Synchronization Costs in Parallel Programs and Concurrent Data Structures

Vitalii Aksenov

Thèse de doctorat d'Informatique
ITMO University and Université Paris-Diderot

L'École Doctorale Sciences Mathématiques de Paris Center, ED 386
INRIA Paris, GALLIUM

Synchronization Costs in Parallel Programs and Concurrent Data Structures

Vitalii Aksenov
under the supervision of Petr Kuznetsov, Anatoly Shalyto and Carole Delporte

Defense: September 26, 2018
Last manuscript update: September, 2018

Jury:	President of Jury	Hugues Fauconnier, HDR, Paris 7 Diderot
	Advisor	Carole Delporte, HDR, Paris 7 Diderot
	Co-advisor	Petr Kuznetsov, PhD, Télecom ParisTech
	Advisor	Anatoly Shalyto, DSc, ITMO University
	Reviewer	Danny Hendler, PhD, Ben-Gurion University
	Reviewer	Guerraoui Rachid, PhD, EPFL
	Examinator	Sylvie Delaët, HDR, Université Paris Sud

Abstract

To use the computational power of modern computing machines, we have to deal with *concurrent* programs. Writing efficient concurrent programs is notoriously difficult, primarily due to the need of harnessing *synchronization costs*. In this thesis, we focus on synchronization costs in parallel programs and concurrent data structures.

First, we present a novel *granularity control* technique for parallel programs designed for the dynamic multithreading environment. Then in the context of concurrent data structures, we consider the notion of *concurrency-optimality* and propose the first implementation of a concurrency-optimal binary search tree that, intuitively, accepts a concurrent schedule *if and only if* the schedule is correct. Also, we propose *parallel combining*, a technique that enables efficient implementations of concurrent data structures from their *parallel batched* counterparts. We validate the proposed techniques via experimental evaluations showing superior or comparable performance with respect to state-of-the-art algorithms.

From a more formal perspective, we consider the phenomenon of *helping* in concurrent data structures. Intuitively, helping is observed when the order of some operation in a linearization is *fixed* by a step of another process. We show that no wait-free linearizable implementation of stack using read, write, compare&swap and fetch&add primitives can be *help-free*, correcting a mistake in an earlier proof by Censor-Hillel et al. Finally, we propose a simple way to *analytically* predict the throughput of data structures based on coarse-grained locking.

Keywords: parallel programs, concurrent data structures, synchronization, performance, granularity control, algorithms

Resumé

Pour utiliser la puissance de calcul des ordinateurs modernes, nous devons écrire *des programmes concurrents*. L'écriture de programme concurrent efficace est notoirement difficile, principalement en raison de la nécessité de gérer *les coûts de synchronization*. Dans cette thèse, nous nous concentrons sur les coûts de synchronisation dans les programmes parallèles et les structures de données concurrentes.

D'abord, nous présentons une nouvelle technique de *contrôle de la granularité* pour les programmes parallèles conçus pour un environnement de multi-threading dynamique. Ensuite, dans le contexte des structures de données concurrentes, nous considérons la notion d'*optimalité de concurrence* (*concurrency-optimality*) et proposons la première implémentation concurrence-optimal d'un arbre binaire de recherche qui, intuitivement, accepte un ordonnancement concurrent *si et seulement si* l'ordonnancement est correct. Nous proposons aussi la *combinaison parallèle* (*parallel combining*), une technique qui permet l'implémentation efficace des structures de données concurrences à partir de leur version *parallèle par lots*. Nous validons les techniques proposées par une évaluation expérimentale, qui montre des performances supérieures ou comparables à celles des algorithmes de l'état de l'art.

Dans une perspective plus formelle, nous considérons le phénomène d'*assistance* (*helping*) dans des structures de données concurrentes. On observe un phénomène d'assistance quand l'ordre d'une opération d'un processus dans une trace linéarisée est *fixée* par une étape d'un autre processus. Nous montrons qu'aucune implémentation sans attente (*wait-free*) linéarisable d'une pile utilisant les primitives `read`, `write`, `compare&swap` et `fetch&add` ne peut être "sans assistance" (*help-free*), corrigeant une erreur dans une preuve antérieure de Censor-Hillel *et al.* Finalement, nous proposons une façon simple de prédire *analytiquement* le débit (*throughput*) des structures de données basées sur des verrous à gros grains.

Mots clefs: programmes parallèles, structures de données simultanées, synchronisation, performance, contrôle de la granularité, algorithmes

Contents

1 Introduction

In 1965, Intel co-founder Gordon E. Moore made a prediction [119], widely known as Moore's law, that the number of transistors per integrated circuit would increase exponentially leading to an exponential increase of a chip performance. However, around 2004, such increase in performance became economically inefficient due to hardware limitations [146]. To resolve this, hardware manufacturers had to find another way to improve the performance. They decided to combine multiple computational units (*cores*) on the same processor, named *multicore processor*. Further, we avoid the term *core* and use the term *process* instead: a *core* is a physical object while a *process* is an abstraction that represents a sequential code running on a core.

1.1 Concurrent Programs

The invention of multicore processors, known as the "multicore revolution" [110], changed the whole paradigm of computing. To benefit from the new design of processors, sequential algorithms had to be replaced by *concurrent* ones.

In this thesis, we consider two types of concurrent programs: *parallel programs* and *concurrent data structures*. The distinction between these types of programs is based on the relation between inputs and outputs.

Similarly to sequential programs, a parallel program, given an input, should produce an output that satisfies a *specification*. A specification maps each possible input to a set of allowed outputs. The main advantage of parallel programs in comparison to sequential programs is that they can use multiple processes to improve performance (Figure 1.1).

Consider a sorting algorithm as an example. Its specification is: given an input array a the algorithm should output an array of elements of a in the ascending order. The sequential *mergesort* algorithm sorts an array of size m in $O(m \log m)$ time, while its parallel version [49] only needs $O(\frac{m}{n} \cdot \log m)$ time where n is the number of processes.

In concurrent data structures, e.g., concurrent stacks and concurrent binary search trees, inputs and outputs are distributed across the concurrent processes (Figure 1.2). The input of a process consists of operations called on the data structure by that process. The process performs each operation and appends its result to the output. Of course, these results should satisfy some correctness property. Typically, the correctness properties of concurrent data structures are split into two categories: *safety* or *liveness*.

A safety property asserts that "nothing bad ever happens". The most reknown safety property is *linearizability* [93] — the invocations and responses of high-level operations observed in the execution should constitute a correct sequential history.

A liveness property asserts that "something good eventually happens". The most common liveness properties are:

- *wait-freedom* — every process eventually completes its operation;
- *lock-freedom* — at least one process eventually completes its operation;
- *starvation-freedom* — every process eventually completes its operation if each process takes an infinite number of steps;
- *deadlock-freedom* — at least one process eventually completes its operation if each process takes an infinite number of steps.

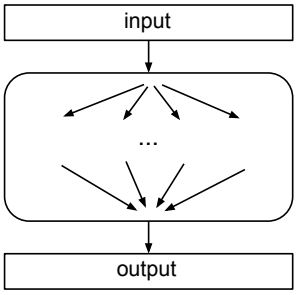


Figure 1.1: Execution of a parallel program

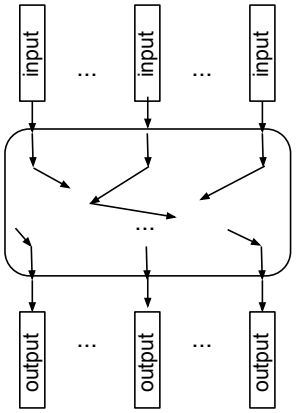


Figure 1.2: Execution of a concurrent data structure

Note that starvation-freedom and deadlock-freedom allow the implementation to use locks, while wait-freedom and lock-freedom do not [91].

1.2 Synchronization

To ensure correctness of parallel programs and concurrent data structures, we need *synchronization*, i.e., coordination between the processes. Informally, synchronization is used to handle conflicts on the *shared data*, e.g., resolving data races, and on *shared resources*, e.g., memory allocation and deallocation.

Let us have a look at how synchronization is instantiated in parallel programs. Parallel programs are typically written for two environments: for *static multithreading* and for *dynamic multithreading* [50].

In static multithreading, each process is provided with its own program. These programs are written as a composition of *supersteps*. During a superstep, a process performs independent local computations. After the superstep is completed, the processes synchronize to accumulate the results of their independent computations.

In dynamic multithreading, the program is written using *fork-join* mechanism (or a similar one, for example, `# pragma omp parallel` in OpenMP [34]). *Fork-join* takes two functions, named *branches*, as arguments and executes them in parallel until their completion. One *fork-join* call incurs the *synchronization overhead* at least on [72]:

1. Allocating and deallocating a thread: the shared memory should be able to process concurrent requests.

2. Scheduling of a thread. The scheduler should accept concurrent requests to execute threads and should control migration of threads. The migration happens when a thread changes the process-owner: this typically involves significant overhead spent on transfering the necessary data.

3. Joining of the two threads. A process that completes a thread decrements a concurrent counter of the corresponding *fork-join* call. If the counter is zero the process proceeds with the code after the *fork-join*, otherwise, it flags the scheduler that it is idle.

An important property of most parallel programs (written either for static or dynamic multithreading) is that they are designed in a way that there are no data races except for the races described above.

In contrast, data races arising in concurrent data structures, typically, are way more diverse because each process has its own input and invocations of operations on different

processes are not synchronized. As a consequence, synchronization is individually crafted from scratch for each concurrent data structure. For example, lock-based data structures get rid of data races using locks that enable an exclusive access to data. In wait-free and lock-free data structures, data races are allowed and they are resolved using different techniques, for example, *helping* where one process may perform some work on behalf of other processes. This complexity of synchronization makes concurrent data structures more challenging to design than parallel programs.

1.3 Overview of the Results

Synchronization, when it is not implemented properly, can induce an overhead overwhelming the benefits of parallelism. Ideally, the programmer should use just the right amount of synchronization to ensure correctness. In this thesis, we discuss how to improve performance in different settings by reducing the synchronization overhead.

At first, we provide a new practical way to solve the *granularity control problem* [100]. Suppose that we are provided with a parallel program written for dynamic multithreading using the *fork-join* mechanism. Informally, the granularity control problem consists in deciding, for each *fork-join* in a given program, whether to call it or to execute its branches sequentially in order to achieve the best performance. On one hand, if too many calls are executed, the total synchronization overhead (spent on allocation, scheduling, and joining of threads) can take a considerable fraction of the total execution time. On the other hand, if too many calls are executed sequentially, we lose potential benefits of multiprocessor machine.

Then we describe two techniques to build concurrent data structures with potentially low synchronization overhead:

- *concurrency-optimality*, a technique that constructs a concurrent data structure from a sequential implementation with the theoretically "optimal" synchronization;
- *parallel combining*, a technique that uses explicit synchronization to gather all concurrent accesses to a concurrent data structure and then calls a parallel algorithm that requires a small amount of synchronization, thus, reducing the total synchronization overhead.

Further, we consider synchronization in wait-free and lock-free data structures that typically appears in a form of *helping*. We identify a mistake in an earlier proof by Censor-Hillel et al. [44] of the fact that stack does not have a wait-free help-free implementation and, then, we reprove that fact.

Finally, we describe a simple method that theoretically measures the throughput of simple coarse-grained lock-based concurrent data structures exporting one method: a critical section of size C guarded by CLH lock [51] followed by a parallel section of size P.

Below, we describe the results in more details.

1.3.1 Granularity Control Problem

Over past decades many programming languages and systems for parallel computing have been developed, such as Cilk, OpenMP, Intel Threading Building Blocks, etc., to provide *dynamic multithreading* [50] (sometimes called *implicit parallelism*). Although these systems raise the level of abstraction and hide synchronization details from the user, the programmer still has to perform extensive optimization in order to improve performance. One such optimization is *granularity control*, which requires the programmer to determine when and how parallel tasks should be sequentialized.

We present an automatic algorithm to control granularity using an *oracle*: the oracle predicts the execution time of a parallel code and helps the algorithm decide whether the code should be sequentialized or not. We prove that this algorithm has the desired

theoretical properties, i.e., the overhead on synchronization is small compared to the total execution time. Finally, we implement it in C++ as an extention to Cilk and evaluate its performance. The results show that our technique can eliminate hand tuning in many cases and closely match the performance of hand-tuned code.

To get an intuition behind granularity control, suppose that a dynamic multithreading is implemented via the *fork-join* primitive [50]. This primitive takes two functions, called *program branches*, as arguments and executes them in parallel until their completion. Despite its simplicity, *fork-join* is powerful enough to implement other common parallel expressions, including parallel loops and nested parallel computations [115].

One important aspect of dynamic multithreading is that it is possible to express the efficiency of a parallel program using so-called *Work-Span Framework*: *work* is the time spent to execute the program using an idealized machine with one process and *span* is the time spent to execute a program using an idealized machine with an infinite supply of processes.

Widely accepted as complexity metrics for parallel programs, work and span do not account for the real overhead hidden in a *fork-join* call — they consider this call to cost one unit of work (one instruction). However, in practice, this cost is non-negligible — one *fork-join* call induces at least an allocation of a new thread, its scheduling and, finally, the joining. Frequent calls of this primitive can take a considerable fraction of the total execution time. Thus, for each *fork-join* call, we have to decide whether to call it or to execute an alternative sequential algorithm, e.g., execute its branches sequentially. The resulting *granularity control problem* is about the synchronization overhead: we want to call *fork-join*s rarely enough so that the time spent on the overhead of these calls is a small fraction of the total execution time, and, on the other hand, we want to call them often enough to make processes busy.

The most popular solution to this problem is to cover each *fork-join* call with a special *condition-guard* [100]: if the work in both branches exceeds some *grain-size constant*, i.e., the approximation of the execution time is much bigger than the overhead in *fork-join*, then we are allowed to call it; otherwise, we execute an alternative sequential algorithm (e.g., execute branches sequentially).

However, such an approach is quite inconvenient. First, a program can have several *fork-join* calls that have distinct condition-guards with different grain-size constants. To improve performance, we have to tune all these constants. This tuning is tedious, especially when the constants cannot be tuned separately. For example, two constants of different condition-guards that correspond to two nested *fork-join* calls depend on each other. There is a further performance issue: everytime we want to check whether the chosen constants provide good performance the program has to be re-executed. Second, the tuned constants are *non-portable*, i.e., the constants tuned for one machine can provide bad performance on other machine.

In order to resolve these two issues, difficulty of tuning and non-portability, we present an automatic granularity control technique in a form of a *sp-guard* (a substitution of a condition-guard).

At first, the user should provide the asymptotic cost function of the code covered by each sp-guard. Notice that, as these functions depend only on the algorithm, they are portable.

Second, the user should provide a machine dependent variable κ. It represents the smallest amount of work that is big enough in comparison to an overhead in a *fork-join* call.

Our automatic granularity control algorithm works with each sp-guard g separately. At first, it determines the biggest value N_g of the cost function for which the sequential execution time of the covered *fork-join* does not exceed κ. Then, when the *fork-join* is about to be called, the cost is compared with N_g. If N_g is smaller than the cost, then the execution time exceeds κ and we can call the *fork-join*, otherwise, an alternative sequential

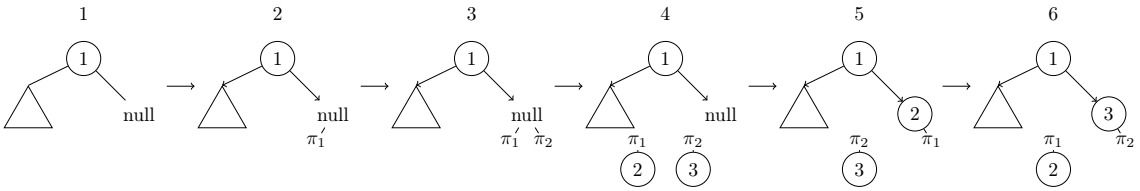


Figure 1.3: A non-linearizable schedule

algorithm is executed (e.g., two branches are executed sequentially).

For our granularity control technique, we provide a theoretical analysis in the Work-Span Framework [104] together with a rigorous experimental analysis on a wide range of parallel programs taken from the PBBS suite [142] on three machines with different architectures. Experiments show that the performance of programs written with our automatic granularity control is comparable or even better (up to 32% faster and no more than 6% slower) than the performance of hand tuned code.

1.3.2 A Concurrency-Optimal Binary Search Tree

Implementing a concurrent data structure typically begins from a sequential implementation. When used *as is* in a concurrent environment, a sequential data structure may expose incorrect behaviour: lost updates, incorrect responses, etc. One way to ensure correctness is to protect portions of the sequential code operating on the shared data with synchronization primitives. By that, certain *schedules*, i.e., interleavings of the steps of the sequential implementation, are rejected, and the more synchronization we use the more *schedules* we reject.

Using this notion of schedules, we can compare the "concurrency" of algorithms: linearizable implementation A is "more concurrent" than linearizable implementation B if the set of schedules accepted by A is a strict superset of the set of schedules accepted by B. Thus, a linearizable implementation is *concurrency-optimal* [79] if it accepts *all* linearizable schedules. We expect this implementation to perform well since it uses exactly as little synchronization as necessary to ensure the linearizability of the data structure. Note that the notion of concurrency-optimality is theoretical and, thus, we expect concurrency-optimal implementations to perform well on any machine.

As an example, consider the sequential partially-external binary search tree and its following schedule (Figure 1.3):

1. initially the tree consists of one node with value 1;
2. $\pi_1 =$ `insert`(2) finds that 2 should be in the right child of the node with 1;
3. $\pi_2 =$ `insert`(3) finds that 3 should be in the right child of the node with 1;
4. π_1 and π_2 create new nodes;
5. π_1 links its node as the right child of the node with 1;
6. π_2 overwrites the right child of the node with 1.

This schedule is non-linearizable: operation `insert`(2) is lost and, thus, it should be rejected by any concurrency-optimal implementation.

We design a concurrency-optimal binary search tree. We implement it and compare against the state-of-the-art implementations [39, 52, 57, 62]. To check the portability of *concurrency-optimal* technique we perform experiments on two machines with different architectures. The evaluation shows that our implementation is one of the best and, thus, the concurrency-optimal approach can be an adequate design principle for building efficient portable data structures.

1.3.3 Parallel Combining

We discussed two portable methods that try to provide optimal synchronization: an automatic granularity control for parallel programs and the concurrency-optimal design principle for concurrent data structure. The first method is automatic: it uses the fact that the synchronization in parallel programs appears in the special form and, by that, it almost does not require to change the original program. By contrast, concurrency-optimality requires the programmer to design the data structure from scratch, which might be nontrivial. It would be interesting to check whether it is possible to automatically build efficient concurrent data structures from parallel programs.

Efficient "concurrency-friendly" data structures (e.g., sets based on linked lists [84, 85] or binary search trees [52, 57]) are, typically, not subject to frequent data races and are designed using hand-crafted fine-grained locking.. In contrast, "concurrency-averse" data structures (e.g., stacks and queues) are subject to frequent sequential bottlenecks and solutions based on combining (e.g., [86]), where requests are synchronized into batches and each batch is applied sequentially, perform surprisingly well compared to fine-grained ones [86]. A general data structure typically combines features of "concurrency-friendliness" and "concurrency-averseness", and an immediate question is how to implement it in the most efficient way.

We propose *parallel combining*, a technique that can be used to build a concurrent data structure from a *parallel batched* counterpart [13]. A *parallel batched* data structure is implemented as a parallel program (`apply` function) that applies a set of data-structure operations to the underlying sequential data structure. As a correctness criteria, typically, after the execution of `apply` function we expect that the results of operations and the state of the underlying sequential data structure satisfy some sequential application of the operations. Parallel batched data structures appeared a long time ago, and, perhaps, the oldest one is 2-3 search tree by Paul et al. [129]. As other examples, we can consider parallel batched priority queues (e.g., [137]) and parallel batched graphs (e.g., [6]).

In *parallel combining*, processes share a set of active requests using any *combining* algorithm [58]. One of the active processes becomes a *combiner* and forms a *batch* from the requests in set. Then, under the coordination of the combiner, owners of the collected requests, called *clients*, execute `apply` method on the parallel batched data structure.

This technique becomes handy when the overhead on combining algorithm is compensated by the advantages of using the parallel batched data structure. We show that combining and parallel batching pay off for data structures that offer some degree of parallelism, but still have sequential bottlenecks.

We discuss three applications of parallel combining. First, we design concurrent implementations optimized for *read-dominated* workloads given sequential data structures. Intuitively, updates are performed sequentially by the combiner and read-only operations are performed by the clients in parallel. In our performance analysis we considered a sequential *dynamic graph* data structure D [95], that allows adding and removing edges (updates), and checking a connectivity between two vertices (read-only). We implement the resulting concurrent dynamic graph and compare it against three implementations: the first uses the global lock to access D; the second uses the read-write lock, i.e., a connectivity query takes the read lock and other queries take the write lock; and, finally, the last one uses *flat combining* [86] to access D. The experimental analysis shows that our implementation has throughput up to six times higher on the loads consisting mostly of connectivity queries that can be parallelized.

Second, we apply parallel combining to the parallel batched *binary search tree* [32]. The resulting concurrent algorithm performs somewhat worse than the state-of-the-art implementations [39, 41, 52, 57, 62, 97, 120, 121], which is expected: a binary search tree is a concurrency-friendly data structure. But from the theoretical point of view, our concurrent tree provides a guarantee (inherited from the parallel batched algorithm) that it is always strictly balanced compared to the state-of-the-art concurrent implementations

with relaxed balancing schemes [39, 41, 52, 57].

Finally, we apply parallel combining to *priority queue*. As a side contribution, we propose a novel parallel batched priority queue. We compare the resulting concurrent algorithm with the state-of-the-art algorithms [90, 113, 139] and show that our implementation is among the best ones.

To summarize, our performance analysis shows that parallel combining can be used to construct efficient concurrent data structures.

1.3.4 Helping as a Synchronization Technique

Up to this point we only considered synchronization costs in *lock-based* (deadlock-free or starvation-free) concurrent data structures. In such data structures, data races are resolved by protecting a code operating on the shared data with the mutual exclusion mechanisms, i.e., *locks*.

Wait-freedom or lock-freedom guarantees cannot be achieved using locks and should use other types of synchronization. To ensure these progress guarantees when one process blocks indefinitely, it might be necessary for the remaining processes to complete the operation of the blocked process. This behaviour is a special type of synchronization known as *"helping"*.

Censor-Hillel et al. [44] proposed a natural formalization of helping, based on the notion of linearization: a process p helps an operation of a process q in a given execution if a step of p determines that an operation of q linearizes before some other operation by a different process in any possible extention. It was claimed that in a system provided with read, write, compare&swap and fetch&add shared memory primitives helping is required for any wait-free linearizable implementation of an *exact order* data type. Informally, a sequential data type is exact order if for some operation sequence any change in the relative order of two operations affects the result of at least one other operation. As examples of exact order data types, Censor-Hillel et al. gave queue and stack.

However, we observe that the stack data type is not exact order. As we show, in any sequential execution on stack, we can reorder any two operations in such a way that no other operation will see the difference. Hence, the original proof for exact order types by Censor-Hillel et al. does not apply to stack.

In this thesis, we present a direct proof that stack does not have a wait-free help-free implementation. Our proof is similar to the original one: we choose any help-free implementation and build an execution history such that one process makes an infinite number of steps, but never completes, proving that the implementation is not wait-free.

1.3.5 Performance Prediction of Coarse-Grained Programs

Reasoning about *concurrency-optimality* provides an *analytical* way to compare synchronization overheads of implementations using sets of schedules. However, this metric might be not always useful capturing synchronization costs. Implementations are, in this sense, incomparable when their sets of schedules are not related by containment, though their actual performance may differ a lot. Furthermore, the scope of implementations that enable the schedule-based metric is limited, e.g., lock-free algorithms are hard to model this way.

The common way to compare two implementations is to compare their *throughput*, i.e., the number of operations per unit of time. In the last technical chapter of this thesis, we discuss an analytical way to evaluate throughput of an implementation. Typically, throughput is measured through experiments, and soundness of the results is subject to experimental setting, the workload, the machine, etc. It is therefore interesting whether we can quantify throughput theoretically.

We describe a simple model that can be used to predict throughput of *coarse-grained* lock-based algorithm. We show that our model works well for the simple data structures

that export one method: perform a critical section of size C guarded by CLH lock [51] (acquire the lock, perform work of size C and release the lock) followed by a parallel section of size P (simply, work of size P).

1.4 Publications

The results presented in this thesis appeared originally in the following papers.

[1] Umut A Acar, Vitaly Aksenov, Arthur Charguéraud, and Mike Rainey. Performance challenges in modular parallel programs. In *Proceedings of the twenty third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 381–382. ACM, 2018

[2] Vitaly Aksenov, Vincent Gramoli, Petr Kuznetsov, Anna Malova, and Srivatsan Ravi. A concurrency-optimal binary search tree. In *European Conference on Parallel Processing (Euro-Par)*, pages 580–593. Springer, 2017

[3] Vitaly Aksenov, Petr Kuznetsov, and Anatoly Shalyto. Parallel combining: Benefits of explicit synchronization. *arXiv preprint arXiv:1710.07588*, 2018

[4] Vitaly Aksenov, Petr Kuznetsov, and Anatoly Shalyto. On helping and stacks. In *Proceedings of NETYS 2018*

[5] Vitaly Aksenov, Dan Alistarh, and Petr Kuznetsov. Brief-announcement: Performance prediction of coarse-grained programs. In *Proceedings of the thirty seventh annual ACM Symposium on Principles of distributed computing (PODC)*, pages 411–413, 2018

In parallel with this doctoral work, the author was also involved in the following paper:

[1] Umut A Acar, Vitaly Aksenov, and Sam Westrick. Brief-announsement: Parallel dynamic tree contraction via self-adjusting computation. In *Proceedings of the twenty-ninth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 275–277. ACM, 2017

1.5 Roadmap

In Chapter 2, (1) we present different models of computation (Section 2.1); (2) we describe schedulers and complexity models for parallel programs written for dynamic multithreading (Section 2.2); (3) we provide definitions concerning concurrent data structures: correctness and progress guarantees, complexity models and data types (Section 2.3).

In Chapter 3, we describe state-of-the-art implementations of binary search tree, skip-list and priority queue, used for experimental analyses in Chapters 5 and 6.

Chapter 4 presents our automatic granularity control algorithm with the theoretical bounds on the execution time of parallel programs using it and with the rigorous performance analysis.

Chapter 5 presents the concurrency-optimal binary search tree.

Chapter 6 presents the parallel combining technique together with several applications.

Chapter 7 presents a proof of the fact that stack does not have a wait-free help-free implementation in systems with reads, writes, compare&swap and fetch&add primitives, correcting a mistake in an earlier proof by Censor-Hillel et al.

Chapter 8 presents a simple method that predicts the throughput of simple coarse-grained lock-based data structures.

We conclude in Chapter 9 with a discussion of open questions and future work.

2 Background: Models and Definitions

In this chapter we present models, definitions and notations that will be used throughout the thesis. Individual chapters will be provided with additional definitions and notations specific to the chapter.

2.1 Parallel Models

In each considered parallel model there are n processes $p_1, \ldots, p_n$ that communicate via *reads*, *writes* and, possibly, other primitives (defined explicitly) on shared atomic registers. Each process is provided with its own program.

2.1.1 Parallel Random Access Machine

Parallel Random Access Machine (PRAM) model was proposed by Fortune and Wyllie [71] as a simple extension of the Random Access Machine model used in the design and analysis of sequential algorithms. A global clock synchronizes the operations of processes: one operation (or skip) of each process is executed during one tick of the clock. The memory model of PRAM is the strongest known consistency model [10]: a write during clock tick t becomes globally visible to all processes in the beginning of clock tick $t + 1$.

There are several variations of PRAM model based on how the concurrent writes and reads are handled. The exclusive read exclusive write (EREW) PRAM does not allow any simultaneous access to a single memory location. The concurrent read exclusive write (CREW) PRAM allows for concurrent reads but not concurrent writes. The concurrent read concurrent write (CRCW) PRAM allows for both concurrent reads and writes. The CRCW models can be distinguished further: *common* allows concurrent writes only when all processors are attempting to write the same value; *arbitrary* allows an arbitrary write to succeed; and *priority* allows the process with the smallest index to succeed.

There are further variations of PRAM model. For example, QRQW PRAM [74] in which reads and writes are added into the queue and are served one per tick; and scan PRAM [26], a version of EREW PRAM in which scan (prefix sum) operation takes one tick of the global clock.

PRAM model is unique in that it supports deterministic parallel computation, and it can be regarded as one of the most programmer-friendly models available. Numerous algorithms have been developed for PRAM model, for example, see the book by JáJá [104]. Unfortunately, this model is not useful in practice since it induces high overhead on the synchronization of processes during each tick.

2.1.2 Bulk Synchronous Parallelism

The *bulk-synchronous shared-memory parallel (BSP)* model, proposed by Valiant [154], is an extension of PRAM. The computation is split into *supersteps*. Each superstep consists of a computation phase, during which the processes peform local computations, and a global interprocesor comunication phase, during which processes can share a data. After each superstep, the processes are synchronized using a barrier.

2.1.3 Asynchronous Shared Memory

An *asynchronous shared memory* model resembles the PRAM model, but the processes run asynchronously and all potentially conflicting accesses to the shared memory must be

resolved by the programmer. If we extend this model with a special instruction — the synchronization of the subset of processes, it becomes Asynchronous PRAM [73].

In this thesis, we use the Asynchronous PRAM model. Typically, in addition to standard primitives on shared atomic registers (reads and writes), we provide this model with more powerful ones such as *compare&swap* and *fetch&add*, etc.

Useful Primitives

The *compare&swap* primitive takes a target location, an expected value and a new value. The value stored in the location is compared to the expected value. If they are equal, then the value in the location is replaced with the new value and `true` is returned (we say that the operation is *successfull*). Otherwise, the operation *fails* (i.e., the operation is *failed*) and returns `false`.

The *fetch&add* primitive takes a target location and an integer value. The primitive augments the value in the location by the provided value and returns the original value.

2.2 Expressions of Parallel Programs

We consider two ways of expressing the parallel programs: for static multithreading or for dynamic multithreading.

2.2.1 Static Multithreading

For static multithreading the user writes a program for each process separately [50]. Typically, it is assumed that the number of processes executing the program is kept constant during the execution (note, that this number has to be less than n, the total number of processes).

2.2.2 Dynamic Multithreading

For dynamic multithreading, or nested fork-join parallelism, the programs are written using two constructions: *fork* that specifies procedures that can be called in parallel, and *join* that specifies a synchronization point among procedures [50]. Fork and join constructs can be nested, making this type of programs useful for divide-and-conquer algorithms.

An execution of programs written for dynamic multithreading can be modeled as a directed acyclic graph (dag) that unfolds dynamically. In this execution dag, each node represents a unit-time sequential subcomputation and each edge represents control-flow dependencies between nodes. A node that corresponds to a "fork" has two or more outgoing edges, and a node corresponding to a "join" has two or more incoming edges.

To execute a program some processes are chosen to work under guidance of a *scheduler* [2]. The scheduler is responsible for choosing which nodes to execute on each process during each timestep. It chooses for execution only *ready* nodes: unexecuted nodes whose predecessors have all been executed.

Schedulers

There exist two types of schedulers for dynamic multithreading: *offline* and *online*. *Online* scheduler works with a dynamically unfolding dag, while *offline* scheduler works with a pre-provided full computation dag. Typically, online schedulers are used in practice, since the execution dag is not known a priori and offline scheduler cannot be used. However, offline schedulers are easier to implement and analyse.

The commonly used offline schedulers [2] are *greedy* schedulers. A scheduler is *greedy* if it never leaves a process idle unless there are no ready nodes.

Online schedulers obey the following scheme. The algorithm maintains a *pool of work*, consisting of ready nodes. Execution starts with the root node in the pool. It ends when

the pool becomes empty. During the execution, a process withdraws a node from the pool, executes it and puts the ready successor nodes into the pool.

One of the most popular online scheduler is the *work-stealing* scheduler [33]. Each process maintains a *deque*, doubly ended queue, of nodes. Each process tries to work on its local deque as much as possible: a process pops the node from the bottom of its deque and executes it. When a node is executed, the process puts all ready successor nodes to the bottom of the deque. If a process finds its deque empty, it becomes a *thief*. A thief picks a *victim* process at random and attempts to steal a node from the top of the victim's deque. If the victim's deque is not empty, the steal attempt succeeds, otherwise, it fails.

Implementations

The support of nested parallelism dates back at least to Dijkstra's *parbegin-parend construct*. Many parallel languages support nested parallelism includling NESL [27], Cilk [72], the Java fork-join framework [70], OpenMP [34], X10 [47], Habanero [42], Intel Threading Building Blocks [100], and Task Parallel Library [112].

Algorithmic Complexity Model

Parallel algorithms are generally analyzed using Work-Span Framework [28]. Work W is the total number of nodes in the computation dag and span S is the length of the longest path in the dag.

There exist several theorems that bound the execution time of a parallel program with work W and span S with different schedulers.

The first one is the renown Brent's theorem [37].

Theorem 2.2.1 (Brent's theorem)**.** *With any greedy scheduler the execution time of program on n processes does not exceed $\lfloor \frac{W}{n} \rfloor + S$.*

The bound on the execution time with the work-stealing scheduler was provided by Aurora et al. [20].

Theorem 2.2.2. *With the work-stealing scheduler the expected execution time of program on n processes is $O(\frac{W}{n} + S)$. Moreover, for any $\epsilon > 0$, with probability at least $1 - \epsilon$, the execution time on n processes is $O(\frac{W}{n} + S + \log \frac{1}{\epsilon})$.*

Work and Span of PRAM Algorithm

In this thesis, we will talk about *work* and *span* of PRAM algorithms even though this notation is only applicable to programs written for dynamic multhreading. Here, we explain how to convert PRAM algorithm to its counterpart written to dynamic multhreding.

Instead of presenting the precise transformation algorithm, we explain how the computation dag should look like for each input (this should be enough to talk about *work* and *span*). At first, we execute the PRAM algorithm on that input and suppose that it takes T ticks. Our dag will have T layers of at most n nodes, the i-th layer contains nodes that correspond to the operations performed by processes during the i-th tick of the PRAM algorithm, and the nodes in the i-th level depend on the nodes in $i-1$-th level. Now, we can state that the *work* and *span* of the PRAM algorithm is *work* and *span* of that dag.

However, typically, the *span* of the PRAM algorithm, re-written for dynamic mutithreading, is $\log n$ times bigger. This is due to the fact that our dag uses a complex dependency between layers, while in practice we only can have nodes with a constant number of predecessors and successors. This replacement increases the *span* of the algorithm by $\log n$, while the work increases by no more than a constant.

An Example

To give an intuition behind the expression of parallel programs for dynamic multithreading and the use of the Work-Span Framework, consider an example: *exclusive scan operation*. Given an array a of s integers *exclusive scan operation* returns an array of length s such that its i-th value is the sum of the first $i-1$ values of a. Below, we provide a standard algorithm of exclusive scan written for dynamic multithreading.

In the pseudocode, instead of two constructions *fork* and *join*, we use one primitive *fork2join* that takes two functions, named *branches*, as arguments and executes them in parallel until their completion.

```
1  phase1(a, tmp, l, r):
2    if l = r - 1:
3      return a[l]
4
5    left ← 0
6    right ← 0
7    m ← (l + r) / 2
8    fork2join(
9      [&] {
10       left ← phase1(a, tmp, l, m)
11     },
12     [&] {
13       right ← phase1(a, tmp, m, r)
14     }
15   )
16   tmp[m] ← left
17   return left + right
18
19 phase2(tmp, ans, l, r, sum):
20   if l = r - 1:
21     ans[l] ← sum
22     return
23   m ← (l + r) / 2
24   fork2join(
25     [&] {
26       phase2(tmp, ans, l, m, sum)
27     },
28     [&] {
29       phase2(tmp, ans, m, r, sum + tmp[m])
30     }
31   )
32   return
33
34 exclusive_scan(a):
35   s ← a.length
36   tmp ← new int[s]
37   ans ← new int[s]
38
39   phase1(a, tmp, 0, s)
40   phase2(tmp, ans, 0, s, 0)
41
42   return ans
```

We briefly explain how the algorithm works. During the execution, we implicitly build a segment tree with height $O(\log s)$, i.e., a binary tree whose root corresponds to segment $[0, s)$, a node corresponding to a segment $[l, r)$ is either a leaf, if l equals to $r-1$, or a parent of two nodes corresponding to segments $[l, \frac{l+r}{2})$ and $[\frac{l+r}{2}, r)$. The algorithm consists of two phases. During the first phase, we calculate an array tmp of length s: suppose that $m = \frac{l+r}{2}$ is a split point of a segment $[l, r)$ in the segment tree, then $tmp[m]$ equals to $a[l] + \ldots + a[m-1]$ (note that each $i \in [1, n)$ is a split point of exactly one segment). During the second phase, we calculate the results array ans: $ans[m]$ is the sum of the values on the segments to the left of m.

The total work performed by the algorithm is $O(s)$: we visit each node in the segment tree $O(1)$ times and the number of nodes is $O(s)$. The span of the algorithm is $O(\log s)$ as the height of the segment tree. Thus, the expected execution time of the algorithm on n processes with work-stealing scheduler by Theorem 2.2.2 is $O(\frac{s}{n} + \log s)$.

2.3 Data Structures

A *data type* is a tuple $(\Phi, \Gamma, Q, q_0, \theta)$, where Φ is a set of operations, Γ is a set of responses, Q is a set of states, q_0 is an initial state and $\theta \subseteq Q \times \Phi \times Q \times \Gamma$ is a transition function, that determines, for each state and each operation, the set of possible resulting states and produced responses.

A *sequential implementation* (or *sequential data structure*) corresponding to a given data type specifies, for each operation, a sequential read-write algorithm, so that the specification of the data type is respected in every sequential execution.

A *batched version* [122] of a data type $\tau = (\Phi, \Gamma, Q, q_0, \theta)$ is specified by the transition function $\beta \subseteq \theta \cup \bigcup_{i=2}^{+\infty} Q \times \Phi^i \times Q \times \Gamma^i$ that determines, for each state and a batch of requests, the set of possible resulting states and produced responses for each request in the batch. In the simplest batched version of the data type the resulting state and the responses to the operations from the batch correspond to the sequential application of the operations in some order.

A *batched implementation* (or *batched data structure*) corresponding to a given data type τ specifies an algorithm of `apply` operation that applies a batch of operations, so that the specification of the batched version of τ is respected.

When `apply` operation is implemented as a parallel program, we call the batched implementation *parallel*.

A *concurrent implementation* (or *concurrent data structure*) corresponding to a given data type assigns for each process and each operation type a deterministic state machine that specifies the sequence of *steps* (primitives on the shared atomic registers) that the process needs to perform to complete the operation.

2.3.1 Corectness, Progress Guarantees and Complexity Model of Concurrent Data Structures

Correctness

Here, we describe common corectness criteria of concurrent implementations in the asynchronous shared memory model.

A *low-level history* (or an *execution*) is a finite or infinite sequence of primitive steps: invocations and responses of high-level operations, invocations and responses of primitives on the shared registers (reads, writes, etc.). We assume that executions are *well-formed*: no process invokes a new primitive, or high-level operation before the previous primitive, or a high-level operation, respectively, returns or takes steps outside its operation's interval.

An operation is *complete* in execution α if the invocation event is followed by a *matching* response; otherwise, it is *incomplete*.

A *high-level history* (or simply a *history*) of execution α on high-level object O is the subsequence of α consisting of all invocations and responses of operations on O.

Here we define three correctness criteria that appear in the thesis: quiescent consistency [21], linearizability [93] and set-linearizability [122].

Definition 2.3.1 (Quiescent consistency)**.** *A period of quiescence in a history h is a period between two consecutive events such that all invoked operations are complete.*

A history h defines a partial order on operations: op_1 preceeds op_2 ($op_1 \prec_h op_2$) if they are separated by a period of quiescence. A quiescence *given a history h is a sequential high-level history of operations Q such that:*

1. *Q consists of all the operations completed in h and, possibly, some operations that have started but not yet completed in h;*
2. *the operations have the same input and the same output as the corresponding operations in h;*
3. *for every two operations $op_1 \prec_h op_2$ if op_2 is included in Q, then op_1 preceeds op_2 in Q ($op_1 \prec_Q op_2$).*

A concurrent implementation of a data type is quiescently consistent if, for each of its histories, there exists a quiescence.

Definition 2.3.2 (Linearizability)**.** *A history h defines a partial order on operations: op_1 preceeds op_2 ($op_1 \prec_h op_2$) if op_1 is completed before op_2 begins. A* linearization *of a history h is a sequential high-level history of operations L such that:*

1. *L contains all the operations completed in h and, possibly, some operations that have started but not yet completed in h;*
2. *the operations have the same input and the same output as the corresponding operations in h;*
3. *for every two operations $op_1 \prec_h op_2$ if op_2 is included in L, then op_1 preceeds op_2 in L ($op_1 \prec_L op_2$).*

A concurrent implementation of a data type is linearizable if, for each of its histories, there exists a linearization.

Definition 2.3.3 (Set-linearizability)**.** *A* set-linearization *of a history h is a sequence S of batches of operations with set-linearization points such that:*

1. *a union of batches in S contains all the operations completed in h and, possibly, some operations that have started but not yet completed in h;*
2. *for any batch b from S, its set-linearization point lies between invocation and response of all operations of b in h;*
3. *the operations have the same input and the same output in h as if they are applied sequentially in batches of S.*

A concurrent implementation of a batched data type is set-linearizable if, for each of its histories, there exists a set-linearization.

In an infinite execution a process is *faulty* if it stops taking steps before completing its operation. A process which is not *faulty* is called *correct*.

Progress Guarantees

A method implementation is *non-blocking* if a failure or a suspension of any other process does not prevent other threads from making progress during the execution of this method. A method that is not non-blocking is called blocking.

Non-blocking. A method implementation is *wait-free* if it guarantees that every call of this method eventually completes. A method implementation is *bounded wait-free* if every call of this method completes in a bounded number of steps. We say that a data structure is wait-free if its methods are wait-free.

A method implementation is *lock-free* if it guarantees that some call of this method eventually completes. We say that a data structure is lock-free if its methods are lock-free. Wait-freedom is stricter than lock-freedom: a wait-free method is lock-free, but not vice versa.

We say that a method call is executed in *isolation* for a duration if no other threads take steps during that time. A method implementation is *obstruction-free* if it guarantees that every call completes if the corresponding thread executes in isolation for long enough.

Blocking. Blocking implementations can provide two progress guarantees: starvation-freedom and deadlock-freedom.

A method is *starvation-free* if it guarantees that every call of this method eventually completes if each process takes an infinite number of steps. We say that a data structure is starvation-free if its methods are starvation-free.

A method is *deadlock-free* if it guarantees that an infinite number of calls of this method completes if each process takes an infinite number of steps We say that a data structure is deadlock-free if its methods are deadlock-free. Note that starvation-freedom implies deadlock-freedom.

Complexity for Concurrent Data Structure

Two architecture paradigms are considered in the literature that allow shared variables to be locally accessed: the *cache-coherent (CC)* and the *distributed shared memory (DSM)* machines [18].

In a CC machine, each process has a private cache, and some hardware protocol is used to enforce cache consistency (i.e., to ensure that all copies of the same variable in different local caches are consistent). A shared variable becomes locally accessible by migrating to a local cache line. Memory references to the variable stored on the cache line in a process's cache are called *local* and are much faster than ones that has to load a cache line (i.e., cache misses), called *remote memory references (RMRs)*.

In a DSM machine, each processor has its own memory module that can be accessed without accessing the global interconnection network. On such a machine, a shared variable can be made localy accessible by storing it in a local memory module. An access to a memory location in a process's own memory module is *local* and a reference to another process's memory module is an *RMR*.

Complexity of concurrent data structures' operations, typically, cannot be expressed in steps: the standard spin-wait loop (the loop that spins on a local boolean variable) can take infinite number of them. Instead, complexity is measured in RMRs and, for example, the described spin-wait loop takes 1 RMRs. However, the algorithm can have different complexities on CC and DSM machines. For example, on DSM machine, the spin-wait loop on a non-local variable can take more than 1 RMRs.

2.3.2 Data Types Considered in This Thesis

Stack

Stack is a data type that maintains a multiset of values and supports two operations:

- `push`(v) — inserts value v into the set;
- `pop`() — extracts the most recently added element that was not yet extracted, or returns $\perp$ if the set is empty.

Queue

Queue is a data type that maintains a multiset of values and supports two operations:

- `enqueue`(v) — inserts value v into the set;
- `dequeue`() — extracts the earliest added element that was not yet extracted, or returns $\perp$ if the set is empty.

Set

Set is a data type that maintains a set of elements and supports three operations:

- `insert`(v) — inserts value v into the set, returns `true`, if and only if v is absent in the set;
- `delete`(v) — deletes value v from the set, returns `false`, if and only if v is present in the set;
- `contains`(v) — returns whether v is in the set or not.

Sets are generally implemented in two ways: using hash tables or some sort of balanced (tree-like) data structures, e.g., binary, B-, (a, b)- trees and skip-lists. In the thesis we are interested in binary search trees and skip-lists implementations. Such implementations support operations in logarithmic time. We discuss their state-of-the-art implementation in Section 3.1 and 3.2.

Priority Queue

Priority queue is a data type that maintains an ordered multiset and supports two operations:

- `insert`(v) — inserts value v into the set;
- `extractMin`() — extracts the smallest value from the set, or returns $\perp$, if the set is empty.

We discuss its state-of-the-art implementations in Section 3.3.

Dynamic graph

Dynamic graph is a data type that maintains a graph on N vertices and supports three operations:

- `insert`(u, v) — adds an edge between vertices u and v;
- `delete`(u, v) — removes an edge between vertices u and v;
- `isConnected`(u, v) — returns whether u and v are connected.

In the thesis, we use only one implementation of dynamic graph presented by Thorup et al. [95]: on a graph with N vertices insertion and deletion take $O(\log^2 N)$ amortized time, and connectivity query takes $O(\log N)$ time.

Dynamic forest

Dynamic forest is a special case of dynamic graph: the sequence of operations can be appled only if after each operation a graph is a forest, i.e., after each operation there exists no more than one simple path between any pair of vertices.

3 Overview of Data Structure Implementations

In this chapter, we overview the state-of-the-art sequential, parallel batched and concurrent implementations of set and priority queue data types. Most of the implementations described here are used to compare against our novel data structures, either empirically or theoretically. However, if you are not interested in the overview, you can safely skip this chapter.

We are primarily interested in set implementations based on binary search tree and priority queue implementations. Since the majority of priority queue implementations are based on skip-list, we also discuss set implementations based on skip-list.

3.1 Binary Search Trees

Binary search tree (BST) is a rooted binary tree, whose nodes each store a value (and, possibly, other information) and have two subtrees, commonly denoted *left* and *right* (the roots of these subtrees are called *left and right children*, respectively, if subtree is not empty). The tree additionally satisfies the *order property*: the value in each node is strictly greater than values in its left subtree and strictly smaller than values in the right subtree. The node is named a *leaf*, if it does not have any child. Otherwise, the node is called *inner*.

Three types of binary search trees are distinguished in literature: *internal*, *external* and *partially-external*. In an internal tree, the set of an internal tree consists of the values in all nodes. In an external tree, the set consists of the values in the leaves; and each inner node has two children and is called *routing*. In a partially-external tree, each node can be either *routing* or *data*; each routing node has exactly two children; and the set consists of the values in data nodes.

To provide logarithmic bounds for the operations, a binary search tree is augmented with a *balancing scheme*. In a few words, *balancing scheme* is an algorithm that explains how to restructure the binary tree after an update, so that a special condition, called an *invariant*, is satisfied.

Some of the schemes are:

- AVL scheme [9]. Invariant: for any node the heights of left and right subtrees differ by at most one.
- Red-Black scheme [81]. Each node has a color: red or black. Invariant consists of two parts: 1) no red node has a red child; and 2) the number of black nodes on every path from the root down to any leaf is the same.
- Weight-balanced scheme or BB[α] [123]. Let $w(T)$ be the number of nodes in tree T and let T_l be the left subtree of T. Invariant: for any node with subtree T, $\alpha \leq \frac{w(T_l)}{w(T)} \leq 1 - \alpha$.
- Treap scheme [138]. Each node has a uniformly random priority. Invariant: the priority of each node is not less than the priorities of its children.

One of the most important features of the presented trees is that they allow an operation *join* in logarithmic time. *Join* unites two trees, such that the maximal key of the first is less than the minimal key of the second.

3.1.1 Parallel Batched Implementations

The first parallel batched binary search tree was presented by Blelloch and Reid-Miller [29]: they used treap balancing scheme. The algorithm is implemented by a recursive function $\texttt{apply}(T, A)$ that returns a tree after applying operations A, sorted by the arguments, to tree T. $\texttt{apply}(T, A)$ splits the tree T by the argument of the middle operation $A[m \leftarrow \frac{l+r}{2}]$ to T_l and T_r, calls recursively in parallel $T_l \leftarrow \texttt{apply}(T_l, A[1, \ldots, m-1])$ and $T_r \leftarrow \texttt{apply}(T_r, A[m+1, \ldots, |A|])$, and, finally, returns the join of T_l, argument of $A[m]$ and T_r, if $A[m]$ is Insert, and return the join of T_l and T_r, if $A[m]$ is Delete.

It was shown that given the size of the tree k and the size of the batch m the algorithm is work-efficient, i.e., the work is $O(m \cdot \log(\frac{k}{m} + 1))$, and is highly parallel, i.e., the span is $O(\log^2 k)$. We called this algorithm work-efficient, because in order to apply a batch of size m to a tree of size k any sequential algorithm should make at least $\lceil \log \binom{k+m}{m} \rceil = \theta(m \cdot \log(\frac{k}{m} + 1))$ comparisons.

The second paper by Blelloch et al. [32] is a generalization of the previous one. They took the same algorithm and applied it to four different implementations of binary search tree (one for each balancing scheme described earlier). For all four algorithms the work is $O(m \cdot \log(\frac{k}{m} + 1))$ and the span is $O(\log k \cdot \log m)$, where k is the size of the tree and m is the size of the batch.

3.1.2 Concurrent Implementations

Lock-Based Partially-External BST by Bronson et al.

We start with the first *practical* balanced concurrent BST by Bronson et al. [39]. The tree is a partially-external BST and it uses relaxed AVL balancing scheme [35]. Any operation starts with a traversal that is implemented as a recursive function: it goes optimistically down the tree and the recursion saves for us a path of nodes that we traversed from the root. If someone has updated the links on the nodes in such a way that we cannot continue the traversal correctly, e.g., the current node is removed or there was a "bad" rotation, we have to backtrack (return from recursive calls) and find the lowest position on the traversed path from the root, saved by the recursion, from which we can continue the traversal down. After the traversal, an Insert or Delete operation properly updates the node and goes up the tree to fix every node on the path to the root using locks: remove routing nodes with one child and rebalance if necessary. Rebalance is performed using relaxed AVL scheme [35]: we optimistically calculate the difference in heights (possibly, stale) between the children; if the absolute difference is bigger than one, we lock necessary nodes (the node, the parent and the proper child) and perform the proper rotation; and, finally, we update the height of the node from the heights of its children. It is argued that when there is no pending operation, the tree is a strict AVL tree.

