FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Variations on Task Scheduling for Shared Memory Systems

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Martin Wimmer
Registration Number 0007126

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Jesper Larsson Träff

The dissertation has been reviewed by:

(Kunal Agrawal) (Marina Papatriantafilou)

Wien, 28.04.2014

(Martin Wimmer)

Erklärung zur Verfassung der Arbeit

Martin Wimmer
Dr. A. Schärf-Straße 3/1/7, 2353 Guntramsdorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

First and foremost, I want to thank my advisor, Jesper Larsson Träff, without whom this thesis would not have been possible. He was an academic mentor to me, was always there for me when I needed help, but also gave me a lot of freedom to develop independently.

I want to thank Philippas Tsigas and Daniel Cederman for our long and fruitful cooperation and several research visits that helped shape many of the research results in this work.

Countless discussions and conversations with Sascha Hunold and Francesco Versaci helped me gain a deeper understanding of many topics. They also provide for great conversations on non-academic topics, thus making sure I never want to miss out on a lunch-break with them.

My students Manuel Pöter, Jakob Gruber and Martin Kalany helped me by building up in-depth expertise on particular topics related to my work, and improving my understanding of these topics in discussions.

I thank the PEPPHER project, University of Vienna and TU Vienna for providing funding and infrastructure for this work. Apart from giving me a place to work and machines to work on, this enabled valuable research visits, the participation in conferences and summer schools.

In advance, I want to thank the reviewers Kunal Agrawal and Marina Papatriantafilou for taking the time to review this thesis, and to travel to Vienna for my defense. I hope for many interesting and challenging comments.

I also want to thank all other colleagues from the parallel computing research group at TU Vienna: Alexandra Carpen-Amarie, Christine Kamper, Markus Levonyak, Angelos Papatriantafyllou, Margret Steinbuch, Antoine Rougier. Thanks as well to my colleagues from University of Vienna where I spent the first years of my academic career: Enes Bajrovic, Siegfried Benkner, Wilfried Gansterer, Yuriy Kaniovskyi, Martin Köhler, Erich Marth, Eduard Mehofer, Sabri Pllana, Martin Sandrieser, Alexander Wöhrer. I also want to thank the colleagues from Chalmers University, which I visited several times, for making me feel at home there: Yiannis Nikolakopoulos, Thomas Petig and Valentin Tudor

Finally, I want to thank my parents and my sister for their continuous support and for always believing in me, my friends Gerhard, Milena, Nathalie and Patrick for encouraging my academic pursuits, and Armin, Ramona and Simon for always being there for me, for keeping me sane while I was working on this thesis, as well as for the countless evenings that were documented in *the book*.

Abstract

This thesis provides an in-depth discussion of task scheduling for shared memory systems. The topic is approached in a vertical manner, starting with high-level programming model aspects, and moving down to low-level implementation details. On the programming model level existing task-parallel programming models are discussed, as well as programming patterns that can be used on top of task parallel models. This is supplemented by extensions to these models and patterns developed in the context of this work. One such extension is strategy scheduling, which allows for tighter interaction between a task-parallel program and the scheduling system. Strategy scheduling is intended to bridge the gap between specialized scheduling systems, optimized for specific applications, and generic task schedulers.

Theoretically and practically efficient run-time systems are required to support task scheduling. While work-stealing has been proven to be an efficient approach for general task scheduling, more complex techniques are required to support the presented extensions to the task parallel model. The main goal here is to provide good load balancing, while at the same time reducing communication to increase the scalability of applications. Another concern in task scheduling is memory usage. For a large class of task parallel applications efficient greedy schedules exist that will only use the same space per processor as a space-efficient execution on one processor.

Both the run-time system and programming patterns require supporting concurrent data structures. Such data structures have been developed in the context of this work and are discussed in detail. To enable good scalability, the concurrent data structures in this work all provide strong progress guarantees. Most are lock-free, which guarantees that at least one participant will progress in a bounded number of steps. Some data structures presented in this work are wait-free, guaranteeing progress for all participants.

The techniques presented in this work have been used to create a task scheduling framework called Pheet. Pheet was originally designed as a vehicle for evaluation of new scheduling techniques, and therefore has a highly flexible plug-in architecture based on C++ template meta-programming. This allows to replace any single component in the task scheduling system, while keeping the rest of a configuration the same, in order to enable direct (performance) comparison between different implementations of a specific component. Pheet is accompanied by the Pheet benchmark suite containing a variety of task parallel micro benchmarks. The Pheet benchmark suite is used to evaluate the performance of Pheet components. While Pheet was originally developed as a tool for research and teaching on task parallel programming, it has developed into a fully fledged scheduling framework, which has been released as open source software.

Kurzfassung

Diese Arbeit beschäftigt sich mit dem Problem des Task-Scheduling, der Ablaufplanung für parallel ausführbare Aufgaben, auf parallelen Systemen mit gemeinsamen Speicher. Für den Aufbau der Arbeit wurde ein vertikaler Ansatz gewählt, bei dem das Thema auf allen Ebenen diskutiert wird. Begonnen wird auf der Ebene der Programmiermodelle mit einer Beschreibung existierender Modelle, sowie von Entwurfsmustern für task-parallele Programme. Diese Modelle und Entwurfsmuster werden in Folge erweitert. Eine dieser Erweiterungen baut auf sogenannten Strategien auf, einem Konstrukt das eine engere Interaktion zwischen task-parallelen Programmen und dem Task-Scheduler ermöglicht. Der Einsatz von Strategien zielt darauf ab, die Brücke zwischen generischen Task-Schedulern und spezialisierten, auf eine bestimmte Applikation optimierten Task-Schedulern zu schließen.

