87	1451514990.13	46240439.33
52	64238033.60	122543525.30	484756656.03	2514987713.47	379651287.00	1024468571.63	1413332353.37	47095151.57
56	66547657.57	126754099.80	507507288.80	2540588509.17	402499664.00	1118652395.07	1402048064.73	47973167.30
60	65885871.70	127109938.03	505315540.43	2575744556.40	416756883.57	1140158428.23	1337866333.20	46690629.67
64	65178512.30	126730354.90	502870762.67	2607573629.43	428069875.03	1140572453.10	1295963945.30	43235588.63

Table B.5: Mean times for Enqueue - Dequeue outer experiment for 64 threads.

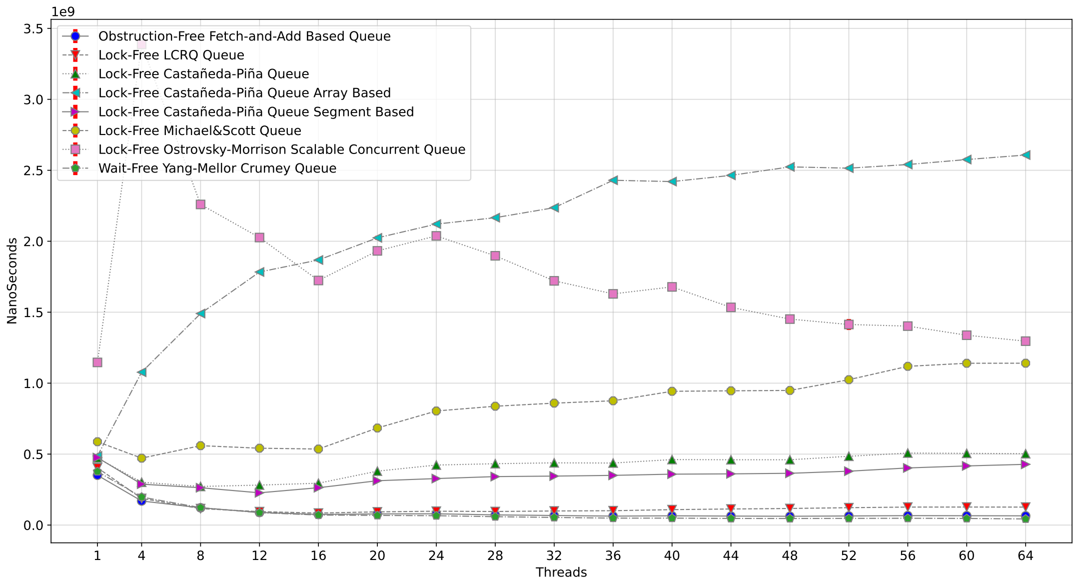


Figure B.3: 1,000,000 of interspersed Enqueue - Dequeue calls for 64 threads.

	Fetch-and-Add	LCRQ	Castañeda-Piña	Castañeda-Piña Array	Castañeda-Piña Segments	Michael and Scott	Ostrovsky-Morrison	YMC
1	6.92	-6.96	-25.25	-29.12	-26.03	-55.65	-203.81	0.00
4	14.03	4.85	-52.91	-445.68	-46.12	-138.88	-1616.00	0.00
8	0.87	5.61	-121.32	-1112.60	-114.00	-355.21	-1738.11	0.00
12	-1.83	-8.44	-222.33	-1940.49	-160.13	-519.71	-2218.35	0.00
16	-5.59	-19.50	-316.54	-2537.73	-271.74	-656.75	-2333.73	0.00
20	-15.73	-36.96	-459.30	-2878.93	-359.82	-906.76	-2743.84	0.00
24	-20.50	-48.03	-540.67	-3112.91	-396.35	-1117.93	-2986.94	0.00
28	-21.59	-61.70	-632.09	-3567.19	-478.29	-1317.84	-3112.34	0.00
32	-25.66	-85.23	-717.10	-4067.25	-542.98	-1500.82	-3106.55	0.00
36	-29.14	-106.69	-792.62	-4861.60	-615.13	-1688.49	-3226.92	0.00
40	-33.56	-122.33	-843.99	-4849.80	-633.71	-1827.70	-3332.12	0.00
44	-36.13	-143.17	-887.19	-5184.94	-673.90	-1929.14	-3190.10	0.00
48	-34.40	-152.73	-895.45	-5358.40	-689.08	-1951.75	-3039.06	0.00
52	-36.40	-160.20	-929.31	-5240.23	-706.14	-2075.32	-2901.01	0.00
56	-38.72	-164.22	-957.90	-5195.85	-739.01	-2231.83	-2822.57	0.00
60	-41.11	-172.24	-982.26	-5416.62	-792.59	-2341.94	-2765.39	0.00
64	-50.75	-193.12	-1063.09	-5931.08	-890.09	-2538.04	-2897.45	0.00

Table B.6: Percentage improvement of Enqueue - Dequeue respect to Yang and Mellor-Crummey Queue from 1 to 64 threads of execution.