Non-Blocking External BST by Ellen et al.

The first practical lock-free BST was proposed by Ellen et al. [62]. The tree is implemented as an external BST. The blocking version of external BST is quite straightforward and Barnes's technique [24] is used to make the algorithm lock-free. Each node of a tree is expanded with a pointer *info* to *Info record*. Each operation starts with a simple traversal. When an operation reaches the effective node *node* the operation indicates that it wants to change *node*: it creates *Info record* and sets the pointer *info* to this record. *Info record* contains enough information to help complete the original operation so that the node can be unflagged. Typically, helping often leads to poor performance because several processes try to perform the same piece of work. In this implementation a process p helps another process's operation only if this operation prevents p's progress. Because of this policy traversals do not help at all.

Wait-Free Red-black Tree by Natarajan et al.

Natarajan et al. [121] presented a wait-free red-black tree. As a basis they took an external red-black tree with top-down operations by Tarjan [149]. To translate a sequential algorithm into wait-free concurrent one they used Tsay and Li's framework [152]. In this framework each operation holds a *window* (subtree) that moves down the tree. If the window of an operation intersects with some lower window of a different operation, the process tries to move the lower window first. To move a window a process copies the subtree from the window, updates the copy, links the copy instead of the original subtree and, finally, moves a window lower. The straightforward application of Tsay and Li's framework is very expensive, because of that some optimizations are proposed which we do not discuss here.

Non-Blocking Internal BST by Howley et al.

Howley et al. [97] presented an internal lock-free binary BST. They used the similar technique of *Info record*s as in BST by Ellen et al. [62]. Nevertheless, this BST is more complicated to implement: the traversal should be accurately taken care of since the desired value can be moved up the tree; furthermore, a delete operation now has to search for the replacement value. Traversals and insertions in Howley and Jones's algorithm are generally faster than in Ellen et al.'s algorithm: the traversed path in an external tree terminates at a leaf node while in an internal tree it may terminate at an internal node. However, delete operations in an internal tree are generally slower than those in an external tree.

Lock-Based Partially-External BST with Helper Thread by Crain et al.

Crain et al. [52] presented a contention-friendly lock-based partially-external BST with relaxed AVL balancing scheme. Given that the traversal is the longest part of the operation concurrent BST should avoid unnecessary verifications and backtracking which happen to be an issue of the previous balanced BST by Bronson et al. Crain et al. simplified an implementation: the operations are decoupled from structural adaptations, e.g., physical removal and rebalancing. These adaptations are performed in a separate daemon thread that continually recursively restructures the tree. To get rid of verifications during the traversals the daemon thread performs restructures as follows: 1) to remove a routing node it takes locks on the node and its parent, reroutes pointers to the children to point onto the parent and marks the node as deleted; 2) to rebalance a node it takes necessary locks (on the node, its parent and the proper child), replaces a current node with a new node and marks the current node as deleted. By that all operations on a data structure become short and lightweight.

Lock-Based Internal BST with Logical Ordering by Drachsler et al.

Drachsler et al. [57] proposed a practical lock-based internal concurrent BST with relaxed AVL balancing scheme. Their main goal was also to reduce overheads during the traversals. For that they stitch the nodes of BST in sorted order with a linked list. Now, to find a node with the given value in BST an operation starts a wait-free traversal, stops at some node, and then moves to the left or to the right of this node in the linked list. Note that the overhead of any modification increases since the algorithm in addition to updating links of the tree updates links in the linked list.

Lock-Free External BST with Low Overhead by Natarajan et Mittal

Natarajan and Mittal [120] proposed an external lock-free BST as an improvement of BST by Ellen et al. We describe three key ideas of this improvement. At first, the algorithm

operates on edge-level (marks edges) whereas Ellen et al.'s algorithm operates at node-level (marks nodes). This leads to the smaller contention window: any two operations that can be executed concurrently in Ellen et al.'s algorithm can also be executed concurrently in Natarajan et al.'s algorithm, but not vice versa. For example, the algorithm by Natarajan et al. allows setting two children of a node independently. The second improvement is that *Info record*s are not used, instead only edges are being marked, and the help is performed only for delete operations (insertions do not need help). Finally, they optimized the operations: the number of allocated objects and the number of executed atomic instructions are less than in lock-free algorithms by Ellen et al. and Howley et al.

Lock-Free External Balanced BST by Brown et al.

Brown et al. [41] presented a lock-free balanced BST based on an external chromatic BST [125], which is a relaxed version of the red-black tree. To implement BST they introduced generalized versions of LL and SC primitives: LLX and SCX [40] which operate on *Record*s consisting of a fixed number of mutable and immutable fields. LLX(r) attempts to take a snapshot of the mutable fields of a Record r. SCX(V, R, fld, new) takes as arguments a sequence V of Records, a subsequence R of V, a pointer fld to a mutable field of one Record in V, and a new value new for that field. SCX tries to atomically store the value new in the field that fld points to and to finalize each Record in R, i.e., make all fields in these Records to be immutable. A traversal in this algorithm is performed in a standard wait-free manner by following links from the root. Insertions and deletions use LLX and SCX (on two and three Records, correspondingly) to update the nodes, then they start a cleanup phase: continually rebalance the first unbalanced node on the path from the root to the position of the value. Brown et al. proved that the height of the tree at any time is $O(c + \log k)$, where k is the size of the tree and c is the number of updates in progress.

3.2 Skip-Lists

A skip-list data structure was proposed in 1990 by Pugh [135]. It maintains a collection of sorted linked lists, which mimics, in a subtle way, a balanced search tree.

Each list has a level, ranging from 0 to a maximum (constant `MAX_LEVEL`). Each value has a corresponding node that is linked into a subset of lists. The bottom-level list contains all nodes, and each higher-level list is a sublist of the lower-level lists. The higher-level lists are *shortcuts* into lower-level lists.

The structure itself is randomized: the node is created with a random *top level* (or simply *level*), and belongs to all lists up to that level. Top levels are chosen in such a way that the expected number of nodes decreases exponentially with the increase of level. For example, if the probability of the node to appear at level i is $\frac{1}{2^i}$, then, roughly speaking, each link at level i skips 2^i nodes in the bottom-level list.

To simplify implementations of skip-lists we prepopulate them with head (value $-\infty$) and tail (value $+\infty$) sentinel nodes that appear in all the lists. Each node has an array `next` of references: one reference for each list the node belongs to.

In the algorithms we use the following additional notations: in a sorted linked list, the node with the biggest value smaller than value v is called the *predecessor* of v, while the node with the smallest value bigger than value v is called the *successor* of v. Also, we sometimes call the *top level* of a node simply as the *level* of a node.

The sequential algorithm works as follows. Given value v each operation starts with the traversal. The traversal starts from the top-level at sentinel node *head* and goes down the levels one after the other until it reaches the predecessor of v at the bottom level: at each level i, the algorithm takes the last visited node and follows `next` pointers until it reaches the predecessor of v; we store the found predecessor in `preds[i]` and the successor of v in `succs[i]`. Then `insert`, if a node with value v does not exist (i.e., value

in preds[0].next[0] is not v), creates a new node `node` (with random top level as described above) and links it from bottom up to its level: `preds[i].next[i]` is set to the new node and `node.next[i]` is set to `succs[i]`. And `delete`, if a node `node` with value v exists (i.e., value in preds[0].next[0] is v), unlinks `node` at each level up to the level of `node`: `preds[i].next[i]` is set to `succs[i].next[i]`.

3.2.1 Concurrent Implementations

Lock-Based Skip-List by Pugh

The first concurrent lock-based skip-list was presented by Pugh [134]. At first, he described a lock-based linked list. To find a position to insert or delete, an operation with argument v does a lock-free lookup for the node `pred` of v by following `next` pointers. Then the operation takes a lock on the next pointer of `pred`, and using hand-over-hand locking follows `next` pointers to adjust the predecessor `pred` of v. `Insert` operation creates a new node and links it after `pred` in the list. `Delete` operation takes an additional lock on the next pointer of the node `node` with value v, sets the next pointer of `node` to `pred` and unlinks `node`.

The skip-list algorithm works similarly to the described linked list. An operation with an argument v traverses the skip-list down to the bottom level using the sequential algorithm and stores the predecessors `preds` at each level. `Insert` creates a new node with level ℓ and iterates through all the lists from level 0 to level ℓ: at each level i, it finds the real predecessor of v using the estimation `preds[i]` (as in lock-based linked list algorithm) and links the new node. `Delete` finds a node `node` with value v and iterates through all the lists from the level of the node downto level 0: at each level i, it finds the real predecessor using the estimation `preds[i]` (as in lock-based linked list algorithm), sets the next pointer `node[i].next` to the predecessor and unlinks `node`.

Lock-Free Skip-List by Sundell and Tsigas

Sundell and Tsigas [144] presented the first lock-free skip-list. They extended nodes with: (1) a reference `prev` that points to some predecessor (not necessarily, exact) at some level if the node is under deletion; and (2) an integer `validLevel` that indicates the current level up to which the node is linked. Also, for each node, each pointer in `next` array is now extended with a boolean mark that indicates whether the node is under deletion.

The algorithm uses auxiliary functions `scan` and `helpDelete`. Given a node `node`, a level and a value v, `scan` follows `next` pointers from `node` on that level until it finds the predecessor of v, and at the same time it helps to delete nodes with the marked `next` pointer at that level using `helpDelete`. The helping function `helpDelete`, given a node with value v and a level ℓ, deletes the node and returns the current predecessor of v at level ℓ: mark all `next` pointers from top-level up to level ℓ; then find a current predecessor of v: traverse either from the node at `prev` pointer or from the head, if `prev` has a small level; and, finally, unlink a node at level ℓ.

Each operation with an argument v starts with the traversal: start at the top level, then, at each level, the traversal calls `scan` function given the predecessor from the previous (upper) level. An insertion then creates and links a new node: at each level from bottom to top, find a current predecessor of v using `scan` on the estimation from the traversal and link the new node. A deletion sets `prev` pointer to the estimated predecessor of v at level $node.level/2$ from the traversal, marks the `next` pointers from top to bottom and, finally, unlinks a node: at each level from top to bottom, find a current predecessor of v using `scan` given the estimation from the traversal and unlink the node. Note that the traversal call in `contains` can skip the help to delete marked nodes.

Lock-Free Skip-List by Fomitchev and Ruppert

Fomitchev and Ruppert [69] designed a lock-free skip-list similar to the lock-based skip-lit by Pugh. At first, they present a lock-free linked list with the bounds on the running time of operations. They augment nodes with a backlink pointer and augment each `next` pointer with two bits: *flag* and *mark*. *Mark bit* is used to register the deletion of the node, while *flag bit* is a warning that a deletion of the next node is in progress. Flag bit is introduced only to ensure the bounds on the running time.

Each operation starts with the traversal that finds a predecessor `pred` of an argument v: start from the head, follow `next` pointers and help to delete marked nodes on its way. `Insert` then creates a new node and tries to link it. If the attempt to link is unsuccessful then: (1) if the `next` pointer of `pred` is flagged, then the operation help the concurrent delete operation; then, (2) while the `next` pointer of `pred` is marked the operation follows the backlink pointers; finally, (3) the operation adjusts `pred`. `Delete` takes node `node` with value v and attempts to perform four steps: 1) flag `prev.next`; 2) set the backlink of `node` to `pred`; 3) mark `node.next`; 4) unlink `node` and unmark `node.next`. If the attempt is unsuccessful the operation helps the concurrent delete operation, adjusts `prev` and retries.

This lock-free linked list is a basic building block of the proposed lock-free skip-list. This skip-list has a non-standard representation of a node: a node is represented with a linked list (*a tower*) from the top-level subnode (`tower_root`) to the bottom-level subnode. Each subnode stores a successor (`next` pointer) in the level-list, a subnode below and a link to `tower_root`. Additionally, `tower_root` is extended with the value field, other subnodes do not contain values. Each operation starts with the traversal: the standard traversal algorithm that helps to delete subnodes from top to bottom of a node which `tower_root` marked for deletion. `Insert` links a new tower from bottom to top: at each level i, it inserts a subnode in the linked list trying with `preds[i]` as the predecessor. `Delete` unlinks a tower from top to bottom: at each level i, it deletes a subnode from the linked list trying with `preds[i]` as the predecessor.

Lock-Based Lazy Skip-List by Herlihy et al.

Herlihy et al. [92] proposed a simple lock-based "lazy" skip-list. Each node is extended with a mark that indicates whether the node is under deletion. Each operation with an argument v starts with the standard traversal that finds the predecessors `preds` and the successors `succs` of v.

`Insert` then tries to lock the predecessors at each level from bottom to top. When all the locks are taken successfully, the algorithm checks that: (1) the predecessors are not marked; and (2) for each level i, `preds[i].next` still equals to `succs[i]`. If the conditions are satisfied, the insertion creates a new node and links it from the bottom level; otherwise, it restarts from the beginning.

`Deletion` locks the node `node` with value v and marks it. Then it locks the predecessors at each level from bottom to top and checks: (1) the predecessors are not marked; and (2) for each level i, `prev[level].next` still equals to `node`. If the conditions are satisfied, the deletion unlinks a `node` from top to bottom; otherwise, the operation restarts.

Lock-Free Skip-List by Herlihy et at.

The last skip-list implementation described here is a simple lock-free skip-list by Herlihy et al [88]. `Next` pointers are augmented with a mark that indicates whether the node is under deletion. Each modification operation with an argument v starts with the traversal: the standard traversal algorithm that at the same time unlinks all the traversed nodes with a marked next pointer. `Insert` creates a new node and links it from bottom to top, but if at some level the real predecessor and successor of v are not the same as found by the previous traversal — the traversal is repeated. `Delete` goes from top to bottom and

only marks the next pointers of a node with value v obtained from the traversal. If it marked the `next` pointer at the bottom level it calls the traversal to physically delete the node. `Contains` function uses the standard traversal that does not modify anything.

3.3 Priority Queues

3.3.1 Sequential Binary Heap

We start with the simplest sequential priority queue and its simple modification. This implementation (or its modification) is a basis of various priority queue implementations, either parallel batched or concurrent, including a novel parallel batched implementation presented in Section 6.4.

A binary heap of size m is represented by a complete binary tree with nodes indexed from 1 to m. Each node v has at most two children: $2v$ and $2v+1$ (to exist, $2v$ and $2v+1$ should be less than or equal to m). For each node, the *heap property* should be satisfied: the value stored in the node is less than or equal to the values stored in its children.

The heap is represented with *size* m and an array a where `a[v]` is the value at node v. At first, we describe the classical binary heap algorithm [50]:

- `extractMin()` records the value $a[1]$ as a response, copies $a[m]$ to $a[1]$, decrements m and performs the *sift down* procedure to restore the heap property. Starting from the root, for each node v on the path, we check whether value $a[v]$ is less than values $a[2v]$ and $a[2v+1]$. If so, the heap property is satisfied and we stop the operation. Otherwise, we choose the child c, either $2v$ or $2v+1$, with the smallest value, swap values $a[v]$ and $a[c]$, and continue with c.

- `insert`(x) increments m, sets $a[m]$ to x and performs the *sift up* procedure to restore the heap property. Starting from the node m, for each node v on the path to the root, we check whether value $a[v]$ is bigger than value $a[v/2]$ in the parent. If so, the heap property is satisfied and we stop the operation. Otherwise, we swap $a[v]$ with $a[v/2]$ and continue with $v/2$.

An important modification of that algorithm was presented by Gonnet and Munro [75]. `extractMin()` remains the same while `insert`(x) works from top to bottom. `insert`(x) initializes a variable *val* with x, increments m and traverses the path from the root to a new node m. For each node v on the path, if $val < a[v]$, then the two values are swapped. Upon reaching node m the operation sets $a[m]$ to *val*.

3.3.2 Parallel Batched Implementations

Parallel Batched Heap By Pinotti and Pucci for CREW PRAM

The first parallel batched priority queue was presented by Pinotti and Pucci [132]. The algorithm works in CREW PRAM with n processes. The priority queue is represented by a complete binary heap and each node contains a sorted array of n values. The arrays in nodes satisfy extended heap property: the minimal value in a node is bigger or equal than the maximal value in the parent. This data structure supports an insertion of n values and an extraction of n minimums.

To insert n values a new node is created, the provided n values are preliminarily sorted using Cole's algorithm [49] in $O(\log n)$ time and merged with all values in the ancestors of the new node using Kruskal's merging algorithm [106] in $O(\log m + \log\log n)$ time, where m is the number of nodes. Then the sorted array is distributed between the nodes on the path from the new node to the root.

To extract n values, at first, an array in the root is saved as a response and is replaced with an array from the last node. Then we construct the minimal path μ: suppose we are now at node v, we add the child c with the smallest maximal value to μ and continue with

c. For each node v in μ, we merge an array in v with an array in its sibling, split this array into halves, give the smallest half to v and the other half to the sibling in $O(\log n)$ total time. Then we merge arrays of all the nodes v on a path μ and distribute the resulting array to the nodes at μ.

Summarizing, insertions and extractions in batches of size n take $O(\log m + \log\log n)$ time, and $O(n \cdot (\log m + \log\log n))$ work where m is the number of nodes.

Parallel Batched Heap by Deo and Prasad for EREW PRAM

Next we discuss a parallel batched priority queue by Deo and Prasad [56]. The algorithm is described for EREW PRAM with n processes. As in the previous algorithm the priority queue is represented by a complete binary tree where each node contains n sorted values and satisfies extended heap property.

An insertion of $c \leq n$ keys works similarly to the sequential algorithm by Gonnet and Munro. At first, the operation sorts arguments using Cole's algorithm and stores them in a new array x. The operation starts at the root and traverses towards the last leaf: at each node the array at the node is merged with array x; then, first n values are stored in the node, while x is populated with the values left.

An extraction of $c \leq n$ minimums retrieves the first c values from the root as the answer, then replaces them with c values from the last leaf. Then the operation starts a traversal from the root towards some leaf. Suppose the traversal is currently at node *curr* and let *child* be the child with the biggest maximal value. The array at *curr* is merged with arrays in children; the smallest n values are stored in *curr*, the next n values are stored in *child*; and the traversal continues with the other child.

Thus, a batch of size $c \leq n$ can be applied in $O(\log m \cdot \log n)$ time and $O(n \cdot \log m \cdot \log n)$ work where m is the number of nodes. In the same paper, they discuss how the pipelining approach can reduce time to $O(\log m)$. Unfortunately, this idea can be applied only if there is a continuous flow of batches to apply.

Parallel Batched Priority Queue by Sanders et al. for Asynchronous PRAM

We finish the survey of parallel batched priority queue with the algorithm by Sanders [137]. The algorithm is designed for asynchronous shared memory model with n processes. Each process has its own queue which is split into two parts: a sorted array and a heap.

An insertion of $c \leq n$ values randomly chooses c out of n local queues and inserts each value into the array of the chosen queue. Once in $\log n$ insertions to a local queue the values from the array are flushed into the heap. By that the size of the array in any local queue does not exceed $\log n$.

The algorithm to apply a batch of $c \leq n$ `extractMin` operations is more involved. At first, each process operates on its own local queue: move the smallest values one-by-one from the heap to the array until all arrays of local queues together contain c smallest values (they can also contain other values). Then the smallest c values from the union of arrays are found using the algorithm similar to quicksort and, finally, these values are returned.

An application of a batch of $c \leq n$ `insert` operations takes $O(c \cdot \log m)$ work and $O(\log m)$ span, while an application of a batch of $c \leq n$ `extractMin` operations takes $O(n \cdot \log m)$ work and $O(\log m \cdot \log n)$ span, where m is the size of the priority queue.

3.3.3 Concurrent Implementations

Lock-Based Heap by Hunt et al.

We start with the coarse-grained heap-based priority queue by Hunt et al. [99]. They used an algorithm similar to the sequential binary heap that we described earlier. At first, an operation takes a global lock, changes the number of nodes m, calculates $\bar{m}$ as a bit-reverse

of m (the least significant becomes the most significant and etc.), locks node $\bar{m}$ and, if the operation is `extractMin`, also locks root, and, finally, releases the global lock.

`Insert` writes a value into $a[\bar{m}]$ and sifts up from $\bar{m}$. To guarantee correctness the operation uses hand-over-hand locking: takes the lock on the parent, then, possibly, swaps values, releases the lock on the node and continues with the parent.

`ExtractMin` swaps $a[1]$ with $a[\bar{m}]$, unlocks node $\bar{m}$ and sifts down. To guarantee correctness the operation uses hand-over-hand locking: takes a lock on both children, then, possibly, swaps values with one of them, releases the lock on the node and continues with the proper child.

Priority Queue from Skip-Lists

Typically, all concurrent skip-lists algorithms can be transformed into concurrent priority queue implementations in a manner firstly proposed by Lotan and Shavit [139]. They took the concurrent skip-list by Pugh [134] and augmented it with a `findMin` function (a subfunction of `extractMin`) that simply goes through the bottom level searching for a non-marked node. When such node is found the operation performs its deletion.

It is argued that such an implementation is not linearizable, but quiescently consistent: suppose that the skip-list contains only one value 2 and the first process performing `findMin` is currently at that node; then the second process inserts 1, inserts 3, and marks 2 for deletion; the first process wakes up and finds 3 which never was a minimum. To ensure the linearizability they added the timestamp for each operation and `findMin` function only searches for the nodes that were inserted before the `extractMin` operation starts.

Next, Sundell and Tsigas proposed a lock-free priority queue [145] which is based on their lock-free skip-list [144]. Finally, the lazy lock-based skip-list by Herlihy [92] and the lock-free skip-list by Herlihy [88] were transformed into priority queues in [90]. We name them Lazy and SkipQueue, correspondingly.

Lock-Free Priority Queue by Johnson and Linden

The concurrent priority queues obtained from concurrent skip-lists in a manner described above are subject to severe contention: several processes can try to mark the same node at the same time, where the losers proceed together trying to mark the next node and so on; plus the nodes to be physically removed are likely to be neighbours and, consequently, are likely to share predecessors, thus, leading to contention.

Johnson and Linden addressed the issue with contention during physical removals and presented a lock-free priority queue [113]. A key idea is: instead of physically removing each logically deleted node separately, we remove them in batches, i.e., several at a time. As a basis for priority queue they take a skip-list similar to the lock-free skip-list by Herlihy [88]. As usual the nodes can be logically and physically deleted. The major distinction from the previous skip-list algorithms, is that the logically-deletion flag is stored not in the next pointer of the node, but in the next pointer of its predecessor. By that the algorithm can guarantee that logically deleted nodes represent a prefix of the skip-list.

Given a value v an insertion starts with a traversal that finds, at each level i, the first not logically deleted node `succs[i]` with the value bigger than v and its predecessor `preds[i]`. Since logically deleted nodes always form the prefix, this traversal does not differ much from the standard one. Then the insertion creates a new node and inserts it starting from the bottom: at each level i, inserts between `preds[i]` and `succs[i]` using compare&swap, if unsuccessful then restarts the traversal to get new predecessors.

An extraction of minimum iterates through the lower level in order to mark the first not logically deleted node. When found, the node is marked and the number of traversed nodes is compared to the threshold. If it exceeds some predefined threshold, the next

pointers from the head node are changed in order to unlink the prefix of logically deleted nodes.

Lock-Based Priority Queue with Elimination and Combining by Calciu et al.

The last data structure still suffers from the contention during the marking of the nodes. Calciu et al. [43] tried to resolve this issue by splitting the queue into two parts: sequential and concurrent skip-lists. The first skip-list contains values smaller than a threshold and is accessed through a request buffer. The requests from the buffer are performed sequentially by a dedicated server thread. The second skip-list stores all other values and is instantiated as some concurrent skip-list.

`Insert` decides in which skip-list to insert: if a value is bigger than the threshold it is inserted into the concurrent skip-list, otherwise, the operation lefts a request in the buffer to the sequential skip-list. `ExtractMin` simply publishes a request in the buffer to the sequential skip-list. After an operation is performed the threshold can be adjusted in order to improve performance: the sequential skip-list should be not very small and also not very big.

Lock-Free Cache-Friendly Priority Queue by Braginsky et al.

Braginsky et al. [36] choose some lock-free skip-list implementation and make a node contain not one value but an array of values with the size of one cache line. The skip-list is logically split into two parts: the first node, which array is sorted, and the others.

`Insert` a node `curr` in the skip-list that should contain the argument and tries to insert it. There are two cases: `curr` is the first node or not. In the first case, the operation adds its request into a special buffer associated with the first node. Then the operation *freezes* `curr`, creates a new node with the sorted values from `curr` and the buffer, and tries to replace `curr` with a new node in the skip-list. In the second case `Insert` appends the value to the array in `curr` using one fetch&add instruction. In both cases, if the array becomes full the operation *freezes* node `curr` to split it into two. The freeze procedure is implemented in such a manner that other operations working on `curr` help to unfreeze it.

`ExtractMin` tries to take the first element from the first node using one fetch&add instruction. If the first node becomes empty, the operation freezes the first node and the second node in order to sort values in the second node and move them to the first node.

Lock-Based and Lock-Free Priority Queues by Liu and Spear

We finish the overview of concurrent priority queues with Mounds by Liu and Spear [114]. This priority queue is represented by a binary tree with a sorted list of values in each node and extended heap property: the smallest value at a node is less than or equal to the smallest values in the children (note that this extended property differs from the one appeared before).

`Insert(`v`)` starts with several attempts to find a node with the first value bigger than v: take a random leaf and find the highest node on the path to root that has the first value bigger than v using binary search. If such node is not found, the binary tree is extended with a new level of nodes and the operation restarts. The operation simply finishes by inserting v at the beginning of the list of the found node.

`ExtractMin` operation removes the first value from the list in the root and marks the root as dirty. All dirty nodes have to be *moundified*, or in other words they have to be *sift down* that compares the first values of the lists.

Two implementations of this data structure were described: lock-based and lock-free. Lock-based implementation simply uses hand-over-hand locking as the priority queue by Hunt et al. In lock-free implementation operations help each other to moundify nodes they are working on.

4 Automatic Oracle-Guided Granularity Control

4.1 Introduction

At the very beginning of the era of multicore processors parallel programs were written for *static multithreading*: each process is provided with its own program. These programs are written as a composition of *supersteps*. During a superstep, each process performs independent local computations. After each superstep, the processes synchronizae to accumulate the results of their independent computations. When the number of processes exceeds the number of cores, the programmer has to think about low-level execution details, such as scheduling the processes onto cores and controlling the overhead spent on the synchronization between processes.

The intricacy of these low-level details has motivated interest in *dynamic multithreading* (also known as *implicit parallelism*). Dynamic multithreading seeks to make parallel programming easier by delegating the tedious details, such as tasks scheduling, to compiler or run-time system. Much work has been done in that area resulting in various implementations: OpenMP [34], Cilk [72], Fork/Join Java [70], Habanero Java [42], X10 [47], Intel TBB [100], NESL [27], TPL [112], parallel ML [103] and parallel Haskell [46].

Most of these implementations provide the opportunity for parallelism via lightweight language constructs. In this work, we will use Cilk Plus [101] runtime system that provides just two keywords: `spawn` and `sync`. `Spawn` indicates a computation that can be executed in parallel and `sync` indicates a computation that must be synchronized with. These keywords are enough to express common parallel constructs such as nested parallelism and parallel loop [115].

As a classic application of implicit parallelism, consider a `map` function that takes an array a and a function f, and returns an array b with $b_i = f(a_i)$. Using templates (a powerful feature of `C++`) our `map` implementation (Figure 4.1) enables different forms of mapping. For example, a can be an array of integers and f can be an increment. Alternatively, a can be an array of vectors from $\mathbb{R}^n$ and f can be a "dot product" with vector specified beforehand. As can be seen, templates simplify a life of the programmer: it is unnecessary to write the separate code for each type of a and f. However, as discussed further this unification introduces a new obstacle on the path to good performance.

Even though implicit parallelism allows writing high-level code, it is non-trivial to make the code perform well. During the execution of a parallel program the run-time system induces overheads related to creation and scheduling of tasks. These overheads can be large enough to wipe out benefits of parallelism. For example, the simple `map` example above could run as much as 10-100x slower than an optimized implementation (Section 4.2).

There exists two complementary types of approaches to reduce the overheads of parallelism. An approach of the first type (e.g., [107]) optimizes the run-time system itself, for example, by reducing overheads spent per task creation, and does not require any changes to the code. An approach of the second type (e.g., [100]) controls *granularity*. *Granularity control* requires the programmer to *tune* the code in order to ensure that only big enough computations are parallelised and for small and only small computations *an alternative sequential algorithm* is executed. By that, the total overhead spent by run-time system takes a small fraction of the total execution time.

As an example of granularity control, imagine our `map` function that takes an integer array as a and an increment function as f. To perform this operation efficiently, the pro-

```
template <F, T, S>
map(F f, T* a, S* b, int n):
  map(f, a, b, 0, n)
  return


template <F, T, S>
map(F f, T* a, S* b, int l, int r):
  if r - l = 1:
    b[l] ← f(a[l])
    return


  mid ← (l + r) / 2
  spawn [&] { map(f, a, b, l, mid) }
        [&] { map(f, a, b, mid, r) }
  sync
  return
```

Figure 4.1: Parallel map, naive implementation.

```
template <F, T, S>
map(F f, T* a, S* b, int n):
  map(f, a, b, 0, n)
  return

int grain = ... // to be determined
template <F, T, S>
map(F f, T* a, S* b, int l, int r):
  if r - l ≤ grain:
    for i in l..r - 1:
      b[i] ← f(a[i])
    return

  mid ← (l + r) / 2
  spawn [&] { map(f, a, b, l, mid) }
        [&] { map(f, a, b, mid, r) }
  sync
  return
```

Figure 4.2: Parallel map, coarsened implementation.

grammer should group operations on elements into blocks that are executed sequentially and that are large enough to amortize the cost of parallelism. Figure 4.2 shows such a tuned implementation of `map`, where `GRAIN_SIZE` determines the block size.

The only thing left unspecified is how to choose the setting for grain-size constant `GRAIN_SIZE`. While it may appear simple, this process turns out to be a challenge, because the optimal setting for `GRAIN_SIZE` depends on the the architecture as well as the specific inputs [25, 55, 66, 94, 136, 153]. The programmer has to perform the tuning process, i.e., repeatedly execute the program with different inputs so that the right setting can be found [100, 153].

In simple parallel programs, such as `map` discussed above, grain-size constants are independent from each other and can be tuned separately. However, when programs become more complicated, for example, by using the nested parallelism: the grain-size constants that correspond to recursive calls become dependent. The dependency between the constants greatly complicates the tuning process.

Furthermore, for generic code, such as `map`, it can be impossible to perform such tuning, because grain-size constants depend on the template parameters as well as the actual arguments.

These issues emerge an interest in an automatic algorithms that help the programmer to control granularity [55, 98, 130, 155]. However, all these approaches fall short to deliver strong theoretical and practical efficiency guarantees. A more recent paper on automatic oracle-guided algorithm [5] show that strong bounds can be achieved if we are provided with an oracle that can predict the execution time precisely. The authors present an implementation of such an oracle, but this implementation cannot be used in programs with nested parallelism. This limitation excludes nearly all interesting parallel programs expressible in high-level parallel programs.

In this chapter, we present provably and practically efficient automatic granularity control technique for the class of nested parallel programs. To use our technique the programmer is asked to provide an "asymtotic" cost function of each piece of parallel code. Our novel online algorithm works as follows: (1) uses these cost functions to predict the expected execution time of that code on one process; (2) uses this prediction to decide whether to execute the parallel or sequential version of the code.

We prove that our algorithm has the desired theoretical properties, i.e., the overhead

on synchronization is small in comparison to the total execution time, for a broad class of parallel programs. We implement the algorithm as a C++ library that extends Cilk Plus [101] and compare it against the hand-tuned code from the PBBS suite [142]. As the results show, our automatic approach to granularity control can eliminate the need for hand-tuning in many cases.

To the best of our knowledge, this is the first result that provides the provable and practically efficient automatic granularity control.

Roadmap

In Section 4.2, we present several challenges of the granularity problem and present the high-level idea behind our solution. In Section 4.3, we present our online algorithm for automatic granularity control. This algorithm is fully online and does not require additional tuning or specific compiler support. Then we present end-to-end tight bounds on the execution time of a parallel algorithm that uses our algorithm. In Section 4.4, we provide the proofs of the bounds. In Section 4.5, we present the results of the evaluation on a broad collection of hand-tuned benchmarks. In Section 4.6, we discuss the related work. We conclude in Section 4.7.

4.2 Overview

We present an overview of the challenges of the granularity problem and our proposed solution.

Example Program

Consider a simple, data-mining problem: given an array of elements of type `T`, find the number of elements that satisfy a given predicate `p`. Using C++ templates, we specify such a generic function as follows:

```
template<T, P>
match(T* lo, T* hi, P p): int
  ...
```

Because `match` is generic, we can set `T` to be `char`, and define the predicate to be a function (a C++ lambda function [143, Sec.11.4]) that tests equality of the character to '#' as follows.

```
p = [&] (char* c) { return *c = '#' }
```

Similarly, we can perform matches over arrays whose elements are 1024-character strings, by setting `T` to be `char[1024]`. For example, we may count the strings whose hash code matches a particular value, say 2017, by instantiating the predicate as follows.

```
p = [&] (T* x) { return hash(x, x + sizeof(T)) = 2017 }
```

A classic way to implement a parallel `match` is to divide the input array into two halves, recursively call on each half, and compute the sum of the number of occurrences from both halves. Figure 4.3 shows the code for such an implementation, where `match` takes as arguments the input array, specified by a reference `a` and length `l`, and a predicate function, `p`. To control granularity, we stop the recursion when the input contains fewer than `grain` elements and switch to a fast sequential algorithm, `match_seq`. When the input array is large, it is divided in half and solved recursively in parallel using `fork2join`.

Choice of the Grain Size

To ensure good performance, the programmer must choose the setting for `grain`; but what should this setting be? The challenge is that there is no a priori suitable setting for `grain`,

Type `T`	Size	`grain`	Time	Comment
`char`	800M	1	1.963	100x slower
		10	0.330	17x slower
		5000 (TBB-rec.)	0.020	optimal
		auto (ours)	0.020	optimal
`char[64]`	200M	1	0.129	78% slower
		10 (TBB-rec.)	0.077	6% slower
		5000	0.072	optimal
		auto (ours)	0.073	optimal
`char[2048]`	0.4M	1 (TBB-rec.)	0.049	optimal
		10	0.050	optimal
		5000	0.057	16% slower
		auto (ours)	0.050	optimal
`char[131072]`	0.01M	1 (TBB-rec.)	0.075	optimal
		10	0.075	optimal
		5000	1.419	19x slower
		auto (ours)	0.075	optimal

Table 4.1: Running times on 40 cores for inputs of various sizes, for manually-fixed grain sizes, including the one obtained following Intel's TBB manual, and for our algorithm.

because the suitability depends on the particular hardware and software environment and, in fact, on the inputs to the function `match`. Applying the established practice of manual granularity control leads to poor results even on the same machine and with the same software environment.

Table 4.1 illustrates the issue. It shows the 40-core run times for different types `T` and different `grain` settings. (The experiment is run on an Intel machine described in Section 4.5.) The input size (total number of elements) is chosen to ensure a sequential execution time of a few seconds. When `T` is the type `char`, we use character equality as predicate. When `T` is an array of characters, we compare the hash values, for a standard polynomial hashing function. For each setting of `T`, we consider various values of `grain`, including the "recommended" grain value determined by following the process described in Intel's TBB manual [100]: start by setting the `grain` to the value $10,000$ and halve it until the 1-processor run-time stops decreasing by more than 10%. Such tuning maximizes the exposed parallelism by considering the smallest grain value for which the overheads are not prohibitive.

At first, observe that the TBB-recommended value of `grain` changes for different settings of `T`. For `char` it is `5000`; for `char[64]` it is `10`; for `char[2048]` it is `1`. Secondly, observe that a grain optimal in one setting may induce a very significant slowdown in a different setting. For example, when `T` is `char`, setting the grain to `1` instead of `5000` results in a 100-fold slowdown. Thus, we conclude that, there is no one value of `grain` that works well for all instances of `match`.

One might attempt to select the `grain` size based on the arguments provided to the `match` function. For example, the `grain` could be set to `C/sizeof(T)`, for some constant `C`, to ensure use of a smaller grain size when processing bigger elements. This approach helps in some cases, but it does not solve the problem in general: note that in `match`, the grain depends not only on the type `T`, but also on the predicate passed as the second argument. If `T` is set to `char[64]` and the predicate is instantiated as a standard polynomial hash function, the optimal grain size is `10`; however, providing a different, more computationally expensive predicate function, for example, the number of different substrings, causes

```
int grain = ... // determined by tuning
template <T, P>
match(T* lo, int n, P p): int
  res ← 0
  if n > grain:
    sl ← n / 2
    sr ← (n + 1) / 2
    T* mid ← lo + sl
    res1 ← 0
    res2 ← 0
    fork2join(
      [&] {
        res1 ← match(lo, sl, p)
      },
      [&] {
        res2 ← match(mid, sr, p)
      }
    )
    res ← res1 + res2
  else:
    res ← match_seq(lo, hi, p)
  return res
```

Figure 4.3: Find matches with manual granularity control.

```
template <T, P>
match(T* lo, int n, P p): int
  res ← 0
  spguard([&] { // complexity function
    return n
  }, [&] { // parallel body
    if n ≤ 1:
      result ← match_seq(lo, hi, p)
    else:
      sl ← n / 2
      sr ← (n + 1) / 2
      T* mid ← lo + sl
      res1 ← 0
      res2 ← 0
      fork2join(
        [&] {
          res1 ← match(lo, sl, p)
        },
        [&] {
          res2 ← match(mid, sr, p)
        }
      )
      res ← res1 + res2
  }, [&] { // sequential body
    res ← match_seq(lo, hi, p)
  })
  return res
```

Figure 4.4: Find matches with automatic granularity control.

the optimal grain size to be 1. Selecting the right `grain` for different predicates would require the ability to predict the execution time of a function, an intractable problem. To control granularity in cases of nested-parallel programs, the programmer will likely have to specialize the code for each predicate and apply granularity control to each such specialization, thus losing the key benefits of generic functions.

The problems illustrated by the simple example above are neither carefully chosen ones nor isolated cases. They are common; more realistic benchmarks exhibit even more complex behavior. In fact, as discussed in more detail in Section 4.5, in the state of the art PBBS benchmarking [142], nearly every benchmark relies on carefully written, custom granularity control techniques.

Our Approach

Our goal is to delegate the task of granularity control to a smart library implementation. To this end, we ask the programmer to provide for each parallel function a *series-parallel guard*, by using the keyword `spguard`. A spguard consists of: a *parallel body*, which is a lambda function that performs a programmer-specified parallel computation; a *sequential body*, which is a lambda function that performs a purely sequential computation equivalent to the parallel body, i.e. performing the same side-effects and delivering the same result; and, a *cost function*, which gives an abstract measure, as a positive number, of the work (run-time cost) that would be performed by the sequential body.

At a high level, a spguard exploits the result of the cost function to determine whether the computation involved is small enough to be executed sequentially, i.e., without attempting to spawn any subcomputation. If so, the spguard executes the sequential body.

Otherwise, it executes the parallel body, which would typically spawn smaller subcomputations, each of them being similarly guarded by a spguard.

The cost function may be any programmer-specified piece of code that, given the context, computes a value in proportion to the one-processor execution time of the sequential body. Typically, the cost function depends on the arguments provided to the current function call. A good choice for the cost function is the average asymptotic complexity of the sequential body, e.g., $n \lg n$, or n, or $\sqrt{n}$, where n denotes the size of the input. The programmer need not worry about constant factors because `spguards` are able to infer them on-line, with sufficient accuracy.

In a real implementation, the sequential body can be left implicit in many cases, because it can be inferred automatically. For example, the sequential body for a parallel-for loop can be obtained by replacing the parallel-for primitive with a sequential for. Likewise, in many instances, the complexity function is linear, allowing us to set it to the default when not specified. In our library and experiments, we use this approach to reduce dramatically the annotations needed.

Figure 4.4 shows the code for our example `match` function using spguard. Compared with the original code from Figure 4.3, the only difference is the code being structured as a `spguard` with three arguments: cost function, parallel body, and sequential body. There, the cost function simply returns the input size, written `n`, because the sequential body (`match_seq`) uses a linear-time, sequential matching algorithm.

As we show in this chapter, once a parallel algorithm is modified with the insertion of spguards like in Figure 4.4, the information provided by the cost function suffices for our run-time system to control granularity effectively for all settings of the parameters. As shown in Table 4.1, our oracle-guided version matches the performance achieved by the `grain` settings recommended by the TBB method, but without any of the manual tuning effort and code modifications.

4.3 Algorithmic Granularity Control

Our algorithm aims at sequentializing computations that involve no more than a small amount of work. To quantify this amount, let us introduce the *parallelism unit*, written κ, to denote the smallest amount of work (as units of time) that would be profitable to parallelize on the host architecture. The value of κ should be just large enough to amortize the cost of creating and managing a single parallel task. On modern computers, this cost can be from hundreds to thousands of cycles. Practical values for κ therefore range between 25 and 500 microseconds.

To see the basic idea behind our algorithm, consider the following simple example, involving a single spguard. Suppose that we have a parallel function `f(x)` which, given an argument `x`, performs some computation in divide-and-conquer fashion, by recursively calling itself in parallel, as shown below. Assume the body of this function to be controlled by a spguard, with `c(x)` denoting the cost function, and `g(x)` denoting the sequential body, that is, a purely sequential function that computes the same result as `f(x)`.

```
f(x):
  spguard(
          [&] { c(x) }, // cost function
          [&] {         // parallel body
              if x = 1:
                ...
              else:
                 (x1, x2) ← divide(x)
                 r1 ← null
                 r2 ← null
                 spawn [&] { r1 ← f(x1) }
```

```
                    [&] { r2 ← f(x2) }
                sync
                conquer(r1, r2)
            },
            [&] { g(x) } // sequential body
    )
```

Intuitively, we aim at enforcing the following policy: if the result of `f(x)` can be obtained by evaluating the sequential body `g(x)` in time less than κ, then `g(x)` should be used. Under a small number of assumptions detailed further in this chapter, this policy leads to provably efficient granularity control.

4.3.1 Making Predictions

One central question is how to predict whether a call to `g(x)` would take less than κ units of time. Assume, to begin with, a favorable environment where (1) the hardware is predictable in the sense that the execution time is in proportion to the number of instructions, and (2) the cost function `c(x)` gives an estimate of the asymptotic number of instructions involved in the execution of `g(x)`. For a given input `x`, let N denote the value of `c(x)`, and let T denote the execution time of `g(x)`. By definition of "asymptotic", there exists a constant C such that: $T \approx C \cdot N$. Our algorithm aims at computing C by sampling executions, and then it exploits this constant to predict whether particular calls to `g(·)` take less than κ units of time.

More precisely, for an input `x_i` being evaluated sequentially, that is, through a call to `g(x_i)`, we may measure the execution time of `g(x_i)`, written T_i, and we may compute the value of `c(x_i)`, written N_i. From a collection of samples of the form (T_i, N_i), we may evaluate the value of the constant C by computing the ratios T_i/N_i. In reality, the actual value of the constant can vary dramatically depending on the size of the computation, in particular due to cache effects — we will later return to that point. There is a much bigger catch to be addressed first.

In order to decide which computations are safe to execute using the sequential body `g(·)`, our algorithm needs to first know the constant C. Indeed, without a sufficiently accurate estimate of the constant, the algorithm might end up invoking `g(·)` on a large input, thereby potentially destroying all available parallelism. Yet, at the same time, in order to estimate the constant C, the algorithm needs to measure the execution time of invocations of `g(·)`. Thus, determining the value of C and executing the algorithm `g(·)` are interdependent. Resolving this critical circular dependency is a key technical challenge.

At a high level, our algorithm progressively sequentializes larger and larger computations. It begins by sequentializing only the base case, and ultimately converges to computations of duration κ. Each time that we sequentialize a computation by calling `g(·)` instead of `f(·)`, we obtain a new time measure. This measure may be subsequently used to predict that another, slightly larger input may also be processed sequentially. We are careful to increase the input size progressively, in order to always remain on the safe side, making sure that our algorithm never executes sequentially a computation significantly longer than κ.

As our algorithm increases the cost (as measured by the cost function) of sequentialized computations each time at most by a multiplicative factor, called α, it converges after just a logarithmic number of steps. The *growth rate*, α controls how fast sequentialized computations are allowed to grow. Any $\alpha > 1$ could be used; values between 1.2 and 3 work well in practice.

4.3.2 Dealing with Nested Parallelism

When dealing with a single spguard, the process described above generally suffices to infer the constant associated with that spguard. However, the process falls short for programs

involving nested parallelism, e.g., nested loops or mutually-recursive functions. To see why, consider a function `h(·)` that consists of a spguard whose parallel body performs some local processing then spawns a number of calls to completely independent functions. Because `h(·)` is not a recursive function with a base case, the algorithm described so far is unable does not have a chance to follow the convergence process; the spguard of `h(·)` would have no information whatsoever about its constant, and it would always invoke the parallel body, failing to control granularity.

To address this issue and support the general case of nested parallelism, we introduce an additional mechanism. When executing the parallel body of a spguard, our algorithm computes the sum of the durations of all the pieces of sequential computation involved in the execution of that parallel body. This value gives an upper bound on the time that the sequential body would have taken to execute. This upper bound enables deriving an over-approximation of the constant. Our algorithm uses this mechanism to make *safe* sequentialization decisions, i.e., to sequentialize computations that certainly require less than κ time. By measuring the duration of such sequential runs, our algorithm is then able to refine its estimate of the constant. It may subsequently sequentialize computations of size closer to κ.

Overall, our algorithm still progressively sequentializes larger and larger subcomputations, only it is able to do so by traversing distinct spguards with different constant factors.

4.3.3 Dealing with Real Hardware

Our discussion so far assumes that execution times may be predicted by the relationship $T \approx C \cdot N$. But in reality, this assumption is not the case. The reason is that the ratios T/N may significantly depend on the input size. For example, we observed in several programs that processing an input that does not fit into the L3 cache may take up to 10 times longer to execute than a just slightly smaller input that fits in the cache. In fact, even two calls to the same function on the same input may have measured time several folds apart, for example, if the first call needs to load the data into the cache but not the second one.

We design our algorithm to be robust in the face of large variations of the execution times typical of modern hardware. In addition to validating empirically that our algorithm behaves well on hundreds of runs of our full benchmark suite, we formally prove its robustness property. To that end, we consider a relatively realistic model that takes into account the variability of execution times typical of current hardware. More precisely, we develop our theory with respect to an abstract notion of "work", which we connect both to runtime measures and to results of costs functions, as described next.

At first, we assume that runtime measures may vary by no more than a multiplicative factor E from the work, in either direction. Second, we require that, for the program and the growth rate α considered, there exists a value β such that, for any cost function involved in the program, the following property holds: if the cost function applied to an input J returns a value no more than α bigger than for input I, then the work associated with input J is at most β times bigger than the work for input I. Intuitively, this property ensures that the work increases with the cost, but not exceeding a maximal rate of increase. In practice, for $\alpha \leq 2$, we typically observe $\beta \leq 10$. The value 10 typically corresponds to the maximal slowdown occurring when the input data exceeds the size of the cache. Note that the algorithm does not require knowledge of β; it is only involved in the analysis.

4.3.4 Analysis Summary

Our algorithm relies on several important assumptions on programs. These assumptions, whose formal statements may be found in Analysis Section 4.4.1, are matched by a large class of realistic parallel algorithms. At first, our algorithm assumes that, for each spguard,

the sequential body evaluates no slower than the corresponding parallel body on a single process. (The sequential body might always be obtained by omitting spawns in the parallel body.) But at the same time, the sequential body should not run arbitrarily faster than the parallel body executed with all its inner spguards being sequentialized. This assumption is required to know that it is safe to exploit the execution time of the parallel body to over-approximate that of the sequential body.

Second, our algorithm assumes that spguards are called regularly enough through the call tree. Without this assumption, the program could have a spguard called on an input that takes much longer than κ to process, with the immediately nested spguard called on an input that takes much less than κ, leaving no opportunity for our algorithm to sequentialize computations of duration close to κ.

Last, our algorithm assumes some balance between the branches of a fork-join. Without this assumption, the program could be a right-leaning tree, with all left branches containing tiny computations. Such an ill-balanced program is a poorly-designed parallel program that would be slow in any case.

As we prove through a careful analysis detailed in Section 4.4, under these assumptions, our algorithm is efficient. Our bound generalizes Brent's Theorem 2.2.1 ($T_P \leq \frac{W}{P} + S$), by taking into account the overheads of thread creation (cost of spawning and managing threads) and granularity control (including the cost of evaluating cost functions, which are assumed to be executed in constant time). A simplified statement of the bound, using big-O notations, appears next.

Theorem 4.3.1. *Under the above assumptions, with parallelism unit κ, the runtime on P processes using any greedy scheduler of a program involving work w and span s is bounded by:*

$$T_P \leq \left(1 + \frac{O(1)}{\kappa}\right) \cdot \frac{w}{P} + O(\kappa) \cdot s + O\left(\log^2 \kappa\right).$$

The most important term in this bound is the first term: the overheads impact the work term w by only a small factor $O(1)/\kappa$, which can be reduced by controlling κ. The second term states that doing so increases the span by a small factor $O(\kappa)$, and that the granularity control comes at a small logarithmic overhead $O(\log^2 \kappa)$, which is due to our granularity control algorithm. We note that while our theorem is stated for simplicity in terms of greedy schedulers, it could easily be adapted to other schedulers, e.g., to work-stealing schedulers using Theorem 2.2.2.

4.3.5 High-Level Pseudo-Code for the Estimator and Spguard

In what follows, we present pseudo-code for the core of our algorithm. For each syntactic instance of a spguard, we require a unique *estimator* data structure to store the information required to estimate the corresponding constant factor. (In the case of templated code, we instrument the template system to ensure that each template instantiation allocates one independent estimator data structure.) The estimator data structure appears on the left of Figure 4.5. It maintains only two variables: the variable `C`, a running estimate of the constant, and `Nmax`, which tracks the maximum (abstract) work sampled thus far.

The function `report` provides the estimator with samples. It takes as argument `T`, the execution time and `N`, the abstract work. If `T` is less than the parallelism threshold κ and `N` is greater than `Nmax`, then `C` is atomically updated as `T/N` and `Nmax` is updated to `N`. The function `report` protects against data races by using an `atomic` block. Our library implements this atomic block using a single compare-and-swap (CAS) operation, by packing the variables `C` (represented in single-precision) and `Nmax` (represented on 32 bits) on a single machine word. In the case of a race, compare-and-swap would fail and our code would try again until it succeeds, or `Nmax` becomes greater than `N`.

The function `is_small`, takes as argument a cost `N` and returns a boolean indicating whether the time to execute this much work is predicted to be less that $\alpha \cdot \kappa$. (Allowing

```
const double κ   // parallelism unit
const double α   // growth factor

class estimator
  // constant for estimations
  double C
  // max complexity measure
  int Nmax

  report(int N, time T):
    atomic {
      if T ≤ κ and N > Nmax:
        C ← T / N
        Nmax ← N
    }
  return

  is_small(int N): bool
    return (N ≤ Nmax) or
           (N ≤ α · Nmax and N · C ≤ α · κ)
```

```
template <Body_left, Body_right>
fork2join(Body_left bl, Body_right br):
  spawn [&] { bl() }
        [&] { br() }
  sync
  return

template <Complexity, Par_body, Seq_body>
spguard(estimator* es, Complexity c,
    Par_body pb, Seq_body sb):
  int N ← c()
  if es.is_small(N):
    work ← measured_run(sb)
  else:
    work ← measured_run(pb)
  es.report(N, work)
  return
```

Figure 4.5: Pseudocode for the core algorithm.

sequentialization of computations of size slightly larger than κ is needed to obtain proper estimates near κ.) If `N` is smaller than `Nmax`, then the function returns true. Intuitively, this computation is smaller than a previously-sampled computation that took less than κ time. If `N` exceeds `Nmax`, but no more than by a factor α, the function extrapolates using its most current information: if the product `C · N` does not exceeds $\alpha\kappa$, then it returns `true`.

The functions `fork2join` and `spguard` appear on the right of Figure 4.5. The `fork2join` function executes its two branches in parallel and resumes after completion of both branches. The function `spguard` takes an estimator, a parallel body, a sequential body and a cost function that return the abstract work of the sequential body. It begins by computing the abstract cost `N` for the sequential body and consults the estimator. If the work is predicted to be small, it runs the sequential body, else the parallel body. It relies on a function called `measured_run` to measure the sum of the duration of the pieces of sequential code executed. This function is not entirely trivial to implement, but can be implemented efficiently with little code as detailed further.

4.3.6 Implementing Time Measurements

Above, for simplicity, we have abstracted an important detail: measuring the work of a sequential or a parallel function. Now, we present a refinement of the algorithm that performs such measurements. The basic strategy is to keep several pieces of information as processor local state to accumulate relevant timer information and combine them carefully to compute the total work of a function. Figure 4.6 shows the full specification, which we describe below.

Time Measures

We assume a function called `now()` that returns the current time. Importantly, we never need to assume a global clock synchronized between the various processors, since all our time measurements are always local to a processor. We nevertheless need to assume that the clocks on the various processors deliver homogeneous results, that is, that the clocks tick (roughly) at the same pace. The precision of the clock should be sufficient to

```
core_local time total
core_local time timer

total_now(time t): time
  return total + (now() - t)

measured_run(f): time
  total ← 0
  timer ← now()
  f()
  return total_now(timer)

template <Body_left, Body_right>
fork2join(Body_left bl, Body_right br):
  t_before ← total_now(timer)
  t_left ← 0
  t_right ← 0
  spawn [&] { t_left ← measured_run(bl) }
        [&] { t_right ← measured_run(br) }
  sync
  total ← t_before + t_left + t_right
  timer ← now()
  return
```

```
template <Complexity, Par_body, Seq_body>
spguard(estimator* es, Complexity c,
    Par_body pb, Seq_body sb):
  N ← c()
  if es.is_small(N):
    t ← now()
    sb()
    es.report(N, now() - t)
  else:
    t_before ← total_now(timer)
    t_body ← measured_run(pb)
    es.report(N, t_body)
    total ← t_before + t_body
    timer ← now()
  return
```

Figure 4.6: Implementation of time measurements.

measure time interval one or two order of magnitude smaller than κ, that is, roughly in the thousands of cycles. In practice, we rely on hardware cycle counters, which are both very precise and very cheap to query.

Sequential Work Time

We define the *sequential work time* of a computation to be the sum of the durations of all the sequential subcomputations performed over the scope of that computation.

Invariants

Our algorithm maintains some information as processor-local state. More precisely, each processor manipulates two variables, called `total` and `timer`. These two variables are used for evaluating the sequential work time associated with the innermost call to the `measured_run` function, according to the following two invariant. First, the `timer` variable stores either the time of beginning of the innermost `measured_run` call, or a point in time posterior to it. Second, the `total` variable stores the sequential work time that was performed between the timestamp associated with the beginning of the innermost `measured_run` call and the timestamp stored in the variable `timer`.

The auxiliary function `total_now` returns the sequential work time since the beginning of the innermost call to `measured_run`. It is implemented by computing the sum of the contents of the `total` variable and the width of the time interval between the `timer` and the current time.

Remark: when outside of the scope of any `measured_run` call, the two processor local variables keep track of the sequential work time since the beginning of the program. Our code never exploits such values.

Transitions

When entering the scope of a new call to `measured_run` (Line 7), the variable `total` is set to zero, and the variable `timer` is set to the current time. When exiting the scope of a `measured_run` call (Line 11), the auxiliary function `total_now` is used to compute the total sequential work time over this call.

When entering the scope of a new call to `measured_run`, it is essential to save the relevant information associated with the current (immediately outer) call, otherwise this valuable information would get lost. The variable `t_before` serves this purpose, by saving the sequential work time performed so far in the current call, just before entering the scope of the new call (Lines 15 and 33). When subsequently leaving the scope of this new call, we restore the invariant by setting `total` to `t_before` plus the time spent during the new call (Lines 21 and 36), and by setting `timer` to the current time (Lines 22 and 37).

In case the innermost `measured_run` call executes a fork-join (Line 14), the value of the local variable `t_before` is captured by the join continuation. In technical terms, the value of `t_before` is part of the call frame associated with the join thread that executes after the `sync`. This join thread could be executed on a different processor than the one that initiated the spawn, but such possibility does not harm the correctness of our algorithm. Regardless of potential thread migration, we correctly set the `timer` and the `total` immediately after the `sync` (Lines 21 and 22). More precisely, the processor that executes the join continuation sets its `total` variable to be the sum of the sequential work time performed before the spawn, plus that performed in each of the two branches (which may execute on different processors), and it resets its `timer` variable to the current time.

In the simple case of a spguard executing its sequential body (Lines 28-31), our algorithm does not bother calling `measured_run`. Instead, it directly measures the time before and after the sequential body, and then compute the difference between the two values. This simpler scheme applies because the sequential body involves no spguard nor any fork-join call.

4.3.7 Programming Interface

We highlight two interesting features of our `C++` implementation. At first, our implementation supports spguard without an explicitly-provided sequential body. In this case, the parallel body is executed, with all spawns processed sequentially. Second, our implementation supports defining higher-level abstractions on top of `fork2join`, such as `parallel-for`, `map`, `reduce`, `map_reduce`, `scan`, `filter`, etc. For these abstractions, the programmer may either indicate a custom cost function or simply rely on the default one, which assumes a constant-time processing for each item or iteration. This default mechanism saves a large number of cost functions.

For example, we implemented a more concise version of the `match` function in Figure 4.4 using `map_reduce`, as follows.

```
1  template <T, P>
2  match(T* lo, int l, P p): int
3    return map_reduce(lo, lo + l, 0,
4      [&] (int x, int y) { // associative combining operator
5        return x + y
6      },
7      [&] (T* i) { // leaf-level operation
8        return p(*i)
9      }
10   )
```

4.4 Analysis

4.4.1 Definitions and Assumptions

Work and Span

To take into account the actual overheads of parallelism that granularity control aims to amortize, our analysis accounts for the cost of parallel task creation, written by τ, as well as the cost of evaluating cost functions, written by ϕ. Although these costs may vary in practice, we assume τ and ϕ to be upper bounds for them. In particular, we assume cost functions to evaluate in constant time, and we exploit the fact that our algorithm makes predictions and handles time measurements in constant time for each spguard call. For the latter, we make the simplifying assumption that the resolution of data races during reports in an estimator incurs no more than a fixed cost. (In practice, CAS-conflicts are quite rare in our experiments; a more refined cost model taking contention into account would be required to account for the cost of CAS conflicts.)

Our analysis establishes bounds on the work and on the span, including the overheads of parallelism, of the execution of a program under the guidance of our automatic granularity control algorithm. We name these entities *total work* and *total span*. Our bounds on the total work and span are expressed with respect to the work and span of the *erasure* version of that program, in which all spguards are replaced with their parallel bodies. The erasure of a program can be viewed as a program that is not granularity controlled at all, but instead exposes all available parallelism.

We define the work and span of the erasure program, written w and s respectively, as the work and the span of its erasure, where work and span are defined in the standard manner [2]. In short, the work of two expressions composed sequentially is the sum of the work of the two expressions; the work of two expressions composed in parallel is the sum of the work of the two plus one. The span of two expressions composed sequentially is the sum of the spans of the two; the span of two expressions composed in parallel is the maximum of the spans of the two plus one. Note that to avoid ambiguity we call w and s the *raw work* and the *raw span*, respectively.

We define the *total work*, written $\mathbb{W}$, and the *total span*, written $\mathbb{S}$, to measure the actual work and span of an execution (rather than a program), accounting also for the cost of parallelism and granularity control. More specifically, total work and total span include the *overheads*: each `fork2join` is assumed to incur an extra cost τ (covering in particular the cost of spawning threads and dealing with their scheduling), and each estimator operation (including the evaluation of the cost function, the call to `is_small` and to `report`, and the time measurements) is assumed to incur an extra cost ϕ. When computing total work and span, we compute work/span of a spguard based on the branch that it took: if the sequential branch is taken, then the total work/span of the spguard is the total work/span of the sequential branch plus ϕ. Otherwise, if the parallel branch is taken, then the total work/span of the spguard is the total work/span of the parallel branch plus "$\tau + \phi$".

Finally, for the purpose of stating the assumptions of our theorems, we define the *sequential work*, written $W_s(t, I)$, as the work of a piece of code t executed on input I when this term executes fully sequentially, that is, where all spguards are forced to execute the sequential body. For such a sequential execution, no parallelism is created and no cost function is evaluated, thus, the sequential work is equal both to the work and to the total work.

Accuracy of Time Measurements

We assume that time measured for a sequential execution may diverge from the sequential work by up to some multiplicative factor E, in either direction. Formally, we assume the existence of constant E such that, for any sequential execution of a purely sequential term

S on input I, its measured time, written $M(S, I)$, satisfies:

$$\frac{M(S,I)}{W_s(S,I)} \in [\frac{1}{E}, E].$$

Well-Defined Spguards

We say that a spguard is *well-defined* if the relative sequential work cost of the sequential body and the parallel body of each spguard can be lower and upper bounded by a constant, i.e., are asymptotically the same. Specifically, consider a spguard g, with a cost function F, sequential body S, and parallel body B, executed on some input I. Let $W_s(S, I)$ denote the sequential work of the sequential body of this call. Let $W_s(B, I)$ denote the sequential work of parallel body of this call, that is, the sequential execution time of the parallel body when all spguards choose their sequential body. The spguard is well-defined if there exists a constant D such that:

$$1 \leq \frac{W_s(B,I)}{W_s(S,I)} \leq D.$$

For the analysis, we consider programs where each spguard is well-defined with respect to the same constant D.

The lower bounds assert that the sequential body induces no more work than the parallel body. This is justified because the parallel body can always serve as a sequential body by replacing `fork2join` calls with function calls. Note that without this assumption, the sequentialization of subcomputations can increase the total work making it impossible to control granularity.

The upper bound asserts that the sequential body cannot be arbitrarily faster than executing the parallel body sequentially. This requires the parallel body to use spguards to fall back to a sequential body without doing too much work. This is not difficult to achieve for many nested-parallel algorithms, because they present an opportunity to use a sequential body at each nest level, which usually corresponds to a recursive call. Without this assumption, our algorithm is not guaranteed to converge to the desired constant factor due to the gap between the parallel and sequential bodies.

Accurate Cost Functions

We make several accuracy assumptions on cost functions. At first, as mentioned previously, we assume that cost functions are evaluated in constant time, and that this amount of time does not exceed ϕ.

Second, we assume that the work increases with the cost. Formally, we assume that, for any spguard g and associated cost function F, and for any pair of input I and J such that $F(I) \leq F(J)$, we have: $W_s(g, I) \leq W_s(g, J)$.

Finally, we assume that the work increases with the cost no faster than at some maximal rate (recall Section 4.3). Formally, we require that, for the program and the value considered for the parameter α, there exists a value β such that, for any guard g and associated cost function F involved in the program, and for any pair of inputs I and J such that $F(I) \leq \alpha \cdot F(J)$, we have: $W_s(g, I) \leq \beta \cdot W_s(g, J)$.

Syntactic Forms of Spguards

For the analysis, we assume that we are given a program written by using the two parallelism primitives that we offer: `fork2join` and `spguards` (described in Section 4.3). To facilitate and simplify the analysis, we make two syntactic assumptions about programs. These assumptions are not necessary for the algorithm or our implementation but are used purely to facilitate analysis.

First, we assume that the parallel body of a spguard consists of fork-join call surrounded by two pieces of sequential code, one to split the input and one to merge the output. In `C++` syntax:

$$\texttt{spguard}(F,\ \texttt{[\&]\{}\ S_p\texttt{;fork2join}(L,R)\texttt{;}S_m\ \texttt{\}},\ S).$$

This assumption causes no loss of generality because more complex expressions can be guarded by sequencing and nesting `spguards`.

Second, we treat the body of spguards as functions that operate on some input. More precisely, an execution of the guard above requires some input I and proceeds by first executing $F(I)$ to compute the cost, and then running with I either the sequential body $S(I)$ or the parallel body based on the outcome of the estimator as described in Section 4.3. The parallel body is of the form:

$$S_p(I);\ \texttt{fork2join}(L(I_L),R(I_R));\ S_m(I_m)$$

where I_L and I_R are the inputs to branches and I_m is the input to S_m. The inputs I_L and I_R are obtained after processing of the input I by S_p, and I_m is an output produced by branches L and R.

Regularity of Forks

As explained in Section 4.3, to allow efficient granularity control, we need to rule out ill-balanced programs. We assume the existence of a constant γ (with $\gamma \geq 1$), called the *regularity factor*, satisfying the following requirements.

- When considering a fork-join, there must be at least some balance between the left and the right branch. Formally, for any execution on input I of a spguard with sequential body S and branches L and R, we assume:

$$\frac{W_s(L,I)}{W_s(S,I)} \quad \text{and} \quad \frac{W_s(R,I)}{W_s(S,I)} \quad \in \left[\frac{1}{\gamma},\ 1-\frac{1}{\gamma}\right].$$

  Without this assumption, the program can be a right-leaning tree, with all left branches containing only a tiny subcomputation; in such a program the overheads of forks cannot be amortized.

- spguards must be called sufficiently frequently in the call tree. Formally, for any call to a spguard that has sequential body S and executes on input I, if the immediate outer spguard call has a sequential body S' and executes on input I', we assume:

$$\frac{W_s(S,I)}{W_s(S',I')} \geq \frac{1}{\gamma}.$$

  Without this assumption, the program can have a spguard called on an input that takes much longer than κ to execute, with the immediate inner spguard called on an input that takes much less than κ, leaving no opportunity to sequentialize a subcomputation that takes approximately time κ to execute.

- For an outermost call to a spguard, we need to assume that it involves a nontrivial amount of work. Formally, if an outermost call to a spguard with sequential body S executes on input I, we assume $W_s(S,I) \geq \kappa$. Without this assumption, the program can consist of a sequential loop that iterates calls to a spguard on tiny input; and, again, the overheads would not be amortized. (Note that, technically, the requirement $W_s(S,I) \geq \frac{\kappa}{\gamma DE}$ would suffice.)

4.4.2 Results Overview

To establish a bound on the parallel execution time of our programs, we first bound the total span and total work. We then derive a bound on the parallel execution time under a greedy scheduler. For the bounds, we assume programs that are 1) γ-regular, 2) where spguards are well-defined, and 3) where all cost functions are accurate. We express our bounds with respect to work and span, which do not account for the overheads as well as the parameters of the analysis that account for various overheads and factors.

Theorem 4.4.1 (Bound on the total span)**.** $\mathbb{S} \leq (1 + \phi + \max(\tau,\, E\beta\kappa)) \cdot s$.

The span may grow by a multiplicative factor, but no more. As our final bound will show, for a program with sufficient parallelism (i.e., with $\frac{w}{s} \gg P$), this increase in the span has no visible impact on the parallel run time.

Theorem 4.4.2 (Bound on the total work)**.** *Let* $\kappa' = \frac{\kappa}{DE\gamma}$*, and* $F = 1 + \log_\alpha \frac{\kappa}{DE}$*, and* $H = \log_{\gamma/(\gamma-1)} \frac{\kappa}{DE}$*, and* P *denote the number of processes, and* G *denote the number of spguards occuring in the code. Then, we have*

$$\mathbb{W} \leq \left(1 + \frac{\tau + 2\phi}{\kappa'}\right) \cdot w + PFGH \cdot (\tau + 2\gamma\phi).$$

The first component of the left hand side asserts that the work grows by no more than a multiplicative factor $1 + \epsilon$, where ϵ may be tamed to a couple of percents by the choice of a sufficiently large κ. The second component asserts that the total overheads associated with the sequentialization of tiny subcomputations during the convergence phase of the estimators are bounded by some constant cost, proportional to the number of estimators and to the product $P \cdot F \cdot H$, which is $P \cdot O(\log^2 \kappa)$.

To bound the parallel run time, we exploit Brent's Theorem 2.2.1, which asserts that, for any greedy scheduler, $T_p \leq \frac{\mathbb{W}}{P} + \mathbb{S}$, where P stands for the number of processes. To simplify the final statement, we make some over-approximation, and we also exploit several inequalities that are always satisfied in practice. We assume $\kappa \geq \tau$ and $\kappa \geq 1 + \phi$, since in practice the user always sets $\kappa \gg \max(\tau, \phi)$ to ensure small overheads.

Theorem 4.4.3 (Bound on the parallel run time)**.** *Let* P *denote the number of processes and* G *denote the number of spguards.*

$$T_P \;\leq\; \left(1 + \frac{\gamma ED \cdot (\tau + 2\phi)}{\kappa}\right) \cdot \frac{w}{P} \;+\; (E\beta + 1) \cdot \kappa \cdot s \;+\; O\left(G \cdot \log^2 \kappa \cdot (\tau + 2\gamma\phi)\right).$$

Theorem 4.3.1 (Simplified bound on the parallel run time)**.** *For fixed hardware and any program, all parameters of the analysis except for* κ *(the unit of parallelism) can be replaced with constants, leaving us with the following bound:*

$$T_P \;\leq\; \left(1 + \frac{O(1)}{\kappa}\right) \frac{w}{P} \;+\; O(\kappa) \cdot s \;+\; O\left(\log^2 \kappa\right).$$

The most important term in this bound is the first term that says that the overhead of various practical factors impact the work term w by only a small factor $O(1)/\kappa$, which can be reduced by controlling κ. The second term states that doing so increases the span by a small factor $O(\kappa)$ and that all of this comes at a small logarithmic overhead, which is due to our granularity control algorithm $O(\log^2 \kappa)$.

Our theorem establishes that a non-granularity controlled nested parallel program with work w and span s can be guaranteed to be executed fast on a real machine using our granularity control algorithm. Our experiments (Section 4.5) show that the analysis appears to be valid in practice.

4.4.3 Additional Definitions

Syntax of Programs

The BNF grammar for programs with spguards is thus as follows. For brevity, we focus on the language constructs that matter to the analysis: sequences, conditionals, and spguards combined with fork-join.

- B ::= *a boolean variable*
- S ::= *a purely sequential piece of code, without forks nor guards*
- t ::= S | (t; t) | `if` B `then` t `else` t | `spguard`(S, p, S)
- p ::= S; `fork2join`(t, t); S

Detailed Definitions of Work and Span

The table below shows the definition of *raw* work and span, which correspond to the standard definitions used in parallel program analysis, as well as the definition of *total* work and span, which include the overheads of thread creation, written τ, and the overheads associated with the estimator, written ϕ. Note that in the definitions below we are ultimately referring to the work of a sequential piece of code, written $W(S)$, which we assume to be defined as the number of instructions involved in the execution of S, or as the number of cycles involved if the considered execution model assigns a cost to each instruction.

Definition 4.4.1 (Work and span, raw and total, for each construct)**.**

t: source expression	$W(t)$*: raw work*	$S(t)$*: raw span*	$\mathbb{W}(t)$*: total work*	$\mathbb{S}(t)$*: total span*
S	$W(S)$	$W(S)$	$W(S)$	$W(S)$
$(t_1; t_2)$	$W(t_1) + W(t_2)$	$S(t_1) + S(t_2)$	$\mathbb{W}(t_1) + \mathbb{W}(t_2)$	$\mathbb{S}(t_1) + \mathbb{S}(t_2)$
`if` B `then` t_1 `else` t_2 *when* B *is* `true`	$1 + W(t_1)$	$1 + S(t_1)$	$1 + \mathbb{W}(t_1)$	$1 + \mathbb{S}(t_1)$
`if` B `then` t_1 `else` t_2 *when* B *is* `false`	$1 + W(t_2)$	$1 + S(t_2)$	$1 + \mathbb{W}(t_2)$	$1 + \mathbb{S}(t_2)$
`spguard`$(F, (S_p;$ `fork2join`$(L, R); S_m), S)$ *when parallel body is chosen*	$W(S_p) + W(L) + W(R) + 1 + W(S_m)$	$S(S_p) + \max(S(L), S(R)) + 1 + S(S_m)$	$\mathbb{W}(S_p) + \mathbb{W}(L) + \mathbb{W}(R) + \phi + \tau + \mathbb{W}(S_m)$	$\mathbb{S}(S_p) + \max(\mathbb{S}(L), \mathbb{S}(R)) + \phi + \tau + \mathbb{S}(S_m)$
`spguard`$(F, (S_p;$ `fork2join`$(L, R); S_m), S)$ *when sequential body is chosen*	$W(S)$	$W(S)$	$W(S) + \phi$	$W(S) + \phi$

Due to the fact that spguard may dynamically select between the execution of the sequential or the parallel branch, it does not make sense to speak about the work and span of a program. We may only speak about the work and span of a particular execution of the program, that is, of the work and span of an execution trace describing which spguards have been sequentialized and which have not.

Definition 4.4.2 (Execution trace)**.** *A particular execution of a source term t on an input I corresponds to a trace, written X, that describes, for each evaluation of a spguard during the program execution, whether the sequential body or the parallel body is selected by the spguard.*

Definition 4.4.3 (Work and span, raw and total, of an execution trace)**.** *For an execution of a source term t on an input I producing a trace X, we let:*

- $W(t, I, X)$ *denote the* raw work*,*
- $S(t, I, X)$ *denote the* raw span*,*
- $\mathbb{W}(t, I, X)$ *denote the* total work*,*
- $\mathbb{S}(t, I, X)$ *denote the* total span*.*

The definitions are obtained by applying the appropriate rules from the previous table.

When the arguments t, I and X are obvious from the context, we write simply W, S, $\mathbb{W}$, and $\mathbb{S}$.

Definition 4.4.4 (Sequential work)**.** *For a term t and an input I, we define the* sequential work*, written $W_s(t, I)$, as the raw work $W(t, I, X)$, where X is the trace that systematically selects the sequential bodies.*

Definition 4.4.5 (Work and span)**.** *For a term t and an input O, we define:*

- *the* work $w(t, I)$ *as the raw work $W(t, I, X)$ where X is the trace that systematically selects parallel bodies.*
- *the* span $s(t, I)$ *as the raw span $S(t, I, X)$ where X is the trace that systematically selects parallel bodies.*

For a purely sequential subcomputation, the work is equal to the raw work. (It also matches the span since there is no parallelism involved.) We use this fact implicitly in several places through the proofs.

Time Measurements

In addition to the definition already given for the measurement of a sequential execution time, written $M(t, I)$, we need for the analysis to quantify the total sequential work time involved in a parallel execution, written $M(t, I, X)$. (Recall Appendix 4.3.6.)

Definition 4.4.6 (Measured time)**.** *We let:*

- $M(t, I)$ *denote the measured time of the sequential execution of the term* t *on input* I*.*
- $M(t, I, X)$ *denote the sum of the measured time of all the pieces of sequential sub-computations involved in the evaluation of the term* t *on input* I *according to trace* X*. The measure thus ignores the overheads of fork-join operations and the overheads associated with our runtime decisions. Note that this is exactly the time we measure in our algorithm.*

Classification of Spguard Calls

Definition 4.4.7 (Small call)**.** *A call to a spguard* g *on input* I *is said to be* small *whenever* $W_s(g, I) \leq \frac{\kappa}{DE}$*. Otherwise, it is said to be* non-small*.*

Definition 4.4.8 (Domination of a spguard call)**.**

- *We say that a spguard call* a *is* dominated *by a spguard call* b *if* a *is executed as part of the execution of the parallel body of* b*.*
- *We say that* a *is* directly dominated *by* b *if there are no spguard in-between, i.e. if there does not exist a spguard* c *that dominates* a *and at the same time is dominated by* b*.*

Definition 4.4.9 (Covered sequential call)**.** *A sequential call to a spguard is said to be* covered *if it is directly dominated by a parallel small call. Otherwise, it is* non-covered*.*

Definition 4.4.10 (Classification of spguard calls)**.** *Every spguard call falls in one of the four categories:*

- parallel non-small call *(when the spguard executes its parallel body on a non-small call).*
- parallel small call *(when the spguard executes its parallel body on a small call).*
- covered sequential call *(when the spguard executes its sequential body and is dominated by a parallel small call; note that a covered sequentiall call is a small call).*
- non-covered sequential call *(when the spguard executes its sequential body but is not directly dominated by a parallel small call; in this case, it can be dominated by a parallel non-small call, or not dominated at all).*

Definition 4.4.11 (Critical parallel calls)**.** *A small parallel call to a spguard is said to be a* critical *if all the calls that it dominates are sequential small calls.*

This classification simplifies the presentation of the proof.

Any non-small call amortizes the overheads in a standard manner, i.e., provides multiplicative factor to the work and span. So, our major task is to bound the overhead spend during small calls. Our bound is obtained in several steps.

1. We bound the number of critical parallel calls (that is, a parallel small call that dominates only sequential small calls) in Lemma 4.4.7. To that end, we exploit the fact that, after each critical parallel call, the algorithm performs a report that increases the value of `Nmax` by at least some constant factor.

2. We show that the number of parallel small calls is bounded in terms of the number of critical parallel calls in Lemma 4.4.9. To that end, we observe that each parallel small call features at least one nested critical parallel call, and that, reciprocally, each critical parallel call is nested in at most a logarithmic number of small parallel calls.

3. We bound the number of covered sequential calls in terms of the number of parallel small calls in Lemma 4.4.10. We do so by arguing that each parallel small call can have at most a logarithmic number of nested sequential calls.

4. We independently show that the non-covered sequential calls correspond to non-small calls, so their overheads are properly amortized in Lemma 4.4.11.

All together, we derive the bound on the overheads associated with all small calls in Lemma 4.4.13.

Parameters Involved in the Statement of the Bounds

Definition 4.4.12 (Bound on the number of processes)**.** *Let P denote the number of processes involved in the evaluation.*

Definition 4.4.13 (Bound on the number of spguards)**.** *Let G denote the total number of different spguards occuring in the source code of the program.*

Definition 4.4.14 (Auxiliary parameter F)**.** *We define $F = 1 + \log_\alpha \frac{\kappa}{DE}$, where κ is expressed in number of machine cycles.*

Definition 4.4.15 (Auxiliary parameter H)**.** *We let $H = \log_{\gamma/(\gamma-1)} \frac{\kappa}{DE}$, where κ is expressed in number of machine cycles.*

In the definitions of the auxiliary constants F and H, we assume that the number of cycles involved in the sequential execution of a spguard always exceeds the number returned by the cost function. The bounds can be easily adapted by adding a constant factors to F and H if this was not the case.

To bound the total work $\mathbb{W}$, our proof first bounds the total work excluding overheads involved in parallel small calls and covered sequential calls, which we call $\mathbb{W}'$, then bounds the excluded overheads, that is, the value of $\mathbb{W} - \mathbb{W}'$.

Definition 4.4.16 (Total work excluding the overheads involved in parallel small calls)**.** *We let $\mathbb{W}'$ denote the subset of the total work $\mathbb{W}$ obtained by excluding the overheads (τ and ϕ) involved in small parallel calls, in the sense that when reaching such a call, we only count the raw work involved in this call, regardless of the decisions involved in the subcomputations.*

4.4.4 Basic Auxiliary Lemmas

Lemma 4.4.1. *If term S does not contain any spguard, then $w(S, I) = W_s(S, I)$.*

Proof. Immediate from the definitions. □

Lemma 4.4.2 (Sequential work associated with a spguard)**.** *In the particular case where the term t corresponds to some spguard g, the sequential work is that of its sequential body S, which does not contain any spguard. Thus, we have:*

$$W_s(g, I) = W_s(S, I) = W(S, I, X) = \mathbb{W}(S, I, X) - \phi \qquad \text{for any trace } X.$$

Lemma 4.4.3 (Relationship between work and measured time of a computation)**.** *For any term t executed on input I according to trace X, we have:*

$$\frac{M(t, I, X)}{W(t, I, X)} \in [\frac{1}{E}, E].$$

Proof. Recall our hypothesis: $\frac{M(S,I)}{W(S,I)} \in [\frac{1}{E}, E]$ for all sequential computation S. First, we prove that $M(t,I,X) \leq E \cdot W(t,I,X)$ by induction on the execution tree.

- Case t = S. By assumption on the variability of time measurements, we have: $M(t,I,X) = M(S,I,X) = M(S,I) \leq E \cdot W_s(S,I) = E \cdot W_s(t,I)$. Using the fact $W_s(S,I) = W(S,I,X)$ from Lemma 4.4.2 we get $M(t,I,X) \leq E \cdot W(t,I,X)$.
- Case t $= (t_1, t_2)$. We have: $M(t,I,X) = M(t_1,I,X) + M(t_2,I,X) \leq E \cdot W(t_1,I,X) + E \cdot W(t_2,I,X) = E \cdot W(t,I,X)$.
- Case t = `if` B `then` T_1 `else` T_2. Similar to the previous case, after performing the case analysis.
- Case t = `spguard`$(F, (S_p;$ `fork2join`$(L, R); S_m), S)$.
  - First case: spguard chooses the parallel body. Then, $M(t,I,X) = M(S_p,I,X) + M(L,I,X) + M(R,I,X) + 1 + M(S_m,I,X) \leq E \cdot W(S_p,I,X) + E \cdot W(L,I,X) + E \cdot W(R,I,X) + 1 + E \cdot W(S_m,I,X) \leq E \cdot (W(S_p,I,X) + W(L,I,X) + W(R,I,X) + 1 + W(S_m,I,X)) = E \cdot W(t,I,X)$.
  - Second case: spguard chooses the sequential body. Similar to the case $t = S$.

The second inequality $W(t,I,X) \leq E \cdot M(t,I,X)$ again can be proved by induction on the execution tree. The proof is identical to the proof of the inequality above with the only difference: M and W should be swapped. □

We assumed, for each spguard, that the execution of the sequential body is faster than the sequential execution of the parallel body, thus, when the trace chooses the parallel body the execution should be slower than the corresponding sequential execution.

Lemma 4.4.4. *Consider a term t with well-defined spguards. For any execution t on input I with trace X, we have: $W_s(t,I) \leq W(t,I,X)$.*

Proof. We prove this by induction on the execution tree.

- Case t = S. We have: $W_s(S,I) = W(S,I,X)$.
- Case t $= (t_1, t_2)$. We have: $W_s(t,I) = W_s(t_1,I) + W_s(t_2,I) \leq W(t_1,I,X) + W(t_2,I,X) = W(t,I,X)$.
- Case t = `if` B `then` t_1 `else` t_2. Similar to the previous case, after performing case analysis.
- Case t = `spguard`$(F, (S_p;$ `fork2join`$(L, R); S_m, S)$. We distinguish two cases:
  - First case: spguard chooses the parallel body. Then, using the property of well-defined spguards, $W_s(t,I) \leq W_s(S_p,I) + 1 + W_s(L,I) + W_s(R,I) + W_s(S_m,I) \leq W(S_p,I,X) + 1 + W(L,I,X) + W(R,I,X) + W(S_m,I,X) = W(t,I,X)$.
  - Second case: spguard chooses the sequential body. Then, $W_s(t,I) = W_s(S,I) = W(S,I,X) = W(t,I,X)$.

□

Because work considers full parallelization, it never selects the sequential bodies of spguards, which are assumed to execute faster than parallel bodies on one process. Thus, the work always exceeds the raw work.

Lemma 4.4.5. *Consider a term t with well-defined spguards. For any execution of t on input I with trace X, we have: $w(t,I) \geq W(t,I,X)$.*

Proof. We prove this by induction on the execution tree.

- Case t = S. We have: $w(S, I) = W(S, I, X)$.
- Case t = (t_1, t_2). We have: $w(t, I) = w(t_1, I) + w(t_2, I) \geq W(t_1, I, X) + W(t_2, I, X) = W(t, I, X)$
- Case t = `if` B `then` t_1 `else` t_2. Similar to the previous case, after performing the case analysis.
- Case t = `spguard`(F, (S_p; `fork2join`(L, R); S_m), S). We distinguish two cases:
  - First case: spguard chooses the parallel body. Then, $w(t, I) = w(S_p, I) + 1 + w(L, I) + w(R, I) + w(S_m, I) \geq W(S_p, I, X) + 1 + W(L, I, X) + W(R, I, X) + W(S_m, I, X) = W(t, I, X)$.
  - Second case: spguard chooses the sequential body. Thus, the raw work corresponds to the sequential work, i.e. $W(t, I, X) = W_s(t, I)$. Besides, by definition of work, we know that $w(t, I) = W(t, I, X')$, where X' is the trace that systematically selects parallel bodies. By Lemma 4.4.4, $w(t, I) = W(t, I, X') \geq W_s(t, I) = W(t, I, X)$

□

4.4.5 Proofs

Lemma 4.4.6. *If a spguard g executes its sequential body, then its sequential work is bounded as follows:*

$$W_s(g, I) \leq E \cdot \beta \cdot \kappa.$$

Proof. At first, observe that $W_s(g, I) = W_s(S, I)$, where S denotes the sequential body of the spguard.

Let N denote the value of $F(I)$. According to the implementation of function `is_small`, a spguard selects the sequential body according to the boolean condition: ($\mathtt{N} \leq \mathtt{Nmax}$) or (($\mathtt{N} \leq \alpha \cdot \mathtt{Nmax}$) and ($\mathtt{N} \cdot \mathtt{C} \leq \alpha \cdot \kappa$)), where `Nmax` and `C` are the values stored in the estimator for this spguard.

Since N is non-negative, the condition may only evaluate to true if `Nmax` is non-zero, indicating that at least one previous report has been stored for this spguard. Let J be the input on which this previous report has been obtained, in other words $\mathtt{Nmax} = F(J)$, let X denote the trace of spguards choices, and let $M(g, J, X)$ denote the measured time at this previous report. Since the report was stored, according to the implementation of function `report`, we know that $M(g, J, X) \leq \kappa$ and that $\mathtt{C} = \frac{M(g,J,X)}{F(J)}$.

By Lemma 4.4.2 on sequential work we know that $W_s(S, J) = W_s(g, J)$. Lemma 4.4.4 provides us with $W_s(g, J) \leq W(g, J, X)$. And Lemma 4.4.3 on measured time in the execution on input J gives $W(g, I, X) \leq E \cdot M(g, J, X)$. Thus, the raw work on input J can be bounded as follows: $W_s(S, J) = W_s(g, J) \leq W(g, J, X) \leq E \cdot M(g, J, X)$.

In what follows, we establish the inequality $W_s(S, I) \leq \beta \cdot W_s(S, J)$. By exploiting that $W_s(S, J) \leq E \cdot M(g, J, X)$ and $M(g, J, X) \leq \kappa$, this inequality allows us to conclude as follows:

$$W_s(g, I) = W_s(S, I) \leq \beta \cdot W_s(S, J) \leq \beta \cdot E \cdot M(g, J, X) \leq E \cdot \beta \cdot \kappa.$$

The desired inequality is deduced from the fact that the boolean condition evaluates to true. Indeed, we know that $\mathtt{N} \leq \mathtt{Nmax}$ is true, or that $\mathtt{N} \leq \alpha \cdot \mathtt{Nmax}$ and $\mathtt{N} \cdot \mathtt{C} \leq \alpha \cdot \kappa$ are true.

- In the first case, the condition reformulates to $F(I) \leq F(J)$. By the property of well-defined spguards, we deduce $W_s(S, I) \leq W_s(S, J)$. By exploiting $\beta > 1$, we conclude $W_s(S, I) \leq \beta \cdot W_s(S, J)$, as desired.

- In the second case, the condition reformulates to: $F(I) \leq \alpha \cdot F(J)$ and $F(I) \cdot \frac{M}{F(J)} \leq \alpha \cdot \kappa$. By the property of well-defined spguards, exploiting the first inequality, we deduce: $W_s(S, I) \leq \beta \cdot W_s(S, J)$, the desired inequality.

□

Theorem 4.4.1 (Bound on the total span)**.** *For any execution of a program with well-defined spguards, we have:*

$$\mathbb{S} \leq (1 + \phi + \max(\tau,\, E\beta\kappa)) \cdot s$$

Proof. Let ρ be a shorthand for $1+\phi+\max(\tau,\, E\beta\kappa)$. We establish the inequality $\mathbb{S} \leq \rho \cdot s$ by induction on the execution tree.

- Case t = S. The program is sequential, so $\mathbb{S} = s \leq \rho \cdot s$.
- Case t = $(t_1; t_2)$. We have: $\mathbb{S} = \mathbb{S}(t_1) + \mathbb{S}(t_2) \leq \rho \cdot s(t_1) + \rho \cdot s(t_2) = \rho \cdot s$.
- Case t = `if` B `then` t_1 `else` t_2. Similar to the previous case, after considering the two cases.
- Case t = `spguard`$(F,\, (S_p;$ `fork2join`$(L,\, R);\, S_m),\, S)$. We have two cases:
  - First case: the spguard chooses the parallel body. Then, $\mathbb{S} = \mathbb{S}(S_p) + \max(\mathbb{S}(L), \mathbb{S}(R)) + \phi + \tau + \mathbb{S}(S_m) \leq \rho \cdot s(S_p) + \max(\rho \cdot s(L), \rho \cdot s(R)) + \rho + \rho \cdot s(S_m) = \rho \cdot (s(S_m) + \max(s(L), s(R)) + s(S_p) + 1) = \rho \cdot s$.
  - Second case: the spguard chooses the sequential body. In this case by Lemma 4.4.6 we know that the sequential work of this spguard call is bounded: $W_s(g, I) \leq E\beta\kappa$. Thus, using the fact that the span is nonnegative ($s \geq 1$), we have:

$$\mathbb{S} = W_s(S, I) + \phi \leq E\beta\kappa + \phi \leq \rho \leq \rho \cdot s.$$

□

Lemma 4.4.7 (Bound on the number of critical calls)**.** *A given estimator involves no more than $P \cdot F$ critical calls.*

Proof. Let us consider the critical call on input I with measured time M by process p.

At first, we prove that M does not exceed κ. From Lemma 4.4.3, we have $\frac{M}{W} \leq E$. Because the critical call is small, we know that $W_s(S) \leq \frac{\kappa}{DE}$ and that all directly dominated calls are sequentialized, thus $W = W_s(S_p) + W_s(L) + W_s(R) + 1 + W_s(S_m)$. By the property of well-defined spguards $W_s(S_p) + W_s(L) + W_s(R) + 1 + W_s(S_m) \leq D \cdot W_s(S)$. Combining all these facts, we get $M \leq E \cdot W = E \cdot (W_s(S_p) + W_s(L) + W_s(R) + 1 + W_s(S_m)) \leq E \cdot D \cdot W_s(S) \leq E \cdot D \cdot \frac{\kappa}{DE} = \kappa$.

Second, after the report `Nmax` is at least $F(I)$: either another process updated `Nmax` to be not less than $F(I)$, or this report successfully updates `Nmax` and sets it to $F(I)$. This report can be successful since the reported time M does not exceed κ.

Next, we show that the cost of the next critical call by process p increases at least by a factor α. Suppose that the input of the next critical call is J. Our goal is to show $F(J) > \alpha \cdot F(I)$. For the spguard to choose parallel body on J, the boolean condition in `is_small` function ($F(J) \leq F(Y)$, or (($F(J) \leq \alpha \cdot F(Y)$) and ($F(J) \cdot T / F(Y) \leq \alpha \cdot \kappa$), where Y and T are the latest reported input and the measured time, respectively), needs to evaluate to `false`. Note that $F(I) \leq F(Y)$, since after the last report by process p the value of `Nmax` was at least $F(I)$, and $T \leq \kappa$, since the measure for Y was reported. Thus, at least one of the two following conditions needs to be satisfied: $F(J) > \alpha \cdot F(Y)$, or $F(J) > F(Y)$ and $F(J) > \frac{\alpha\kappa \cdot F(Y)}{T}$. The first case directly gives us the desired inequality, since $F(Y) \geq F(I)$.

For the second case, exploiting $T \leq \kappa$, we have: $F(J) > \frac{\alpha \cdot \kappa \cdot F(Y)}{T} \geq \frac{\alpha \cdot \kappa \cdot F(I)}{\kappa} = \alpha \cdot F(I)$. In summary, $F(J) > \alpha \cdot F(I)$.

Since critical calls are small calls, their work cannot exceeed $\frac{\kappa}{DE}$. By the assumption that the number of cycles exceeds the cost, the value of the cost function cannot exceed $\frac{\kappa}{DE}$, with κ expressed in cycles. The cost associated with the first critical report is at least 1 unit, and since the value increases by a factor α at least between every two consecutive critical call by the same process, the number of such critical calls cannot exceed $P \cdot (1 + \log_\alpha \frac{\kappa}{DE})$.

We denoted $1 + \log_\alpha \frac{\kappa}{DE}$ as F. □

Lemma 4.4.8 (Bound on the number of nested small calls)**.** *Small spguard calls may be nested at no more than on depth H.*

Proof. Consider a set of nested small calls. The outermost call is small, so it involves work at most $\frac{\kappa}{D \cdot E}$. As we argue next, the ratio between the work of a call directly nested into another one is at least $\frac{\gamma}{\gamma - 1}$. Combining the two, we can deduce that the number of nested small calls is bounded by $\log_{\gamma/(\gamma-1)} \frac{\kappa}{D \cdot E}$.

To bound the ratio between two directly nested calls, we proceed as follows. Consider a spguard with sequential body S on input I, directly dominated by a call to a spguard with sequential body S' on input I'. Assume, without loss of generality, that the inner spguard call occurs in the left branch, call it L, of the outer spguard call. From the γ-regularity assumption, we know that $\frac{W_s(L', I')}{W_s(S', I')} \leq 1 - \frac{1}{\gamma}$. Furthermore, since S executes as a subcomputation of the sequential execution of L', we have: $W_s(S, I) \leq W_s(L', I')$. Combining the two inequalities gives: $W_s(S, I) \leq (1 - \frac{1}{\gamma}) \cdot W_s(S', I')$, which can be reformulated as $\frac{W_s(S', I')}{W_s(S, I)} \geq \frac{\gamma}{\gamma - 1}$, meaning that the ratio between the two nested calls is at least $\frac{\gamma}{\gamma - 1}$. □

Lemma 4.4.9 (Bound on the number of parallel small calls)**.** *A given spguard involves no more than $P \cdot F \cdot H$ parallel small calls.*

Proof. First, we observe that each parallel small call must dominate at least one critical call. Indeed, when following the computation tree, there must be a moment at which we reach a spguard such that all dominated spguards choose sequential bodies, or such that there are no dominated spguards in the body.

Thus, we may bound the number of parallel small calls by multiplying the number of critical calls with the number of parallel small calls that dominate it (including the critical call, which is itself a parallel small call). Of course, we may be counting a same parallel call several times, but this over-approximation is good enough to achieve our bound. More precisely, we multiply the bound from Lemma 4.4.8 with the bound from Lemma 4.4.7. □

Lemma 4.4.10 (Bound on the number of covered sequential calls)**.** *A given spguard involves no more than $P \cdot F \cdot H \cdot (2\gamma - 2)$ covered sequential calls.*

Proof. A covered sequential call is dominated by a parallel small call. The claimed bound follows from the bound of Lemma 4.4.9 and the fact that, for each parallel small call, we can have at most $2\gamma - 2$ directly dominated sequential small calls. We next prove this last claim.

Consider a parallel small call on input I' to a spguard with sequential body S' and branches L' and R'. Consider a directly dominated call on input I_c to a spguard with sequential body S_c. By γ-regularity, we have: $W_s(S_c, I_c) \geq \frac{1}{\gamma} \cdot W_s(S', I')$. From the structure of the program, we know: $W_s(L') + W_s(R') \geq \sum_{c \in C} W_s(S_c, I_c)$, where C is the set of all directly dominated sequential small calls. Also, by the first property of γ-regular programs: $W_s(L'), W_s(R') \leq (1 - \frac{1}{\gamma}) \cdot W_s(S', I')$, giving us additional inequality $W_s(L') + W_s(R') \leq (2 - \frac{2}{\gamma}) \cdot W_s(S', I')$. By combining the facts, we get: $|C| \cdot \frac{1}{\gamma} \cdot W_s(S', I') \leq \sum_{c \in C} W_s(S_c, I_c) \leq W_s(L') + W_s(R') \leq (2 - \frac{2}{\gamma}) \cdot W_s(S', I')$. Thus, we deduce that the number of directly dominated sequential small calls $|C|$ does not exceed $2\gamma - 2$. □

Lemma 4.4.11 (Work involved in a non-covered sequential call)**.** *If a call to a spguard g on input I is a non-covered sequential call, then it involves at least some substantial amout of work, in the sense that:*

$$W_s(g, I) \geq \frac{\kappa}{\gamma DE}.$$

Proof. A call can be a non-covered sequential call for one of two reasons.

- First case: the call is directly dominated by another spguard call. This spguard call is necessarily parallel, and by assumption it is a non-small call (otherwise, the inner call would be covered). This non-small parallel call occurs on some spguard g' with sequential body S' executed on input I'. This call is not small, meaning that $W_s(S', I') > \frac{\kappa}{DE}$ holds. Besides, by the definition of γ-regularity, we have: $\frac{W_s(S,I)}{W_s(S',I')} \geq \frac{1}{\gamma}$. Combining the two inequalities gives: $W_s(S, I) > \frac{\kappa}{\gamma DE}$.
- Second case: the call is not dominated by any other spguard call. Then, by the last assumption from the definition of γ-regularity, we have $W_s(S) \geq \frac{\kappa}{\gamma DE}$.

□

Lemma 4.4.12 (Bound on the work excluding overheads during small calls)**.**

$$\mathbb{W}' \leq \left(1 + \frac{DE\gamma \cdot (\tau + 2\phi)}{\kappa}\right) \cdot w$$

Proof. Let us introduce the shorthand $\kappa' = \frac{\kappa}{\gamma DE}$. The bound is equivalent to $\mathbb{W}' \leq \left(1 + \frac{1}{\kappa'}\tau + \frac{2}{\kappa'}\phi\right) \cdot w$.

Let $B(w) = w + \frac{(w-\kappa')^+}{\kappa'}\tau + \frac{(2w-\kappa')^+}{\kappa'}\phi$, where x^+ is defined as x if x is non-negative, and 0, otherwise.

We establish the slightly tighter inequality $\mathbb{W}' \leq B(w)$, by induction on the execution tree.

- Case t = S. The program is sequential, so $\mathbb{W}' = \mathbb{W} = w \leq B(w)$.
- Case t = $(t_1; t_2)$. $\mathbb{W}' = \mathbb{W}'(t_1) + \mathbb{W}'(t_2) \leq B(w(t_1)) + B(w(t_2)) \leq B(w_1 + w_2)$. The last inequality holds, because $B(x) + B(y) \leq B(x + y)$ due to the fact $(x - k)^+ + (y - k)^+ \leq (x + y - k)^+$.
- Case t = `if` B `then` t_1 `else` t_2. Considering two different cases, similar to the previous case.
- Case t = `spguard`$(F, (S_p;$ `fork2join`$(L, R);\ S_m), S)$. Let I be the input and X be the trace for this call. Thereafter, we write $W_s(S)$ as short for $W_s(S, I)$ ad w as short for $w(t, I)$.

  By Lemma 4.4.4, we know that: $W_s(t, I) \leq W(t, I, X)$. By Lemma 4.4.5, we have $W(t, I, X) \leq w$. Combining the two gives $W_s(t) \leq w$.

  We then distinguish four cases:

  - First case: the call is the parallel non-small call. By definition of *small*, this means: $W_s(S) > \frac{\kappa}{DE}$. Let us focus first on the left branch. By the definition of γ-regularity, we know $\frac{W_s(L)}{W_s(S)} \geq \frac{1}{\gamma}$. Besides, for the same reason as we have $W_s(t) \leq w$, we have $W_s(L) \leq w(L)$. Combining these results, we get: $w(L) \geq \frac{1}{\gamma} \cdot W_s(S) > \frac{\kappa}{\gamma DE} = \kappa'$. By symmetry, we have $w(R) > \kappa'$. Also, we know that $B(x) + B(y) \leq B(x + y)$. Putting everything together gives: $\mathbb{W}' = \mathbb{W}(S_p) + \mathbb{W}'(L) + \mathbb{W}'(R) + \tau + \phi + \mathbb{W}(S_m) \leq B(w(S_p)) + B(w(L)) + B(w(R)) + \tau + \phi + B(w(S_m)) = B(w(S_p)) + (w(L) + w(R)) + \frac{(w(L)-\kappa')+(w(R)-\kappa')+\kappa'}{\kappa'}\tau + \frac{(2w(L)-\kappa')+(2w(R)-\kappa')+\kappa'}{\kappa'}\phi + B(w(S_m)) \leq B(w(S_p)) + B(w(L)+w(R)) + B(w(S_m)) \leq B(w(S_p)+w(L)+w(R)+w(S_m)) = B(w-1) \leq B(w)$.

– Second case: the call is the parallel small call. By definition of $\mathbb{W}'$, we do not count the overheads within the scope of this call and only count the raw work, thus $\mathbb{W}' \leq W(t, I, X)$. Recall that $W(t, I, X) \leq w$. Besides, by definition of B, we have: $w \leq B(w)$. Combining these results gives: $\mathbb{W}' \leq B(w)$.

– Third case: the call is the covered sequential call. In this case, the call is dominated by a parallel small call. Thus, such calls are never reached by our proof by induction, because to reach them one would necessarily first go through the case that treats parallel small calls, case which does not exploit an induction hypothesis. (Recall that the definition of $\mathbb{W}'$ excludes all the overheads involved throughout the execution of a parallel small call.)

– Fourth case: the call is the non-covered sequential call. In this case, the work equals to the sequential work, $w = W_s(S)$, and $\mathbb{W}'$ does not exclude the overheads, so $\mathbb{W}' = W_s(S) + \phi$. By Lemma 4.4.11, we have: $W_s(S) \geq \frac{\kappa}{\gamma DE}$. In other words, $W_s(S) \geq \kappa'$. This inequality may be reformulated as: $\frac{2W_s(S)-\kappa'}{\kappa'} \geq 1$. Recall that we have $W_s(S) \leq w$, since here $W_s(t) = W_s(S)$. Combining all these results yields:

$$\mathbb{W}' = W_s(S) + \phi \leq W_s(S) + \frac{2W_s(S) - \kappa'}{\kappa'}\phi \leq B(W_s(S)) \leq B(w).$$

□

Lemma 4.4.13 (Bound on the overheads associated with small calls)**.**

$$\mathbb{W} - \mathbb{W}' \;\leq\; PFGH \cdot (\tau + (2\gamma - 1) \cdot \phi)$$

Proof. This bound on the overheads associated with small calls is obtained as the sum of the overheads associated with covered sequential calls and the overheads associated with parallel small calls. For the former, each covered sequential call induces an overhead of ϕ, and there are at most $PFH \cdot (2\gamma - 2)$ of them per spguard, by Lemma 4.4.10. For the latter, each parallel small call induces an overhead of $\tau + \phi$, and there are at most PFH of them per spguard, by Lemma 4.4.9. Multiplying by G, the number of spguards, and factorizing the sum leads to the aforementioned bound. □

Theorem 4.4.2 (Bound on the total work using our algorithm)**.**

$$\mathbb{W} \;\leq\; \left(1 + \frac{DE\gamma \cdot (\tau + 2\phi)}{\kappa}\right) \cdot w \;+\; PFGH \cdot (\tau + 2\gamma \cdot \phi)$$

Proof. Obtained by summing the bound on $\mathbb{W}'$ obtained from Lemma 4.4.12 and the bound on $\mathbb{W} - \mathbb{W}'$ obtained from Lemma 4.4.13 with the fact that $2\gamma - 1 < 2\gamma$. □

Theorem 4.4.4 (Bound on the running time of γ-regular program)**.** *Consider the γ-regular program with well-defined spguards. Let T_P be the running time of the program on a machine with P processes and a greedy scheduler. Let w and s be the work and span of the program, correspondingly. G, F and H are as defined in 4.4.13, 4.4.14 and 4.4.15. We have:*

$$T_P \;\leq\; \left(1 + \frac{\gamma ED \cdot (\tau + 2\phi)}{\kappa}\right) \cdot \frac{w}{P} \;+\; (1 + \phi + \max(\tau, E\beta\kappa)) \cdot s \;+\; FGH \cdot (\tau + 2\gamma\phi).$$

Proof. Combining Theorem 4.4.1 about the total span and Theorem 4.4.2 about the total work together with Brent's Theorem 2.2.1, we get our main theorem. □

Theorem 4.4.3 (Bound on the parallel run time)**.** *Under two assumptions $\kappa \geq \tau$ and $\kappa \geq 1 + \phi$, the bound on the parallel running time from the previous theorem can be slightly simplified:*

$$T_P \;\leq\; \left(1 + \frac{\gamma ED \cdot (\tau + 2\phi)}{\kappa}\right) \cdot \frac{w}{P} \;+\; (E\beta + 1) \cdot \kappa \cdot s \;+\; O\left(G \cdot \log^2 \kappa \cdot (\tau + 2\gamma\phi)\right).$$

Proof. The proof is straightforward. At first, since $\kappa \geq \tau$, $\kappa \geq 1 + \phi$ and $E, \beta \geq 1$ we obtain: $(1 + \phi + \max(\tau, E\beta\kappa)) \leq (1 + E\beta) \cdot \kappa$. Secondly, $F = 1 + \log_\alpha \frac{\kappa}{DE}$ and $H = \log_{\gamma/(\gamma-1)} \frac{\kappa}{DE}$, thus $F \cdot H = O(\log^2 \kappa)$. □

Theorem 4.3.1 (Simplified bound on the parallel run time)**.** *For fixed hardware and any program, all parameters of the analysis except for κ which represents the unit of parallelism can be replaced with constants, leaving us with the following bound:*

$$T_P \leq \left(1 + \frac{O(1)}{\kappa}\right) \frac{w}{P} + O(\kappa) \cdot s + O\left(\log^2 \kappa\right).$$

Proof. Immediate from the previous theorem. □

4.5 Experimental Evaluation

For our experiments we chose 11 benchmarks from the PBBS benchmark suite [142], which covers diverse algorithms and is designed to compare different parallel programming methodologies in terms of performance. Here we list the benchmarks: *blockradix-sort* stably sorts fixed-length unsigned integer keys with the ability to carry along fixed-length auxiliary data; *comparison-sort* sorts a sequence of any type given a comparison function; *suffix-array* sorts all suffixes of a given string; *convex-hull* build a convex hull over a set of points in 2D; *nearest-neighours* given a set of points in 2D or 3D finds k nearest neighbours for each point; *ray-cast* given a set of triangles inside a 3D bounding box and a set of rays calculates for each ray the first triangle it intersects; *delaunay* given a set of points in 2D calculates the Delaunay triangulation for them; *mis* given an undirected graph calculates a maximal independent set, i.e., a set of vertices, such any pair of points are not adjacent and no other vertex can join this set; *mst* given a weighted undirected graph calculates a minimum spanning tree (or forest), i.e., a subgraph without cycles, in which any pair of vertices is connected, and which has the minimum possible total edge weight; *spanning* given an undirected graph calculates any spanning tree (or forest); *BFS* given an unweighted undirected graph and a source calculates the shortest path from the source to all vertices.

PBBS includes only four more benchmarks: *delaunay-refine* given a Delaunay triangulation in 2D calculates new Dealunay triangulation with all original points plus points such that no triangle has an angle smaller than a threshold θ; *n-body* given n points in 3D calculates the gravitational force vector on each point due to all other points; *remove-duplicates* removes duplicates from a sequence of any type; *maximal-matching* given an undirected graph calculates a maximal matching, i.e., the set edges such that any pair of edges does not share a vertex and that no other edge can join this set. We do not consider them in our evaluation analysis because: *delaunay-refine* and *n-body* have critical code regions that do not readily admit a cost function; *remove-duplicates* and *maximal-matching* exhibit a very high variability in the running time of their critical loops, due to heavy use of atomic operations that trigger massive bus contention. We leave it to future work to investigate how to generalize our approach to support these algorithms.

The code provided for these benchmarks in the PBBS suite is highly-optimized. It relies on a number of sophisticated techniques to control granularity, with careful engineering to select the technique and hand-tune threshold settings. Two of the benchmarks we consider, namely *ray-cast* and *delaunay*, are among the largest and most complex codes in PBBS. The *BFS* and *nearest-neighbour* benchmarks are also notable because these benchmarks were identified by the PBBS authors as problem cases for granularity control.

For our programs, we ported the original code by modifying only what was needed to set up our automatic granularity. Overall, we frequently relied on default cost functions provided by our library, and only provided 24 explicit cost functions. Changes are summarized as follows.

- Several divide-and-conquer algorithms involved a manually-fixed grain to control granularity (like in Figure 4.3). We replaced them with a spguard (like in Figure 4.4), thereby eliminating the hardware-dependent magic numbers.

- A number of loops were parallelized by splitting in fixed size blocks of 2048 items each — a pattern widely used throughout the PBBS sequence library. We replaced all of them with our automatically-controlled parallel-for loop.

- Other loops exploited Cilk parallel for-loops, which essentially split the loop range in $8P$ blocks, where P is the number of cores. We also replaced all of them with our automatically-controlled loops.

- A number of inner loops were forced to be sequential, even though they could have been parallel. As we learned from private communication with the authors, the purpose was to tame the overheads. We restored parallelism, using our automatically-controlled loops.

- A number of loops were forced to always make one spawn per iteration, using a Cilk for-loop with grain size 1. Doing so is needed in situations were the loop body itself may need to generate parallel spawns, because nontrivial grain size might dramatically reduce parallelism. Again, we replaced all such loops with our automatically-controlled loops, which properly support nested parallelism.

In summary, from a programmer's perspective, automatic granularity control enables replacing careful selection of techniques with a single, uniform technique for controlling granularity. Furthermore, our automatic approach does not require labor-intensive tuning of grain sizes, and it automatically adapts to the hardware — performance is portable.

These significant benefits are not completely free. Indeed, our algorithm needs to infer granularity thresholds online, through a convergence phase. Nevertheless, as established by our analysis (and, as we confirm through our experimental results), the cost of the convergence phase generally accounts for at most a few percent of the parallel run time.

In addition to its engineering benefits, automatic granularity control may also deliver better results. Depending on the situations, the elimination of Cilk for-loops may either increase the number of spawns, possibly increasing utilization without noticeably increasing the overheads; or it may decrease the number of spawns, possibly decreasing overheads without noticeably decreasing parallelism. Likewise, the parallelization of inner sequential loops may increase the parallelism available without incurring significant overheads. Thus, we may expect to see, on a number of benchmarks, significant performance improvement.

4.5.1 Experimental Setup

Our primary test harness is an Intel machine with 40 cores. We compiled the code using GCC (version 6.3) using the extensions for Cilk Plus (options `-O2 -march=native -fcilkplus`). Our 40-core machine has four 10-core Intel E7-4870 chips, at 2.4GHz, with 32Kb of L1 and 256Kb L2 cache per core, 30Mb of L3 cache per chip, and 32GB RAM, and runs Ubuntu Linux kernel v3.13.0-66-generic. For this machine, we used $\kappa = 25$ microseconds and $\alpha = 1.5$. For each data point, we report the average running time over 30 runs. The variation in the running times is negligible: overall, we observed only a few cases where the standard deviation is 5%, but it was usually below 3%.

The benchmarks in the PBBS suite were originally implemented in `C++`, using Cilk extensions to realize parallelism. As baselines we use the author's original code for each benchmark. Note that that code was tuned by the authors offline, on a collection of inputs, using a test machine similar to ours, and using GCC compiler like we did.

4.5.2 Input Data Description

For input data, we used mostly the same data sets used in the original PBBS study [142], but in a number of cases using newly acquired data.

For *radix-sort*, we used a variety of inputs of 10^8 items. The *random* input consists of 32-bit integers $\in [0, 2^{31})$ drawn from the uniform distribution and *exponential* from the exponential distribution. The input *random kvp 256* consists of integer pairs (k, v) such that $k \in [0, 2^{31})$ and $v \in [0, 256)$, and *random kvp* 10^8 with $v \in [0, 10^8)$.

For *comparison-sort*, we used a variety of inputs of 10^8 items. The *random* input consists of 64-bit floats $\in [0, 1)$ from a uniform distribution, and *exponential* from an exponential distribution. The *almost sorted* input consists of a sorted sequence 64-bit floats $\in [0, 10^8)$ that is updated with 10^4 random swaps. The *trigrams* input consists of strings generated using trigram distribution.

For *suffix-array*, we used one synthetic and three non-synthetic inputs. The *trigrams* input consists of a string of length 10^8 generated using trigram distribution. The *dna* input consists of a DNA sequence and has about 32 million charachters. The *text* input consists of about 105 million characters drawn from Project Gutenberg. The *wiki* input consists of 100 million characters taken from wikipedia's xml source files.

For *convex-hull*, we used a variety of inputs of 10^8 2D points. The *in circle* input consists of points inside the unit circle centered at the origin, *on circle* consists of points on the unit circle centered at the origin, and *kuzmin* consists of points from Kuzmin's distribution [30].

For *nearest neighbors*, we used a variety of inputs of 10^8 2D and 3D points. The inputs *in square* and *on square* consist of 2D points in and on, respectively, the unit square centered at the origin. The input *kuzmin* consists of 2D points drawn from the Kuzmin distribution. The input *plummer* consists of 3D points drawn from the Plummer distribution [133]. The inputs *in sphere* and *on sphere* consist of 3D points in and on, respectively, the unit sphere centered at the origin.

For *ray-cast*, we used a variety of synthetic and non-synthetic inputs. The input *in cube* consists of 10^6 triangles with vertices drawn from the unit cube centered at the origin, and *on sphere* consists of 10^6 triangles with vertices drawn from the unit circle centered at the origin. The input *happy* consists of happy Buddha mesh from the Stanford 3D Scanning Repository, and it consists of 1087716 triangles. The input *xyz-rgb-manusript* comes from the same repository and consists of 4305818 triangles. The input *turbine* comes from a different repository and consists of 1765388 triangles.[1] For each of the mesh with n triangles, n rays were generated: the start of each ray is randomly drawn from the lowest side of the bounding box of the mesh and the end of each ray is randomly drawn from the upper side of the bounding box of the mesh.

For *delaunay*, we used two inputs consisting of 10^7 2D points. The input *in square* consists of points in the unit square, and *kuzmin* consists of points drawn from the Kuzmin distribution.

For *mis* and *spanning*, we used three input graphs. The graph *cube-grid* (33076161 vertices and 99228483 edges) is represented as follows: the vertices are the unit cubes of the cube with side 321 and an edge exist between two vertices if the cubes share the side. The graph *rMat24* (*rMat27*) is a synthetic graph on 16777216 vertices and 119845397 (118768790) generated with power-law distribution degrees [45] with settings $a = 0.5, b = c = 0.1$ and $d = 0.3$ ($a = 0.57, b = c = 0.19, d = 0.05$).

For *mst*, we used the same three graphs as above. A weight of each edge is generated uniformly at random from $[0, 2^{31})$.

For *bfs*, we used 13 different graphs that is a representative subset of the graphs used in a different performance study in [4]. This subset includes small-world graphs, such as the networks of livejournal and twitter, and high-diameter graphs, such as a map of Europe

[1] Large Geometric Models Archive at Georgia Institute of Technology http://www.cc.gatech.edu/projects/large_models/blade.html

and cube-grid. We already described three graphs *cube-grid*, *rMat24* and *rMat27* above. The graphs *livejournal* and *twitter* describe social networks [1, 109]: the first has 4 847 571 vertices and 68 993 773 edges, and the second has 41 652 231 vertices and 1 468 365 182 edges. The graph *wikipedia* (as of 6 February 2007) is taken from University of Florida sparse-matrix collection [54]: 3 566 907 vertices and 45 030 389 edges. The graph *europe* is chosen from DIMACS challenge problems [124]: 50 912 018 vertices and 108 109 320 edges. The remaining six graphs are synthetic. The directed graph *square-grid* (49 999 041 vertices and 99 998 082 edges) is represented as follows: the vertices are the cells of the square with side 7071 and a cell has at most two out-edges, to the right neighbour cell and to the up neighbour cell. The directed graph *par-chains-100* (50 000 002 vertices and 50 000 100 edges) is 100 directed chains with 500 000 vertices each that start in one source vertex and finish in one sink vertex. The directed graph *trunk-first* (10 000 001 vertices and 10 000 000 edges) is a directed chain of length 5 000 000 each vertex of which (except for the last one) has additional out-edge to a vertex with out-degree zero. The directed graph *phases-10-d-2* (33 333 331 vertices and 93 333 306 edges) has a source vertex and 10 layers each with 3 333 333 vertices, a source has out-edges to all vertices of the first layer, exactly one vertex of each layer has out-edges to all the vertices of the next layer, and any other vertex has out-edges to 2 random vertices of the next layer. The directed graph *phases-50-d-5* (40 000 001 vertices and 196 800 000 edges) has a source vertex and 50 layers each with 800 000 vertices, a source has out-edges to all vertices of the first layer, exactly one vertex of each layer has out-edges to all the vertices of the next layer, and any other vertex has out-edges to 5 random vertices of the next layer. The directed graph *trees-524k* (99 712 001 vertices and 99 712 000 edges) is a directed chain of length 381 each vertex of which (except for the last one) has 524 287 additional out-edges to vertices with out-degree zero. The last three graphs are *cube-grid*, *rMat24* and *rMat27* described above.

4.5.3 Main PBBS Results

The performance results appear in Table 4.2. The first column indicates the execution time of the baseline, while the second one indicates the relative performance of our version featuring automatic granularity control. For example, -10% indicates that our code is 10% faster. Overall, our oracle-guided algorithm performs effectively the same and, in six cases, between 23% and 36% better than the unaltered authors' code.

Interestingly, we learned from private communication that the PBBS authors were aware of granularity problems affecting nearest-neighbors and *BFS*, but not for other codes, such as blockradix-sort, sample-sort, and convex-hull. Even carefully hand-tuned codes can contain inefficiencies related to granularity control that are not necessarily obvious and can easily escape detection.

4.5.4 Parallel BFS

In addition to the other PBBS programs, we considered the non-deterministic nested *BFS* benchmark, named ndBFS in the PBBS paper. We chose ndBFS because it is the fastest among the alternative *BFS* implementations in the PBBS suite and, to the best of our knowledge, the fastest publicly available Cilk implementation of *BFS*.

There are two versions of *BFS*: the *flat* and the *nested* version (ndBFS). In the flat version, the traversal over the neighbors of each vertex is performed sequentially. In the nested version, it is performed using a parallel loop, allowing to process in parallel the out-edges of vertices with high out-degree.

PBBS currently uses the flat version. Via personal communication, we learned that the authors sequentialized the inner loop because (1) the graphs they considered did not feature such high-degree vertices, and (2) they observed that performance improved for some of their test graphs when this loop was serialized. Columns 2 and 4 from Figure 4.7 give the execution time for the flat PBBS *BFS*, and for its nested counterpart, using a

Application/input	PBBS (s)	Ours
blockradix-sort		
random	0.20	−7.4%
exponential	0.19	−8.4%
random kvp 256	0.49	−23.9%
random kvp 10^8	0.49	−27.7%
comparison-sort		
random	1.13	−36.4%
exponential	0.82	−31.3%
almost sorted	0.63	−18.8%
suffix-array		
trigrams	3.58	−6.3%
dna	1.29	−6.7%
text	4.11	−7.4%
wiki	3.66	−5.3%
convex-hull		
in circle	0.61	+5.8%
kuzmin	0.41	−6.9%
on circle	8.26	−32.4%
nearest-neighbours		
in square	5.75	−2.2%
kuzmin	22.00	−2.5%
in cube	7.90	−6.5%
on sphere	14.60	−31.2%
plummer	23.54	−2.5%
ray-cast		
in cube	7.90	−1.9%
on sphere	0.87	−0.2%
happy	0.50	−1.9%
xyz-rgb manuscript	9.46	+0.3%
turbine	4.10	−2.1%
delaunay		
in square	3.39	−4.1%
kuzmin	3.99	−4.4%
mis		
cube-grid	0.12	+1.2%
rMat24	0.07	+2.7%
rMat27	0.06	+2.7%
mst		
cube-grid	2.28	−9.9%
rMat24	2.21	−13.3%
rMat27	1.89	−16.3%
spanning		
cube-grid	0.62	−5.8%
rMat24	0.44	−0.6%
rMat27	0.33	−5.0%

Table 4.2: Results from PBBS benchmarks, executed on 40 cores, averaged over 30 runs. Figures show that automatic granularity control can be achieved at the expense of small overheads, and that in fact it often decreases the parallel runtime compared with carefully hand-tuned code.

	Flat		Nested		Ours nested vs. PBBS flat
Graph	PBBS (s)	Ours	PBBS (s)	Ours	
livejournal	0.12	+10.3%	0.18	−26.3%	+12.4%
twitter	1.74	−11.3%	2.14	−29.3%	−13.0%
wikipedia	0.11	+1.2%	0.15	−27.8%	−0.4%
europe	3.62	−30.6%	3.62	−22.9%	−23.0%
rmat27	0.19	+2.5%	0.33	−35.7%	+8.4%
rmat24	0.32	+6.0%	0.32	+6.1%	+4.4%
cube-grid	0.71	−2.2%	0.67	+7.6%	+0.3%
square-grid	3.81	−29.0%	3.82	−30.4%	−30.3%
par-chains-100	65.4	−80.9%	66.9	−70.7%	−70.0%
trunk-first	16.2	−52.1%	15.4	−51.2%	−53.4%
phases-10-d-2	1.33	+16.0%	0.49	+7.3%	−60.1%
phases-50-d-5	0.63	+2.2%	0.58	+8.2%	+0.4%
trees-524k	10.8	+14.8%	1.14	+27.6%	−86.5%

Figure 4.7: Parallel BFS experiment on 40 cores (30 runs).

parallel `cilk_for` loop over the edges. The figures confirm that overheads of parallelization using `cilk_for` are significant, sometimes above 50%.

In contrast, our algorithm, which supports nested parallelism, delivers significantly better results for nested BFS, as reported in the 5th column from Figure 4.7 ("nested/ours"). The last column from this figure shows our main result: it compares the performance of the original authors' flat *BFS* versus our nested *BFS* with automatic granularity control. Our version either performs about as well or up to 86% faster.

4.5.5 Portability Study

We run the same set of experiments on two additional machines. As baselines we use the PBBS author's code, unaltered.

Portability: 48-core AMD machine This machine has four 12-core AMD Opteron 6172 processors running at 2.1GHz. Each core has 64KB of L1 instruction and data cache and a 512KB L2 cache. Each processor has two 6MB L3 cache that is shared with the four cores of the processor. The system has 128Gb of RAM and runs Ubuntu Linux 10.04.2 LTS. We tuned κ and α to be 30 microseconds and 1.1, respectively. Results are shown in Table 4.10 and Figure 4.8.

Portability: 72-core Intel Xeon machine This machine features four Intel Xeon E7-8867 chips, each with 18 cores. Each core has a 32Kb L1 cache and a 256Kb L2 cache, and there is one 45Mb shared cache per chip. The system has 1Tb of RAM and runs is running Ubuntu v16.04 LTS (with Linux kernel version 4.10.0-27). We tuned κ and α to be 40 microseconds and 1.2, respectively. Results are shown in Table 4.11 and Figure 4.9.

Although the architecture of this machine is similar to our primary 40-core Intel machine, this 72-core machine is also useful to test the scalability of our granularity control approach.

One outlier value it the +14% for the BFS algorithm on the cube-grid graph. This slowdown is due to the particular graph structure and its particular size: the BFS traversal of this graph exhibits a critically-low amount of parallelism, explaining why the slowdown did not show up on the other two test machines. For this input graph, the `cilk_for` loop manages to exploit slightly more parallelism within each round of a succession of several BFS frontiers. It does so by spawning $8P$ parallel tasks, each of size less than

	Flat		Nested		Ours nested vs. PBBS flat
Graph	PBBS (s)	Ours	PBBS (s)	Ours	
livejournal	0.26	+4.9%	0.33	−14.6%	+6.9%
twitter	3.15	−5.2%	3.40	−17.1%	−10.4%
wikipedia	0.22	+6.1%	0.27	−13.7%	+3.0%
europe	4.92	−14.2%	4.95	−12.8%	−12.3%
rmat27	0.35	+10.6%	0.49	−24.2%	+5.2%
rmat24	0.55	+10.1%	0.57	+4.6%	+8.8%
cube-grid	1.04	+4.3%	1.04	+4.6%	+4.7%
square-grid	4.77	−25.6%	4.77	−24.7%	−24.8%
par-chains-100	62.8	−66.3%	62.9	−66.2%	−66.2%
trunk-first	10.1	+0.9%	10.1	+0.1%	−0.3%
phases-10-d-2	1.99	+9.4%	0.71	+9.9%	−60.9%
phases-50-d-5	1.43	+18.2%	1.24	+25.3%	+8.1%
trees-524k	37.9	+0.6%	2.64	+32.9%	−90.7%

Figure 4.8: Results from the BFS experiment, executed on the 48-core AMD machine, averaged over 30 runs.

κ. Generally, doing so would result in noticeable overheads, but here it gives a small advantage, because on each of the frontiers considered in turns by the BFS algorithm, there is barely enough work to feed all cores. The relative conservatism of our algorithm gives it in such circumstances a slight disadvantage; yet, this same conservatism is what gives our algorithm a major speedup on many other graphs.

4.5.6 Summary

These results show that, as expected, individual benchmarks could perform slightly differently on different machines, but also that the overall trends remain the same. Therefore, we conclude that, as suggested by our theoretical results, our technique appears to be portable across different machines.

4.6 Related Work

Controlling the overheads of parallelism has been an important open problem since the early 1980's, when it was first recognized that practical overheads can overwhelm the benefits of parallelism [83]. Since then, researchers have explored two separate approaches that reduce overheads by reducing the number of created threads. Note that these approaches are complementary to approaches that reduce overheads by reducing per-task costs [3, 72, 94, 107, 136, 148].

Granularity Control

The granularity control is a technique that reduces the number of threads by selectively switching from a parallel code to its, possibly faster, sequential alternative. Perhaps the oldest granularity control technique is to use *condition-guards* for each task creation: "cut-off" conditions that switch from parallel to a sequential mode of execution. Researchers have addressed the limitations of such manual granularity control by using various forms of automation.

Weening et al. [130, 155] proposed two methods based on *height* and *depth* cutoffs. The "height cutoff" approach approximates the height of the current subcomputation in the execution tree and then compares it against the single properly chosen cutoff constant. The

Graph	Flat		Nested		Ours nested vs. PBBS flat
	PBBS (s)	Ours	PBBS (s)	Ours	
livejournal	0.09	−0.4%	0.13	−34.2%	+3.3%
twitter	1.10	−17.5%	1.29	−34.4%	−22.5%
wikipedia	0.08	−12.7%	0.10	−33.2%	−11.3%
europe	2.77	−22.9%	2.77	−6.7%	−6.9%
rmat27	0.12	+6.3%	0.20	−39.2%	+4.9%
rmat24	0.17	+2.1%	0.18	+0.1%	+4.1%
cube-grid	0.41	+7.9%	0.41	+14.1%	+14.3%
square-grid	2.67	−25.7%	2.67	−20.8%	−20.9%
par-chains-100	49.4	−88.4%	49.3	−87.4%	−87.4%
trunk-first	13.8	−56.9%	14.0	−56.1%	−55.6%
phases-10-d-2	0.93	+11.0%	0.21	+9.0%	−75.9%
phases-50-d-5	0.52	+13.9%	0.45	+19.0%	+4.6%
trees-524k	8.28	+7.5%	0.80	+23.4%	−88.1%

Figure 4.9: Results from the BFS experiment, executed on the 75-core Intel machine, averaged over 30 runs.

two drawbacks are: (1) sometimes the height of the subcomputation cannot be deduced; (2) the increase of the problem size results in the increase of the number of created threads and, thus, the cutoff constant should be adjusted. The "depth cutoff" approach compares the *depth*, i.e., the distance from the root to a current subcomputation in a computation tree, with a single properly chosen cutoff constant. Its disadvantages are: (1) the depth cannot provide an approximation of the size of the task and on skewed computation trees this approach can create very small tasks; (2) when the code has several functions that depend on each other, it becomes non-trivial to find the optimal cutoff constant.

Suppose that a function f has a sole argument — a list l. The algorithm by Huelsbergen et al. [98] compares the length of l with a cutoff constant. This cutoff constant C is deduced statically by abstractly executing f on lists of different sizes: C is the smallest value such that the execution time of f on the list of length C exceeds the overhead threshold T. Note that this approach works only for functions which execution time depends on the length of exactly one argument.

Lopez et al. [55] proposed an approach similar to ours, but in the context of logic programming: the programmer is required to provide cost functions and, then, the approach decides to execute in parallel or sequentially by comparing costs with cutoff constants. However, these cutoff constants are precomputed statically, while our algorithm does not require preliminary tuning.

The algorithm by Duran et al. [60] uses an average execution time of subcomputations on each depth to make predictions: if the average time for the current depth exceeds some cutoff constant then the code is executed in parallel, otherwise, sequentially. To find the average time for each depth, the algorithm measures the sequential execution time of the subcomputations (as the sum of the execution times of sequential pieces) and, then, reports it. If the number of reports for the current depth is big enough, e.g., 100, then the average time is computed as an average time of reports. Despite the fact that this approach is fully automatic, the average time is not a proper approximation of the work. For example, if we received reports from 100 nodes on depth d with very little work then any node on depth d with a lot of work will be executed sequentially leading to the loss of parallelism.

Another approach to control granularity is proposed by Thoman et al. [150]. It radically differs from the previous techniques. At first, it compiles different versions of each function:

Application/input	PBBS (s)	Ours
blockradix-sort		
random	0.33	−2.3%
exponential	0.32	−2.5%
random kvp 256	0.58	+1.1%
random kvp 10^8	0.58	−0.0%
comparison-sort		
random	1.24	−31.2%
exponential	0.96	−25.5%
almost sorted	0.82	−19.9%
suffix-array		
trigrams	7.07	+0.8%
dna	4.01	+2.0%
text	9.21	+0.9%
wiki	7.60	+2.2%
convex-hull		
in circle	1.04	+5.6%
kuzmin	0.73	+2.1%
on circle	10.53	−18.1%
nearest-neighbours		
in square	8.65	+1.6%
kuzmin	32.57	+1.3%
in cube	10.75	+0.3%
on sphere	33.38	+0.0%
plummer	36.51	+0.3%
ray-cast		
in cube	12.94	−3.5%
on sphere	2.09	+1.8%
happy	1.75	+4.8%
xyz-rgb manuscript	15.78	−0.4%
turbine	9.80	−0.2%
delaunay		
in square	7.33	−1.0%
kuzmin	8.73	−4.5%
mis		
cube-grid	0.25	−1.4%
rMat24	0.17	+0.4%
rMat27	0.14	−4.5%
mst		
cube-grid	3.94	−9.6%
rMat24	4.09	−9.0%
rMat27	3.29	−6.5%
spanning		
cube-grid	1.09	+0.0%
rMat24	0.81	+0.1%
rMat27	0.58	−3.0%

Figure 4.10: Results from PBBS benchmarks, executed on the 48-core AMD machine, averaged over 30 runs.

Application/input	PBBS (s)	Ours
blockradix-sort		
random	0.12	−10.0%
exponential	0.11	−7.8%
random kvp 256	0.24	−20.6%
random kvp 10^8	0.24	−20.6%
comparison-sort		
random	0.50	−34.9%
exponential	0.38	−29.6%
almost sorted	0.26	−15.3%
suffix-array		
trigrams	2.17	+0.4%
dna	1.19	−3.2%
text	2.80	−1.1%
wiki	2.34	+0.7%
convex-hull		
in circle	0.39	−4.2%
kuzmin	0.25	−11.9%
on circle	3.86	−30.5%
nearest-neighbours		
in square	2.69	−1.1%
kuzmin	12.54	+1.9%
in cube	3.42	−2.4%
on sphere	14.40	+5.0%
plummer	16.39	−0.3%
ray-cast		
in cube	3.01	−4.4%
on sphere	0.77	+4.3%
happy	0.69	+5.6%
xyz-rgb manuscript	5.81	+2.8%
turbine	2.40	+2.6%
delaunay		
in square	2.43	−4.3%
kuzmin	2.89	−8.9%
mis		
cube-grid	0.09	−5.7%
rMat24	0.06	+3.0%
rMat27	0.05	+1.8%
mst		
cube-grid	1.34	−13.0%
rMat24	1.24	−8.4%
rMat27	1.12	−11.6%
spanning		
cube-grid	0.37	−8.3%
rMat24	0.27	−3.8%
rMat27	0.21	−2.3%

Figure 4.11: Results from PBBS benchmarks, executed on the 72-core Intel machine, averaged over 30 runs.

the parallel version p; the sequentialized version s; the version p_1 where all immediate calls in p are inlined; the version p_2 where all immediate calls in p_1 are inlined; etc. These versions are used to avoid overheads related to function calls. Second, the algorithm maintains two variables: a *task demand* (keeps track of other worker's unfulfilled attempts to steal tasks from the worker) and a *queue length* (how many tasks the worker currently has). Using a task demand and a queue length the algorithm decides which version of a function to execute: s, p, p_1, p_2, etc. Unfortunately, this approach can oversequentialize. Suppose a process withdraws a huge task (for example, the program is irregular, and this is the biggest task created) to execute and sees that all processes are busy. The process decides to sequentialize that task and, thus, the parallelism is lost.

The latest technique was proposed by Iwasaki et al. [102]. For each function it uses a static analysis to find a condition on arguments in which the height of the subcomputation is within a specifically chosen constant height H. Thus, when a function is executed the condition is checked: if it is satisfied, the function is executed sequentially, otherwise, it is executed in parallel. Further, the technique uses a sophisticated compiler support to provide static optimizations, such as *code-bloar-free inlining* (inlining of the immediate function calls) and *loopification* (transforming tasks into vectorizable loops). The major drawback of this approach is that it uses height cutoffs: as shown by Weening [155] the height is not a good approximation of the sequential execution time. In addition, in complex programs the static analysis cannot deduce the conditions and the approach has to decide on runtime information such as depth and the number of ready tasks: as shown in prior works none of these properties can reasonably approximate the sequential execution time.

Lazy Task Creation

In contrast to granularity control, *lazy task creation* (or, sometimes, *lazy scheduling*) focuses on reducing the number of tasks by observing the load in the system and creating tasks only when necessary [25, 66, 118, 153]. Although, lazy task creation can reduce overheads of parallelism, it has some important limitations. First, there is no way to account for the fact that an alternative sequential algorithm is more efficient than the original parallel one executed on one process. So, it is reasonably common, that an algorithm under lazy task creation works 3-10 times slower than the same algorithm under manual granularity control.

Another issue with lazy task creation is that it requires regular checks whether some process made a work request. Unfortunately, the simplest way to implement it is to use a compiler support that injects polling checks [67].

However, we think that granularity control, such as our algorithm, and lazy task creation are complementary. For example, when it is impossible to provide a cost function lazy task creation can improve performance, and when it is possible our granularity control can improve performance by switching to a sequential alternative.

4.7 Conclusion

The problem of managing parallelism-related overheads effectively is an important problem facing parallel programming. The current state of the art in granularity control places the burden of the tuning on the programmer. The tuning process is labor intensive and results in highly engineered, possibly performance-brittle code. In this paper, we show that it is possible to control granularity in a more principled fashion. The key to our approach is an online algorithm for efficiently and accurately estimating the work of parallel computations. We show that our algorithm can be integrated into a state-of-the-art, implicitly parallel language and can deliver executables that compete with and sometimes even beat expertly hand-tuned programs.

5 A Concurrency-Optimal Binary Search Tree

5.1 Introduction

To meet modern computational demands and to overcome the fundamental limitations of computing hardware, the traditional single-CPU architecture is being replaced by a concurrent system based on multi-cores or even many-cores. Therefore, at least until the next technological revolution, the only way to respond to the growing computing demand is to invest in smarter concurrent algorithms.

Synchronization, one of the principal challenges in concurrent programming, consists in arbitrating concurrent accesses to shared *data structures*: lists, hash tables, trees, etc. Intuitively, an efficient data structure must be *highly concurrent*: it should allow multiple processes to "make progress" on it in parallel. Indeed, every new implementation of a concurrent data structure is usually claimed to enable such parallelism. But what does "making progress" means precisely?

Optimal Concurrency

If we zoom in the code of an operation on a typical concurrent data structure, we can distinguish *data accesses*, i.e., reads and updates to the data structure itself, performed as though the operation works on the data in the absence of concurrency. To ensure that concurrent operations do not violate correctness of the implemented high-level data type (e.g., *linearizability* [89] of the implemented set abstraction), data accesses are "protected" with *synchronization primitives*, e.g., acquisitions and releases of locks or atomic read-modify-write instructions like compare&swap. Intuitively, a process makes progress by performing "sequential" data accesses to the shared data, e.g., traversing the data structure and modifying its content. In contrast, synchronization tasks, though necessary for correctness, do not contribute to the progress of an operation.

Hence, "making progress in parallel" can be seen as allowing concurrent execution of pieces of locally sequential fragments of code. The more synchronization we use to protect "critical" pieces of the sequential code, the less *schedules*, i.e., interleavings of data accesses, we accept. Intuitively, we would like to use exactly as little synchronization as sufficient for ensuring linearizability of the high-level implemented abstraction. This expectation brings up the notion of a *concurrency-optimal* implementation [79] that only rejects a schedule if it does violate linearizability.

To be able to reason about the "amount of concurrency" exhibited by implementations employing different synchronization techniques, we consider the recently introduced notion of "local serializability" (on top of linearizability) and the metric of the "amount of concurrency" defined via sets of accepted (locally serializable) schedules [79]. Local serializability, intuitively, requires the sequence of sequential steps locally observed by every given process to be consistent with *some* execution of the sequential algorithm. Note that these sequential executions can be different for different processes, i.e., the execution may not be *serializable* [127]. Combined with the standard correctness criterion of *linearizability* [22, 93]), local serializability implies our basic correctness criterion called *LS-linearizability*. The concurrency properties of LS-linearizable data structures can be compared on the same level: implementation A is "more concurrent" than implementation B if the set of schedules accepted by A is a strict superset of the set of schedules accepted

by B. Thus, a *concurrency-optimal* implementation accepts *all* correct (LS-linearizable) schedules.

A Concurrency-Optimal Binary Search Tree

It is interesting to consider binary search trees (BSTs) from the optimal concurrency perspective, as they are believed, as a representative of *search* data structures [48], to be "concurrency-friendly" [147]: updates concerning different keys are likely to operate on disjoint sets of tree nodes (in contrast with, e.g., operations on queues or stacks).

We present a novel LS-linearizable concurrent BST-based set implementation. We prove that the implementation is concurrency-optimal with respect to a standard partially-external sequential tree [79]. The proposed implementation employs the optimistic "lazy" locking approach [85] that distinguishes *logical* and *physical* deletion of a node and makes sure that read-only operations are *wait-free* [89], i.e., cannot be delayed by concurrent processes.

The algorithm also offers a few algorithmic novelties. Unlike most implementations of concurrent trees, the algorithm uses multiple locks per node: one lock for the *state* of the node, and one lock for each of its descendants. To ensure that only conflicting operations can delay each other, we use *conditional* read-write locks, where the lock can be acquired only under certain condition. Intuitively, only changes in the relevant part of the tree structure may prevent a thread from acquiring the lock. The fine-grained conditional read-write locking of nodes and edges allows us to ensure that an implementation rejects a schedule only if it violates linearizability.

Concurrency-Optimality and Performance

Of course, optimal concurrency does not necessarily imply performance nor maximum progress (à la *wait-freedom* [91]). An extreme example is the transactional memory (TM) data structure. TMs typically require restrictions of serializability as a correctness criterion. And it is known that rejecting a schedule that is rejected only if it is not serializable (the property known as *permissiveness*) requires very heavy local computations [80, 108]. But the intuition is that looking for concurrency-optimal search data structures like trees pays off. In this chapter, we show that optimal concurrency can pay off: we demonstrate empirically that the Java implementation of our concurrency-optimal BST outperforms state-of-the-art BST implementations [39, 52, 57, 62] on most workloads. Apart from the obvious benefit of producing a highly efficient BST, this work suggests that optimizing the set of accepted schedules of the sequential code can be an adequate design principle for building efficient concurrent data structures.

Roadmap

The rest of the paper is organized as follows. In Section 5.2, we describe the details of our concurrency-optimal BST implementation together with a novel conditional read-write lock abstraction. In Section 5.3, we formalize the notion of concurrency-optimality and sketch the corresponding proof. In Section 5.4, we give a proof of correctness of our concurrent algorithm and, in Section 5.5, we show its concurrency-optimality. In Section 5.6, we provides details of our experimental methodology and extensive evaluation of our Java implementation. In Section 5.7, we presents concluding remarks.

5.2 Binary Search Tree Implementation

This section consists of two parts. At first, we describe in details the sequential implementation of the set based on partially-external binary search tree which we briefly mentioned in Section 3.1. Then, we build the concurrent implementation on top of the sequential one

by adding synchronization separately for each field of a node. Our implementation takes only the locks that are neccessary to perform correct modifications of the tree structure. Moreover, if the field is not going to be modified, the algorithm takes the read lock instead of the write lock. The last is done only to improve the performance of the algorithm, i.e., allow more schedules of the concurrent algorithm, while, as we shall see, this does not affect the optimality.

5.2.1 Sequential Implementation

As for a sequential implementation we chose the well-known *partially-external* binary search tree. The partially-external tree supports two types of nodes: *routing* and *data*. The set is represented by the values stored by the data nodes. To bound the number of routing vertices by the number of data nodes the tree should satisfy the following condition: all routing nodes should have exactly two children.

The pseudocode of the sequential implementation is presented in Figure 5.1. Here, we give a brief description. The **traversal** function takes a value v and traverses down the tree from the root following the corresponding links as long as the current node is not `null` and its value is not v. This function returns the last three visited nodes. The **contains** function takes a value v and returns whether the last visited node is `null` or not.

The **insert** function takes a value v and uses the traversal function to find the place to insert the value. If the last node returned by the traversal is not `null`, the algorithm checks whether the node is data or routing: in the former case it is impossible to insert; in the latter case, the algorithm simply changes the state of the node from routing to data. If the last node is `null`, then the algorithm assumes that value v is not in the set and inserts a new node with value v as the child of the latest non-`null` node visited by the traversal.

The **delete** function takes a value v and uses the traversal function to find the node with value v to delete. If the last node returned by the traversal is `null` or its state is routing, the algorithm assumes that value v is not in the set and finishes. Otherwise, there are three cases depending on the number of children that the found node has: (i) if the node has two children, then the algorithm changes its state from data to routing; (ii) if the node has one child, then the algorithm unlinks the node; (iii) finally, if the node is a leaf then the algorithm unlinks the node, and if the parent is a routing node then it also unlinks the parent.

5.2.2 Concurrent Implementation

As the basis of our concurrent implementation we took the idea of optimistic algorithms, where the algorithm reads all necessary variables without synchronization and right before the modification, the algorithm takes all the locks and checks the consistency of all the information it read. As we show in the next section, the obtained concurrent partially-external BST appears to be concurrency-optimal. The algorithm is presented at Figure 5.4 while the necessary read-write lock and node classes are presented at Figures 5.2 and 5.3, respectively. Now, we discuss the details.

Deleted Mark

As usual in concurrent algorithms with wait-free traversals, the deletion of any node happens in two stages. At first, the delete operation logically removes a node from the tree by setting the boolean flag **deleted** (Figure 5.4 Lines 59, 63, 78, 83, 94-95 and 102-103). Secondly, the delete operation updates the links to physically remove the node (Figure 5.4 Lines 60, 64, 79, 84, 96 and 104).

```
   Shared variables
   node is a record with fields:
     val, its value
     left, its pointer to the left child
     right, its pointer to the right child
     state ∈ {DATA, ROUTING}, its state

   Initially the tree contains one node root,
   root.val ← +∞
   root.state ← DATA

1  traversal(v): ⟨Node, Node, Node⟩
2    // wait-free traversal
3    gprev ← null
4    prev ← null
5    curr ← root    // start from root
6    while curr ≠ null:
7      if curr.val = v:
8        break
9      else:
10       gprev ← prev
11       prev ← curr
12       if curr.val < v:
13         curr ← curr.left
14       else:
15         curr ← curr.right
16   return ⟨gprev, prev, curr⟩

17 contains(v): bool  // wait-free contains
18   ⟨gprev, prev, curr⟩ ← traversal(v)
19   return curr ≠ null and curr.state = DATA

20 insert(v): bool
21   ⟨gprev, prev, curr⟩ ← traversal(v)
22   if curr ≠ null:  // curr has value v
23     Lines 28-30
24   else:           // prev is a place to insert
25     Lines 32-36
26   return true

   Update existing node
28   if curr.state = DATA:
29     return false     // v is already in the set
30   curr.state ← DATA

   Insert new node
32   newNode.val ← v // allocate a new node
33   if v < prev.val:
34     prev.left ← newNode
35   else:
36     prev.right ← newNode

37 delete(v): bool
38   ⟨gprev, prev, curr⟩ ← traversal(v)
39   if curr = null or curr.state ≠ DATA
40     return false        // v is not in the set
41   if curr has exactly 2 children:
42     Line 51
43   if curr has exactly 1 child:
44     Lines 52-60
45   if curr is a leaf:
46     if prev.state = DATA:
47       Lines 62-65
48     else:
49       Lines 67-74
50   return true

   Delete node with two children
51   curr.state ← ROUTING

   Delete node with one child
52   if curr.left ≠ null:
53     child ← curr.left
54   else:
55     child ← curr.right
56
57   if curr.val < prev.val:
58     prev.left ← child
59   else:
60     prev.right ← child

   Delete leaf with DATA parent
62   if curr is left child of prev:
63     prev.left ← null
64   else:
65     prev.right ← null

   Delete leaf with ROUTING parent
       // save second child of prev into child
67   if curr is left child of prev:
68     child ← prev.right
69   else:
70     child ← prev.left
71   if prev is left child of gprev:
72     gprev.left ← child
73   else:
74     gprev.right ← child
```

Figure 5.1: Sequential implementation

```
1  class RWLock:
2    int lock ← 0
3
4    writeLock():
5      while true:
6        if compare&swap(lock, 0, 1):
7          break
8
9    readLock():
10     while true:
11       now ← lock
12       if now % 2 = 0 and
13           compare&swap(lock, now, now + 2):
14         break
15   unlock():
16     if lock = 1:
17       compare&swap(lock, 1, 0)
18     else:
19       while true:
20         now ← lock
21         if compare&swap(lock, now, now - 2):
22           break
```

Figure 5.2: Read-write lock implementation

Traversal function

The traversal function is slightly different from the sequential implementation, but we omit it from the pseudocode. It traverses as the sequential algorithm, but if the target node is "under-deletion", i.e., its `deleted` mark is set, the traversal is restarted.

Locks

In the beginning of the section we noted that we have locks separately for each field of a node: `llock` for the reference to the left child, `rlock` for the reference to the right child and `slock` for the state (Figure 5.3 Lines 8-10). Also, as mentioned, the algorithm uses read-write locks, implemented using one integer variable `lock` (Figure 5.2). The least significant bit of `lock` indicates whether the write lock is taken or not, the remaining bits represent the number of readers that have taken the lock. In other words, `lock` is zero if the lock is not taken, `lock` is one if the write lock is taken, otherwise, `lock` divided by two represents the number of times the read lock is taken. The locking and unlocking are done using the atomic compare&swap primitive.

Throughout the algorithm the locks are acquired using helper functions that are provided by the `Node` class (Figure 5.3). These functions take a lock and verify several conditions: they check whether the future modification is possible. If the conditions are not satisfied, the guarded modification will corrupt the data structure (in the next subsection we explain what we mean by corruption). In our concurrent algorithm when the helping function returns `false` (Figure 5.4 Lines 15, 20, 24, 42, 47, 48, 51, 71, 76, 81, 90-92 and 98-100), all locks taken up to this moment (including this one) by the high-level "parent" function call, i.e., `contains`, `insert` and `delete`, are released and that "parent" call is restarted.

These are the helper functions:

- `tryReadLockState()` (Figure 5.3 Lines 36-40) takes a read lock on `slock` and checks that the node is not marked as deleted;
- `try(Read|Write)LockState(exp)` (Figure 5.3 Lines 42-46 and Lines 48-52) takes a (read|write) lock on `slock`, checks that the state of the node is equal to `exp` and the node is not marked as deleted;
- `tryLock(Left|Right)Ref(exp)` (Figure 5.3 Lines 12-16) takes a write lock on `(llock|rlock)`, checks that the reference to the (left|right) child is equal to `exp` (to verify that the child did not change) and the node is not marked as deleted;
- `tryLock(Left|Right)Val(exp)` (Figure 5.3 Lines 24-29) takes a write lock on `(llock|rlock)`, checks that the value in the corresponding child (`left.val` or `right.val`) is equal

```
1  class Node:
2    V val, its value
3    Node left, its pointer to the left child
4    Node right, its pointer to the right child
5    state ∈ {DATA, ROUTING}, its state
6
7    bool deleted, deleted mark
8    RWLock llock, the lock on the left field
9    RWLock rlock, the lock on the right field
10   RWLock slock, the lock on the state field
11
12   tryLock(Left|Right)EdgeRef(expRef): bool
13     (llock|rlock).writeLock()
14     if deleted or (left|right) ≠ expRef:
15       return false
16     return true
17
18   tryLockEdgeRef(exp): bool
19     if v < exp.v:
20       return tryLockRightEdgeRef(exp)
21     else:
22       return tryLockLeftEdgeRef(exp)
23
24   tryLock(Left|Right)EdgeVal(expVal): bool
25     (llock|rlock).writeLock()
26     if deleted or (left|right) = null or
27         (left|right).val ≠ expVal:
28       return false
29     return true
30   tryLockEdgeVal(exp): bool
31     if v < exp.v:
32       return tryLockRightEdgeVal(exp.val)
33     else:
34       return tryLockLeftEdgeVal(exp.val)
35
36   tryReadLockState(): bool
37     slock.readLock()
38     if deleted:
39       return false
40     return true
41
42   tryReadLockState(expState): bool
43     slock.readLock()
44     if deleted or expState ≠ state:
45       return false
46     return true
47
48   tryWriteLockState(expState): bool
49     slock.writeLock()
50     if deleted or expState ≠ state:
51       return false
52     return true
```

Figure 5.3: Node clasess

to exp (to verify that the value in the child did not change) and the node is not marked as deleted.

We use the notation of bar as a shorthand notation for avoiding name duplication; such notation should be read as either choosing the first option or the second option.

Momentarily we make two remarks. At first, in the algorithm we typically use `tryLockEdge(Ref|Val)(node)` (Figure 5.3 Lines 18-22 and Lines 30-34) instead of `tryLock(Left|Right)Edge(Ref|Val)(exp)`. Such a substitution, given non-null `node`, decides whether the `node` is the left child or the right child of the current node and calls the corresponding function providing `node` or `node.val`. Secondly, to improve the performance we implement these functions using double-verification: check the conditions, only if they are satisfied take a lock and then check the conditions again.

5.3 Concurrency-Optimality and Correctness. Overview

In this section, we discuss correctness and *concurrency-optimality* [79] of our implementation (the proofs of these properties are provided in Sections 5.4 and 5.5, respectively). Intuitively, a concurrency-optimal implementation employs as much synchronization as necessary for ensuring correctness of the implemented high-level abstraction — in our case, the linearizable set object [89].

Recall our *sequential* BST implementation and imagine that we run it in a *concurrent* environment. We refer to an execution of this concurrent algorithm as a *schedule*. A schedule, thus, consists of reads, writes, node creation events, and invocation and responses of high-level operations. Note that each operation performs at most one write to which we can simply refer as a write of the operation.

Notice that in every such schedule, any operation locally witnesses a *consistent* tree state, since it cannot distinguish the execution from a sequential one. It is easy to see that the local views *across operations* may not be mutually consistent, and this simplistic con-

```
1  contains(v): bool
2    ⟨gprev, prev, curr⟩ ← traversal(v)
3    return curr ≠ null and curr.state = DATA

4  insert(v): bool
         // All restarts are from this Line
5    ⟨gprev, prev, curr⟩ ← traversal(v)
6    if curr ≠ null:
7      Lines 13-16
8    else:
9      Lines 18-26
10   Release all locks
11   return true

     Update existing node
13   if curr.state = DATA:
14     return false
15   curr.tryWriteLockState(ROUTING)
16   curr.state ← DATA

     Insert new node
18   newNode.val ← v
19   if v < prev.val:
20     prev.tryReadLockState()
21     prev.tryLockLeftEdgeRef(null)
22     prev.left ← newNode
23   else:
24     prev.tryReadLockState()
25     prev.tryLockRightEdgeRef(null)
26     prev.right ← newNode

27 delete(v): bool
         // All restarts are from this Line
28   ⟨gprev, prev, curr⟩ ← traversal(v)
29   if curr = null or curr.state ≠ DATA:
30     return false
31   if curr has exactly 2 children:
32     Lines 42-45
33   if curr has exactly 1 child:
34     Lines 52-64
35   if curr is a leaf:
36     if prev.state = DATA:
37       Lines 75-84
38     else:
39       Lines 85-104
40   Release all locks
41   return true

     Delete node with two children
42   curr.tryWriteLockState(DATA)
43   if curr does not have 2 children:
44     Restart operation
45   curr.state ← ROUTING

     Lock acquisition routine for vertex
         with one child
47   prev.tryLockEdgeRef(curr)
48   curr.tryWriteLockState(DATA)
49   if curr has 0 or 2 children:
50     Restart operation
51   curr.tryLockEdgeRef(child)

     Delete node with one child
52   leftChild ← curr.left
53   if leftChild ≠ null:
54     child ← leftChild
55   else:
56     child ← curr.right
57   if curr.val < prev.val:
58     perform lock acquisition at Lines 47-51
59     curr.deleted ← true
60     prev.left ← child
61   else:
62     perform lock acquisition at Lines 47-51
63     curr.deleted ← true
64     prev.right ← child

     Lock acquisition routine for leaf
66   prev.tryLockEdgeVal(curr)
67   if v < prev.key:   // get current child
68     curr ← prev.left
69   else:
70     curr ← prev.right
71   curr.tryWriteLockState(DATA)
72   if curr is not a leaf:
73     Restart operation

     Delete leaf with DATA parent
75   if curr.val < prev.val:
76     prev.tryReadLockState(DATA)
77     perform lock acquisition at Lines 66-73
78     curr.deleted ← true
79     prev.left ← null
80   else:
81     prev.tryReadLockState(DATA)
82     perform lock acquisition at Line 66
83     curr.deleted ← true
84     prev.right ← null

     Delete leaf with ROUTING parent
85   if curr.val < prev.val:
86     child ← prev.right
87   else:
88     child ← prev.left
89   if prev is left child of gprev:
90     gprev.tryLockEdgeRef(prev)
91     prev.tryWriteLockState(ROUTING)
92     prev.tryLockEdgeRef(child)
93     perform lock acquisition at Lines 66-73
94     prev.deleted ← true
95     curr.deleted ← true
96     gprev.left ← child
97   else:
98     gprev.tryLockEdgeRef(prev)
99     prev.tryWriteLockState(ROUTING)
100    prev.tryLockEdgeRef(child)
101    perform lock acquisition at Lines 66-73
102    prev.deleted ← true
103    curr.deleted ← true
104    gprev.right ← child
```

Figure 5.4: Concurrent implementation

current algorithm is not linearizable. For example, two insert operations that concurrently traverse the tree may update the same node so that one of the operations "overwrites" the other (so called the "lost update" problem). Obviously, such a schedule is "incorrect" since it is not linearizable.

To reach the concurrency-optimality we want the concurrent algorithm to avoid such "incorrect" schedules, while *accepting* all others. More precisely, a schedule σ is accepted by an algorithm if it has an execution in which the sequence of high-level invocations and responses, reads, writes, and node creation events (modulo the restarted fragments) is σ [79]. Now, we formalize what does it mean for the schedule to be "incorrect" by introducing the opposite notion of *observable correctness*.

Definition 5.3.1. *A schedule is observably correct if each of its prefixes σ satisfies the following conditions:*

- *subsequence of high-level invocations and responses of operations that performed their write in σ has a linearization with respect to the* set *type;*
- *a set of nodes reachable from the* root *after performing σ is a BST B: (i) they form a tree rooted at node* root*; (ii) this tree satisfies the* order property*: for each node with value v all the values in the left subtree are less than v and all the values in the right subtree are bigger than v; (iii) each routing node in this tree has two children.*
- *BST after performing σ does not contain a node x such that there exist σ' and σ'', such that σ' is a prefix of σ'', σ'' is a prefix of σ ($\sigma' \prec \sigma'' \prec \sigma$), x is in the BST after σ', and x is not in the BST after σ''.*

In other words, a schedule is *observably correct* if it is linearizable, during its execution the tree is always a partially-external BST and, finally, the unlinked node is never linked back.

Now we can define the concurrency-optimality property.

Definition 5.3.2. *A concurrent implementation is concurrency-optimal if it accepts all observably correct schedules.*

At first, in Section 5.4, we prove the correctness of our algorithm.

Theorem 5.3.1 (Correctness)**.** *The algorithm is correct:*

- *The schedule corresponding to any execution of our BST implementation is observably correct.*
- *The algorithm is deadlock-free.*

Further, in Section 5.5 we prove that, in a strict sense, our algorithm accepts *all* correct schedules.

Theorem 5.3.2 (Optimality)**.** *Our BST implementation is concurrency-optimal.*

The intuition behind the proof of Theorem 5.3.2 is the following. We show that for each observably correct schedule there exists a matching execution of our implementation. Therefore, only schedules not observably correct can be rejected by our algorithm. The construction of an execution that matches an observably correct schedule is possible, in particular, due to the fact that every critical section in our algorithm contains exactly one event of the schedule. Thus, the only reason to reject a schedule is that some condition on a critical section does not hold and, as a result, the operation must be restarted. By accounting for all the conditions under which an operation restarts, we show that the restart may only happen if the schedule violates observable correctness.

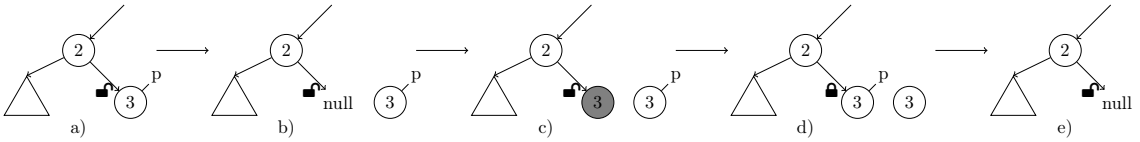


Figure 5.5: Scenario depicting an execution of two concurrent `delete(3)` operations, followed by a successful `insert(3)`; rejected by all the popular BSTs [39, 52, 57, 62, 120], it is accepted by a concurrency-optimal BST

Suboptimality of Related BST Algorithms

To understand the hardness of building linearizable concurrency-optimal BSTs, we present a schedule that is rejected by current state-of-the-art BST algorithms against which we evaluate the performance of our algorithm. This schedule is shown in the Figure 5.5. There is one operation $p =$ `delete`(3) performed on a tree shown in part a). It traverses to leaf v with value 3. Then, some concurrent operation `delete`(3) unlinks node v (part b)). Later, another concurrent operation inserts a new leaf with value 3 (part c)). Operation p wakes up and locks a link since the value 3 is the same (part d)). Finally, p unlinks the node with value 3 (part e)). Note that this is a correct schedule since both delete operations can be successful; however, all the BSTs we are aware of reject this schedule or similar ones [39, 52, 57, 62, 120]. By contrast, there is an execution of our concurrency-optimal BST that accepts this schedule.

5.4 Proof of Correctness

In general, the correctness of the parallel algorithm is carried by the proofs of linearizability and deadlock-freedom. Here, we have additional constraints on the possible executions of our algorithm: they have to carry the observably correct schedules (Definition 5.3.1).

The theorem about the correctness of the algorithm can be stated as follows.

Theorem 5.3.1. *The algorithm is correct if:*

- *the schedule corresponding to any execution of the algorithm is observably correct.*
- *the algorithm is deadlock-free.*

We split the proof of the first statement into two parts: the structural properties, i.e., the tree is a BST and an unlinked node cannot be linked back and the linearizability. In total our proof consists of three parts.

5.4.1 Structural Correctness

At first, we prove that our search tree satisfies the structural properties at any point in time, i.e., the second and the third properties of observably correctness.

Theorem 5.4.1. *The following properties are satisfied at any point in time during the execution:*

- *The order property of BST is preserved.*
- *Every routing node has two children.*
- *Any non-physically deleted node is reachable from the root.*
- *Any physically deleted node is non-reachable from the root.*

Proof. The first two properties are non-trivial by themselves, but we refer to papers [39] and [52] that use the similar partially-external algorithm.

The last two properties follow straightforwardly from the fact that during physical deletion the algorithm takes locks. □

5.4.2 Linearizability

We prove for each execution that it is linearizable.

High-Level Histories and Linearizability

Suppose we are given a history H. To prove the linearizability of H we perform three steps (see Definition 2.3.2). At first, we complete H obtaining the completion $\tilde{H}$: choose a subset of incomplete operations to complete and discard all other incomplete. Then, we provide linearization points to each remaining operation giving us a total order on these operations. And, finally, we prove that the resulting order makes sense: the responses of operations remain the same in $\tilde{H}$ and in the linearization (order) $\tilde{S}$.

Completions

We obtain a completion $\tilde{H}$ of history H as follows. The invocation of an incomplete `contains` operation is discarded. An incomplete π = `insert` operation that has not performed a write at Lines 16, 22 (26) of Figure 5.4 are discarded; otherwise, π is completed with the response `true`. An incomplete π = `delete` operation that has not performed a write at Lines 45, 60 (64), 79 (84), 96 (104) of Figure 5.4 is discarded; otherwise, π is completed with the response `true`.

Note that the described completions correspond to the completions in which the completed operations made a write of the sequential algorithm.

Linearization Points

We obtain a sequential high-level history $\tilde{S}$ equivalent to $\tilde{H}$ by associating a linearization point l_π with each operation π. In some cases our choice of the linearization point depends on the interval between the invocation and the response of the execution of some operation π, later referred to as the interval of π. For example, the linearization point of π in should lie in the interval of π.

Below we specify the linearization point of the operation π depending on its type.

Insert.

For π = `insert`(v) that returns `true`, we have two cases:

1. A routing node with value v was found in the tree. Then l_π is associated with the write in Line 16 of Figure 5.4.

2. A node with value v was not found in the tree. Then l_π is associated with the writes in Lines 22 or 26 of Figure 5.4, depending on whether the newly inserted node is left or right child.

For π = `insert`(v) that returns `false`, we have three cases:

1. If there exists a successful `insert`(v) whose linearization point lies in the interval of π, then we take the first such π' = `insert(v)` and linearize right after $l_{\pi'}$.

2. If there exists a successful `delete`(v) whose linearization point lies in the interval of π, then we take the first such π' = `delete(v)` and linearize right before $l_{\pi'}$.

3. Otherwise, l_π is the call point of π.

Delete.

For π = `delete`(v) that returns `true` we have four cases, depending on the number of children of the node with value v, i.e., the node `curr`:

1. `curr` has two children. Then l_π is associated with the write in Line 45 of Figure 5.4.

2. `curr` has one child. Then l_π is associated somewhere between the writes in Line 59 (63) and in Line 60 (64) of Figure 5.4, depending on whether `curr` is left or right child. The exact position is calculated as what comes last: Line 59 (63) or the last invocation of unsuccessful `insert`(v) or `contains`(v) that reads the node `curr`.

3. `curr` is a leaf with a data parent. Then l_π is associated somewhere between the writes in Line 78 (83) and in Line 79 (84) of Figure 5.4, depending on whether `curr` is left or right child. The exact position is calculated as what comes last: Line 78 (83) or the last invocation of unsuccessful `insert`(v) or `contains`(v) that reads the node `curr`.

4. `curr` is a leaf with a routing parent. Then l_π is associated between the writes in Line 95 (103) and in Line 96 (104) of Figure 5.4, depending on whether `prev` is left or right child. The exact position is calculated as what comes last: Line 95 (103) or the last invocation of unsuccessful `insert`(v) or `contains`(v) that reads the node *curr*.

For every $\pi =$ `delete`(v) that returns `false`, we have three cases:

1. If there exists a successful `delete`(v) whose linearization point lies in the interval of π, then we take the first such $\pi' =$ `delete`(v) and linearize right after $l_{\pi'}$.

2. If there exists a successful `insert`(v) whose linearization point lies in the interval of π, then we take the first such $\pi' =$ `insert`(v) and linearize right before $l_{\pi'}$.

3. Otherwise, l_π is the invocation point of π.

Contains.

For $\pi =$ `contains`(v) that returns `true`, we have three cases:

1. If there exists a successful `insert`(v) whose linearization point lies in the interval of π, then we take the first such $\pi' =$ `insert(v)` and linearize right after $l_{\pi'}$.

2. If there exists a successful `delete`(v) whose linearization point lies in the interval of π, then we take the first such $\pi' =$ `delete`(v) and linearize right before $l_{\pi'}$.

3. Otherwise, l_π is the invocation point of π.

For $\pi =$ `contains`(v) that returns `false`, we have three cases:

1. If there exists successful `delete`(v) whose linearization point lies in the interval of π, then we take the first such $\pi' =$ `delete`(v) and linearize right after $l_{\pi'}$.

2. If there exists successful `insert`(v) which linearization point lies in the interval of π, then we take the first such $\pi' =$ `insert`(v) and linearize right before $l_{\pi'}$.

3. Otherwise, l_π is the invocation point of π.

To confirm our choice of linearization points, we need an auxiliary lemma.

Lemma 5.4.1. *Consider the call $\pi = traverse(v)$. If BST at the moment of the invocation of π contains the node u with value v and there is no linearization point of successful delete(v) operation in the interval of π, then π returns u.*

Proof. Consider a list $A(u)$ of ancestors of node u: $root = w_1, \ldots, w_{n-1}, w_n = u$ (starting from the root) in BST at the moment of the invocation of π.

Let us prove that at any point in time the child of w_i in the direction of value v is w_j for some $j > i$. The only way for w_i to change the proper child is to perform a physical deletion on this child. Consider the physical deletions of w_i in their order in execution. In a base case, when no deletions happened, our invariant is satisfied. Suppose that we

performed first d deletions and now we consider the deletion of w_j. Let w_i be an ancestor of w_j and w_k be a child of w_j in proper direction. After relinking w_k becomes a child of w_i in proper direction, so the invariant is satisfied for w_i because $i \leq j \leq k$, while the children of other vertices remain unchanged.

Summing up, π starts at *root*, i.e., w_1, and traverses only the vertices from $A(u)$ in strictly increasing order. Thus, π eventually reaches u and returns it. □

Theorem 5.4.2 (Linearizability)**.** *The algorithm is linearizable with respect to* set *type.*

Proof. We split our proof into three parts. First, we prove the linearizability of the subhistory with only successful `insert` and `delete` operations because other operations do not affect the structure of the tree. Then, we prove the linearizability of the subhistory with only update operations, i.e., successful and unsuccessful `insert`(v) and `delete`(v). And finally, we present the proof for the history with all types of operations.

Successful Update Operations.

Let $\tilde{S}^k_{succ}$ be the prefix of $\tilde{S}$ consisting of the first k complete successful operations `insert`(v) or `delete`(v) with respect to their linearization points. We prove by induction on k that the sequence $\tilde{S}^k_{succ}$ is consistent with respect to set type.

The base case $k = 0$, i.e., there are no complete operations, is trivial.

The transition from k to $k + 1$. Suppose that $\tilde{S}^k_{succ}$ is consistent with set type. Let π with argument $v \in \mathbb{Z}$ and its response r_π be the last operation in $\tilde{S}^{k+1}_{succ}$. We want to prove that $\tilde{S}^{k+1}_{succ}$ is consistent with π. For that, we check all possible types of π.

1. $\pi =$ `insert`(v) returns `true`.

   By induction, it is enough to prove that there is no preceding operation with an argument v or the last preceding operation with an argument v in $\tilde{S}^{k+1}_{succ}$ is `delete`(v). Suppose the contrary: let the last preceding operation with an argument v be $\pi' =$ `insert`(v). We need to consider two cases of insertion: whether π finds the node with value v in the tree or not.

   In the first case, π finds a node u with value v. π' should have inserted or modified u. Otherwise, the BST at l_π would contain two vertices with value v and this fact violates the structural correctness. If π' has inserted u, then π has no choice but only to read the state of u as data, which is impossible because π is successful. If π' has changed the state of u to data, then π has to read the state of u as data, because the linearization points of π' and π are guarded by the lock on state. This contradicts the fact that π is successful.

   In the second case, π does not find a node with value v. We know that π and π' are both successful. Suppose for a moment that π wants to insert v as a child of node p, while π' inserts v in some other place. Then the tree at l_π has two vertices with value v, violating the structural correctness. This means, that π and π' both want to insert v as a child of node p. Because $l_{\pi'}$ precedes l_π and these linearization points are guarded by the lock on the corresponding link of p, π' takes a lock first, modifies the link to a child of p and by that forces π to restart. During the second traversal, π finds newly inserted node with value v by Lemma 5.4.1 and becomes unsuccessful. This contradicts the fact that π is successful.

2. $\pi =$ `delete`(v) returns `true`.

   By induction it is enough to prove that the preceding operation with an argument v in $\tilde{S}^{k+1}_{succ}$ is `insert`(v). Suppose the opposite: let the last preceding operation with v be `delete`(v) or there is no preceding operation with an argument v.

   At first, consider the simplest case: there is no such operation. This means that π could not find a node with value v. Because if it finds a node then there exists another operation that should have inserted this node and consequently its linearization point

would have been earlier. In this case, π cannot successfully delete, which contradicts the result of π.

The only remaining possibility is that the previous successful operation is $\pi' =$ `delete`(v). Because π is successful, it finds a non-deleted node u with value v. π' should have found the same node u by Lemma 5.4.1, otherwise, the BST right before $l_{\pi'}$ would contain two vertices with value v, violating the structural correctness. So, both π and π' take locks on the state of u to perform an operation. Because $l_{\pi'}$ precedes l_π, π' has taken the lock earlier and set the state of u to routing or marks u as deleted. When π obtains the lock, it could not read state as data and, as a result, cannot delete the node. This contradicts the fact that π is successful.

Update Operations.

Let $\tilde{S}_m^k$ be the prefix of $\tilde{S}$ consisting of the first k complete operations `insert`(v) or `delete`(v) with respect to their linearization points. We prove by induction on k that the sequence $\tilde{S}_m^k$ is consistent with respect to set type. We already proved that successful operations are consistent, then we should prove that the linearization points of unsuccessful operations are consistent too.

The base case $k = 0$, i.e., there are no complete operations, is trivial.

The transition from k to $k+1$. Suppose that $\tilde{S}_m^k$ is consistent with set type. Let π with argument $v \in \mathbb{Z}$ and response r_π be the last operation in $\tilde{S}_m^{k+1}$. We want to prove that $\tilde{S}_m^{k+1}$ is consistent with π. For that, we check all the possible types of π.

If $k+1$-th operation is successful then it is consistent with the previous operations, because it is consistent with successful operations while unsuccessful operations do not change the structure of the tree.

If $k+1$-th operation is unsuccessful, we have two cases.

1. $\pi =$ `insert`(v) returns `false`. When we set the linearization point of π relying on a successful operation in the interval of π, the linearization point is correct: if we linearize right after successful `insert`(v) then π correctly returns `false`; if we linearize right before successful $\pi' =$ `delete`(v) then by the proof of linearizability for successful operations there exists successful `insert`(v) preceding π', thus, π correctly returns `false`.

   It remains to consider the case when no successful update operation was linearized in the interval of π. By induction, it is enough to prove that the last preceding successful operation with v in $\tilde{S}_m^{k+1}$ is `insert`(v). Suppose the opposite: let the last preceding successful operation with an argument v be `delete`(v) or there is no preceding operation with an argument v. If there is no such operation then π could not find a node with value v, because, otherwise, another operation should have inserted the node and its linearization point would have come earlier. Thus, π can successfully insert a new node with value v, which contradicts the fact that π is unsuccessful.

   The only remaining possibility is that the last preceding successful operation is $\pi' =$ `delete`(v). We know that $l_{\pi'}$ does not lie inside the interval of π, since, otherwise, we linearized π with respect to π'. Thus, π has to find either the routing node with value v or do not find such a node at all, since π' has already successfully unlinked it. (This happened because we set linearization points of successful `delete` operations quite tricky) In both cases, insert operation could be performed successfully. This contradicts the fact that π is unsuccessful.

2. $\pi =$ `delete`(v) returns `false`.

   When we set the linearization point of π relying on the successful operation in the interval of π, the linearization point is correct: if we linearize right after successful `delete`(v) then π correctly returns `false`; if we linearize right before successful

$\pi' = \texttt{insert}(v)$ then by the proof of linearizability for successful operations there exists successful $\texttt{delete}(v)$ preceding π' or there are no successful operation with an argument v in $\tilde{S}_m^{k+1}$ before π', thus, π correctly returns `false`.

It remains to consider the case when no successful operation was linearized in the interval of π. By induction, it is enough to prove that there is no preceding successful operation with v or the last preceding successful operation with v in $\tilde{S}_m^{k+1}$ is $\texttt{delete}(v)$. Again, suppose the opposite: let the previous successful operation with v be $\pi' = \texttt{insert}(v)$.

By Lemma 5.4.1, π finds the data node u with value v and π can successfully remove it because no other operation with argument v has a linearization point during the execution of π. This contradicts the fact that π is unsuccessful.

All Operations.

Finally, we prove the correctness of the linearization points of all operations.

Let $\tilde{S}^k$ be the prefix of $\tilde{S}$ consisting of the first k complete operations ordered by their linearization points. We prove by induction on k that the sequence $\tilde{S}^k$ is consistent with respect to the set type. We already proved that update operations are consistent, then we should prove that the linearization points of contains operations are consistent too.

The base case $k = 0$, i.e., there are no complete operations, is trivial.

The transition from k to $k + 1$. Suppose that $\tilde{S}^k$ is consistent with the set type. Let π with argument $v \in \mathbb{Z}$ be the last operation in $\tilde{S}^{k+1}$. We want to prove that $\tilde{S}^{k+1}$ is consistent for the operation π. For that, we check all the possible types of π.

If $k+1$-th operation is $\texttt{insert}(v)$ and $\texttt{delete}(v)$ then it is consistent with the previous $\texttt{insert}(v)$ and $\texttt{delete}(v)$ operations while $\texttt{contains}(v)$ operations do not change the structure of the tree.

If the operation is $\pi = \texttt{contains}(v)$, we have two cases:

1. π returns `true`.

   When we set the linearization point of π relying on a successful update operation in the interval of π, then the linearization point is correct: if we linearize right after successful $\texttt{insert}(v)$, then π correctly returns `true`; if we linearize right before successful $\pi' = \texttt{delete}(v)$, then, by the proof of the linearizability on successful operations, there exists successful $\texttt{insert}(v)$ preceding π', thus π correctly returns `true`.

   We are left with the case when no successful operation has its linearization point in the interval of π. By induction, it is enough to prove that the last preceding successful operation with v in $\tilde{S}^{k+1}$ is $\texttt{insert}(v)$. Suppose the opposite: the last preceding successful operation with an argument v is $\texttt{delete}(v)$ or there is no preceding successful operation with v. If there is no successful operation then π could not find a node with value v, otherwise, some operation has inserted a node before and its linearization point would have come earlier. This contradicts the fact that π is successful.

   It remains to check if there exists a preceding $\pi' = \texttt{delete}(v)$ operation. Since $l_{\pi'}$ does not lie inside the interval of π then π has to find either the routing node with value v or do not find such node, since π' has unlinked it. This contradicts the fact that π returns `true`.

2. π returns `false`.

   When we set the linearization point of π relying on a successful update operation in the interval of π, then the linearization point is correct: if we linearize right after successful $\texttt{delete}(v)$, then π correctly returns `false`; if we linearize right before successful $\pi' = \texttt{insert}(v)$ then, by the proof of linearizability on successful operations

either there exists a preceding π' successful `delete`(v) or there exists no operation with an argument v in $\tilde{S}$ before π'. Thus π correctly returns `false`.

We are left with the case when no successful operation has its linearization point in the interval of π. By induction, it is enough to prove that there is no preceding successful operation with an argument v or the last preceding successful operation with an argument v in $\tilde{S}^{k+1}$ is `delete`(v). Again, suppose the opposite: the last preceding successful operation with an argument v is $\pi' =$ `insert`(v).

By Lemma 5.4.1 π finds the data node u with value v. This contradicts the fact that π returns `false` and π should return `false`.

□

5.4.3 Deadlock-Freedom

Theorem 5.4.3 (Deadlock-freedom)**.** *The algorithm is deadlock-free: assuming the infinite execution and that no thread fails in the middle of its update operation, the infinite number of operations is completed.*

Proof. `contains` does not take locks at all, so, it is deadlock-free.

We note that during `insert` and `delete` operations the locks are taken in a top-down manner: when the first lock is taken, the other locks are acquired only after the "parent" lock is taken: the "parent" lock for the state lock is the lock on the edge from parent; and the "parent" lock for the edge lock is the lock on the state in the source node. Also, let a *lock-level* of an operation be the number of "ancestors" towards the *root* of the first lock that the operation attempts to acquire.

In such a manner, when it is not possible for an operation to take a lock then there is another operation with a higher *lock-level* that acquired that lock. Then, if the second operation cannot proceed there exists the third blocking operation with *lock-level* even bigger, and so on. Since the height of the tree is finite, there exist the lowest operation that blocks everybody: this operation can proceed. This lowest operation either successfully completes or the conditions are not satisfied and the operation is restarted. However, in the last case, the operation, that forces the restart, made progress. □

5.5 Proof of Concurrency-Optimality

Theorem 5.3.2 (Optimality)**.** *Our binary search tree implementation is concurrency-optimal with respect to the sequential algorithm provided in Figure 5.1.*

Proof. Consider all executions of our algorithm in which all critical sections are executed sequentially, i.e., the primitives of a critical section are executed sequentially. Since all critical sections in our algorithm contain only one operation from the sequential algorithm, the implementation accepts all the schedules in which the operation is not restarted by failing some condition in the critical sections.

Suppose that we are given a schedule σ. We show that each condition that forces the restart is crucial, i.e., if the operation ignores it, then σ is not observably correct.

By the first property of observable correctness (Definition **??**) when we talk about the linearization of the schedule we talk about the linearization of operations that made the write.

To simplify the proof by exhaustion we consider three common situations (later referred to as Cases 1, 2 or 3) that appear under consideration, and show that in each of these cases the schedule σ is not observably correct.

Let two functions $I(T, v)$ and $D(T, v)$ be the number of `insert` and `delete` operations with argument v that made the single write of the sequential algorithm in the prefix of the schedule σ of length T, i.e., T primitives, later referred as $\sigma(T)$.

1. The modification, i.e., a corresponding write, in the critical section of operation $\pi = \texttt{insert}(v)$ (the case of $\texttt{delete}(v)$ is considered similarly) does not change the set of values represented by our tree: (1) for each node reachable from the root the fields (`left` and `right` link, and `state`) do not change; or (2) for each node reachable from the root (except for one) the fields do not change, and for the vertex left one routing child can be unlinked. Note that we do not discuss the `deleted` field, since the sequential algorithm does not know about its existence and, thus, modifications of this field do not belong to schedules.

   Let this modification be the T-th primitive of the current schedule σ. Consider two prefixes of this schedule: $\sigma(T-1)$ and $\sigma(T)$. There are two cases:

   - If the value v is present in the set after $\sigma(T-1)$, then $I(T-1,v) = D(T-1,v)+1$, since $\sigma(T-1)$ is linearizable. As the T-th primitive, π made the write and, since π is `insert`, $I(T,v) = D(T,v)+2$. Any completion of $\sigma(T)$ (considering only operations that made a write) is non-linearizable, because the number of successful `insert` operations cannot exceed the number of successful `delete` operations by more than one. This means that $\sigma(T)$ is non-linearizable, and, thus, σ is not observably correct.
   - If the value v is not present in the set after $\sigma(T-1)$, then $I(T-1,v) = D(T-1,v)$, since $\sigma(T-1)$ is linearizable. As the T-th primitive, operation π made a write and, since π is `insert`, $I(T,v) = D(T,v)+1$,. Also, the node with value v is unreachable from the root, since the set did not change. Let us consider any linearizable completion of $\sigma(T)$. By completing operations, the set cannot change, since all writes by the operations were already performed, and, thus, v does not belong to the set. However, the number of successful insertion exceeds the number of successful deletions, and, thus, $\sigma(T)$ cannot be linearizable. This means that any completion of $\sigma(T)$ is non-linearizable, thus, $\sigma(T)$ is non-linearizable and, consequently, σ is not observably correct.

2. After the modification in the critical section of operation π with argument v a whole subtree of node u with a value different from v becomes unreachable from the root. Let this modification be the T-th primitive of the current schedule σ. Because of the structure of the partially-external tree, subtree of node u should contain at least one data vertex with value x not equal to v: either u is a data leaf, or the subtree contains at least two data nodes. Since x was reachable before the modification and $\sigma(T-1)$ is linearizable, we assume $I(T-1,x) = D(T-1,x)+1$. The number of successful update operations with argument x does not change after the modification, so $I(T,x) = D(T,x)+1$. But the value x is not reachable from the root after $\sigma(T)$, meaning that any completion of (operations that made a write in) $\sigma(T)$ cannot be linearized. Thus, $\sigma(T)$ is non-linearizable and, consequently, σ is not observably correct.

3. After the modification in the critical section of operation π the node u with `deleted` mark becomes reachable from the root. Let this write be the T-th primitive of the current schedule σ. Let the write that was done in the same critical section as the `deleted` mark of u was set be the $\tilde{T}$-th primitive of σ. It could be seen that u is reachable from the root after $\sigma(\tilde{T}-1)$ and after $\sigma(T)$, but u is not reachable from the root after $\sigma(\tilde{T})$. Thus, σ does not satisfy the third requirement for observably correct schedules, meaning that σ is not observably correct.

Now, we want to prove that all conditions that precede each modification operation are necessary and their omission leads to not observably correct schedule. The proof is done by induction on the position of the modification operation in the execution. The base case, when there are no modification operations done, is trivial. Suppose that we showed the correctness of the statement for the first $i-1$ modifications and want to prove it for

the i-th. Let this modification be the T-th primitive of the schedule σ. We ignore each condition that precedes the modification one by one in some order and show that their omission makes σ not observably correct.

- Operation $\pi = \texttt{insert}(v)$ restarts in Line 15 of Figure 5.4. As we explicitely did not write in the pseudocode for simplicity we want to remind that an update operation restarts whenether some condition checked while the locks are held is not satisfied. This means, that at least one of the following condition holds:
  - `curr` is not routing (Line 15). Then the guarded operation does not change the set of values and we can apply Case 1.
  - `deleted` mark of `curr` is set (later, we simply say `curr` is deleted) (Line 15). then `curr` is already unlinked, so the modification in Line 16 does not change the set of values and we can apply Case 1.
- Operation $\pi = \texttt{insert}(v)$ restarts in Lines 20, 21 (24, 25) of Figure 5.4. This means, that at least one of the following conditions holds:
  - `prev` is deleted (Line 20 (24)). Then `prev` is already unlinked and is not reachable from the root. This means, that the modification in Line 22 (26) links the new node to already unlinked node `prev`, not changing the set of values, and we can apply Case 1.
  - The corresponding child of `prev` is not null (Line 21 (25)). Then the write in Line 22 (26) unlinks a whole subtree of the current child and we can apply Case 2.
- Operation $\pi = \texttt{delete}(v)$ restarts in Lines 42 and 44 of Figure 5.4. This means that either `curr` is not a data node, `curr` is deleted or `curr` does not have two children.
  - If `curr` is not a data node or it is deleted (Line 42), then the write at Line 45 does not change the set of values and we can apply Case 1.
  - If `curr` does not have two children (Line 44), then after the write in Line 45 the tree has the routing node `curr` with less than two children. Thus, after σ the tree does not satisfy the second requirement for observably correct schedules.
- Operation $\pi = \texttt{delete}(v)$ restarts in Lines 47-51 of Figure 5.4. This means that at least one of the following conditions holds:
  - `prev` is deleted (Line 47). Then the write at Line 60 (64) does not change the set of values and we can apply. For later cases, we already assume, that `prev` is not deleted.
  - There is no link from `curr` to `child` (Line 51). The link can cease to exist only of one of the nodes are deleted.
    - `child` is deleted. Then after the write at Line 60 (64) the deleted node `child` becomes reachable from the root, since `prev` is not deleted, and we can apply Case 3. From hereon, we assume that `child` is not deleted.
    - `curr` is deleted. We know that `prev` and `child` are not deleted, thus `prev` already has `child` as its child. By that, the write at Line 60 (64) does not change the set of values and we can apply Case 1. From hereon, we assume that `curr` is not deleted.
  - There is no link from `prev` to `curr` (Line 47). This can happen only if `prev` or `curr` is deleted. But we already considered these cases.
  - `curr` is not a data node (Line 48). Then the write in Line 60 (64) does not change the set of values and we can apply Case 1.

- `curr` does not have exactly one child (Line 50). Since none of `curr` and `child` are deleted, the link from `curr` to `child` exists in the tree. Thus, the only possible way to violate is that `curr` has two children. Finally, the write in Line 60 (64) unlinks a whole subtree of the other child of `curr` and we can apply Case 2.

- Operation π = `delete`(v) restarts in Lines 66-73 and 76 (81) of Figure 5.4. This means that at least one of the following conditions holds:
  - `prev` is deleted (Line 76 (81)). Then the write in Line 79 (84) does not change the set of values and we can apply Case 1. From hereon, we assume that `prev` is not deleted.
  - `prev` is not a data node (Line 76 (81)). Then after the write in Line 79 (84) the tree contains a routing node with less than two children. Thus, $\sigma(T)$ does not satisfy the second requirement for observably correct schedules.
  - The child c of `prev` in the direction of v is null (Line 66). Then the write in Line 79 (84) does not change the set of values and we can apply Case 1.
  - The child c of `prev` in the direction of v has a key different from v (Line 66). (Note that c cannot be deleted since the link from `prev` to c is locked and, thus, in the tree now.) The write in Line 79 (84) unlinks a whole subtree of c and we can apply Case 2.
  - The child c of `prev` in the direction of v is not a leaf (Line 73). Then the write in Line 79 (84) removes whole subtree of c with at least one data node with the value different from v and we can apply Case 2. In last case we assume that c is a leaf.
  - The child c of `prev` in the direction of v is a routing node (Line 71). Then before the write in Line 79 (84) the tree contained a routing leaf c. Thus, $\sigma(T-1)$ does not satisfy the second requirement for observably correct schedule.

- Operation π = `delete`(v) restarts in Lines 66-73 and 90-92 (98-100) of Figure 5.4. This means that at least one of the following conditions holds:
  - `gprev` is deleted (Line 90 (98)). Then the write in Line 96 (104) does not change the set of values and we can apply Case 1 σ. From hereon, we assume that `gprev` is not deleted.
  - `child` is deleted, then the write in Line 96 (104) links the deleted node back to the tree and we can apply Case 3. Later, we assume that `child` is not deleted.
  - `prev` is not a child of `gprev`. (Line 90 (98)) Since `gprev` is not deleted, this case can happen only if `prev` is physically deleted. From this follows that the current child of `gprev` in a direction of v is `child`. Thus, the write in Line 96 (104) does not change the set of values and by Case 1 σ is not observably correct. Later, we assume that `prev` is not deleted.
  - `prev` is a data node (Line 91 (99)). Then the write in Line 96 (104) unlinks `prev` from the tree and we can use the same reasoning as in Case 2.
  - `child` is not a current child of `prev` (Line 92 (100)). However, we already showed in the previous cases that `prev` and `child` are not deleted.
  - The last four cases are identical to the last four cases for `delete`(v) that restarts in Lines 66-73 and 91 (99).

We showed that the restart of operation in the execution happens only if the the corresponding sequential schedule is not observably correct. Thus, our algorithm is indeed concurrency-optimal. □

5.6 Implementation and Evaluation

Experimental Setup

For our experiments we used two machines with different architectures. The first is a 4-processor Intel Xeon E7-4870 2.4 GHz server (Intel) with 20 threads per processor (yielding 80 hardware threads in total), 512 Gb of RAM, running Fedora 25. This machine has Java 1.8.0_111-b14 and HotSpot VM 25.111-b14. Second machine is a 4-processor AMD Opteron 6378 2.4 GHz server (AMD) with 16 threads per processor (yielding 64 threads in total), 512 Gb of RAM, running Ubuntu 14.04.5. This machine has Java 1.8.0_111-b14 and HotSpot JVM 25.111-b14.

Binary Search Tree Implementations

We compare our algorithm, denoted as Concurrency-Optimal or CO, against four other concurrent BSTs. They are: 1) the lock-based contention-friendly tree by Crain et al. ([52], Concurrency Friendly or CF), 2) the lock-based logical ordering AVL-tree by Drachsler et al. ([57], Logical Ordering or LO) 3) the lock-based tree by Bronson et al. ([39], BCCO) and 4) the lock-free tree by Ellen et al. ([62], EFRB). All these implementations, including ours, are written in Java and are available in the `synchrobench repository` [76]. In order to make the comparison equitable, we remove rotation routines from the CF-, LO- and BCCO- trees implementations. We are aware of efficient lock-free tree by Natarajan and Mittal ([120]), but unfortunately we were unable to find it written on Java.

Experimental Methodology

For our experiments, we use the environment provided by the synchrobench library. To compare the performance we considered the following parameters:

- **Workloads.** Each workload distribution is characterized by the percent $x\%$ of update operations. This means that the tree will be requested to make $100 - x\%$ of `contains` calls, $x/2\%$ of `insert` calls and $x/2\%$ of `delete` calls. We considered three different workload distributions: 0%, 20% and 100%.
- **Tree size.** On the workloads described above, the tree size depends on the size of the key space (the size is approximately half of the range). We consider three different key ranges $[0, R)$, where R is 2^{15}, 2^{19} or 2^{21}.
- **Degree of contention.** This depends on the number of cores in a machine. We take enough points to reason about the behaviour of curves.

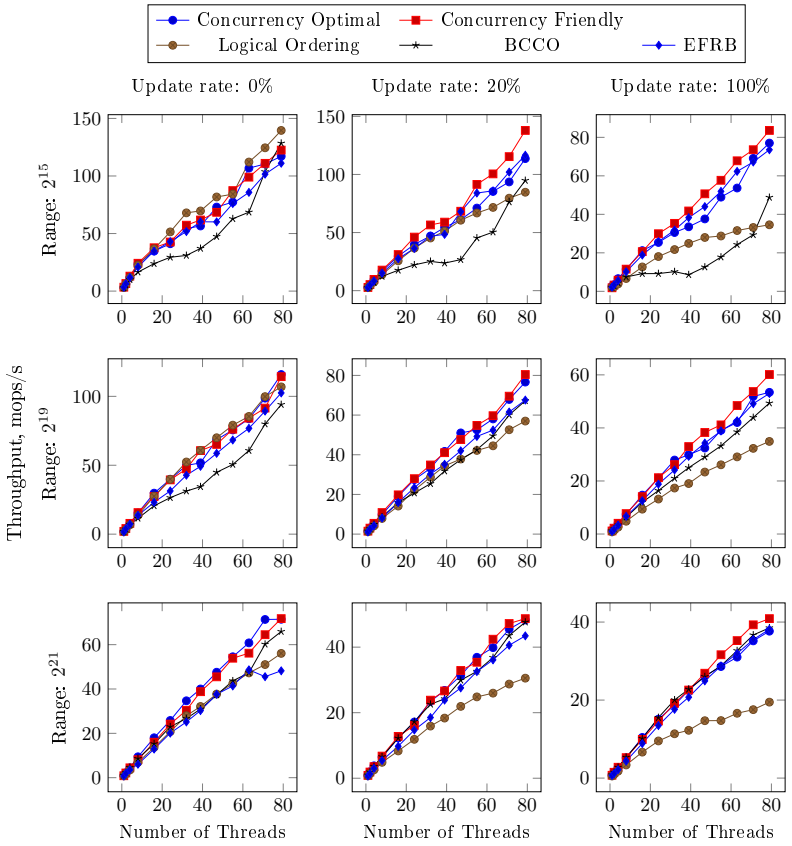
We prepopulate the tree with, approximately, $R/2$ values: we take each value from the range with probability $\frac{1}{2}$. Then we start P processes. Each process repeatedly performs operations: (1) with probability $1 - x$, it searches for a random value taken uniformly from range; (2) with probability $\frac{x}{2}$, it inserts a random value taken uniformly from range; (3) with probability $\frac{x}{2}$, it deletes a random value taken uniformly from range.

Under such workloads, the size of the tree is always approximately half the range, $R/2$, and, thus, only half of update operations succeed.

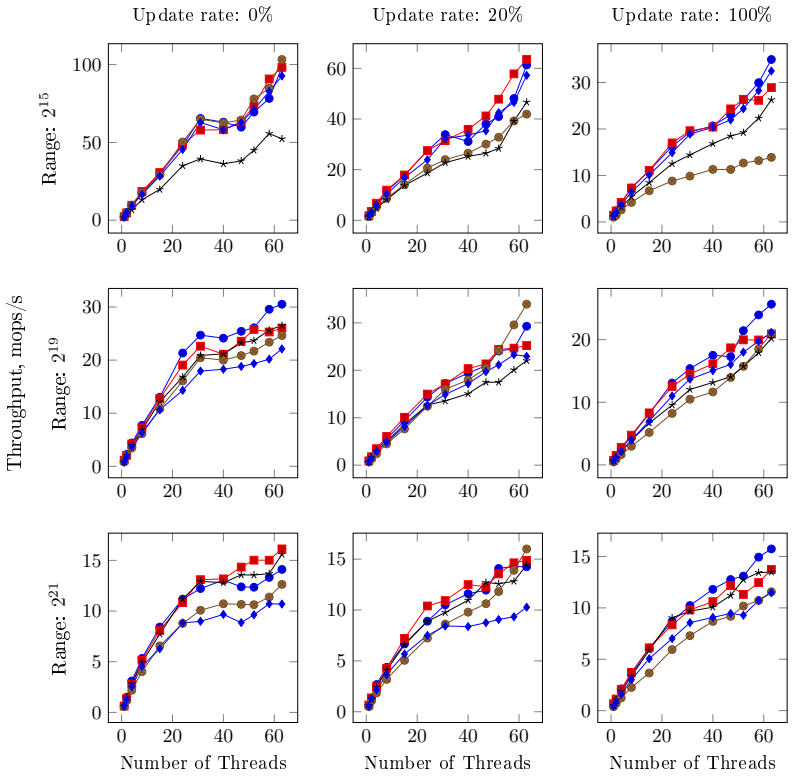
We chose the settings such that we had two extremes and one middle point. For workload, we chose 20% of attempted updates as a middle point.

Results

To get meaningful results we average through up to 25 runs. Each run is carried out for 10 seconds with a warmup of 5 seconds. Figure 5.6a (and resp. 5.6b) contains the results of executions on Intel (and resp. AMD) machine. It can be seen that with the increase of the size the performance of our algorithm becomes better relatively to CF-tree. This is



(a) Evaluation of BST implementations on Intel



(b) Evaluation of BST implementations on AMD

Figure 5.6: Performance evaluation of concurrent BSTs

due to the fact that with bigger size the cleanup-thread in CF-tree implementation spends more time to clean the tree out of logically deleted vertices, thus, the traversals has more

chances to pass over deleted vertices, leading to longer traversals. By this fact and the trend shown, we could assume that CO-tree outperforms CF-tree on bigger sizes. On the other hand, BCCO-tree was much worse on 2^{15} and became similar to CO-tree on 2^{21}. This happened because the races for the locks become more unlikely. This helped much to BCCO-tree, because it uses high-grained locking. On bigger sizes we expect that our implementation will continue to perform similarly to BCCO-tree, because the difference in our and BCCO-tree implementations is only in grabbing locks. By that, we can state that our algorithm works well not depending on the size. Finally, in most workloads our algorithm seems to perform better than other trees.

5.7 Conclusion

Measuring Concurrency

Measuring concurrency via comparing a concurrent data structure to its sequential counterpart was originally proposed [78]. The metric was later applied to construct a concurrency-optimal linked list [77], and to compare synchronization techniques used for concurrent *search* data structures, organizing nodes in a directed acyclic graph [79]. Although lots of efforts have been devoted to improve the performance of BSTs as under growing concurrency, to our knowledge, the existence of a concurrency-optimal BST has not been earlier addressed.

Search for Concurrency-Optimal Data Structures

Concurrent BSTs have been studied extensively in literature; yet by choosing to focus on minimizing the amount of synchronization, we identified an extremely high-performing concurrent BST implementation. We proved our implementation to be formally correct and established the concurrency-optimality of our algorithm. Apart from the intellectual merit of understanding what it means for an implementation to be highly concurrent, our findings suggest a relation between concurrency-optimality and efficiency. We hope this work will inspire the design of other concurrency-optimal data structures that currently lack efficient implementations.

6 Parallel Combining: Benefits of Explicit Synchronization

6.1 Introduction

Efficient concurrent data structures have to balance parallelism and synchronization. Parallelism implies performance, while synchronization maintains consistency. Efficient "concurrency-friendly" data structures (e.g., sets based on linked lists [84, 85] or binary search trees [52, 57]) are conventionally designed using hand-crafted fine-grained locking. In contrast, "concurrency-averse" data structures (e.g., staks and queues) are subject to frequent sequential bottlenecks and solutions based on combining (e.g., [86]), where all requests are synchronized and applied sequentially, perform surprisingly well compared to fine-grained ones [86]. Typically, a general data structure combines features of "concurrency-friendliness" and "concurrency-averseness", and an immediate question is how to implement it in the most efficient way.

In this paper, we suggest a methodology of building a concurrent data structure from its *parallel batched* counterpart [13]. A parallel batched data structure applies a *batch* (set) of operations using parallelism. The algorithm distributes the work between the processes assuming the synchronized application of operations that reduces the number of possible interleavings. As a result, designing parallel batched algorithms is much easier than their concurrent counterparts.

In the technique proposed here, we *explicitly* synchronize concurrent operations, assemble them into batches, and apply these batches on an *emulated* parallel batched data structure. Processes share a set of active requests using any *combining* algorithm [58]: one of the active processes becomes a *combiner* and forms a *batch* from the requests in the set. Under the coordination of the combiner, the owners of the collected requests, called *clients*, apply the requests in the batch to the parallel batched data structure.

This technique becomes handy when the overhead on explicit synchronization of processes is compensated by the advantages of involving clients into the combining procedure using the parallel batched data structure. In the extreme case of concurrency-averse data structures, such as queues and stacks, having control over the batch can be used to *eliminate* certain requests and bypass sequential bottlenecks, even though there are no benefits of parallelizing the execution.[1] But as we show in the paper, combining *and* parallel batching pay off for data structures that offer some degree of parallelism, such as dynamic graphs and priority queues.

We discuss three applications of parallel combining and we experimentally validate the benefits for two of them. First, we design concurrent implementations optimized for *read-dominated* workloads given a sequential data structure. Intuitively, updates are performed sequentially and read-only operations are performed by the clients in parallel under the coordination of the combiner. In our performance analysis, we considered a *dynamic graph* data structure [95] that can be accessed for adding and removing edges (updates), as well as for checking connectivity between pairs of vertices (read-only). Second, we apply parallel combining to the parallel batched *binary search tree* algorithm by Blelloch et al. [32] in Section 3.1. Our concurrent data structure performs worse than the state-of-the-art implementations, since the operations on them are not likely to contend. Despite this, our implementation provides theoretical bounds on the height of the tree and the running

[1] Note that sequential combining [58, 65, 86, 126], typically used in the concurrency-averse case, is a degenerate case of our parallel combining in which the combiner applies the batch sequentially.

time of operations in RMRs. Finally, we apply parallel combining to *priority queue* that is subject to sequential bottlenecks for minimal-element extractions, while most insertions can be applied concurrently. As a side contribution, we propose a novel parallel batched priority queue, as no existing batched priority queue we are aware of can be efficiently used in our context. Our perfomance analysis shows that implementations based on parallel combining may outperform state-of-the-art algorithms.

Roadmap

The rest of the chapter is organized as follows. In Section 6.2, we outline parallel combining and provide several applications. In Section 6.3, we present the parallel batched binary search tree by Blelloch et al. [32] in a form convenient for parallel combining. In Section 6.4, we present a novel parallel batched priority queue in a form convenient for parallel combining. In Section 6.5, we report the results of experiments. In Section 6.6, we overview the related work. We conclude in Section 6.7.

6.2 Parallel Combining and Applications

In this section, we describe the *parallel combining* technique in a parameterized form: the parameters are specified depending on the application.

Combining Data Structure

Our technique relies on a combining data structure $\mathbb{C}$ (e.g., [86]) that maintains a set of requests to data structure and determines which process is a combiner. If the set of requests is not empty then exactly one process should be a combiner.

Elements stored in $\mathbb{C}$ are of `Request` type consisting of the following fields: 1) the method to be called and its input; 2) the response field; 3) the status of the request with a value from an application-specific `STATUS_SET`; 4) application-specific auxiliary fields. In our applications `STATUS_SET` contains at least `INITIAL` and `FINISHED` values: `INITIAL` is set during the initialization and `FINISHED` means that the request is served.

$\mathbb{C}$ supports three operations: 1) `addRequest(r : Request)` inserts request `r` into the set and returns whether this process becomes a combiner or a client; 2) `getRequests()` returns a non-empty set of requests; and 3) `release()` is issued by the combiner to make $\mathbb{C}$ find another process to be a combiner.

In the rest of the paper we assume any black-box implementation of $\mathbb{C}$ [58, 65, 86, 126].

Specifying Parameters

To perform an operation, a process executes the following steps (Figure 6.1): 1) it inserts the request into $\mathbb{C}$ using `addRequest(·)` and checks whether it becomes a combiner; 2) if it is the combiner, it collects requests from $\mathbb{C}$ using `getRequests()`, then it executes algorithm `COMBINER_CODE`, and, finally, calls `release()` to enable another active process to become a combiner; 3) if the process is a client, it waits until the status of the request is not `INITIAL` and, then, executes algorithm `CLIENT_CODE`.

To use our technique, one should therefore specify `COMBINER_CODE`, `CLIENT_CODE`, and appropriately modify `Request` type and `STATUS_SET`.

Note that *sequential combining* [58, 65, 86, 126] is a special case of parallel combining: we simply need to use the proper algorithm behind black box $\mathbb{C}$. Now we discuss two interesting applications of parallel combining.

6.2.1 Read-optimized Concurrent Data Structures

Parallel combining can boost a sequential data structures under *read-dominated* workloads, i.e., when most operations do not modify the data structure. Such operations are called

```
1  Request:
2    method
3    input
4    res
5    status ∈ STATUS_SET
6    ...
7
8  execute(method, input):
9    req ← new Request()
10   req.method ← method
11   req.input ← input
12   req.status ← INITIAL
13   if  ℂ.addRequest(req):
14     // combiner
15     A ← ℂ.getRequests()
16     COMBINER_CODE
17     ℂ.release()
18   else:
19     while req.status = INITIAL:
20       nop
21     CLIENT_CODE
22   return
```

Figure 6.1: Parallel combining: pseudocode

```
1  COMBINER_CODE:
2    R ← ∅
3
4    for r ∈ A:
5      if isUpdate(r.method):
6        apply(D, r.method, r.input)
7        r.status ← FINISHED
8      else:
9        R ← R ∪ r
10
11   for r ∈ R:
12     r.status ← STARTED
13   if req.status = STARTED:
14     apply(D, req.method, req.input)
15     req.status ← FINISHED
16
17   for r ∈ R:
18     while r.status = STARTED:
19       nop
20
21 CLIENT_CODE:
22   if not isUpdate(req.method):
23     apply(D, req.method, req.input)
24     req.status ← FINISHED
```

Figure 6.2: Parallel combining in application to read-optimized data structures

read-only operations, while other operations are called *update* operations.

Suppose that we are given a sequential data structure D that suports update and read-only operations. Now we explain how to set parameters of parallel combining for this application. At first, `STATUS_SET` consists of three elements `INITIAL`, `STARTED` and `FINISHED`. `Request` type does not have auxiliary fields.

In `COMBINER_CODE` (Figure 6.2 Line 1), the combiner iterates through the set of collected requests A: if a request contains an update operation then the combiner executes it and sets its status to `FINISHED`; otherwise, the combiner adds the request to set R. Then the combiner sets the status of requests in R to `STARTED`. After that the combiner checks whether its own request is read-only. If so, it executes the method and sets the status of its request to `FINISHED`. Finally, the combiner waits until the status of the requests in R become `FINISHED`.

In `CLIENT_CODE` (Figure 6.2 Line 21), the client checks whether its method is read-only. If so, the client executes the method and sets the status of the request to `FINISHED`.

Theorem 6.2.1. *Algorithm in Figure 6.2 produces a linearizable concurrent data structure from a sequential one.*

Proof. Any execution can be split into combining phases (Figure 6.2 Lines 2-19) which do not intersect. We can group the operations into batches by the combining phase in which they are applied.

Each update operation is linearized at the point when the combiner applies this operation. Note that this is a correct linearization since all operations that are linearized before are already applied: the operations from preceding combining phases were applied during

the preceding phases, while the operations from the current combining phase are applied sequentially by the combiner.

Each read-only operation is linearized at the point when the combiner sets the status of the corresponding request to `STARTED`. By the algorithm, a read-only operation observes all update operations that are applied before and during the current combining phase. Thus, the chosen linearization is correct. □

To evaluate the approach we apply it to the sequential *dynamic graph* data structure by Holm et al. [95] (Section 6.5.1).

6.2.2 Parallel Batched Algorithms

We discuss below how to build a concurrent implementation given a parallel batched one in one of two forms: for static or dynamic multithreading.

Suppose that we are given a parallel batched implementation for dynamic multithreading. One can turn it into a concurrent one using parallel combining with the *work-stealing* scheduler. We enrich `STATUS_SET` with `STARTED`. In `COMBINER_CODE` the combiner collects the requests and sets their status to `STARTED`. Then the combiner creates a working deque, puts there a new node of computational DAG with `apply` function and starts the work-stealing routine on processes-clients. Finally, the combiner waits for the clients to become `FINISHED`. In `CLIENT_CODE` the client creates a working deque and starts the work-stealing routine.

In the static multithreading case, each process is provided with a distinct version of `apply` function. Again, we enrich `STATUS_SET` with `STARTED`. In `COMBINER_CODE` the combiner collects the requests, sets their status to `STARTED`, performs the code of `apply` and waits for the clients to become `FINISHED`. In `CLIENT_CODE` the client waits until its request has `STARTED` status, performs the code of `apply` and sets the status of its request to `FINISHED`.

We apply this technique to the *binary search tree* and to the *priority queue*. For the binary search tree we present the parallel batched algorithm by Blelloch et al. [32] in a form of `COMBINER_CODE` and `CLIENT_CODE`. For the priority queue we introduce a novel parallel batched implementation in a form of `COMBINER_CODE` and `CLIENT_CODE`. We had to design a new algorithm since no known implementation [38, 56, 132, 137] can be efficiently used in our context: their complexity inherently depends on the total number of processes in the system, regardless of the actual batch size.

6.3 Binary Search Tree with Parallel Combining

In this section, we explain how to implement a parallel batched binary search tree algorithm by Blelloch et al. [32] in the form of `COMBINER_CODE` and `CLIENT_CODE` for the parallel combining framework described in the previous section.

Initially, we choose a sequential balanced binary search tree implementation that supports four operations with logarithmic complexity:

1. $T \leftarrow$ `join`$(T_L,\ k,\ T_R)$ — takes two trees T_L and T_R, and a value k, and returns a new tree for which the in-order values are a concatenation of the in-order values of T_L, k, and the in-order of T_R.

2. $T \leftarrow$ `join2`$(T_L,\ T_R)$ — takes two trees T_L and T_R, and returns a new tree for which the in-order values are a concatenation of the in-order values of T_L and T_R.

3. $(T_L, b, T_R) \leftarrow$ `split`$(T,\ k)$ — takes a tree T and a value k, and returns two trees T_L and T_R, and the boolean b, where T_L contains all the values of T that are less than k, T_R contains all the values of T that are greater than k, and b represents whether k is in T.

4. $b \leftarrow$ `contains(`T`,` k`)` — takes a tree T and a value k, and returns if k is in T.

As shown in the paper by Blelloch et al. [32] at least four state-of-the-art sequential implementations satisfy the above requirements: AVL [9], Red-Black [81], Weight-balanced [123] and Treap [138]. In our parallel batched implementation any sequential tree is represented by an object of `Tree` class. This class together with the functions described above appears in Figure 6.3 Lines 1-25.

We briefly describe the parallel batched algorithm by Blelloch et al. [32]. It is implemented as one recursive function `apply` that takes an interval of requests `S[`l`..`r`]` and a sequential binary search tree T, and applies the requests to the tree. `apply(S[`l`..`r`],` T`)` works in three stages: (1) splits T by the argument of the middle request `S[`mid`]`, where $mid = \frac{l+r}{2}$, into left T_L and right T_R trees; (2) calls fork-join with two branches: `apply(S[`l`..`$mid-1$`],` T_L`)` that returns tree T'_L and `apply(S[`$mid+1$`..`r`],` T_R`)` that returns tree T'_R; and, finally, (3) joins two trees T'_L and T'_R, possibly, with the argument of `S[`mid`]` if it is `insert`. In order to apply all requests we should call `root` $\leftarrow$ `apply(S[1..`$|S|$`],` `root)`, where `root` stores the tree.

This algorithm works correctly: if an Insert request has argument v then the resulting tree contains v since we join trees with v; if a Delete request has argument v then the resulting tree does not contain v since we do not join trees with v; and the resulting tree is perfectly balanced. The algorithm takes $O(c \cdot \log(\frac{m}{c}+1))$ work and $O(\log c \cdot \log m)$ span to apply a batch of size c to a tree of size m.

Now we describe the high-level idea behind the transformation from the described parallel batched data structure into the version written for parallel combining. The main difference between the algorithms is that we wake up clients instead of calling fork-join. Suppose that we have requests `S[1..`$|S|$`]` and corresponding clients-owners `C[1..`$|S|$`]`. When the algorithm by Blelloch et al. calls a branch of fork-join with `apply(S[`l`..`r`],` T`)`, our algorithm wakes up client `C[`$\frac{l+r}{2}$`]` to apply the same requests. Thus, typically, an application of `S[`l`..`r`]` to tree T works in four stages: (1) client `C[`mid`]`, where $mid = \frac{l+r}{2}$, splits T by the argument of `S[`mid`]` into T_L and T_R; (2) `C[`mid`]` wakes up client `C[`$\frac{l+(mid-1)}{2}$`]` to apply `S[`l`..`$mid-1$`]` to T_L and client `C[`$\frac{(mid+1)+r}{2}$`]` to apply `S[`$mid+1$`..`r`]` to T_R; (3) `C[`mid`]` waits until two awoken clients finish their work and return new trees; and, finally, (4) `C[`mid`]` joins two new trees, possibly, with the argument of `S[`mid`]` if it is `insert`. To start the algorithm we wake up client `S[`$\frac{1+|S|}{2}$`]` with `S[1..`$|S|$`]` and tree `root`.

We present the code of the algorithm. The code for necessary classes is presented in Figure 6.3, `COMBINER_CODE` is presented in Figure 6.4 and `CLIENT_CODE` is presented in Figure 6.5.

`STATUS_SET` consists of five elements `INITIAL`, `CONTAINS`, `CONTAINS_FINISHED`, `UPDATE` and `FINISHED`.

A Request object (Figure 6.3 Lines 30-39) consists of: a method *method* to be called (Insert, Delete or Contains) and its input argument v; a result *res* field; a *status* field; a tree *tree* to which the client-owner c of this request should apply requests; a $leftR$ request of the client that is awaken by c on the left part of requests; a $rightR$ request of the client that is awaken by c on the right part of requests.

The purpose of the last three fields will be clear from the code.

6.3.1 Contains Phase

Combiner: preparation (Figure 6.4 Lines 11-19). First, the combiner withdraws requests A from the combining data structure $\mathbb{C}$ (Line 11) and sorts them by their arguments (Line 13). Then it copies all update operations from A into set U and sets the status of all requests in A to `CONTAINS` (Lines 16-19).

Clients (Figure 6.5 Lines 2-6). Each client finds whether its argument appears in the tree (Line 2) and stores the result in *res* field of its request. Then if the request is `contains` (Line 3), the client sets the status of its request to `FINISHED` (Line 4) and ends

```
1  class Tree:
2    V v
3    Tree left
4    Tree right
5
6    // some balancing information
7
8  // joins two trees and value v
9  join(Tree l, V v, Tree r): Tree
10   ...
11
12 // joins two trees
13 join2(Tree l, Tree r): Tree
14   ...
15
16 // splits a tree into
17 // two trees by value v
18 // also returns whether
19 // v is present in the tree
20 split(Tree tree, V v):
21     <Tree, bool, Tree>
22   ...
23 // returns whether tree contains value v
24 contains(Tree tree, V v): bool
25   ...
26
27 // represents the maintained tree
28 global Tree root
29
30 class Request
31   method: { Contains, Insert, Delete }
32   V v
33   bool res
34   STATUS_SET status
35
36   Tree tree
37
38   Request leftR
39   Request rightR
```

Figure 6.3: Binary Search Tree. Classes

the execution. Otherwise, the request is an update and the client sets the status of its request to CONTAINS_FINISHED (Line 6) in order to signal the combiner that it finished the contains phase.

The combiner runs the same code as clients (Figure 6.4 Lines 21-26). Then it considers the contains phase finished when all requests in A changes their state from CONTAINS either to FINISHED (contains requests) or to CONTAINS_FINISHED (update requests) (Lines 28-30).

6.3.2 Update Phase

Combiner: preparation (Figure 6.4 Lines 32-68). As the first step (Lines 32-64), the combiner computes the results of each request and chooses at most one request for each value of the argument to apply. The combiner iterates through batches of update requests in U grouped by argument (Lines 38-51), i.e., $\mathtt{U}[l\,..\,r-1]$. For each batch the combiner reads the current state of the argument (Line 36): whether it is present in the tree. Then it logically applies requests with this argument one by one and calculates their responses. Finally, the combiner chooses at most one request that should be applied to the tree and adds it to set S (Lines 53-62): it is the last successful request, i.e., returns true. All requests that are not in S can be released and their status are set to FINISHED.

Now the combiner prepares the requests in S to be applied to the binary search tree. As discussed earlier in our algorithm client $c = \mathtt{C}[\frac{l+r}{2}]$ that applies $\mathtt{S}[l\,..\,r]$ wakes up two clients L on the left half and R on the right half of requests: in Line 66, the combiner prefills the fields leftR and rightR of the request of c with the requests of L and R, respectively. Finally, the combiner passes the tree argument to the first working client $\mathtt{C}[\frac{1+|S|}{2}]$, i.e., the client that owns the middle request, by writing the tree into tree field of the request (Line 67) and, then, wakes up that client by setting its status to UPDATE (Line 68).

Clients (Figure 6.5 Lines 8-12). Each client waits until its status changes from CONTAINS_FINISHED (Lines 8-9). If the new status is FINISHED then the client finishes its execution, otherwise,

```
1  fill_dependencies(S, l, r): Request
2    if l > r:
3      return null
4    mid ← (l + r) / 2
5    S[mid].leftR ←
6      fill_dependencies(S, l, mid - 1)
7    S[mid].rightR ←
8      fill_dependencies(S, mid + 1, r)
9    return S[mid]
10
11 A ← ℂ.getRequests()
12
13 sort(A, by arguments)
14
15 U ← ∅
16 for r ∈ A:
17   if isUpdate(r):
18     U ← U ∪ r
19   r.status ← CONTAINS
20
21 req.res ← client_contains(req)
22
23 if isContains(req):
24   req.status ← FINISHED
25 else:
26   req.status ← CONTAINS_FINISHED
27
28 for r ∈ A:
29   while r.status = CONTAINS:
30     nop
31
32 S ← ∅
33 l ← 1
34 while l ≤ |U|:
35   r ← l
36   state ← U[l].res
37
38   while r ≤ |U| and U[l].v = U[r].v:
39     if isInsert(U[r]):
40       if state = true:
41         U[r].res ← false
42       else:
43         U[r].res ← true
44         state ← true
45     else:
46       if state = true:
47         U[r].res ← true
48         state ← false
49       else:
50         U[r].res ← false
51     r ← r + 1
52
53   applied ← false
54   for i in r - 1..l:
55     if U[i].res = true:
56       if applied = false:
57         S ← S ∪ U[i]
58         applied ← true
59       else:
60         U[i].status ← FINISHED
61     else:
62       U[i].status ← FINISHED
63
64   l ← r
65
66 main ← fill_dependencies(S, 1, |S|)
67 main.tree ← root
68 main.status ← UPDATE
69
70 while req.status = CONTAINS_FINISHED:
71   nop
72
73 if req.status = UPDATE:
74   client_update(req)
75   req.status ← FINISHED
76
77 while main.status = UPDATE:
78   nop
79 root ← main.tree
```

Figure 6.4: Binary Search Tree. COMBINER_CODE

```
1  CLIENT_CODE:
2    req.res ←client_contains(req)
3    if isContains(req):
4      req.status ← FINISHED
5      return
6    req.status ← CONTAINS_FINISHED
7
8    while req.status = CONTAINS_FINISHED:
9      nop
10
11   if req.status ≠ FINISHED:
12     client_update(req)
13   return
14
15 client_contains(Request req):
16   req.res ← contains(root, req.v)
17   return
18
19 client_update(Request req):
20   (l, res, r) ← split(req.tree, res.v)
21
22   if req.leftR ≠ null:
23     req.leftR.tree ← l
24     req.leftR.status ← UPDATE
25   if req.rightR ≠ null:
26     req.rightR.tree ← r
27     req.rightR.status ← UPDATE
28
29   if req.leftR ≠ null:
30     while req.leftR.status =
31         CONTAINS_FINISHED:
32       nop
33     l ← req.leftR.tree
34
35   if req.rightR ≠ null:
36     while req.rightR.status =
37         CONTAINS_FINISHED:
38       nop
39     r ← req.rightR.tree
40
41   if isInsert(req):
42     req.tree ← join(l, req.v, r)
43   else:
44     req.tree ← join2(l, r)
45
46   req.status ← FINISHED
47   return
```

Figure 6.5: Binary Search Tree. CLIENT_CODE

the new status is UPDATE and the client has to perform an update (Line 12) described by the algorithm below.

At first, the client splits the provided tree in tree field by the argument v into two trees l and r. Then the client writes the tree l to tree field of leftR request (Line 23) and wakes up the owner of leftR by setting its status to UPDATE (Line 24). It does the same with rightR request (Lines 25-27). The client waits until the owner of leftR request finishes its work (Lines 30-32) , i.e., the status changes from CONTAINS_FINISHED, and stores the resulting tree in l (Line 33). It does the same with rightR request (Lines 35-39). Then the client joins trees l and r with the argument of its request if the request is insert (Line 42) and without, otherwise (Line 44). Finally, it sets the status of its request to FINISHED and ends the execution.

The combiner runs the same code as clients (Lines 70-75). Then it waits until the first working client finishes, i.e., its status changes from CONTAINS_FINISHED (Lines 77-78) and, finally, updates the tree, i.e., root (Line 79).

6.3.3 Analysis

We provide theorems on correctness and time bounds.

Theorem 6.3.1. *Our concurrent binary search tree implementation is linearizable.*

Proof. The execution can be split into combining phases (Figure 6.4) which do not intersect. We group the operations into the batches corresponding to the combining phase in which they are applied.

Consider the i-th combining phase. We linearize all the operations from the i-th phase right after the end of the corresponding getRequests() (Figure 6.4 Line 11) in order: at first, contains operations and, then, update operations ordered by sorting in Figure 6.4 Line 13.

To prove that this linearization is correct it is enough to show for each different argument that the operations with that argument are linearizable. Consider operations that have the same argument v. The results of these operations definitely satisfy the sequential execution since we apply them logically in the sorted order knowing whether v is in the tree prior to this combining phase. (Figure 6.4 Lines 32-64) The only thing left to show is that the resulting tree after the i-th combining phase contains v if the last successful operation is an insert, and does not have, otherwise. This directly follows from the algorithm since we apply to the tree only the last successful operation: if it is `insert` then the value is inserted into the tree (Figure 6.5 Line 42), otherwise, the value is not inserted (Figure 6.5 Line 44). Finally, since we join and split trees using the sequential algorithm — the resulting tree is perfectly balanced. □

Theorem 6.3.2. *Suppose that the combiner collects c requests using `getRequests()`. To apply the collected requests to the tree, the combiner spends $O(c + \log m)$ RMRs in CC model, each client spends $O(\log m)$ RMRs in CC model and, in total, the algorithm spends $O(c \cdot \log m)$ RMRs in CC model.*

Proof. Suppose that the batch consists of only update operations. This assumption slightly simplify the presentation, but it does not affect the bounds.

The combiner gathers requests ($O(c)$ RMRs, Line 11) and sorts them ($O(c \cdot \log c)$ primitive steps but $O(c)$ RMRs, Line 13). Then the combiner sets the status of requests to `CONTAINS` ($O(c)$ RMRs, Line 15-19).

The clients and the combiner participate in Contains phase. At first, each client waits for its status to change from `INITIAL` (1 RMR). Then the client calls `contains` function ($O(\log m)$ primitive steps and RMRs, Line 2). It finishes contains phase with changing its status to `CONTAINS_FINISHED` ($O(1)$ RMRs, Lines 3-6).

The combiner waits for the change of the status of the clients ($O(c)$ RMRs, Line 28-30). Then it sets the responses of the clients and choose the requests to apply ($O(c)$ RMRs, Lines 32-64). After that, the combiner fills the dependencies between requests ($O(c)$ RMRs, Line 66) and wakes up the first working client ($O(1)$ RMRs, Lines 67-68).

The clients and the combiner participate in Update phase. The client waits for its status to change (1 RMR, Lines 8-9). Then it splits the tree ($O(\log m)$ primitive steps and RMRs, Line 20), wakes up two clients ($O(1)$ RMRs, Lines 22-27), waits until they finish and change their status ($O(1)$ RMRs, Lines 29-39) and, finally, joins the resulting trees, possibly, with the argument of its request ($O(\log m)$ primitive steps and RMRs, Lines 41-46).

The combiner waits the first working client to finish ($O(1)$ RMRs, Line 77-79) and ends the execution.

Summing up, the combiner performs $O(c + \log m)$ RMRs and each client performs $O(\log m)$ RMRs, giving us in total $O(c \cdot \log m)$ RMRs. □

Remark 6.3.1. *The above bounds also hold in DSM model for the version of the described algorithm. For that we have to simply make spin-loops to loop on the local variables of process. Luckily for us, in our algorithm the purpose of each spin-loop is to wake up some process and at each point when we set the variable we know (or can deduce by a simple modification of the algorithm) which process is going to wake up. Thus, it is enough for each spin-loop to create a separate variable in the memory of the target process. Such transformation (in reality, it is slightly more technical than described above) of our algorithm provides an algorithm with the same bounds on RMRs but in DSM model.*

6.4 Priority Queue with Parallel Combining

Before we introduce our novel parallel batched priority queue algorithm we have to mention that all previous parallel batched algorithms (see Section 3.3) do not satisfy us because

```
1  class InsertSet:
2    List A, B
3
4    split(int l):
5        Pair<InsertSet, InsertSet>
6      L ← min(l, |A| + |B| - l)
7      X ← new InsertSet()
8      if |A| ≥ L:
9        for i in 1..L:
10         a ← X.A.removeFirst()
11         X.A.append(a)
12     else:
13       for i in 1..L:
14         b ← X.B.removeFirst()
15         X.B.append(b)
16
17     if L = l:
18       return (X, this)
19     else:
20       return (this, X)
```

```
21 class Node:
22   V val
23   bool locked
24   InsertSet split
25
26 class Request:
27   method: { ExtractMin, Insert }
28   V v
29   V res
30   STATUS_SET status
31   int start
32
33   // for client_insert(req)
34   // specifies the segment of leaves
35   // in a subtree of start node
36   int l, r
```

Figure 6.6: Parallel Priority Queue. Classes

they are tailored to a fixed number of processes p. Such a restriction leads to two major issues. Each of the known algorithm has a constraint on the size of batch: the algorithm by Pinotti and Pucci [132] allows only batches with size p, the algorithm by Deo and Prasad [56] allows only batches with size not exceeding p, and, finally, the algorithm by Sanders [137] allows a batch to have no more than p `extractMin` operations. Thus it is not possible to introduce new processes to the system and use them efficiently, when the data structure is already constructed, and, consequently, p is chosen to be n, the total number of processes in the system. Secondly, we expect a parallel batched implementation to be work-efficient: it should be able to apply a batch of size c to a priority queue of size m in a time not worse than $O(c \log m)$. Unfortunately, the known algorithms are not work-efficient when the size of the batch is always less than p (which is the common case in our environment, since p is chosen to be n): the algorithm by Pinotti and Pucci [132] induces $O(p \cdot (\log m + \log \log p))$ work, the algorithm by Deo and Prasad [56] induces $O(p \cdot \log m \cdot \log p)$ work and the algorithm by Sanders [137] induces $O(p \cdot \log m)$ work.

We respond to these issues with a new parallel batched algorithm that applies a batch of size c to a queue of size m in $O(c \cdot (\log c + \log m))$ RMRs in CC or DSM models. From this result we can get the theoretical bounds in the Work-Span framework: $O(c \cdot (\log m + \log c))$ work and $O(c \cdot \log c + \log m)$ span. By that, our algorithm can use up to $c \approx \log m$ processes efficiently.

6.4.1 Combiner and Client. Classes

Now, we describe our novel parallel batched priority queue in the form of `COMBINER_CODE` and `CLIENT_CODE` that fits the parallel combining framework described in Section 6.2. It is based on the sequential binary heap by Gonnet and Munro [75] described in Section 3.3. The code of necessary classes is presented in Figure 6.6, `COMBINER_CODE` is presented in Figure 6.7, and `CLIENT_CODE` is presented in Figure 6.8.

We introduce a sequential object `InsertSet` (Figure 6.6 Lines 1-20) that consists of two sorted linked lists A and B supporting *size* operation $|\cdot|$. The size of InsertSet S is $|S| = |S.A| + |S.B|$. InsertSet supports operation `split: (X, Y) ← S.split(`ℓ`)`, which splits InsertSet S into two InsertSet objects X and Y, where $|X| = \ell$ and $|Y| = |S| - \ell$.

```
1  A ← ℂ.getRequests()
2
3  if m ≤ |A|:
4    apply A sequentially
5    for r ∈ A:
6      r.status ← FINISHED
7    return
8
9  E ← ∅
10 I ← ∅
11
12 for r ∈ A:
13   if isInsert(r):
14     I ← I ∪ r
15   else:
16     E ← I ∪ r
17
18 bestE ← new int[|E|]
19 heap ← new Heap<Pair<V, int>>()
20
21 heap.insert((a[1], 1))
22
23 for i in 1..|E|:
24   (v, id) ← heap.extract_min()
25   bestE[i] ← id
26   heap.insert((a[2 · v], 2 · v))
27   heap.insert(
28     (a[2 · v + 1], 2 · v + 1))
29
30 for i in 1..|E|:
31   E[i].res ← a[bestE[i]].val
32   a[bestE[i]].locked ← true
33   E[i].start ← bestE[i]
34
35 l ← min(|E|, |I|)
36 for i in 1..l:
37   a[bestE[i]] ← I[i].v
38   I[i].status ← FINISHED
39
40 for i in l + 1..|E|:
41   a[bestE[i]] ← a[m]
42   m--
43
44 for i in 1..|E|:
45   E[i].status ← SIFT
46
47 if req.status = SIFT:
48   client_extract_min(req)
49
50 for i in 1..|E|:
51   while E[i].status = SIFT:
52     nop
53 I ← I[l + 1..|I|]
54
55 I[1].start ← 1
56 I[1].l ← 2^⌊log2(m+|I|)⌋
57 I[1].r ← 2 · I[1].l - 1
58 for i in 2..|I|:
59   t ← m + i
60   power ← 1
61   while t > 1:
62     p ← t / 2
63     if 2 · p = t:
64       break
65     t ← p
66     power ← 2 · power
67
68   if t = 1:
69     t ← 2
70   I[i].start ← t + 1
71   I[i].l ← I[i].start · power
72   I[i].r ← I[i].l + power - 1
73
74 // L and R are global variables
75 // necessary for client_insert(req)
76 L ← m + 1
77 R ← m + |I|
78
79 m ← m + |I|
80
81 args ← new V[|I|]
82 for i in 1..|I|:
83   args[i] ← I[i].v
84
85 sort(args)
86
87 a[1].split ← new InsertSet()
88 for i in 1..|I|:
89   a[1].split.A.append(args[i])
90
91 for i in 1..|I|:
92   I[i].status ← SIFT
93
94 if req.status = SIFT:
95   client_insert(req)
96
97 for i in 1..|I|:
98   while I[i].status = SIFT:
99     nop
```

Figure 6.7: Parallel Priority Queue. COMBINER_CODE

```
1  CLIENT_CODE:
2    if isInsert(req):
3      if req.status = SIFT:
4        client_insert(req)
5    else:
6      client_extract_min(req)
7    req.status ← FINISHED
8    return
9
10 client_extract_min(Request req):
11   v ← req.start
12   while 2 · v ≤ m:
13     while a[2 · v].locked
14       nop
15     if 2 · v + 1 ≤ m:
16       while a[2 · v + 1].locked:
17         nop
18     c ← 2 · v
19     if 2 · v + 1 ≤ m
20         and a[2 · v] > a[2 · v + 1]:
21       c ← 2 · v + 1
22     if a[c] > a[v]:
23       a[v].locked ← false
24       break
25     else:
26       swap(a[c], a[v])
27       a[c].locked ← true
28       a[v].locked ← false
29       v ← c
30   return
31
32 // Integers that specifies
33 // a segment of target nodes,
34 // i.e., m + 1 and m + |I|
35 global int L, R
36
37 targets_in_subtree(l, r): int
38   return min(r, R) - max(l, L) + 1
39
40 client_insert(Request req):
41   v ← req.start
42   while a[v].split = null:
43     nop
44   S ← a[v].split
45   a[v].split ← null
46
47   l ← req.l
48   r ← req.r
49
50   while v ∉ [L, R]:
51     a ← S.A.first()
52     b ← S.B.first()
53     if a[v] < min(a, b):
54       x ← a[v]
55       a[v] ← min(a, b)
56       if a < b:
57         S.A.pollFirst()
58       else:
59         S.B.pollFirst()
60       S.B.append(x)
61
62     mid ← (l + r) / 2
63     inL ←
64       targets_in_subtree(l, mid)
65     inR ←
66       targets_in_subtree(mid + 1, r)
67
68     if inL = 0:
69       v ← 2 · v + 1
70       l ← mid + 1
71
72     if inR = 0:
73       v ← 2 · v
74       r ← mid
75
76     if inL ≠ 0 and inR ≠ 0:
77       (S, T) ← S.split(inL)
78       a[2 · v + 1].split ← T
79       v ← 2 · v
80       r ← mid
81
82   if |S.A| ≠ 0:
83     a[v] ← S.A.first()
84   else:
85     a[v] ← S.B.first()
86   return
```

Figure 6.8: Parallel Priority Queue. CLIENT_CODE

This operation is executed sequentially in $O(L = \min(\ell, |S| - \ell))$ steps. The split operation works as follows: 1) new InsertSet T is created (Line 7); 2) if $|S.A| \geq L$ then the first L values of $S.A$ are moved to $T.A$ (Lines 9-11); otherwise, the first L values from $S.B$ are moved to $T.B$ (Lines 13-15); note that either $|S.A|$ or $|S.B|$ should be at least L; 3) if $L = \ell$ then (T, S) is returned; otherwise, (S, T) is returned (Lines 17-20).

The heap is defined by its size m and an array a of Node objects. Node object (Figure 6.6 Lines 21-24) has three fields: value *val*, boolean *locked* and InsertSet *split*.

`STATUS_SET` consists of three items: `INITIAL`, `SIFT` and `FINISHED`.

A Request object (Figure 6.6 Lines 26-36) consists of: a method *method* to be called and its input argument v; a result *res* field; a *status* field; a node identifier *start*.

6.4.2 ExtractMin Phase

Combiner: ExtractMin preparation (Figure 6.7 Lines 1-52). First, the combiner withdraws requests A from the combining data structure $\mathbb{C}$ (Line 1). If the size of A is larger than m, the combiner serves the requests sequentially (Lines 3-7). Intuitively, in this case, there is no way to parallelize the execution. For example, if A consists of only Insert requests and if there are more Insert requests than the number of nodes in the corresponding binary tree, we cannot insert them in parallel.

In the following, we assume that the size of the queue is at least the size of A. The combiner splits A into sets E and I (Lines 9-16): the set of ExtractMin requests and the set of Insert requests. Then it finds $|E|$ nodes $v_1, \ldots, v_{|E|}$ of heap with the smallest values (Lines 18-28), e.g., using the Dijkstra-like algorithm in $O(|E| \cdot \log |E|)$ primitive steps or $O(|E|)$ RMRs. For each request $E[i]$, the combiner sets $E[i].res$ to $a[v_i].val$, $a[v_i].locked$ to `true`, and $E[i].start$ to v_i (Lines 30-33).

The combiner proceeds by pairing Insert requests in I with ExtractMin requests in E using the next procedure (Lines 35-38). Suppose that $\ell = \min(|E|, |I|)$. For each $i \in [1, \ell]$, the combiner sets $a[v_i].val$ to $I[i].v$ and $I[i].status$ to `FINISHED`, i.e., this Insert request becomes completed. Then, for each $i \in [\ell + 1, |E|]$, the combiner sets $a[v_i].val$ to the value of the last node $a[m]$ and decreases m, as in the sequential algorithm (Lines 40-42). Finally, the combiner sets the status of all requests in E to `SIFT` (Lines 44-45).

Clients: ExtractMin phase (Figure 6.8 Lines 11-29). Briefly, the clients sift down the values in nodes $v_1, \ldots, v_{|E|}$ in parallel using hand-over-hand locking: the *locked* field of a node is set whenever there is a *sift down* operation working on that node.

A client c waits until the status of its request becomes `SIFT`. c starts sifting down from $req.start$. Suppose that c is currently at node v. c waits until the *locked* fields of the children become `false` (Lines 13-17). If $a[v].val$, the value of v, is less than the values in its children, then *sift down* is finished (Lines 23-24): c unsets $a[v].locked$ and sets the status of its request to `FINISHED`. Otherwise, let w be the child with the smallest value. Then c swaps $a[v].val$ and $a[w].val$, sets $a[w].locked$, unsets $a[v].locked$ and continues with node w (Lines 26-29).

If the request of the combiner is ExtractMin, it also runs the code above as a client (Figure 6.7 Lines 47-48). The combiner considers the ExtractMin phase completed when all requests in E have status `FINISHED` (Lines 50-52).

6.4.3 Insert Phase

Combiner: Insert preparation (Figure 6.7 Lines 53-99). For Insert requests, the combiner removes all completed requests from I (Line 53). Nodes $m+1, \ldots, m+|I|$ have to be leaves, because we assume that the size of I is at most the size of the queue. We call these leaves *target nodes*. The combiner then finds all *split nodes*: nodes for which the subtrees of both children contain at least one target node. (See Figure 6.9 for an example of how target and split nodes can be defined.)

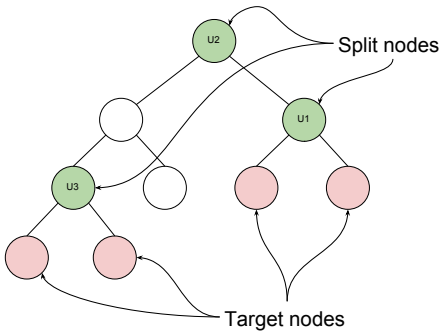


Figure 6.9: Split and target nodes

Since we have $|I|$ target nodes, there are exactly $|I|-1$ split nodes $u_1, \ldots, u_{|I|-1}$: u_i is the lowest common ancestor of nodes $m+i$ and $m+i+1$. They can be found in $O(|I|+\log m)$ primitive steps (Lines 55-72): starting with node $m+i$ go up the heap until a node becomes a left child of some node pr; this pr is u_i. We omit the discussion about the fields l and r of $I[i]$: they represent the smallest and the largest leaf identifiers in the subtree of u_i at the lowest level, and they are used to calculate the number of leaves at the lowest level that are newly inserted, i.e., $m+1, \ldots, m+|I|$, in constant time. The combiner sets $I[1].start$ to the root (the node with identifier 1), (Line 55) and, for each $i \in [2, |I|]$, it sets $I[i].start$ to the right child of u_{i-1} (node $2 \cdot u_{i-1}+1$) (Line 70). Then the combiner creates an InsertSet object X, sorts the arguments of the requests in I, puts them to $X.A$ and sets $a[1].split$ to X (Lines 81-89). Finally, it sets the status fields of all requests in I to `SIFT` (Lines 91-92).

Clients: Insert phase (Lines 41-85). Consider a client c with an incompleted request req. It waits while $a[req.start].split$ is null (Lines 42-43). Now c is going to insert values from InsertSet $a[req.start].split$ to the subtree of $req.start$. Let S be a local InsertSet variable initialized with $a[req.start].split$. For each node v on the path, c inserts values from S into the subtree of v. c calculates the minimum value x in S (Lines 51-53): the first element of $S.A$ or the first element of $S.B$. If $a[v].val$ is bigger than x, then the client removes x from S, appends $a[v].val$ to the end of $S.B$ and sets $a[v].val$ to x (Lines 54-60). Note that by the algorithm $S.B$ contains only values that were stored in the nodes above node v, thus, any value in $S.B$ cannot be bigger than $a[v].val$ and after appending $a[v].val$ $S.B$ remains sorted. Then the client calculates the number inL of target nodes in the subtree of the left child of v and the number inR of target nodes in the subtree of the right child of v (Lines 63-66, to calculate these numbers in constant time we use fields l and r of the request). If $inL = 0$, then all the values in S should be inserted into the subtree of the right child of v, and c proceeds with the right child $2v+1$ (Lines 69-70). If $inR = 0$, then, symmetrically, c proceeds with the left child $2v$ (Lines 73-74). Otherwise, if $inL \neq 0$ and $inR \neq 0$, v is a split node and, thus, there is a client that waits at the right child $2v+1$. Hence, c splits S to `(X, Y)` $\leftarrow$ `S.split(inL)` (Line 77): the values in X should be inserted into the subtree of node $2v$ and the values in Y should be inserted into the subtree of node $2v+1$. Then c sets $a[2v+1].split$ to Y, sets S to X and proceeds to node $2v$ (Lines 78-80). When c reaches a leaf v it sets the value $a[v].val$ to the only value in S (Lines 82-85) and sets the status of the request req to `FINISHED` (Line 7).

If the request of the combiner is an incompleted Insert, it runs the code above as a client (Figure 6.7 Lines 94-95). The combiner considers the Insert phase completed when all requests in I have status `FINISHED` (Lines 97-99).

6.4.4 Analysis

Now we provide theorems on correctness and time bounds.

Theorem 6.4.1. *Our concurrent priority queue implementation is linearizable.*

Proof. The execution can be split into combining phases (Figure 6.7 Lines 1-99) which do not intersect. We group the operations into batches corresponding to the combining phase in which they are applied.

Consider the i-th combining phase. We linearize all the operations from the i-th phase right after the end of the corresponding `getRequests()` (Line 1) in the following order: at

first, we linearize ExtractMin operations in the increasing order of their responses, then, we linearize Insert operations in any order.

To see that this linearization is correct it is enough to prove that the combiner and the clients apply the batch correctly.

Lemma 6.4.1. *Suppose that the batch of the i-th combining phase contains a ExtractMin operations and b Insert operations with arguments $x_1, \ldots, x_b$. Let V be the set of values stored in the priority queue before the i-th phase. The combiner and the clients apply this batch correctly:*

- *The minimal a values $y_1, \ldots, y_a$ in V are returned to ExtractMin operations.*
- *After an execution the set of values stored in the queue is equal to $V \cup \{x_1, \ldots, x_b\} \setminus \{y_1, \ldots, y_a\}$. and the values are stored in nodes with identifiers $1, \ldots, |V| - b + a$.*
- *After an execution the heap property is satisfied for each node.*

Proof. The first statement is correct, because the combiner chooses the smallest a elements from the priority queue and sets them as the results of ExtractMin requests (Lines 18-33).

The second statement about the set of values straightforwardly follows from the algorithm. During ExtractMin phase the combiner finds a smallest elements, replaces them with $x_1, \ldots, x_{\min(a,b)}$ and with values from the last $a - \min(a, b)$ nodes of the heap: the set of values in the priority queue becomes $V \cup \{x_1, \ldots, x_{\min(a,b)}\} \setminus \{y_1, \ldots, y_b\}$ and the values are stored in nodes $1, \ldots, |V| - a + \min(a, b)$. Then, the *sift down* is initiated, but it does not change the set of values and it does not touch nodes other than $1, \ldots, |V| - a + \min(a, b)$. During Insert phase the values $x_{\min(a,b)+1}, \ldots, x_b$ are inserted and new nodes which are used in Insert phase are $|V| - a + \min(a, b) + 1, \ldots, |V| - a + b$. Thus, the final set of values is $V \cup \{x_1, \ldots, x_b\} \setminus \{y_1, \ldots, y_a\}$ and the values are stored in nodes $1, \ldots, |V| - a + b$.

The third statement is slightly tricky. At first, the combiner finds a smallest elements that should be removed and replaces them with $x_1, \ldots, x_{\min(a,b)}$ and with values from the last $a - \min(a, b)$ nodes of the heap. Suppose that these a smallest elements were at nodes $v_1, \ldots, v_a$, sorted by their depth (the length of the shortest path from the root) in non-increasing order. These nodes form a connected subtree where v_a is the root of the heap. Suppose that they do not form a connected tree or v_a is not the root of the heap. Then there exists a node v_i which parent p is not v_j for any j. This means that the values in nodes $v_1, \ldots, v_a$ are not the smallest a values: by the heap property a value at p is smaller than the value at v_i.

Now a processes perform *sift down* from the nodes $v_1, \ldots, v_a$. We show that when a node v is unlocked, i.e., its `locked` field is set to false, the value at v is the smallest value in the subtree of v. This statement is enough to show that the heap property holds for all nodes after ExtractMin phase, because at that point all nodes are unlocked.

Consider an execution of *sift down*. We prove the statement by induction on the number of unlock operations. Base. No unlock happened and the statement is satisfied for all unlocked nodes, i.e., all the nodes except for $v_1, \ldots, v_a$. Transition. Let us look right before the k-th unlock: the unlock of a node v. The left child l of v should be unlocked and, thus, l contains a value that is the smallest in its subtree. The same statement holds for the right child r of v. v chooses the smallest value between the value at v and the values at l and r. This value is the smallest in the subtree of v. Thus, the statement holds for v when unlocked.

After that, the algorithm applies the incompleted $b - \min(a, b)$ Insert operations. We name the nodes with at least one target node in the subtree as *modified*. Modified nodes are the only nodes whose value can be changed and, also, each modified node is visited by exactly one client. To prove that after the execution the heap property for each modified node holds: we show by induction on the depth of a modified node that if a node v is visited by a client with InsertSet S then: (1) $S.A$ is sorted; (2) $S.B$ is sorted and contains only values that were stored in ancestors of v after ExtractMin phase; and (3) v contains

the smallest value in its subtree when the client finishes with it. Base. In the root $S.A$ is sorted, $S.B$ is empty and the new value in the root is either the first value in $S.A$ or the current value in the root, thus, it is the smallest value in the heap. Transition from depth k to depth $k+1$. Consider a modified node v at depth $k+1$ and its parent p. Suppose that p was visited by a client with InsertSet S_p. By induction, $S_p.A$ is sorted and $S_p.B$ is sorted and contains only the values that were in ancestors. Then the client chooses the smallest value in p: either $a[p]$, the first value of $S_p.A$ or the first value of $S_p.B$. Note that after any of these three cases $S_p.A$ and $S_p.B$ are sorted and $S_p.B$ contains only values from ancestors and node p:

- $a[p]$ is the smallest, then $S_p.A$ and $S_p.B$ are not modified;
- we poll the first element of $S_p.A$ or $S_p.B$; $S_p.A$ and $S_p.B$ are still sorted; then we append $a[p]$ to $S_p.B$, and $a[p]$ has to be the biggest element in $S_p.B$, since $S_p.B$ contains only the values from ancestors.

Then the client splits S_p and some client, possibly, another one, works on v with IntegerSet S. Since, S is a subset of S_p then $S.A$ is sorted and $S.B$ is sorted and contains only the values from ancestors (ancestors of p and, possibly, p). Finally, the client chooses the smallest value to appear in the subtree: the first value of $S.A$, the first value of $S.B$ and $a[v]$. □

□

Theorem 6.4.2. *Suppose that the combiner collects c requests using `getRequests()`. Then the combiner and the clients apply these requests to a priority queue of size m using $O(c + \log m)$ RMRs in CC model each and $O(c \cdot (\log c + \log m))$ RMRs in CC model in total.*

Proof. Suppose that the batch consists of a `ExtractMin` operations and b `Insert` operations.

The combiner splits requests into two sets E and I ($O(c)$ RMRs, Lines 9-16). Then it finds a nodes with the smallest values ($O(a \log a)$ primitive steps, but $O(a)$ RMRs, Lines 18-28) using Dijkstra-like algorithm. After that, the combiner sets up ExtractMin requests, sets their status to `SIFT` and pairs some Insert requests with ExtractMin requests ($O(a)$ RMRs, Lines 30-45).

The clients participate in ExtractMin phase. At first, each client waits for its status to change (1 RMR). Then the client performs at most $\log m$ iterations of the loop (Line 12): waits on the *locked* fields of the children ($O(1)$ RMRs, Lines 13-17); reads the values in the children ($O(1)$ RMRs, Line 20); compares these values with the value at the node, possibly, swap the values, lock the proper child and unlock the node ($O(1)$ RMRs, Lines 22-29). When the client stops it changes the status (1 RMR, Line 7).

The combiner waits for the change of the status of the clients ($O(a)$ RMRs, Lines 50-52). Summing up, in ExtractMin phase each client performs $O(\log m)$ RMRs and the combiner performs $O(a + \log m)$ RMRs, giving $O(c + c \cdot \log m)$ RMRs in total.

The combiner throws away completed Insert requests ($O(b)$ primitive steps and 0 RMRs, Line 53). Then it finds the split nodes ($O(\log m + b)$ primitive steps, but 0 RMRs, Lines 55-72). After that the combiner sorts arguments of remaining Insert requests, sets their status to `SIFT` and sets up the initial InsertSet ($O(b \cdot \log b)$ primitive steps, but $O(b)$ RMRs, Line 81 and Line 92).

The clients participate in Insert phase. At first, a client t waits while the corresponding InsertSet is null (1 RMR, Lines 41-43). Suppose that it reads the InsertSet S and starts the traversal down. The client performs at most $\log m$ iterations of the loop (Line 50): choose the smallest value ($O(1)$ RMRs, Lines 51-60), find whether to split InsertSet ($O(1)$ RMRs, Lines 62-74), split InsertSet (calculated below, Line 77) and passe one InsertSet to another client ($O(1)$ RMRs, Lines 78-80). Now let us calculate the number of RMRs spent

in Line 77. Suppose that there are k iterations of the loop and the size of S at iteration i is s_i. At the i-th iteration split works in $O(\min(s_{i+1}, s_i - s_{i+1})) = O(s_i - s_{i+1})$ primitive steps and RMRs. Summing up through all iterations we get $O(s_1) = O(b)$ RMRs spent by t in Line 77. Finally, t sets the value in the leaf ($O(1)$ RMRs, Lines 82-85) and changes the status ($O(1)$ RMRs, Line 7).

The combiner waits for the change of the status of the clients ($O(b)$ RMRs, Lines 97-99). Summing up, in Insert phase the clients and the combiner perform $O(b + \log m)$ RMRs each. Consequently, the straightforward bound on the total number of RMRs is $O(c^2 + c \cdot \log m)$ RMRs.

To get the improved bound we carefully calculate the total number of RMRs spent on the splits of InsertSets in Line 77. This number equals to the number of values that are moved to newly created sets during the splits. For simplicity we suppose that inserted values are bigger than all the values in the priority queue and, thus, each InsertSet contains only the newly inserted values. This assumption does not affect the bound. Consider now the inserted value v. Suppose that v was moved k times and at the i-th time it was moved during the split of InsertSet with size s_i. Because v is moved during split only to the set with the smaller size: $s_1 \geq 2 \cdot s_2 \geq \ldots \geq 2^{k-1} \cdot s_k$. k is less than $\log c$, because $s_1 \leq c$, and, thus, v was moved no more than $\log c$ times. This means, that in total during the splits of InsertSets no more than $c \cdot \log c$ values are moved to new sets, giving $O(c \cdot \log c)$ RMRs during the splits. This gives us a total bound of $O(c \cdot (\log c + \log m))$ RMRs during Insert phase.

To summarize, the combiner and the clients perform $O(c + \log m)$ RMRs each and $O(c \cdot (\log c + \log m))$ RMRs in total. □

Remark 6.4.1. *The above bounds also hold in DSM model for the version of the described algorithm. For that we have to simply make spin-loops to loop on the local variable of processes. In our algorithm the purpose of each spin-loops is to wake up some process. At most places in our algorithm when we set the variable on which we spin we know (or can deduce by a simple modification of the algorithm) which process is going to wake up. For each spin-loop it is enough to create a separate variable in the memory of the target process.*

The only two non-trivial spin-loops are in `CLIENT_CODE` *(Lines 13-17) where we do not know a process that is going to wake up. To elliviate this issue we expand each Node object with the pointer to process proc. When the process wants to* sift-down*, first, it registers itself in $a[v].proc$ and, then, checks $a[2v].locked$ and $a[2v+1].locked$. If some of them are* `true` *then it spins on specifically created local variables: on $notify_{2v}$ if $a[2v].locked$ is* `true`*, and on $notify_{2v+1}$ if $a[2v+1].locked$ is* `true`*. Then, the algorithm standardly performs swapping routine. At the end, it unlocks the node, i.e., sets $a[v].locked$ to* `false`*, then, reads a process $a[v/2].proc$ and notifies it by setting its corresponding variable $notify_v$. Note that the total number of* notify *local variables that is needed by each process is logarithmic from the size of the queue.*

The described transformation (in reality, it is slightly more technical than described above) of our algorithm provides an algorithm with the same bounds on RMRs but in DSM model.

6.5 Experiments

We evaluate Java implementations of our data structures on a 4-processor AMD Opteron 6378 2.4 GHz server with 16 threads per processor (yielding 64 threads in total), 512 Gb of RAM, running Ubuntu 14.04.5 with Java 1.8.0_111-b14 and HotSpot JVM 25.111-b14.

6.5.1 Concurrent Dynamic Graph

To illustrate how parallel combining can be used to construct read-optimized concurrent data structures, we took the sequential dynamic graph implementation by Holm et al. [95]. This data structure supports two update methods: an insertion of an edge and a deletion of an edge; and one read-only method: a connectivity query that tests whether two vertices are connected.

We compare our implementation based on parallel combining (PC) against three others: (1) Lock, based on ReentrantLock from java.util.concurrent; (2) RW Lock, based on ReentrantReadWriteLock from java.util.concurrent; and (3) FC, based on *flat combining* [86]. The code is available at https://github.com/Aksenov239/concurrent-graph.

We consider workloads parametrized with: 1) the fraction x of connectivity queries (50%, 80% or 100%, as we consider read-dominated workloads); 2) the set of edges E: edges of a single random tree, or edges of ten random trees; 3) the number of processes P (from 1 to 64). We prepopulate the graph on 10^5 vertices with edges from E: we insert each edge with probability $\frac{1}{2}$. Then we start P processes. Each process repeatedly performs operations: 1) with probability x, it calls a connectivity query on two vertices chosen uniformly at random; 2) with probability $1-\frac{x}{2}$, it inserts an edge chosen uniformly at random from E; 3) with probability $1-\frac{x}{2}$, it deletes an edge chosen uniformly at random from E.

We denote the workloads with E as a single tree as *Tree* workloads, and other workloads as *Trees* workloads. *Tree* workloads are interesting because they show the degenerate case: the dynamic graph behaves as a dynamic tree. In this case, about 50% of update operations successfully change the spanning tree, while other update operations only check the existence of the edge and do not modify the graph. *Trees* workloads are interesting because a reasonably small number (approximately, 5-10%) of update operations modify the set of all edges and the underlying complex data structure that maintains a spanning forest (giving in total the squared logarithmic complexity), while other update operations only can modify the set of edges and cannot modify the underlying complex data structure (giving in total the logarithmic complexity).

For each setting and each algorithm, we run the corresponding workload for 10 seconds to warmup HotSpot JVM and then we run the workload five more times for 10 seconds. The average throughput of the last five runs is reported in Figure 6.10.

From the plots we can infer two general observations: PC exhibits the highest thoughput over all considered implementations and it is the only one whose throughput scales up with the number of the processes. On the 100% workload we expect the throughput curve to be almost linear since all operations are read-only and can run in parallel. The plots almost confirm our expectation: the curve of the throughput is a linear function with coefficient $\frac{1}{2}$ (instead of the ideal coefficient 1). We note that this is almost the best we can achieve: a combiner typically collects operations of only approximately half the number of working processes. In addition, the induced overhead is still perceptible, since each connectivity query works in just logarithmic time. With the decrease of the fraction of read-only operations we expect that the throughput curve becomes flatter, as plots for the 50% and 80% workloads confirm.

It is also interesting to point out several features of other implementations. At first, FC implementation works slightly worse than Lock and RW Lock. This might be explained as follows. Lock implementations (`ReentrantLock` and `ReentrantReadWriteLock`) behind Lock and RWLock implementations are based on CLH Lock [51] organized as a *queue*: every competing process is appended to the queue and then waits until the previous one releases the lock. Operations on the dynamic graph take significant amount of time, so under high load when the process finishes its operation it appends itself to the queue in the lock without any contention. Indeed, all other processes are likely to be in the queue and, thus, no process can contend. By that the operations by processes are serialized with almost no overhead. In contrast, the combining procedure in FC introduces non-negligible

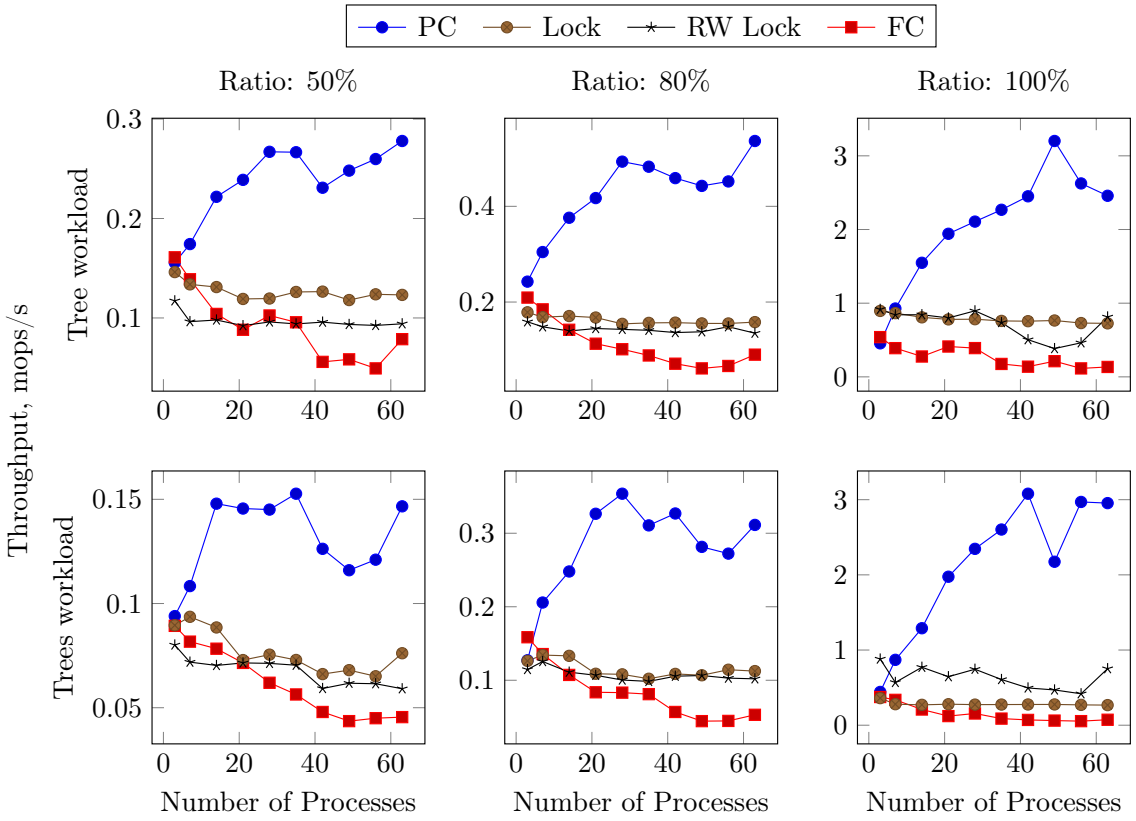


Figure 6.10: Dynamic graph implementations

overhead related to gathering the requests and writing them into requests structures.

Second, it is interesting to observe that, against the intuition, RWLock is not so superior with respect to Lock on read-only workloads. As can be seen, when there are update operations in the workload RWLock works even worse than Lock. We relate this to the fact that the overhead hidden inside `ReentrantReadWriteLock` spent on manipulation with read and write requests is bigger than the overhead spent by `ReentrantLock`. With the increase of the percentage of read-only operations the difference between Lock and RWLock diminishes and RWLock becomes dominant since read operations become more likely to be applied concurrently (for example, on 50% it is normal to have an execution without any parallelization: read operation, write operation, read operation, and so on). However, on 100% one could expect that RWLock should exhibit ideal throughput. Unfortunately, in this case, under the hood `ReentrantReadWriteLock` uses compare&swap on the shared variable that represents the number of current read operations. Read-only operations take enough time but not enough to amortize the considerable traffic introduced by concurrent compare&swaps. Thus, the plot for RWLock is almost flat, getting even slightly worse with the increase of the number of processes, and we blame the traffic for this.

6.5.2 Binary Search Tree

We implement our algorithm using AVL binary search tree as the sequential implementation, since it seems to be the fastest. Then we compare it with the state-of-the-art concurrent binary search trees with relaxed AVL-balancing scheme [39, 52, 57].

We consider typical workloads parametrized by: 1) the fraction x of update queries (0%, 20%, 100%), 2) the key range R ($[0, 2 \cdot 10^6]$ or $[0, 2 \cdot 10^7]$), and 3) the number of processes P (from 1 to 64). We prepopulate the tree with values from the key range: we add a value with probability $\frac{1}{2}$. Then we start P processes. Each process repeatedly performs operations: 1) with probability $1 - x$, it calls a contains query given a value chosen uniformly at random from the range; 2) with probability $\frac{x}{2}$, it inserts a value

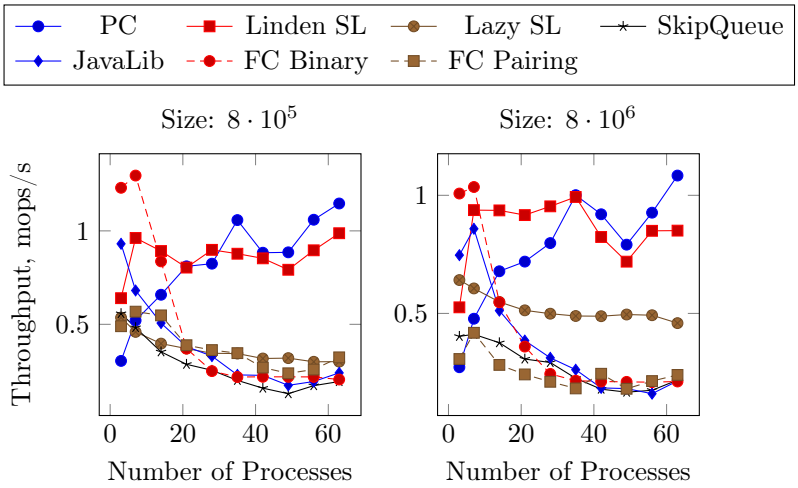


Figure 6.11: Priority Queue implementations

chosen uniformly at random from the range; 3) with probability $\frac{x}{2}$, it deletes a value chosen uniformly at random from the range.

We do not show here the plots, but we give the main conclusion: our implementation works worse than others by up to 7 times. This is expected, since when the number of concurrent operations on binary search tree is moderately small in comparison to the size of the tree, the operations are performed almost without data races. Suppose that there are no data races at all, then we can implement a parallel batched algorithm that calls operations from a batch concurrently giving $O(\log m)$ span, where m is the size of the tree. At the same time, the parallel batched algorithm by Blelloch et al. [32] has $O(\log c \cdot \log m)$ span, where c is the size of the batch and m is the size of the queue. Thus, we can expect the parallel batched algorithm to work $\log c \approx 5$ times slower even not considering the explicit synchronization.

To summarize, the binary search tree is an example of a data structure that does not benefit from parallel combining on small batches (proportional to the number of processes): the synchronization used by the concurrent algorithm is smaller than the synchronization used by the parallel batched algorithm.

6.5.3 Priority Queue

We run our algorithm (PC) against six state-of-the-art concurrent priority queues: (1) the lock-free skip-list by Linden and Johnson (Linden SL [113]), (2) the lazy lock-based skip-list (Lazy SL [90]), (3) the non-linearizable lock-free skip-list by Herlihy and Shavit (SkipQueue [90]) as an adaptation of Lotan and Shavit's algorithm [139], (4) the lock-free skip-list from Java library (JavaLib), (5) the binary heap with flat combining (FC Binary [86]), and (6) the pairing heap with flat combining (FC Pairing [86]). [2] The code is available at https://github.com/Aksenov239/FC-heap.

We consider workloads parametrized by: 1) the initial size S ($8 \cdot 10^5$ or $8 \cdot 10^6$); and 2) the number P of working processes (from 1 to 64). We prepopulate the queue with S random integers chosen uniformly from the range $[0, 2^{31} - 1]$. Then we start P processes, and each process repeatedly performs operations: with equal probability it either inserts a random value taken uniformly from $[0, 2^{31} - 1]$ or extracts the minimum value.

For each setting and each algorithm, we run the corresponding workload for 10 seconds to warmup HotSpot JVM and then we run the workload five more times for 10 seconds. The average throughput of the last five runs is reported in Figure 6.11.

On a small number of processes (< 15), PC performs worse than other algorithms.

[2] We are aware of the cache-friendly priority queue by Braginsky et al. [36], but we do not have its Java implementation.

With respect to Linden SL, Lazy SL, SkipQueue and JavaLib this can be explained by two different issues:

- Synchronization incurred by PC is not compensated by the work done;
- Typically, a combiner collects operations of only approximately half the processes, thus, we "overserialize", i.e., only $\frac{n}{2}$ operations can be performed in parallel.

In contrast, on small number of processes, the other four algorithms can perform operations almost with no contention. With respect to algorithms based on flat combining, FC Binary and FC Pairing, our algorithm is simply slower on one process than the simplest sequential binary and pairing heap algorithms.

With the increase of the number of processes the synchronization overhead significantly increases for all algorithms (in addition to the fact that FC Binary and FC Pairing cannot scale). As a result, starting from 15 processes, PC outperforms all algorithms except for Linden SL. Linden SL relaxes the contention during ExtractMin operations, and it helps to keep the throughput approximately constant. At approximately 40 processes the benefits of the parallel batched algorithm in PC starts prevailing the costs of explicit synchronization, and our algorithms overtakes Linden SL.

It is interesting to note that FC Binary performs very well when the number of processes is small: the overhead on the synchronization is very small, the processes are from the same core and the simplest binary heap performs operations very fast.

6.6 Related Work

To the best of our knowledge, the first attempt to combine concurrent operations was introduced by Yew et al. [156]. They introduced a *combining tree*: processes start at the leaves and traverse upwards to gain exclusive access by reaching the root. If, during the traversal, two processes access the same tree node, one of them adopts the operations of another and continues the traversal, while the other stops its traversal and waits until its operations are completed. Several improvements of this technique have been discussed, such as adaptive combining tree [141], barrier implementations [82, 116] and counting networks [140].

A different approach was proposed by Oyama et al. [126]. Here the data structure is protected by a lock. A thread with a new operation to be performed adds it to a list of submitted requests and then tries to acquire the lock. The process that acquires the lock performs the pending requests on behalf of other processes from the list in LIFO order, and later removes them from the list. Its main drawback is that all processes have to perform CAS on the head of the list. The *flat combining* technique presented by Hendler et al. [86] addresses this issue by replacing the list of requests with a *publication list* which maintains a distinct *publication record* per participating process. A process puts its new operation in its publication record, and the publication record is only maintained in the list if the process is sufficiently active. This way the process generally does not need to perform CAS on the head of the list. Variations of flat combining were later proposed for various contexts [58, 64, 65, 96].

Hierarchical combining [87] is the first attempt to improve the performance of combining using the computation power of clients. The list of requests is split into blocks, and each of these blocks has its own combiner. The combiners push the combined requests from the block into the second layer implemented as the standard flat combining with one combiner. This approach, however, may be sub-optimal as it does not have *all* clients participating. Moreover, this approach works only for specific data structures, such as stacks or unfair synchronous queues, where operations could be combined without accessing the data structure.

Agrawal et al. [13] decide to use a *parallel batched* data structure instead of a concurrent one in a different context. They provide provable bounds on the running time of a dynamic

multithreaded parallel program using P processes and a specified *scheduler*. The proposed scheduler extends the work-stealing scheduler by maintaining separate *batch* work-stealing deques that are accessed whenever processes have operations to be performed on the abstract data type. A process with a task to be performed on the data structure stores it in a *request array* and tries to acquire a global lock. If succeeded, the process puts the task to perform the batch update in its batch deque. Then all the processes with requests in the request array run the work-stealing routine on the batch deques until there are no tasks left. The idea of [13] is similar to ours. However, our goals are different: we aim at improving the performance of a *concurrent* data structure while their goal was to establish bounds on the running time of a *parallel program for dynamic multithreading*. Implementing a concurrent data structure from its parallel batched counterpart for dynamic multithreading is only one of the applications of our parallel combining, as shown in Section 6.2.2.

6.7 Conclusion

In this chapter, we showed that parallel combining can provide performance gains and some theoretical guarantees. We can come up with another important advantage of our technique: it can enable the first ever concurrent data structures of some data types, that are efficient for requests of any type, either read-only or update. As a good example we can provide a *dynamic tree* — a data type that supports the same operations as the dynamic graph, but with the condition that the graph is always a forest. It does not have concurrent implementation but has a parallel batched one [6].

As shown in Section 6.5, our concurrent priority queue performs well compared to state-of-the-art algorithms. We assume that one of the reasons is that the underlying parallel batched implementation is designed for static multithreading and, thus, it has little synchronization overhead. This might not be the case for implementations based on dynamic multithreading, where the overhead induced by the scheduler can be much higher. We intend to check this in the forthcoming work.

7 On Helping and Stacks

7.1 Introduction

In a *wait-free* data structure, every process is guaranteed to make progress in its own speed, regardless of the behavior of other processes [89]. It has been observed, however, that achieving wait-freedom typically involves some *helping* mechanism (e.g., [64, 68, 131, 151]). Informally, helping means that a process may perform additional work on behalf of other processes. But how to define formally the phenomenon of helping? There are three attempts done up to date: the linearization-based helping by Censor-Hillerl et al. [44] and two, valency-based and universal, by Attiya et al. [23].

This chapter is devoted to the linearization-based helping, but at first we explain what makes us take helping into the consideration. Initially, we started with a question that was open for many years: does queue belong to Common2?

To explain the meaning of the question we provide several definitions. In the *consensus* problem, each process proposes a value and is required to decide on a value such that the following properties are satisfied in every execution:

- Termination. Every correct process decides.
- Agreement. No two processes decide on different values.
- Validity. The decided value was proposed by some process.

The *consensus number* of a primitive on a shared register [89] is the maximum number n such that it is possible to solve consensus on n processes in a wait-free manner from reads, writes and this primitive. For example, *test&set*, the primitive that retrieves the boolean value from the memory cell and sets the value in the cell to `true`, has consensus number 2. Finally, Common2 is the family of data types that have wait-free implementions from primitives with consensus number 2 for any number of processes.

It was proven by Afek et al. [12] that stack belongs to Common2: they provided a wait-free algorithm implemented with reads, writes and test&set primitives. For queue, known Common2 implementations work only in special cases: each process either only enqueues or dequeues, and the number of processes that dequeue [111] or the number of processes that enqueue [61] should not exceed two.

But how the question about Common2 queue implementation is related to helping? Attiya et al. [23] searched for the wait-free implementations of stacks and queues that do not have the non-trivial valency-based helping and are implemented only using reads, writes and test&set primities, i.e., a relaxed version of Common2. They found that such an implementation exists for stack, but does not exist for queue. This can give us an intuition that queue does not belong to Common2.

The valency-based helping has the disadvantage that it cannot capture the helping between the operations that return trivial values: for example, in the case of stack, two pushes do "help" each other. Because of that we switched our attention to the linearization-based helping by Censor-Hillel et al. [44]. They proposed a natural formalization based on the notion of *linearization*: a process p helps an operation of a process q in a given execution if a step of p determines that an operation of q takes effect, or *linearizes*, before some other operation in any possible extension. At first glance, there is no connection between the question about Common2 queue and this definition: they claimed that stack and queue are *exact order* types and, then, they proved that any *exact order* type does not have

a wait-free help-free implementation using reads, writes and compare&swap primitives. Informally, a sequential data type is exact order if for some sequence of operations every change in the relative order of two operations affects the result of some other operations. However, we found that that the stack data type is not exact order. Hence, the proof of help-free impossibility for exact order types given in [44] does not apply to stack, and there can exist a distinction between stack and queue!

Unfortunately, we reassert the result by Censor-Hillel et al. via a direct proof that stack does not have a wait-free and help-free implementation using reads, writes, compare&swap and fetch&add primitives. At first, we show the result for implementations using read, write and compare&swap operations in systems with at least three processes, and, then, extend the proof to those additionally using fetch&add in systems with at least four proccesses. The structure of these two proofs resembles the structure of the proofs from the paper by Censor-Hillel et al. [44], but the underlying reasoning is novel. Unlike their approach our proofs argue about the order of operations given their responses *only* after we empty the data structure. As a result, certain steps of the proof become more technically involved.

Roadmap

The chapter is organized as follows. In Section 7.2, we present a computational model and necessary definitions. In Section 7.3, we recall the definition of helping and highlight the mistake in [44]. In Section 7.4, we give our direct proof. In Section 7.5, we explain how to build a help-free wait-free implementation of any data type given *move&increment* primitive. In Section 7.6, we discuss the related work. And, finally, we conclude in Section 7.7.

7.2 Model and Definitions

We briefly remind the model and definitions. All missed definitions can be found in Chapter 2.

We consider a system of n processes $p_1, \ldots, p_n$ communicating via invocations of *primitives* on a shared memory. During this chapter we assume that primitives are *read*, *write*, *compare&swap*, *fetch&add* and *move&increment*. The definition of the last primitive appear in Section 7.5.

In this chapter, we discuss stack implementations. To simplify the reasoning all considered implementations are deterministic. Nevertheless, the provided proofs can be easily extended to randomized implementations. For the rest of the section we fix some implementation of stack.

A *program* of a process specifies a sequence of operations calls on an object. The program may include local computations and can choose which operation to execute depending on the results of the previous operations.

A (low-level) *history* is a finite or infinite sequence of primitive steps. Note that in this chapter when we mention *history* we mean that it is *low-level* (if not specified otherwise explicitly). Each step is coupled with a specific operation that is being executed by the process performing this step. The first step of an operation always comes with the input parameters of the operation, and the last step of an operation is associated with the return of the operation. Given two histories h_1 and h_2 we denote by $h_1 \circ h_2$ the concatenation of h_1 and h_2.

A *schedule* is a finite or infinite sequence of processes' identifiers. Given a schedule, an implementation and programs provided to the processes, one can unambiguously determine the corresponding history. And vice versa, given a history one can always build a schedule by substituting the steps of history to the process that performed it. Assuming a fixed program for each process (these programs will be clear from the context), and a history

h, we denote by $h \circ p_i$ the history derived from scheduling process p_i to take the next step (if any) following its program immediately after h.

The set of histories H induced by an implementation consists of all possible histories induced by all possible processes' programs with all possible schedules. An implementation of a data type is linearizable if each history from the set of histories has a linearization (Definition 2.3.2). A *linearization function* defined over a set of linearizable histories H maps every history in H to its linearization. Note that a linearizable implementations may have multiple linearization functions defined on the set of its histories.

7.3 Helping and Exact Order Types

In this section, we recall the definitions of helping and exact order type in [44], and show that stack is *not* exact order.

Definition 7.3.1 (Decided before)**.** *For a history h in a set of histories H, a linearization function f over H, and two operations op_1 and op_2, we say that op_1* is decided before *op_2 in h with respect to f and H, if there exists no extension $s \in H$ of h such that $op_2 \prec_{f(s)} op_1$.*

Definition 7.3.2 (Helping)**.** *A set of histories H with a linearization function f over H is* help-free *if for every $h \in H$, every two operations op_1, op_2, and a single computation step γ such that $h \circ \gamma \in H$ it holds that if op_1 is decided before op_2 in $h \circ \gamma$ and op_1 is not decided before op_2 in h then γ is a step in the execution of op_1.*

An implementation is help-free*, if there exists a linearization function f such that the set of histories of this implementation with f is help-free.*

Following the formalism of [44], if S is a sequence of operations, we denote by $S(n)$ the first n operations in S, and by S_n the n-th operation of S. We denote by $(S + op?)$ the set of sequences that contains S and all sequences that are similar to S, except that a single operation op is inserted somewhere between (or before, or after) the operations of S.

Definition 7.3.3 (Exact Order Types)**.** *An* exact order type *is a data type for which there exists an operation op, an infinite sequence of operations W, and a (finite or infinite) sequence of operations R, such that for every integer $n \geq 0$ there exists an integer $m \geq 1$, such that for any sequence A from $W(n+1) \circ (R(m) + op?)$ and any sequence B from $W(n) \circ op \circ (R(m) + W_{n+1}?)$ at least one operation in $R(m)$ has different responses in A and B, where $\circ$ is a concatenation of sequences.*

It is shown in [44] that the implementations of exact order types require helping if they use only read, write, and compare&swap primitives. The paper also sketches the proof of a more general result for implementations that, additionally, use fetch&add. Further, it is claimed in [44] that stack and queue are exact order types. Indeed, at first glance, if you swap two subsequent operations, further operations have to acknowledge this difference. However, the definition of an exact order type is slightly more complicated, as it allows not only to swap operations but also move them. This relaxation does not affect queue, but, unfortunately, it affects stack.

Theorem 7.3.1. *Stack is not an exact order type.*

Proof. We prove that for any fixed op, W, R and n there does not exist m that satisfies Definition 7.3.3. Note that the claim is stronger than what is needed to prove the theorem: it would be sufficient to prove that for all op, W and R, the condition does not hold *for some* n. In a sense, this suggests that stack is *far* from being exact order.

Suppose, by contradiction, that there exists m that satisfies Definition 7.3.3 for fixed op, W, R and n. There are four cases for op and W_{n+1}: pop-pop, push-pop, pop-push or push-push. For each of these cases, we find two sequences from $W(n+1) \circ (R(m) + op?)$ and $W(n) \circ op \circ (R(m) + W_{n+1}?)$ for which all operations in $R(m)$ return the same responses.

- $op = \texttt{pop}$, $W_{n+1} = \texttt{pop}$. Then, as two sequences we can choose $W(n+1) \circ op \circ R(m)$ and $W(n) \circ op \circ W_{n+1} \circ R(m)$. In both of them the operations in R return the same responses, since $W_{n+1} \circ op$ and $op \circ W_{n+1}$ perform two pop operations.

- $op = \texttt{push}(a)$, $W_{n+1} = \texttt{pop}$. For the first sequence we take $A = W(n+1) \circ op \circ R(m)$. Now, we choose the second sequence B from $W(n) \circ op \circ (R(m)+W_{n+1}?)$. Let W_{n+1} pop in A the x-th element from the bottom of the stack. We extend $W(n) \circ op$ in B with operations from $R(m)$ until some operation op' tries to pop the x-th element from the bottom. Note that all operations $R(m)$ up to op' (not including op') return the same results in A and B. If such op' does not exist then we are done. Otherwise, we insert W_{n+1} right before op', i.e., pop this element. Subsequent operations in $R(m)$ are not affected, i.e., results of operations in $R(m)$ are the same in A and B.

- $op = \texttt{pop}$, $W_{n+1} = \texttt{push}(b)$. This case is symmetric to the previous one.

- $op = \texttt{push}(a)$, $W_{n+1} = \texttt{push}(b)$. For the first sequence, we take $A = W(n+1) \circ op \circ R(m)$. Now, we build the second sequence B from $W(n) \circ op \circ (R(m) + W_{n+1}?)$. Let W_{n+1} push in A the x-th element from the bottom of the stack. Let us perform $W(n) \circ op$ in B and start performing operations from $R(m)$ until some operation op' pops the x-th element, otherwise a contradiction is established). Note that all operations $R(m)$ up to op' (including op') return the same results in A and B. If such op' does not exist then we are done. Otherwise, right after op' we perform W_{n+1}, i.e., push the element b in its proper position. Subsequent operations in $R(m)$ are not affected and, thus, the results of all operations in $R(m)$ are the same in A and B.

The contradiction implies that stack is not an exact order type. □

7.4 Wait-Free Stack Cannot Be Help-Free

In this section, we prove that there does not exist a help-free wait-free implementation of stack in a system with reads, writes, and compare&swaps. We then extend the proof to the case when a system has one more primitive fetch&add.

7.4.1 Help-Free Stacks Using Reads, Writes and Compare&Swap

Suppose that there exists such a help-free stack implementation Q that uses read, write, and compare&swap primitives. All further lemmas and theorems assume this implementation. We establish a contradiction by presenting a history h in which some operation takes infinitely many steps without completing. All further lemmas and theorems assume the

We start with two observations that immediately follow from the definition of linearizability.

Observation 7.4.1. *In any history h:*

1. *Once an operation is completed it must be decided before all operations that have not yet started;*

2. *If an operation is not started it cannot be decided before any operation of a different process.*

Lemma 7.4.1 (Transitivity)**.** *For any linearization function f and finite history h, if an operation op_2 is completed in h, an operation op_1 is decided before op_2 in h and op_2 is decided before an operation op_3 in h then op_1 is decided before op_3 in h.*

```
1  h ← ϵ
2  op1 ← push(1)
3  id2 ← 2
4  while true:                  // outer loop
5    op2 ← push(id2)
6    while true:                // inner loop
7      if op1 is not decided before op2 in h ∘ p1:
8        h ← h ∘ p1
9        continue
10     if op2 is not decided before op1 in h ∘ p2:
11       h ← h ∘ p2
12       continue
13     break
14   h ← h ∘ p2
15   h ← h ∘ p1
16   while op2 is not completed:
17     h ← h ∘ p2
18   id2 ← id2 + 1
```

Figure 7.1: Constructing the history for the proof of Theorem 7.4.1

Proof. Suppose that op_1 is not decided before op_3 in h then there exists a extension s of h for which $op_3 \prec_{f(s)} op_1$. Since op_2 is linearized in $f(s)$ and op_1 is decided before op_2 then $op_1 \prec_{f(s)} op_2$. Together, $op_3 \prec_{f(s)} op_1 \prec_{f(s)} op_2$ contradicting our assumption that op_2 is decided before op_3 in h. □

Lemma 7.4.2. *For any linearization function f and finite history h, if an operation op_1 of a process p_1 is decided before an operation op_2 of a process p_2, then op_1 must be decided before any operation op that has not started in h.*

Proof. Consider h', the extension of h, in which p_2 runs solo until op_2 completes. Such an extension exists, as our considered implementation Q is wait-free. By Observation 7.4.1 (1), op_2 is decided before op in h', and, consequently, by Transitivity Lemma 7.4.1, op_1 is decided before op in h'.

Since in h', only p_2 takes steps starting from h, op_1 must be decided before op in h — otherwise, h' has a prefix h'' such that op_1 is not decided before op in h'' and op_1 is decided before op in $h'' \circ p_2$ — a contradiction with the assumption that Q is help-free. □

Now we build an *infinite* history h in which p_1 executes infinitely many failed compare&swap steps, yet it never completes its operation. We assume that p_1, p_2 and p_3 are assigned the following programs: p_1 tries to perform $op_1 = \texttt{push}(1)$; p_2 applies an infinite sequence of operations $\texttt{push}(2), \texttt{push}(3), \texttt{push}(4), \ldots$; and p_3 is about to perform an infinite sequence of pop() operations.

The algorithm for constructing this "contradiction" history is given in Figure 7.1. Initially, p_1 invokes $op_1 = \texttt{push}(1)$ and, concurrently, p_2 invokes $op_2 = \texttt{push}(2)$. Then we interleave steps of p_1 and p_2 until a *critical* history h is located: op_1 is decided before op_2 in $h \circ p_1$ and op_2 is decided before op_1 in $h \circ p_2$. We let p_2 and p_1 take the next step and, then, run op_2 after $h \circ p_2 \circ p_1$ until it completes. We will show that op_1 cannot complete and that we can reiterate the construction by allowing p_2 to invoke concurrent operations push(3), push(4), etc. In the resulting infinite history, p_1 takes infinitely many steps without completing op_1.

To ensure that at each iteration op_1 is not completed, we show that, at the start of each iteration of the outer loop (Line 6), the constructed history satisfies the following two invariants:

- op_1 is not decided before op_2 or before any operation of p_3;

- the operations of p_2 prior to op_2 are decided before op_1.

At the first iteration, the invariants trivially hold, since neither op_1 nor op_2 is started.

Observation 7.4.2. *The order between op_1 and op_2 cannot be decided during (and right after) the inner loop (Lines 6-13).*

Lemma 7.4.3. *During (and right after) the execution of the inner loop (Lines 6-13) op_1 and op_2 cannot be decided before any operation of p_3.*

Proof. Suppose that during an execution of the inner loop op_1 or op_2 is decided before some operation of p_3.

Before entering the inner loop, neither op_1 nor op_2 is decided before any operation of p_3: op_1 is not decided because of the first invariant that holds at the beginning of this iteration, while op_2 is not started (Observation 7.4.1 (2)). Thus, at least one step is performed by p_1 or p_2 during the execution of the inner loop.

Let us execute the inner loop until the first point in time when op_1 or op_2 is decided before an operation of p_3. Let this history be h. Note, that because Q is help-free only one of op_1 and op_2 is decided before an operation of p_3 in h. Suppose, that op_1 is decided before some op_3 of p_3, while op_2 is not decided before any operation of p_3. (The case when op_2 is decided before some op_3 is symmetric)

Now p_3 runs pop operations until it completes operation op_3 and then, further, until the first pop operation returns $\perp$, i.e., the stack becomes empty. Let the resulting extension of h be h'.

Recall that op_2 is not decided before any operation of p_3 in h and, since Q is help-free and only p_3 takes steps after h, op_2 cannot be decided before any operation of p_3 in h'. Hence, none of the completed operations of p_3 can return id_2, the argument of op_2, due to the fact that all push operations have different arguments. Since the operations of p_3 empty the stack and they cannot pop id_2, op_2 has to linearize after them, making op_3 be decided before op_2 in h'. By Transitivity Lemma 7.4.1, op_1 is decided before op_2 in h'. Finally, since Q is help-free and only p_3 takes steps after h op_1 has to be decided before op_2 in h, contradicting Observation 7.4.2. □

Lemma 7.4.4. *op_1 and op_2 cannot be completed after the inner loop (Lines 6-13).*

Proof. Suppose the contrary. By Observation 7.4.1 (1), op_1 has to be decided before all operations of p_3, contradicting Lemma 7.4.3. □

Lemma 7.4.5. *The execution of the inner loop (Lines 6-13) is finite.*

Proof. Suppose that the execution is infinite. By Lemma 7.4.4, neither of op_1 and op_2 is completed in h. Thus, in our infinite execution either op_1 or op_2 takes infinite number of steps, contradicting wait-freedom of Q. □

Lemma 7.4.6. *Just before Line 14 the following holds:*

1. *The next primitive step by p_1 and p_2 is to the same memory location.*
2. *The next primitive step by p_1 and p_2 is a compare&swap.*
3. *The expected value of the compare&swap steps of p_1 and p_2 is the value that appears in the designated address.*
4. *The new values of the compare&swap steps of p_1 and p_2 are different from the expected value.*

Proof. Suppose that the next primitive steps by p_1 and p_2 are to different locations. Consider two histories: $h' = h \circ p_1 \circ p_2 \circ \text{complete } op_1 \circ \text{complete } op_2$ and $h'' = h \circ p_2 \circ p_1 \circ \text{complete } op_1 \circ \text{complete } op_2$. Let us look at the first two pop() operations by p_3. Executed after h' they have to return id_2 then 1, since op_1 is decided before op_2 in h' and both of them are completed. While executed after h'' they have to return 1 then id_2. But the local states of p_3 and shared memory states after h' and h'' are identical and, thus, two pops of p_3 must return the same values — a contradiction. The same argument will apply when both steps by p_1 and p_2 are reads.

Suppose that the next operation of p_1 is a write. (The case when the next operation of p_2 is write is symmetric) Consider two histories: $h' = h \circ p_2 \circ p_1 \circ \text{complete } op_1$ and $h'' = h \circ p_1 \circ \text{complete } op_1$. Let the process p_1 perform two pop() operations (op_1' and op_1'') and p_2 complete its operation after h': op_1' and op_1'' have to return 1 and id_2, correspondingly, since op_1 and op_2 are completed and op_2 is decided before op_1 in h'. Again, since the local states of p_1 and the shared memory states after h' and h'' are identical, op_1' and op_1'' performed by p_1 after h'' must return 1 and id_2. Hence, op_2 has to be decided before op_1'' in $\tilde{h} = h'' \circ \text{perform } op_1' \circ \text{perform } op_1''$ and, by Lemma 7.4.2, op_2 has to be decided before any operation of p_3 in $\tilde{h}$. Since only p_1 performs steps after h in $\tilde{h}$ and Q is help-free, op_2 has to be decided before any operation of p_3 at h, contradicting Lemma 7.4.3. Thus, both primitives have to be compare&swap.

By the same argument both compare&swap steps by p_1 and p_2 have the expected value that is equal to the current value in the designated memory location, and the new value is different from the expected. If it does not hold, either the local states of p_1 and the shared memory states after $h \circ p_1$ and $h \circ p_2 \circ p_1$ are identical or the local state of p_2 and the shared memory states after $h \circ p_2$ and $h \circ p_1 \circ p_2$ are identical. □

Observation 7.4.3. *The primitive step of p_2 in Line 14 is a successful compare&swap, and the primitive step of p_1 in Line 15 is a failed compare&swap.*

Observation 7.4.4. *Immediately after Line 14 op_2 is decided before op_1.*

Lemma 7.4.7. *Immediately after Line 15 the order between op_1 and any operation of p_3 is not decided.*

Proof. By Lemma 7.4.3, the order between op_1 and any operation of p_3 is not decided before Line 14. Since Q is help-free, the steps by p_2 cannot fix the order between op_1 and any operation of p_3. Thus, the only step that can fix the order of op_1 and some operation of p_3 is a step by p_1 at Line 15, i.e., a failed compare&swap.

Suppose that op_1 is decided before some operation op_3' of p_3 after Line 15. Let h be the history right before Line 14. Consider two histories $h' = h \circ p_2 \circ p_1$ and $h'' = h \circ p_2$. Let p_3 solo run pop operations after h' until it completes operation op_3' and then, further, until pop operation returns $\perp$, i.e., the stack is empty. Since op_1 is decided before op_3', some completed operation op_3'' of p_3 has to return 1: if we now complete op_1 it should be linearized before op_3'. Now let p_3 to perform after h'' the same number of operations as it did after h'. Since the local states of p_3 and the shared memory states after h' and h'' are identical (p_1 makes the failed compare&swap), op_3'' after h'' has to return 1 as after h'. Thus, op_1 is decided before op_3'' in h''. Since Q is help-free and p_1 does not take steps after h in h'', op_1 has to be decided before op_3'' before Line 14, contradicting Lemma 7.4.3. □

Lemma 7.4.8. *At the end of the outer loop (Line 18) the order between op_1 and next $op_2 = \texttt{push}(id_2 + 1)$ is not yet decided.*

Proof. The operation op_2 is not started, thus, it cannot be decided before op_1 by Observation 7.4.1 (2).

Suppose that op_1 is decided before op_2. By Lemma 7.4.2 op_1 has to be decided before all operations of p_3, contradicting Lemma 7.4.7. □

```
1  h ← ϵ
2  for i in 1..2:
3    op_i ← push(i)
4  id_3 ← 3
5  while true:                  // outer loop
6    op_3 ← push(id_3)
7    while true:                // inner loop
8      moved ← False
9      for i in 1..3:
10       if op_i is not decided before any op_j in h ∘ p_i:
11         h ← h ∘ p_i
12         moved ← True
13     if not moved:
14       break
15
16   h ← h ∘ p_3
17   // let p_k be the process whose next primitive is compare&swap
18   h ← h ∘ p_k
19   while op_3 is not completed:
20     h ← h ∘ p_3
21   id_3 ← id_3 + 1
```

Figure 7.2: Constructing the history for the proof of Theorem 7.4.2

Thus, after this iteration of the loop the two invariants hold (Observation 7.4.4 and Lemmas 7.4.7 and 7.4.8), and p_1 took at least one primitive step.

This way we build a history in which p_1 takes infinitely many steps, but op_1 is never completed. This contradicts the assumption that Q is wait-free.

Theorem 7.4.1. *In a system with at least three processes and primitives read, write and compare&swap there does not exist a wait-free and help-free* stack *implementation.*

7.4.2 Adding Fetch&Add

Now suppose that the implementation is allowed to additionally use fetch&add primitives. We prove that there is no wait-free and help-free stack implementation in a system with at least *four* processes.

Again, by contradiction, suppose that such an implementation Q exists. We build an infinite history h in which either p_1 or p_2 executes infinitely many failed compare&swap steps, yet it never completes its operation, contradicting wait-freedom. In h, processes p_1, p_2, p_3 and p_4 follow the following programs: for $1 \leq i \leq 2$, p_i tries to perform $op_i = \texttt{push}(i)$; p_3 applies an infinite sequence of operations $\texttt{push}(3), \texttt{push}(4), \texttt{push}(5), \ldots$; and p_4 is about to perform an infinite sequence of pop() operations. The algorithm for constructing this "contradiction" history is given in Figure 7.2.

Similar to the proof of Theorem 7.4.1, we show that the following three invariants hold at the beginning of each iteration of the outer loop (Line 6):

- the order between any two operations among op_1, op_2 and op_3 is not decided;
- op_1 and op_2 are not decided before any operation of p_4;
- all the operations of p_3 prior to op_3 are decided before op_1 and op_2.

At the beginning of the first iteration, the invariants hold trivially, since none of op_i is started.

Observation 7.4.5. *The order between op_i and op_j for $1 \leq i \neq j \leq 3$ cannot be decided during (and right after) the inner loop (Lines 7-14).*

Proof. From the first invariant, op_i cannot be decided before op_j prior to the inner loop (Lines 7-14). Since Q is help-free, during the inner loop op_i can become decided before op_j only after a step by p_i which is impossible due to the check in Line 10. □

Lemma 7.4.9. *During (and right after) an execution of the inner loop (Lines 7-14) op_1, op_2 and op_3 cannot be decided before any operation of p_4.*

Proof. Suppose that during an execution of the inner loop op_1, op_2 or op_3 is decided before some operation of p_4.

At the beginning of the loop, none of op_1, op_2 and op_3 is decided before any operation of p_4: op_1 and op_2 are not decided because of the second invariant, while op_3 is not yet started. Suppose that during the execution of the inner loop some op_i becomes decided before some operations of p_4.

Let us look at the execution and find the first point in time when some op_k of p_k is decided before some operation op_4 of p_4. Using the same argument as in the proof of Lemma 7.4.3, we can show that op_k has to be decided before any other op_j contradicting Observation 7.4.5: we let p_4 run until the operation op_4 is completed and, further, while stack is not empty; op_4 becomes decided before op_j; by Transitivity Lemma 7.4.1, op_k is decided before op_j. □

The proofs of the following two lemmas are identical to those of Lemmas 7.4.4 and 7.4.5.

Lemma 7.4.10. *For each i, $1 \leq i \leq 3$, op_i cannot be completed after the inner loop (Lines 7-14).*

Lemma 7.4.11. *The execution of the inner loop (Lines 7-14) is finite.*

Lemma 7.4.12. *For all i, j, $1 \leq i \neq j \leq 3$, op_i is decided before op_j in $h \circ p_i$.*

Proof. Consider an operation of process i. At the end of the inner loop op_i should be decided before some op_k in $h \circ p_i$, otherwise, p_i can make at least one more step during the inner loop. Thus, by Lemma 7.4.2 op_i should be decided before op_4, the first operation of p_4. Let p_4 run pop operations after $h \circ p_i$ until one of them returns $\perp$, i.e., the stack is empty. Let this history be h'.

By Lemma 7.4.9, op_j is not decided before any operation of p_4 in h. Since Q is help-free and only p_i and p_4 takes steps in h' after h, op_j cannot be decided before any operation of p_4 in h', and, consequently, operations of p_4 cannot pop an argument of op_j. Since the operations of p_4 empty the stack, op_j must be linearized after them. Thus, op_4 is decided before op_j in h'. By Transitivity Lemma 7.4.1, op_i is decided before op_j in h'. Finally, since Q is help-free and only p_4 takes steps in h' after $h \circ p_i$, op_i is decided before op_j in $h \circ p_i$. □

Lemma 7.4.13. *Immediately before Line 16 the following holds:*

1. *The next primitive step by p_i for $1 \leq i \leq 3$ is to the same memory location.*
2. *The next primitive step by p_i for $1 \leq i \leq 3$ is fetch&add with a non-zero argument or compare&swap for which the expected value is the value that appears in the designated location and the new value is different from the expected one.*

Proof. Suppose that for some pair p_i and p_j the next steps are to different memory locations. We consider two histories $h' = h \circ p_i \circ p_j \circ$ complete $op_i \circ$ complete op_j and $h'' = h \circ p_j \circ p_i \circ$ complete $op_i \circ$ complete op_j. By Lemma 7.4.12, after h', the two subsequent pop operations by p_4 should return first the argument of op_j and then the argument of op_i, while after h'' they should return the two values in the opposite order. This is impossible, since the local states of p_4 and the shared memory states after h' and h'' are identical. The same argument will apply if the next steps of some pair of processes are read primitives.

Suppose that the next primitive step of some p_i is a write. We take any other process p_j and build two histories: $h' = h \circ p_j \circ p_i \circ$ complete op_i and $h'' = h \circ p_i \circ$ complete op_i. As in the proof of Lemma 7.4.6, p_i performs two pop() operations (op_i' and op_i'') and p_j completes its operation after h': by Lemmas 7.4.1 and 7.4.12, op_i' and op_i'' have to return the argument of op_i and the argument of op_j, respectively. The local states of p_i and the shared memory states after h' and h'' are identical, thus, op_i' and op_i'' after h'' should also return the arguments of op_i and op_j. Hence, op_j has to be decided before op_i'' in $\tilde{h} = h'' \circ$ perform $op_i' \circ$ perform op_i''. By Lemma 7.4.2, op_j is decided before any operation of p_4 in $\tilde{h}$. And, finally, since Q is help-free and p_j does not take steps in $\tilde{h}$ after h, op_j has to be decided before any operation of p_4 in h, contradicting Lemma 7.4.9.

A similar argument applies to the case when the next primitive step of some p_i is fetch&add with argument zero, or compare&swap which expected value differs from the value in the designated location or the new value is equal to the expected. We take any other process p_j ($1 \leq j \leq 3$) and build two histories $h' = h \circ p_i \circ p_j \circ$ complete p_j and $h'' = h \circ p_j \circ$ complete p_j. The proof for the previous case applies except that now the roles of p_i and p_j are swapped. □

Lemma 7.4.14. *At most one out of p_1 and p_2 can have fetch&add as their next primitive step.*

Proof. Suppose that p_1 and p_2 have fetch&add as their next primitive step. Consider two histories $h' = h \circ p_1 \circ p_2$ and $h'' = h \circ p_2 \circ p_1$. From Lemma 7.4.12 op_1 is decided before op_2 in h', thus, by Lemma 7.4.2, op_1 is decided before the first operation op_4 of p_4. After h' p_4 performs k' pop operations until one of them returns $\perp$, i.e., the stack is empty. One pop has to return 1, because if we now complete op_1 it has to be linearized before op_4. The same with h'': p_4 performs k'' pops until one of them returns $\perp$, and one of these pop's return 2. Since the local states of p_4 and the shared memory states after h' and h'' are the same: two pop operations $pop_1()$ and $pop_2()$ of $k'(= k'')$ operations of p_4 after h' and h'' return 1 and 2.

Now we show that op_1 and op_2 are decided before op_3 in h'. The same can be shown for h''. Consider a history $\tilde{h}$: h' continued with k' pop operations by p_4. By Lemma 7.4.12, op_1 is decided before op_3 in h'. From Lemma 7.4.9 and two facts that Q is help-free and op_3 does not make any steps after h in $\tilde{h}$, it follows that op_3 cannot be decided before any operation of p_4 in $\tilde{h}$ and, consequently, the operations of p_4 cannot pop an argument of op_3. Since k' pops of op_4 empty the stack, op_3 has to linearize after them, making operation $pop_2()$ to be decided before op_3. Since $pop_2()$ returns 2, it has to be decided after op_2. By Transitivity Lemma 7.4.1, op_2 is decided before op_3 in $\tilde{h}$. Q is help-free and only p_4 takes steps after h', thus, op_2 is decided before op_3 in h'.

Now consider two histories $h' \circ$ complete op_3 and $h'' \circ$ complete op_3. In both of these histories, op_1 and op_2 are decided before op_3. After the first history let p_4 perform three pop operations and p_1 and p_2 complete push(1) and push(2): the three pops return id_3, 2 and 1, respectively. Analogously, after the second history three pop return id_3, 1 and 2. This is impossible, since the local states of p_4 and the memory states after these two histories are identical. □

Observation 7.4.6. *From the previous lemma we know that the next primitive step of at least one process p_1 or p_2 is compare&swap. Let it be process p_k. By algorithm, p_3 takes a step at Line 16 changing the memory location either by fetch&add or by a successful compare&swap, thus, the next step of p_k at Line 18 should be a failed compare&swap.*

Observation 7.4.7. *Immediately after Line 16, op_3 is decided before op_1 and op_2.*

Lemma 7.4.15. *Immediately after Line 18, op_1 and op_2 are not decided before any operation of p_4.*

Proof. We prove the claim for op_1, the case of op_2 is similar.

If p_2 took a step at Line 18, then, by Lemma 7.4.9 and the fact that the steps by p_2 or p_3 cannot fix the order between op_1 and any operation of p_4 due to help-freedom, op_1 is not decided before any operation of p_4.

If p_1 took a step at Line 18, then, by Lemma 7.4.9 and the fact that the steps by p_3 cannot fix the order between op_1 and any operation of p_4 due to help-freedom, the only step that could fix the order is a step by p_1 at Line 18, i.e., a failed compare&swap. Suppose that op_1 is decided before some op_4' of p_4 after Line 18. We consider two histories $h' = h \circ p_3 \circ p_1$ and $h'' = h \circ p_3$. Let p_4 run solo after h' until it completes op_4', and then further until some pop returns $\perp$, i.e., the stack becomes empty. Since op_1 is decided before op_4', some completed operation op_4'' of p_4 has to return 1: if we now complete op_1 it has to be linearized before op_4'. Now, let p_4 to run the same number of pop operations after h''. Since the local states of p_4 and the shared memory states after h' and h'' are identical, op_4'' returns 1. Thus, op_1 is decided before op_4'' in h''. As Q is help-free and p_1 does not take steps after h in h'', op_1 has to be decided before op_4'' in h, contradicting Lemma 7.4.9. □

Lemma 7.4.16. *At the end of the outer loop (Line 21), the order between any two operations among op_1, op_2 and the next $op_3 = \texttt{push}(id_3 + 1)$ is not yet decided.*

Proof. The operation op_3 is not yet started, thus, it cannot be decided before op_i, $i = 1, 2$, by Observation 7.4.1 (2).

Suppose that op_i, $i = 1, 2$, is decided before op_j, then by Lemma 7.4.2 op_i has to be decided before all operations of p_4, contradicting Lemma 7.4.15. □

We started with three invariants that hold before any iteration of the loop. By Observation 7.4.7 and Lemmas 7.4.15 and 7.4.16, the invariants hold after the iteration, and at least one of p_1 and p_2 made at least one primitive step.

This way we build a history in which one of op_1 and op_2 never completes its operation, even though it takes infinitely many steps. This contradicts the assumption that Q is wait-free.

Theorem 7.4.2. *In a system with at least four processes and primitives read, write, compare&swap and fetch&add, there does not exist a wait-free and help-free* stack *implementation.*

7.5 Universal Construction with Move&Increment

At first, we define atomic *move&increment* primitive. It takes two pointers to integers `to_move` and `to_increment`. The operation copies a value from `to_increment` to `to_move` and increments the value in `to_increment`.

This primitive is obviously more powerful than fetch&add, but it seems to be less complex than atomic double compare&swap: a primitive that takes a pair of arbitrary memory locations, a pair of expected values and a pair of new values, and sets new values in the provided memory locations only if the current values in the memory locations are equal to the provided expected values.

Theorem 7.5.1. *In a system with n processes and primitives read, write and move&increment any data type has a help-free wait-free implementation provided a sequential implementation.*

Proof. As for the proof we simply provide such an implementation in Figure 7.3.

In the algorithm, we use an infinite array of operations `operations`, an identifier array `id_by_p` of size n and an operation array `op_by_p` of size n, where n is the number of processes. The algorithm builds a total order of operations in `operations` array.

```
1  operations[1...]
2  op_by_p[1..n]
3  id_by_p[1..n]
4  id ← 1
5
6  apply(pid, op): // process pid applies operation op
7    op_by_p[pid] ← op
8    move&increment(&id_by_p[pid], &id)
9
10   for j in 1..n:
11     op_id_j ← id_by_p[j]
12     op_j ← op_by_p[j]
13
14     if op_id_j ≤ id_by_p[pid] and
15         operations[op_id_j] = null:
16       operations[op_id_j] ← op_j
17
18   o ← new Object()
19   for j in 1..id_by_p[pid]:
20     o.apply(operations[j])
21
22   return o.apply(op)
```

Figure 7.3: The universal help-free wait-free construction using move&increment

Process `pid` tries to apply an operation `op` by calling `apply(pid, op)`. At first, the process registers his operation in `op_by_p[id]` array (Line 7). Second, it gets an identifier `id_by_p` of its operation in the sequence of all operations, i.e., the array `operations` (Line 8). Then the process helps to copy all the operations into the sequence `operations` that should come prior to `op` together with `op`. For that it iterates through all processes and for each process `j` (p_j) it reads the current identifier (Line 11) and operation (Line 12). Then it verifies whether the operation by $p_j$was not yet applied (Lines 14 and 15): the operation is `op` or should come prior to `op` and the corresponding slot in `operations` is still empty. Finally, it applies the all operations in the order of array `operations` to new sequential object.

Each `c = apply(pid, op)` call has a unique identifier `id(c)` calculated in Line 7. To prove the correctness of the algorithm it is enough to show that: 1) `op` from a call `c = apply(pid, op)` can only appear at `operations[id(c)]`, and 2) before Line 18 of call `c` all cells of `operations` up to `id(c)` are non-null.

We prove this by the identifier of the call k.

Base $k = 0$. No calls are performed and everything is correct.

Consider a call `c = apply(pid, op)` with identifier k. We know that for all calls prior to k the two statements are correct.

Instead of proving the two statements we prove for each p_j separately: all operations of p_j from the calls with identifiers up to k are copied to proper positions in `operations` array before Line 18 of call `c`.

For p_j, $p_{\texttt{pid}}$ reads a current identifier `op_id_j` in Line 11 and a current operation `op_j` in Line 12. Suppose that `op_j` was written to `op_by_p[j]` during a call `apply(j, op')` with identifier $\texttt{uid}_{\texttt{op}}$ and `op_id_j` was written to `id_by_p[j]` during a call `apply(j, op")` with identifier $\texttt{uid}_{\texttt{id}}$.

We note that $\texttt{uid}_{\texttt{id}}$ is always less than or equal to $\texttt{uid}_{\texttt{op}}$, since we read identifier and operation in the opposite order we write them.

Let `lid` be the identifier of the last call by p_j with the identifier less than or equal to k. Note that $\texttt{uid}_{\texttt{id}}$ and $\texttt{uid}_{\texttt{op}}$ cannot be less than `lid`. This is due to the fact that $\texttt{uid}_{\texttt{op}}$ was already set to `lid` before move&increment primitive by `pid` during c which happened

prior to the read of op_id_j.

There are two cases. If $\texttt{uid}_{\texttt{op}}$ is bigger than lid then the call with identifier lid is already completed when op_j is read. This means by induction that all the operations by p_j prior to k are already copied into their proper positions.

Otherwise, $\texttt{uid}_{\texttt{op}}$ is equal to lid. Thus, since $\texttt{uid}_{\texttt{id}} \leq \texttt{uid}_{\texttt{op}}$, we get that $\texttt{uid}_{\texttt{id}} = \texttt{uid}_{\texttt{op}} = \texttt{lid}$. Thus, the write at Line 16 by $\texttt{p}_{\texttt{pid}}$ can copy operation only into proper position and it is always copied if the proper position is empty. Note that this is the only situation when the operation is copied into operations array and it is copied only into the proper position.

Since the operations by p_j from calls before $\texttt{uid}_{\texttt{id}}$ are successfully copied by induction, and the operation from the last call by p_j with identifier less than ort equal to k is copied, after Line 16 all operations by p_j with identifier less than or equal to k are copied.

Note that the presented implementation is obviously wait-free. Also, it is help-free since we can linearize any operation in Line 8. Thus, the operation is linearized by its operation and does not require the linearization-based helping. □

7.6 Related Work

Helping is often observed in wait-free (e.g., [64, 68, 131, 151]) and lock-free implementations (e.g., [19, 97, 117, 120]): operations of a slow or crashed process may be finished by other processes. Typically, to benefit from helping, an operation should register a *descriptor* (either in a dedicated "announce" array or attached in the data items) that can be used by concurrent processes to help completing it.

We are aware of three alternative definitions of helping: (1) *linearization-based* by Censor-Hillel et al. [44] considered here, (2) *valency-based* by Attiya et al. [23] and (3) *universal* by Attiya et al. [23].

Valency-based helping [23] captures helping through the values returned by the operations, which makes it quite restrictive. In particular, for stack, the definition cannot capture helping relation between two push operations. They distinguish *trivial* and *non-trivial* helping: for non-trivial helping, the operation that is being helped should return a data-structure-specific *non-trivial* (e.g., non-empty for stacks and queues) value. It is shown in [23] that any wait-free implementation of queue has non-trivial helping, while there exists a wait-free implementation of stack without non-trivial helping. This is an interesting result, given notorious attempts of showing that queue is in Common2 [11], i.e., that they can be implemented using reads, writes and primitives with consensus number 2, while stack has been shown to be in Common2 [12].

Attiya et al. [23] also introduced a very strong notion of helping — *universal helping* — which essentially boils down to requiring that every invoked operation eventually takes effect. This property is typically satisfied in universal constructions parameterized with object types. But most algorithms that involve helping in a more conventional (weaker) sense do not meet it, which makes the use of universal helping very limited.

Linearization-based helping [44] considered here is based on the order between two operations in a possible linearization. Compared to valency-based definitions, this notion of helping operates on the linearization order and, thus, can be applied to all operations, not only to those that return (non-trivial) values. By relating "helping" to fixing positions in the linearization, this definiton appears to be more intuitive: one process helps another make a "progress", i.e., linearize earlier. Censor-Hillel et al. [44] also introduced two classes of data types: exact order types (queue as an example) and global view types (snapshot and counter as examples). They showed that no wait-free implementation of data types from these two classes can be help-free. By assuming stack to be exact order, they deduced that this kind of helping is required for wait-free stack implementations. In this chapter, we showed that stack is in fact not an exact order type, and give a direct proof of their claim.

7.7 Conclusion

In this chapter, we gave a direct proof that any wait-free implementation of stack in a system with read, write, comare&swap and fetch&add primitives is subject to linearization-based helping. This corrects a mistake in the indirect proof via exact order types in [44].

Let us come back to the original intuition of *helping* as a process performing work on behalf of other processes. One may say that linearization-based helping introduced by Censor-Hillel et al. and used here does not adequately capture this intuition. For example, by examining the wait-free stack implementation by Afek et al. [12], we find out that none of the processes *explicitly* performs work for the others: to perform pop() a process goes down the stack from the current top until it reaches some value or the bottom of the stack; while to perform push(x) a process simply increments the top of the stack and deposits x there. But we just showed that any wait-free stack implementation has linearization-based helping, and indeed this algorithm has it. So, we might think that valency-based helping is superior to linearization-based one, since the algorithm by Afek et al. does not have *non-trivial* valency-based helping. Nevertheless, the aforementioned algorithm has *trivial* valency-based helping, and, thus, the (quite unnatural) distinction between trivial and non-trivial helping seems to be chosen specifically to allow the algorithm by Afek et al. to be help-free.

A very interesting challenge is therefore to find a definition of linearization-based helping that would naturally reflect help-freedom of the algorithm by Afek et al., while queue does not have a wait-free and help-free implementation.

8 Performance Prediction for Coarse-Grained Programs

8.1 Introduction

A standard design pattern found in many concurrent data structures, such as hash tables or ordered containers, is an alternation of parallelizable sections that incur no data conflicts and critical sections that must run sequentially and are protected with locks. A lock can be viewed as a *queue* that arbitrates the order in which the critical sections are executed, and a natural question is whether we can use *theoretical analysis* to predict the resulting throughput. As a preliminary evidence to the affirmative, we describe a simple model that can be used to predict the throughput of *coarse-grained* lock-based algorithms. We show that our model works well for CLH lock, and we expect it to work for other popular lock designs such as TTAS, MCS, etc.

Roadmap

In Section 8.2, we describe the problem in more details. In Section 8.3, we list assumptions on the abstract machine on which data structures are executed. In Section 8.4, we obtain a formula for the throughput. In Section 8.5, we verify our theoretical result through experimental analysis. We conclude in Section 8.6.

8.2 Abstract Coarse-Grained Synchronization

Conventionally, the performance of a concurrent data structure is evaluated via experiments, and it is notoriously difficult to account for all significant experimental parameters so that the outcomes are meaningful. Our motivation here is to complement experimental evaluation with an analytical model that can be used to *predict* the performance rather than measure it. As a first step towards this goal, we attempt to predict the throughput of a class of algorithms that use *coarse-grained* synchronization.

Consider a concurrent system with N processes that obey the following simple *uniform* scheduler: at every time step, each process performs a step of computation. This scheduler, resembling the well-known PRAM model [104], appears to be a reasonable approximation of a real-life concurrent system. Suppose that the processes share a data structure exporting a single `operation()`. If the operation induces a work of size P and incurs no synchronization, the resulting throughput is $N \cdot \alpha/P$ operations in a unit of time: each process performs α/P operations in a unit of time, where α indicates the amount of work that is performed by one process in a unit of time. One way to evaluate the constant α experimentally is to count the total number F of operations, each of work P, completed by N processes in time T. Then we get $\alpha = F/NP$. The longer is T, the more accurate is the estimation of α.

Now suppose that, additionally, the operation performed by each process contains a *critical section* of size C. In the operation, described in Figure 8.1, every process takes a global lock, performs the critical section of size C, releases the lock and, finally, performs the *parallel section* of size P.

Here, as a unit of work, we take the number of CPU cycles spent during one iteration of the loop in Lines 3-4 or 6-7. The iteration consists of a `nop` instruction, an increment

```
1 operation():
2   lock.lock()
3   for i in 1..C:
4     nop
5   lock.unlock()
6   for i in 1..P:
7     nop
```

Figure 8.1: The coarse-grained operation

of a local variable and a conditional jump, giving us, approximately, *four* CPU cycles in total.

8.3 Model Assumptions

Below we list basic assumptions on the abstract machine used for our analytical throughput prediction.

First, we assume that coherence of caches is maintained by a variant of MESI protocol [128]. Each cache line can be in one of four states: `Modified` (`M`), `Exclusive` (`E`), `Shared` (`S`) and `Invalid` (`I`). MESI regulates transitions between states of a cache line and responses depending on the request (read or write) to the cache line by a process or on the request to the memory bus. The important transitions for us are: (1) upon reading, the state of the cache line does not change if it was not `I`, otherwise, the state becomes `S`, and, if the state was `I`, then a *read request* is sent to the bus; (2) upon writing, the state of the cache line becomes `M`, and, if the state was `S` or `I`, an *invalidation request* is sent to the bus.

We assume that the caches are *symmetric*: for each MESI state st, there exist two constants R_{st} and W_{st} such that any read from any cache line with status st takes R_{st} work and any write to a cache line with status st takes W_{st} work. Tudor et al. [53] showed that for an Intel Xeon machine (similar to the one we use in our experimental validation below), given the relative location of a cache line with respect to the process (whether they are located on the same socket or not), the following hypotheses hold: (1) writes induce the same work, regardless of the state of the cache line; (2) `swaps`, not concurrent with other `swaps`, induce the same work as writes. Therefore, we assume that (1) $W = W_M = W_E = W_S = W_I$ and (2) any contention-free `swap` induces a work of size W.

8.4 CLH Lock

Multiple lock implementations have been previously proposed, from simple spinlocks and TTAS to more advanced MCS [116] and CLH [51]. For our analysis, we choose CLH, as the simplest lock among those considered to be efficient. In Figure 8.2, we inline `lock` and `unlock` calls to CLH lock in our abstract coarse-grained operation.

8.4.1 Cost of an Operation

Let us zoom into what happens during the execution of the operation.

Note that at the beginning of an operation (unless it is the very first invocation), `my_node.locked` is loaded into the cache and the corresponding cache line is in state `M`, because of the set in Line 15 during the previous operation by the same process.

1. The operation starts with `swap` (Line 9) that induces a work of size W, if not concurrent with other `swaps`, and a work of size at most X, otherwise.

```
1  class Node:
2    bool locked
3
4  global Node head ← new Node()   // global
5  Node my_node                    // per process
6     my_node.locked ← true
7
8  operation():
9    Node next ← swap(&head, my_node)  // W or X
10   while (next.locked) {}            // R_I or 2 · R_I
11   for i in 1..C:                    // C
12     nop
13   my_node.locked ← false            // W
14   my_node ← next
15   my_node.locked ← true             // W
16   for i in 1..P:                    // P
17     nop
```

Figure 8.2: The coarse-grained operation with inlined `lock` and `unlock` functions

2. In Line 10, the algorithm loops on a field `next.locked`. During this loop one or two cache misses happens.

   One cache miss can happen at the first iteration of the loop if the read of `locked` returns `true`. The last process that grabbed the lock already invalidated this cache line in Line 15 during its penultimate operation. MESI reloads the cache line and changes its state from `I` (or none if it was not loaded previously) to `S`.

   The other cache miss happens in every execution when the operation reads `next.locked` and gets `false`. In this case, the cache line was invalidated in Line 13 during the last operation of the last process that grabbed the lock. MESI reloads the cache line and changes its state from `I` (or none) to `S`.

   Each of the described cache misses induces the work of size R_I. Thus, the work induced in Line 10 is of size R_I (if only the second miss happens) or $2 \cdot R_I$ (if both misses happen).

3. In Lines 11-12, the critical section with work of size C is performed.

4. In Line 13, `my_node.locked` is set to `false`. There are two cases: if `my_node.locked` is not yet loaded by any other process in Line 10 then the state remains `M`; otherwise, MESI changes the state from `S` to `M` and sends a signal to invalidate this cache line. In both cases, the induced work is of size W.

5. In Line 14, the operation performs an assignment on local variables, without contributing to the total work.

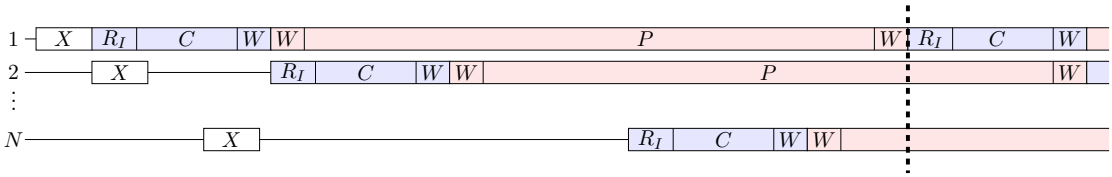
6. In Line 15, `my_node.locked` is set to `true`. From the end of the while loop at Line 10 the corresponding cache line is in state `S`. MESI changes the state to `M` and sends a signal to invalidate this cache line inducing work of size W.

7. In Lines 16-17, the parallel work of size P is performed.

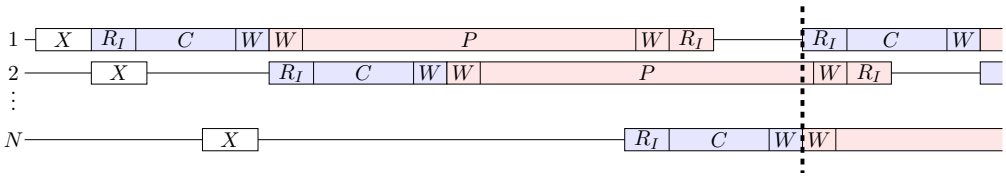
8.4.2 Evaluating Throughput

To evaluate the throughput of the resulting program under the uniform scheduler, take a closer look on how N processes continuously perform the operation from Figure 8.2.

Process 1 executes: its first swap (taking at most X units); the critical section (blue, Lines 10-13): acknowledges the ownership of the lock by reading `false` in Line 10 (takes



(a) Case 1: $W + P + W \geq (N-1) \cdot (R_I + C + W)$. Each process enters the critical section without waiting in the queue.



(b) Case 2: $W + P + W \leq (N-1) \cdot (R_I + C + W)$. Each process waits in the queue before entering the critical section.

Figure 8.3: Examples of executions of the coarse-grained algorithm from Figure 8.2. Blue intervals depict critical sections and red intervals depict parallel sections.

R_I units), performs the work of size C and releases the lock in Line 13 (takes W units); the parallel section (red, Lines 15-17 and 9): sets `my_node.locked` to `true` (takes W), performs the work of size P, performs a non-contended swap (takes W) and, possibly, reads `true` in Line 10 (takes R_I). (Here, the swap operation performed after the very first completed critical section is counted in the parallel work, as it is executed in the absence of contention.) Every other process i operates in the same way: it swaps as early as possible (taking at most X), waits until process $i-1$ releases the lock, and then performs its critical (blue) and parallel (red) sections.

Depending on the parameters N, C, P, W, and R_I, two types of executions are possible.

In Case 1 (Figure 8.3a), at the moment when process 1 finishes its parallel section, process N already finished its critical section, i.e., $P + 2 \cdot W > (N-1) \cdot (C + R_I + W)$. Therefore, in the steady case, at every moment of time, each process does not wait and executes either the parallel or critical section, and the read in Line 10 cannot return `true` because the lock is already released. Thus, the throughput, measured as the number of operations completed in a unit of time, equals to $N \cdot \frac{\alpha}{(P+2 \cdot W)+(C+R_I+W)}$.

In Case 2 (Figure 8.3b), before proceeding to the next operation, process 1 has to wait until process N completes its critical section from the previous round of operations; process 2 waits for process 1, process 3 waits for process 2, etc. Thus, there is always some process in the critical section, giving the throughput of $\frac{\alpha}{C+R_I+W}$.

Therefore, given the number of processes N, the sizes C and P of critical and parallel sections, the throughput can be calculated as follows:

$$\begin{cases} \frac{\alpha}{C+R_I+W} & \text{if } P + 2 \cdot W \leq (N-1) \cdot (C + R_I + W) \\ \frac{\alpha \cdot N}{(P+2 \cdot W)+(C+R_I+W))} & \text{otherwise} \end{cases}$$

8.5 Experiments

For our measurements, we used a server with four 10-core Intel Xeon E7-4870 chips of 2.4 GHz (yielding 40 hardware processes in total), running Ubuntu Linux kernel v3.13.0-66-generic. We compiled the code with MinGW GCC 5.2.0 (with -O0 flag to avoid compiler optimizations, such as function inlining, that can screw up our benchmarking environment). The code is available at
https://github.com/Aksenov239/complexity-lock-with-libslock.

We considered the following experimental settings: the number of processes is 39; the size of the critical section $C \in \{100, 500, 5000\}$; and the multiplier $x \in [1, 150]$ (we choose

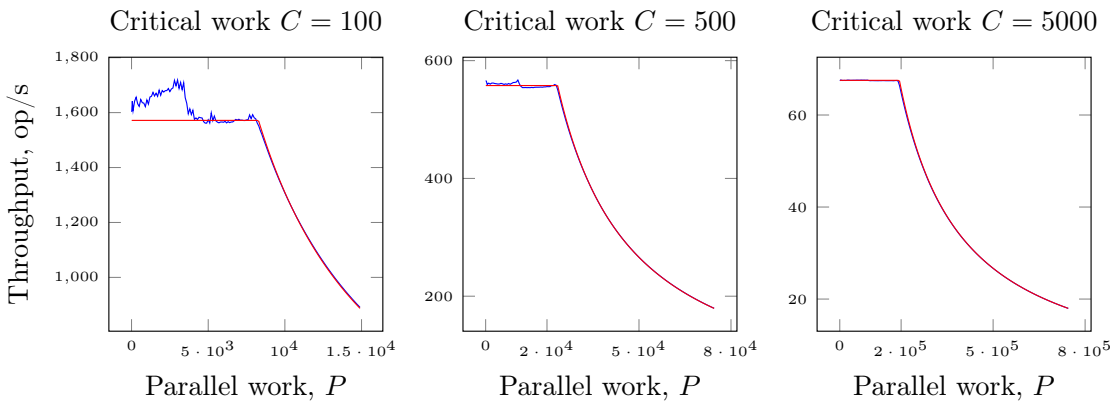


Figure 8.4: Throughput on 39 processes for $C \in \{100, 500, 5000\}$

all integer values) that determined the size of the parallel section $P = x \cdot C$. For each setting, we measured the throughput for 10 seconds. Our experimental evaluation gives $\alpha \approx 3.5 \cdot 10^5$, $W \approx 40$, and $R_I \approx 80$. The ratio between W and R_I correlates with the experimental results provided by Tudor et al. [53].

In Figure 8.4, we show our experimental results (blue curves) compared with our theoretical prediction (red curves). The two curves match very closely, except for the case of small C and P where our predicted throughput underestimates the real one. We relate this to the fact that we oversimplified the abstract machine: any write induces the work of constant size W, regardless of the relative location of the cache line with respect to the process. For small C and P, two processes from the same socket are more likely to take the lock one after the other and, thus, on average, a write might induce less work than W, and, consequently, the throughput can be higher than predicted.

8.6 Conclusion

In this short chapter, we showed that a simple theoretical analysis may quite accurately predict the throughput of data structures implemented using coarse-grained synchronization. For the moment, our analysis is restricted to algorithms using CLH-based locking in systems obeying the uniform scheduler. As a future work, we intend to extend the analysis to more realistic algorithm designs, lock implementations and architectures.

9 Conclusion and Future Work

In this thesis, we study how to balance synchronization and parallelism in a range of concurrent programs. We presented an automatic granularity control for parallel programs and two novel ways (concurrency-optimality and parallel combining) to design concurrent data structures with potentially low synchronization overhead. Then we discussed synchronization in a form of helping in wait-free and lock-free data structures. Finally, we proposed a way to predict the throughput of simple coarse-grained concurrent data structure.

Each of these contributions provides us with open questions for future research. In this chapter, we briefly recall these questions.

Automatic Granularity Control

In the current proposal, the programmer has to provide an asymptotic cost function for each sp-guard separately. These functions can be non-trivial and the programmer might not be able to provide them. However, he often can provide the sizes of the data on which the code is going to work. An interesting question is whether we can design a similar granularity control technique that uses only the information about sizes?

The intuition tells that we can find the appropriate cost functions using some machine learning algorithm and, after that, we can use our original algorithm. At first, new algorithm produces data points for each sp-guard by measuring the execution time: these points map a size of the data to the execution time. When the set of sample points for the sp-guard becomes representative enough, the algorithm fits the cost function using some machine learning algorithm, e.g., symbolic regression by genertic programming [105].

Another open question is what to do when complexity functions are not helpful, e.g., the execution time of the code with the same cost function fluctuates a lot. In this case, our predictions can be useless and the performance can degrade.

As an example, consider the simplest algorithm that compares two strings, symbol by symbol. For this algorithm, we expect the cost function to depend on the lengths of the strings, but not on the data itself (otherwise, the cost function is not lightweight). Thus, for the following two inputs the cost is the same while the execution time differs much: (1) if one string consists of n symbols `'a'` and the other consists of n symbols `'b'` then the execution time is constant; (2) if both strings consist of n symbols `'a'` then the execution time depends on n.

As a second example, we consider a maximal matching algorithm from the PBBS suite [142]: given a graph it finds a set of edges such that no pair of edges share a vertex and any other edge shares a vertex with some edge from the set. The algorithm uses the compare&swap primitive [31]. Because the pattern of the compare&swap invocations is not fixed, the contention during the execution of the same code and with same complexity can differ much, which results in the fluctuation of the execution time. We did not present the results of the execution of this algorithm in Chapter 4 since our approach did not perform well on some inputs. Sometimes our version worked up to 30% worse than the PBBS one. Is it possible to improve our granularity control algorithm in a way that it could handle such behaviour?

Concurrency-Optimal Binary Search Tree

In Chapter 5, we provided a concurrency-optimal implementation of a binary search tree. This is the second known concurrency-optimal implementation after linked-list based set [77]. Do there exist other data structures that can benefit from concurrency-optimality? We expect that skip-list should be a good candidate because it can be seen as a composition of linked lists.

Right now the concurrency-optimality proposes to measure "concurrency" of algorithms in *sets* of schedules. The immediate question is whether it is possible to express this metric quantitatively, so that it would be easier to compare "concurrency" of implementations? For example, as a new integer metric we can propose the number of *dihomotopy* classes of executions [63]. Intuitively, two executions are equivalent (or *dihomotopic*) if one can be obtained from the other by permuting independent instructions. Is it the reasonable metric?

Finally, we can imagine a design principle similar to concurrency-optimality which intends to maximize the set of exported *executions*, i.e., interleavings of steps of a concurrent algorithm, instead of schedules. Below we explain this idea on the example.

Consider two concurrent implementations of partially-external binary search trees. In the first one, there is one lock *per node* and an insertion of a new node is performed as follows: take a lock on the parent and set the proper child to the newly created node. In the second one, there is a read-write lock per node and a lock per each *link* to the children, and an insertion of a new node is performed as follows: take a read lock on the parent, take a lock on a link to the proper child and set a proper child to the newly created node.

It is obvious that the second implementation is more fine-grained and it allows more executions than the first one: two new nodes can be added as a left and a right child almost concurrently. This separation of one lock per node into two locks on the links was proven to provide better performance in practice [120].

Can we provide a formal guide for this approach: specify the ways to increase the number of executions and specify the granularity, i.e., the number of executions, after which the performance starts to degrade?

Parallel Combining

In Chapter 6 we provide three application of the parallel combining technique. However, for two of them, binary search trees and priority queues, there already existed efficient fine-grained concurrent implementations. An immediate question is: which new concurrent data structures can we create from their parallel batched counterparts? For sure, we can build a concurrent dynamic tree since its parallel batched version is known [6]: it exports the same operations as a dynamic graph but must remain a tree (forest) at any moment of time. Also, we expect that it should be possible to design a parallel batched dynamic graph from a parallel batched dynamic tree and, consequently, it should be possible to design a concurrent dynamic graph that is more efficient than its coarse-grained implementations.

Right now, to use parallel combining, we have to provide a code of a parallel batched data structure in a form of `COMBINER_CODE` and `CLIENT_CODE`. It is less convenient than using the original code, typically written for dynamic multithreading, without changes. It is known that, the algorithms for dynamic multithreading are typically executed using some scheduler, for example, *work-stealing* [115]. Thus, the follow-up question is: can we implement efficient scheduler that uses only the processes from a specified subset, i.e., *clients*? This question is not entirely trivial, since up-to-date schedulers use all processes, and not the specific subset.

In Chapter 6 we consider only the linearizability correctness criterion. We present a new weaker notion of correctness: for each operation op there exist a linearizable set of operations S that contains: (1) op, (2) all operations finished before the invocation of op,

and, (3) possibly, some other operations.

As an example, we build a simple concurrent implementation of a *persistent* binary search tree that satisfies our criterion while the linearized implementation is unknown and we expect it to be non-trivial. A *persistent* data structure [59] is a data structure that creates a new version on an update and provides an access to all the previous versions.

We get our concurrent persistent binary search tree using parallel combining. Suppose that before the current combining phase the tree is T. During the combining phase the algorithm works in two phases: (1) we apply each operation to T separately in parallel using standard sequential persistent algorithm, i.e., as a resulting version an operation gets a tree T with this operation applied; (2) we apply all operations together to T using the parallel batched persistent algorithm similar to the algorithm by Blelloch et al. [32]. This implementation satisfies our criterion: for each operation *op*, the set S contains *op* and all operations that were applied in preceding combining phases — the version of the tree returned by *op* is a tree that satisfies some sequential application of operations from S.

It would be interesting to investigate this correctness criterion further and find other data structures that can benefit from it, i.e., the implementation with new criteria is more efficient than the linearizable implementation.

On Helping and Stacks

In Chapter 7, we only considered helping in wait-free implementations. Thus, an immediate question is whether it is possible to consider helping for *lock-free* algorithms. Can we identify data types that do not have a lock-free help-free implementation?

As shown in this thesis, *linearization-based* helping does not distinguish stack and queue: none of these data structures has a wait-free help-free implementation. But examining the wait-free stack by Afek et al. [12], we find out that none of the processes *explicitly* performs work for the others: to perform `pop()` a process goes down the stack from the current top until it reaches some value or the bottom of the stack; while to perform `push(`x`)` a process simply increments the top of the stack and deposits x there. Thus, this algorithm does not have helping in the simplest intuitive form. However, there is no paradox, since linearization-based helping is still present in this algorithm.

Therefore, we might think that *valency-based* helping [23] is more suitable. The algorithm by Afek et al. does not have *non-trivial* (valency-based) helping. Nevertheless, valency-based helping has two issues: (1) the aforementioned algorithm by Afek et al. does have *trivial* helping, i.e., one `pop` can make another `pop` to return $\perp$, and the quite unnatural distinction between trivial and non-trivial helping seems to be chosen specifically to allow this algorithm to be help-free; (2) the definition does not capture helping in operations that do not return non-trivial responses, for example, no operation helps `push` operation on stack by default.

Thus, it is appealing to find a variant of linearization-based helping that would capture help-freedom of the algorithm by Afek et al.

Performance Prediction of Coarse-Grained Programs

To perform our simplistic theoretical analysis in Chapter 8 we posed several constraints and it would interesting to get rid of them.

At first, can we replace CLH Lock by other popular lock designs such as TTAS, MCS, etc.?

Second, we assumed that the architecture has symmetric cache and read and write costs are independent on the status of a cache line. Can we introduce more details and make the architecture more realistic?

Finally, all considered programs have only one lock, one critical and one parallel sections of fixed size. It would be interesting to investigate throughput of more complex programs, for example, programs with several locks or with the parallel section of variable size.

10 Bibliography

[1] http://snap.stanford.edu/. 65

[2] Umut A Acar and Guy E Blelloch. *Algorithm Design: Theory and Practice*. 2017. 20, 49

[3] Umut A Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *ACM SIGPLAN Notices*, volume 48, pages 219–228. ACM, 2013. 68

[4] Umut A Acar, Arthur Charguéraud, and Mike Rainey. A work-efficient algorithm for parallel unordered depth-first search. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pages 1–12. IEEE, 2015. 64

[5] Umut A Acar, Arthur Charguéraud, and Mike Rainey. Oracle-guided scheduling for controlling granularity in implicitly parallel languages. *Journal of Functional Programming*, 26, 2016. 38

[6] Umut A Acar, Vitaly Aksenov, and Sam Westrick. Brief announcement: Parallel dynamic tree contraction via self-adjusting computation. In *Proceedings of the twenty-ninth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 275–277. ACM, 2017. 16, 116, 138

[7] Umut A Acar, Vitaly Aksenov, and Sam Westrick. Brief-announsement: Parallel dynamic tree contraction via self-adjusting computation. In *Proceedings of the twenty-ninth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 275–277. ACM, 2017.

[8] Umut A Acar, Vitaly Aksenov, Arthur Charguéraud, and Mike Rainey. Performance challenges in modular parallel programs. In *Proceedings of the twenty third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 381–382. ACM, 2018.

[9] G Adelson-Velsky and E Landis. An algorithm for the organization of information. 145:263–266, 1962. 27, 99

[10] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *computer*, 29(12):66–76, 1996. 19

[11] Yehuda Afek, Eytan Weisberger, and Hanan Weisman. A completeness theorem for a class of synchronization objects. In *Proceedings of the twelfth annual ACM symposium on Principles of distributed computing*, pages 159–170. ACM, 1993. 129

[12] Yehuda Afek, Eli Gafni, and Adam Morrison. Common2 extended to stacks and unbounded concurrency. *Distributed Computing*, 20(4):239–252, 2007. 117, 129, 130, 139

[13] Kunal Agrawal, Jeremy T Fineman, Kefu Lu, Brendan Sheridan, Jim Sukha, and Robert Utterback. Provably good scheduling for parallel programs that use data structures through implicit batching. In *Proceedings of the twenty-sixth annual ACM symposium on Parallelism in algorithms and architectures*, pages 84–95. ACM, 2014. 16, 95, 115, 116

[14] Vitaly Aksenov, Petr Kuznetsov, and Anatoly Shalyto. On helping and stacks. In *Proceedings of NETYS 2018*.

[15] Vitaly Aksenov, Vincent Gramoli, Petr Kuznetsov, Anna Malova, and Srivatsan Ravi. A concurrency-optimal binary search tree. In *European Conference on Parallel Processing (Euro-Par)*, pages 580–593. Springer, 2017.

[16] Vitaly Aksenov, Dan Alistarh, and Petr Kuznetsov. Brief-announcement: Performance prediction of coarse-grained programs. In *Proceedings of the thirty seventh annual ACM Symposium on Principles of distributed computing (PODC)*, pages 411–413, 2018.

[17] Vitaly Aksenov, Petr Kuznetsov, and Anatoly Shalyto. Parallel combining: Benefits of explicit synchronization. *arXiv preprint arXiv:1710.07588*, 2018.

[18] James H Anderson, Yong-Jik Kim, and Ted Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed computing*, 16(2-3):75–110, 2003. 25

[19] Maya Arbel-Raviv and Trevor Brown. Reuse, don't recycle: Transforming lock-free algorithms that throw away descriptors. In *31 International Symposium on Distributed Computing*, volume 91, pages 4:1–4:16, 2017. 129

[20] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001. 21

[21] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks. *Journal of the ACM (JACM)*, 41(5):1020–1048, 1994. 23

[22] Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004. 73

[23] Hagit Attiya, Armando Castañeda, and Danny Hendler. Nontrivial and universal helping for wait-free queues and stacks. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 46, 2016. 117, 129, 139

[24] Greg Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 261–270. ACM, 1993. 28

[25] Lars Bergstrom, Mike Rainey, John Reppy, Adam Shaw, and Matthew Fluet. Lazy tree splitting. In *ACM Sigplan Notices*, volume 45, pages 93–104. ACM, 2010. 38, 72

[26] Guy E Blelloch. Scans as primitive parallel operations. *IEEE Transactions on computers*, 38(11):1526–1538, 1989. 19

[27] Guy E Blelloch. Nesl: A nested data-parallel language. 1991. 21, 37

[28] Guy E Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996. 21

[29] Guy E Blelloch and Margaret Reid-Miller. Fast set operations using treaps. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, pages 16–26. ACM, 1998. 28

[30] Guy E Blelloch, Gary L Miller, Jonathan C Hardwick, and Dafna Talmor. Design and implementation of a practical parallel delaunay algorithm. *Algorithmica*, 24 (3-4):243–269, 1999. 64

[31] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *ACM SIGPLAN Notices*, volume 47, pages 181–192. ACM, 2012. 137

[32] Guy E Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 253–264. ACM, 2016. 16, 28, 95, 96, 98, 99, 114, 139

[33] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999. 21

[34] OpenMP Architecture Review Board. Openmp application interface. http://www.openmp.org, 2008. 12, 21, 37

[35] Luc Bougé, Joaquim Gabarro, Xavier Messeguer, Nicolas Schabanel, et al. Height-relaxed avl rebalancing: A unified, fine-grained approach to concurrent dictionaries. *Research Report 1998-18, LIP, ENS Lyon*, 1998. 28

[36] Anastasia Braginsky, Nachshon Cohen, and Erez Petrank. Cbpq: High performance lock-free priority queue. In *European Conference on Parallel Processing*, pages 460–474. Springer, 2016. 36, 114

[37] Richard P Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM (JACM)*, 21(2):201–206, 1974. 21

[38] Gerth Stølting Brodal, Jesper Larsson Träff, and Christos D Zaroliagis. A parallel priority queue with constant time operations. *Journal of Parallel and Distributed Computing*, 49(1):4–21, 1998. 98

[39] Nathan G Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *ACM Sigplan Notices*, volume 45, pages 257–268. ACM, 2010. 15, 16, 17, 28, 74, 81, 91, 113

[40] Trevor Brown, Faith Ellen, and Eric Ruppert. Pragmatic primitives for non-blocking data structures. In *Proceedings of the 2013 ACM symposium on Principles of distributed computing*, pages 13–22. ACM, 2013. 30

[41] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *ACM SIGPLAN Notices*, volume 49, pages 329–342. ACM, 2014. 16, 17, 30

[42] Zoran Budimlić, Vincent Cavé, Raghavan Raman, Jun Shirako, Sağnak Taşırlar, Jisheng Zhao, and Vivek Sarkar. The design and implementation of the habanero-java parallel programming language. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications*, pages 185–186. ACM, 2011. 21, 37

[43] Irina Calciu, Hammurabi Mendes, and Maurice Herlihy. The adaptive priority queue with elimination and combining. In *International Symposium on Distributed Computing*, pages 406–420. Springer, 2014. 36

[44] Keren Censor-Hillel, Erez Petrank, and Shahar Timnat. Help! In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 241–250. ACM, 2015. 13, 17, 117, 118, 119, 129, 130

[45] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, pages 442–446. SIAM, 2004. 64

[46] Manuel MT Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18. ACM, 2007. 37

[47] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. 40(10):519–538, 2005. 21, 37

[48] Vinay K Chaudhri and Vassos Hadzilacos. Safe locking policies for dynamic databases. *Journal of Computer and System Sciences*, 57(3):260–271, 1998. 74

[49] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988. 11, 33

[50] Thomas H Cormen, Charles Eric Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. The MIT press, 3rd edition, 2009. 12, 13, 14, 20, 33

[51] Travis Craig. Building fifo and priorityqueuing spin locks from atomic swap. Technical report, Technical Report TR 93-02-02, University of Washington, 02 1993., 1993. 13, 18, 112, 132

[52] Tyler Crain, Vincent Gramoli, and Michel Raynal. A contention-friendly binary search tree. In *European Conference on Parallel Processing*, pages 229–240. Springer, 2013. 15, 16, 17, 29, 74, 81, 91, 95, 113

[53] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 33–48. ACM, 2013. 132, 135

[54] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1, 2011. 65

[55] SK Debray, Manuel V Hermenegildo, and Pedro López García. A methodology for granularity-based control of parallelism in logic programs. *Journal of symbolic computation*, 21(4-6):715–734, 1996. 38, 69

[56] Narsingh Deo and Sushil Prasad. Parallel heap: An optimal parallel priority queue. *The Journal of Supercomputing*, 6(1):87–98, 1992. 34, 98, 104

[57] Dana Drachsler, Martin Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. *ACM SIGPLAN Notices*, 49(8):343–356, 2014. 15, 16, 17, 29, 74, 81, 91, 95, 113

[58] Dana Drachsler-Cohen and Erez Petrank. Lcd: Local combining on demand. In *International Conference On Principles Of Distributed Systems*, pages 355–371. Springer, 2014. 16, 95, 96, 115

[59] James R Driscoll, Neil Sarnak, Daniel D Sleator, and Robert E Tarjan. Making data structures persistent. *Journal of computer and system sciences*, 38(1):86–124, 1989. 139

[60] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. An adaptive cut-off for task parallelism. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11. IEEE, 2008. 69

[61] David Eisenstat. Two-enqueuer queue in common2. *arXiv preprint arXiv:0805.0444*, 2008. 117

[62] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 131–140. ACM, 2010. 15, 16, 28, 29, 74, 81, 91

[63] Lisbeth Fajstrup, Eric Goubault, Emmanuel Haucourt, Samuel Mimram, and Martin Raussen. *Directed algebraic topology and concurrency*. Springer, 2016. 138

[64] Panagiota Fatourou and Nikolaos D Kallimanis. A highly-efficient wait-free universal construction. In *Proceedings of the twenty-third annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 325–334. ACM, 2011. 115, 117, 129

[65] Panagiota Fatourou and Nikolaos D Kallimanis. Revisiting the combining synchronization technique. In *ACM SIGPLAN Notices*, volume 47, pages 257–266. ACM, 2012. 95, 96, 115

[66] Marc Feeley. A message passing implementation of lazy task creation. In *US/Japan Workshop on Parallel Symbolic Computing*, pages 94–107. Springer, 1992. 38, 72

[67] Marc Feeley. Polling efficiently on stock hardware. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 179–187. ACM, 1993. 72

[68] Steven Feldman, Pierre Laborde, and Damian Dechev. A wait-free multi-word compare-and-swap operation. *International Journal of Parallel Programming*, 43 (4):572–596, 2015. 117, 129

[69] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59. ACM, 2004. 32

[70] Java Fork-Join. https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html. 21, 37

[71] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM, 1978. 19

[72] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. *ACM Sigplan Notices*, 33(5):212–223, 1998. 12, 21, 37, 68

[73] Phillip B Gibbons. A more practical pram model. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168. ACM, 1989. 20

[74] Phillip B Gibbons, Yossi Matias, and Vijaya Ramachandran. The queue-read queue-write pram model: Accounting for contention in parallel algorithms. *SIAM Journal on Computing*, pages 638–648, 1997. 19

[75] Gaston H Gonnet and J Ian Munro. Heaps on heaps. *SIAM Journal on Computing*, 15(4):964–971, 1986. 33, 104

[76] Vincent Gramoli. More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms. In *ACM SIGPLAN Notices*, volume 50, pages 1–10. ACM, 2015. 91

[77] Vincent Gramoli, Petr Kuznetsov, Srivatsan Ravi, and Di Shang. Brief announcement: a concurrency-optimal list-based set. In *International Symposium on Distributed Computing*, page 659, 2005. 93, 138

[78] Vincent Gramoli, Petr Kuznetsov, and Srivatsan Ravi. From sequential to concurrent: correctness and relative efficiency (brief announcement). In *Principles of Distributed Computing (PODC)*, pages 241–242, 2012. 93

[79] Vincent Gramoli, Petr Kuznetsov, and Srivatsan Ravi. In the search for optimal concurrency. In *Structural Information and Communication Complexity - 23rd International Colloquium, SIROCCO*, pages 143–158, 2016. 15, 73, 74, 78, 80, 93

[80] Rachid Guerraoui, Thomas A Henzinger, and Vasu Singh. Permissiveness in transactional memories. In *International Symposium on Distributed Computing*, pages 305–319. Springer, 2008. 74

[81] Leo J Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Foundations of Computer Science, 1978., 19th Annual Symposium on*, pages 8–21. IEEE, 1978. 27, 99

[82] Rajiv Gupta and Charles R Hill. A scalable implementation of barrier synchronization using an adaptive combining tree. *International Journal of Parallel Programming*, 18(3):161–180, 1989. 115

[83] Robert H Halstead Jr. Implementation of multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 9–17. ACM, 1984. 68

[84] Timothy L Harris. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing*, pages 300–314. Springer, 2001. 16, 95

[85] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *International Conference On Principles Of Distributed Systems*, pages 3–16. Springer, 2005. 16, 74, 95

[86] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 355–364, 2010. 16, 95, 96, 112, 114, 115

[87] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Scalable flat-combining based synchronous queues. In *DISC*, pages 79–93. Springer, 2010. 115

[88] M Herlihy, Y Lev, and N Shavit. A lock-free concurrent skiplist with wait-free search. *Unpublished Manuscript, Sun Microsystems Laboratories, Burlington, Massachusetts*, 2007. 32, 35

[89] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991. 73, 74, 78, 117

[90] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2011. 17, 35, 114

[91] Maurice Herlihy and Nir Shavit. On the nature of progress. In *International Conference On Principles Of Distributed Systems*, pages 313–328. Springer, 2011. 12, 74

[92] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A simple optimistic skiplist algorithm. In *International Colloquium on Structural Information and Communication Complexity*, pages 124–138. Springer, 2007. 32, 35

[93] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990. 11, 23, 73

[94] Tasuku Hiraishi, Masahiro Yasugi, Seiji Umatani, and Taiichi Yuasa. Backtracking-based load balancing. In *ACM Sigplan Notices*, volume 44, pages 55–64. ACM, 2009. 38, 68

[95] Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)*, 48(4):723–760, 2001. 16, 26, 95, 98, 112

[96] Brandon Holt, Jacob Nelson, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Flat combining synchronized global data structures. In *7th International Conference on PGAS Programming Models*, page 76, 2013. 115

[97] Shane V Howley and Jeremy Jones. A non-blocking internal binary search tree. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 161–171. ACM, 2012. 16, 29, 129

[98] Lorenz Huelsbergen, James R Larus, and Alexander Aiken. Using the run-time sizes of data structures to guide parallel-thread creation. In *ACM SIGPLAN Lisp Pointers*, volume 7, pages 79–90. ACM, 1994. 38, 69

[99] Galen C Hunt, Maged M Michael, Srinivasan Parthasarathy, and Michael L Scott. An efficient algorithm for concurrent priority queue heaps. *Information Processing Letters*, 60(3):151–157, 1996. 34

[100] Intel. Intel threading building blocks. https://www.threadingbuildingblocks.org, 2011. 13, 14, 21, 37, 38, 40

[101] Intel. Intel cilk plus manual. https://software.intel.com/en-us/node/684236/, 2017. 37, 39

[102] Shintaro Iwasaki and Kenjiro Taura. A static cut-off for task parallel programs. In *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 139–150. IEEE, 2016. 72

[103] Suresh Jagannathan, Armand Navabi, KC Sivaramakrishnan, and Lukasz Ziarek. The design rationale for multi-mlton. In *ML'10: Proceedings of the ACM SIGPLAN Workshop on ML*, 2010. 37

[104] Joseph JáJá. *An introduction to parallel algorithms*, volume 17. Addison-Wesley Reading, 1992. 15, 19, 131

[105] John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994. 137

[106] Clyde P Kruskal. Searching, merging, and sorting in parallel computation. *IEEE Transactions on Computers*, (10):942–946, 1983. 33

[107] Vivek Kumar, Daniel Frampton, Stephen M Blackburn, David Grove, and Olivier Tardieu. Work-stealing without the baggage. In *ACM SIGPLAN Notices*, volume 47, pages 297–314. ACM, 2012. 37, 68

[108] Petr Kuznetsov and Srivatsan Ravi. On the cost of concurrency in transactional memory. In *International Conference On Principles Of Distributed Systems*, pages 112–127. Springer, 2011. 74

[109] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th international conference on World wide web*, pages 591–600. ACM, 2010. 65

[110] Charles E Leiserson and Ilya B Mirman. How to survive the multicore software revolution (or at least survive the hype). *Cilk Arts*, 1, 2008. 11

[111] Zongpeng Li. Non-blocking implementations of queues in asynchronous distributed shared-memory systems. *Masters thesis, University of Toronto*, 2001. 117

[112] Task Parallel Library. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl. 21, 37

[113] Jonatan Lindén and Bengt Jonsson. A skiplist-based concurrent priority queue with minimal memory contention. In *International Conference On Principles Of Distributed Systems*, pages 206–220. Springer, 2013. 17, 35, 114

[114] Yujie Liu and Michael Spear. Mounds: Array-based concurrent priority queues. In *41st International Conference on Parallel Processing*, pages 1–10. IEEE, 2012. 36

[115] Michael D McCool, Arch D Robison, and James Reinders. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012. 14, 37, 138

[116] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991. 115, 132

[117] Maged M Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82. ACM, 2002. 129

[118] Eric Mohr, David A Kranz, and Robert H Halstead. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE transactions on Parallel and Distributed Systems*, 2(3):264–280, 1991. 72

[119] Gordon E Moore. Cramming more components onto integrated circuits. *Electronic Magazine*, pages 82–85, 1965. 11

[120] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *ACM SIGPLAN Notices*, volume 49, pages 317–328. ACM, 2014. 16, 29, 81, 91, 129, 138

[121] Aravind Natarajan, Lee H Savoie, and Neeraj Mittal. Concurrent wait-free red black trees. In *Symposium on Self-Stabilizing Systems*, pages 45–60. Springer, 2013. 16, 29

[122] Gil Neiger. Set-linearizability. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, page 396. ACM, 1994. 23

[123] Jürg Nievergelt and Edward M Reingold. Binary search trees of bounded balance. *SIAM journal on Computing*, 2(1):33–43, 1973. 27, 99

[124] The nineth dimacs implementation challenge. http://www.dis.uniroma1.it/challenge9/, 2013. 65

[125] Otto Nurmi and Eljas Soisalon-Soininen. Chromatic binary search trees. *Acta informatica*, 33(5):547–557, 1996. 30

[126] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of the International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, volume 16, 1999. 95, 96, 115

[127] Christos H Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, 26(4):631–653, 1979. 73

[128] Mark S Papamarcos and Janak H Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ACM SIGARCH Computer Architecture News*, volume 12, pages 348–354. ACM, 1984. 132

[129] Wolfgang Paul, Uzi Vishkin, and Hubert Wagener. Parallel dictionaries on 2–3 trees. In *International Colloquium on Automata, Languages, and Programming*, pages 597–609. Springer, 1983. 16

[130] Joseph D Pehoushek and Joseph S Weening. Low-cost process creation and dynamic partitioning in qlisp. In *US/Japan Workshop on Parallel Lisp*, pages 182–199. Springer, 1989. 38, 68

[131] Yaqiong Peng and Zhiyu Hao. Fa-stack: A fast array-based stack with wait-free progress guarantee. *IEEE Transactions on Parallel and Distributed Systems*, 2017. 117, 129

[132] Maria Cristina Pinotti and Geppino Pucci. Parallel priority queues. *Information Processing Letters*, 40(1):33–40, 1991. 33, 98, 104

[133] Henry Crozier Plummer. On the problem of distribution in globular star clusters. *Monthly notices of the royal astronomical society*, 71:460–470, 1911. 64

[134] William Pugh. Concurrent maintenance of skip lists. *Technical Report CS-TR-2222.1, University of Maryland*, 1990. 31, 35

[135] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990. 30

[136] Daniel Sanchez, Richard M Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. In *ACM Sigplan Notices*, volume 45, pages 311–322. ACM, 2010. 38, 68

[137] Peter Sanders. Randomized priority queues for fast parallel access. *Journal of Parallel and Distributed Computing*, 49(1):86–97, 1998. 16, 34, 98, 104

[138] Raimund Seidel and Cecilia R Aragon. Randomized search trees. *Algorithmica*, 16 (4-5):464–497, 1996. 27, 99

[139] Nir Shavit and Itay Lotan. Skiplist-based concurrent priority queues. In *Proceeding of 14th International Parallel and Distributed Processing Symposium*, pages 263–268. IEEE, 2000. 17, 35, 114

[140] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Transactions on Computer Systems (TOCS)*, 14(4):385–428, 1996. 115

[141] Nir Shavit and Asaph Zemach. Combining funnels: a dynamic approach to software combining. *Journal of Parallel and Distributed Computing*, 60(11):1355–1387, 2000. 115

[142] Julian Shun, Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the problem based benchmark suite. In *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, pages 68–70. ACM, 2012. 15, 39, 41, 62, 64, 137

[143] Bjarne Stroustrup. The c++ programming language, 2013. 39

[144] Håkan Sundell and Philippas Tsigas. Scalable and lock-free concurrent dictionaries. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1438–1445. ACM, 2004. 31, 35

[145] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65(5): 609–627, 2005. 35

[146] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's journal*, 30(3):202–210, 2005. 11

[147] Herb Sutter. Choose concurrency-friendly data structures, June 2008. 74

[148] Olivier Tardieu, Haichuan Wang, and Haibo Lin. A work-stealing scheduler for x10's task parallelism with suspension. *ACM SIGPLAN Notices*, 47(8):267–276, 2012. 68

[149] Robert E Tarjan. Efficient top-down updating of red-black trees. *Technical Report TR-006-85*, 1985. 29

[150] Peter Thoman, Herbert Jordan, and Thomas Fahringer. Adaptive granularity control in task parallel programs using multiversioning. In *European Conference on Parallel Processing*, pages 164–177. Springer, 2013. 69

[151] Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *International Conference On Principles Of Distributed Systems*, pages 330–344. Springer, 2012. 117, 129

[152] Jyh-Jong Tsay and Hsin-Chi Li. Lock-free concurrent tree structures for multiprocessor systems. In *International Conference on Parallel and Distributed Systems*, pages 544–549. IEEE, 1994. 29

[153] Alexandros Tzannes, George C Caragea, Rajeev Barua, and Uzi Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. *ACM Sigplan Notices*, 45(5):179–190, 2010. 38, 72

[154] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990. 19

[155] Joseph S Weening. Parallel execution of lisp programs. Technical report, Stanford University, Department of Computer Science, 1989. 38, 68, 72

[156] Pen-Chung Y, Nian-Feng T, et al. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, 100(4):388–395, 1987. 115