Task-parallele Programmiermodelle bedingen den Einsatz von Task-Schedulern die sowohl theoretisch als auch praktisch effizient sind. Diese Effizienz wurde für Work-Stealing, einen weit verbreiteten Ansatz zum Task-Scheduling, bewiesen, doch viele der in dieser Arbeit gezeigten Erweiterungen des task-parallelen Programmiermodells benötigen Task-Scheduler die über Work-Stealing hinausgehen. Das Primärziel hierbei ist die Ausführungszeit und den Speicherverbrauch einer task-parallelen Applikation zu beschränken, und dabei die Kommunikation zwischen Prozessoren klein zu halten.

In einem weiteren Teil dieser Arbeit werden Ansätze zur Synchronisation, sowie Datenstrukturen die parallele Zugriffe erlauben, diskutiert. Diese sind für die Implementierung von effizienten Task-Schedulern notwendig, sowie als Unterstützung der in dieser Arbeit diskutierten Entwurfsmuster. Der Fokus dieser Arbeit liegt auf Ansätzen die Fortschrittsgarantien für Synchronisation geben können. Die meisten der vorgestellten Synchronisationsalgorithmen und Datenstrukturen sind *lock-free*, was den Fortschritt zumindest eines Synchronisationsteilnehmers in einer beschränkten Anzahl an Schritten garantiert. Manche der vorgestellten Algorithmen sind auch *wait-free*, und garantieren somit den Fortschritt für alle Teilnehmer.

Die vorgestellten Algorithmen und Techniken wurden genutzt, um eine Programmierbibliothek namens Pheet zu implementieren. Pheet wurde ursprünglich als Plattform entwickelt, die es erlaubt, schnell neue Task-Scheduler und Synchronisationsalgorithmen zu entwickeln und mit anderen zu vergleichen. Um dies zu ermöglichen, setzt Pheet auf einer Plug-In-Architektur auf, die auf C++ Template-Metaprogrammierung basiert. Durch diese Architektur ist es möglich, einen großen Teil der Komponenten von Pheet auszutauschen, ohne die restliche Konfiguration zu verändern. Dies ermöglicht den direkten Vergleich mehrerer Implementierungen einer Komponente. Um diesen Vergleich zu vereinfachen, enthält Pheet eine Reihe von kleinen Benchmark-Applikationen über die verschiedene Implementierungen verglichen werden können. Pheet war ursprünglich als Forschungs- und Unterrichtswerkzeug gedacht, ist aber in Zwischenzeit zu einer vollwertigen Programmierbibliothek zur Entwicklung von task-parallelen Applikationen angewachsen und ist als Quelloffene Software öffentlich verfügbar.

Contents

1 Introduction

Task scheduling has become a standard model for exploiting parallelism, and is especially popular for shared memory systems. It has the advantage of being oblivious of the amount of parallelism available in a system.

The focus of this thesis is to present extensions to the task-parallel programming model, both to make the model suitable for more applications, as well as to simplify the implementation of efficient task-parallel programs. Efficient non-blocking data structures and synchronization primitives were developed to support this.

1.1 Motivation and Inspiration

One topic of interest for this thesis is the task-parallel programming model. What makes the task-parallel programming model so interesting is that programs written in this model are oblivious of the parallelism available in a system. Programmers need to be concerned with understanding the concrete machine their program is running on and do not have to deal with load balancing. Instead, the model exposes all potential parallelism in an algorithm, and has a run-time system dynamically distribute available work to idle processors. Of particular interest to this thesis are the task-parallel programming languages *Cilk* [25], *Cilk++* [97] and *X10* [38]. Many ideas found in this work are based on, or inspired by work done on these programming languages.

Our work on schedulers was originally inspired by work-stealing schedulers, in particular the so-called *ABP work-stealing* algorithm by Arora, Blumofe and Plaxton [11]. What makes work-stealing an interesting algorithm to study is its decentralized nature and the small amount of communication required for load-balancing, which is essential for scalability. Even though later work in this thesis moved away from pure work-stealing approaches, work-stealing still serves as a benchmark for scalability. Furthermore, the main ideas behind work-stealing were a significant inspiration to scheduler designs developed over the course of this work.

There were multiple factors that sparked our interest in non-blocking synchronization primitives and data structures. First and foremost, the implementation of the ABP work-stealing algorithm requires the use of a lock-free deque implementation for the work-stealing algorithm. The book *The Art of Multiprocessor Programming* by Herlihy and Shavit [81] provided a good introduction into non-blocking algorithms and data structures that allowed the practical implementation of ABP work-stealing. Herlihy and Shavit's book also inspired the lock-free implementation of all other synchronization primitives required by a task-parallel programming model, some of which could not be found in the literature. Finally, an intense cooperation with Philippas Tsigas and Daniel Cederman helped to push forward our work on lock-free synchronization.

1.2 History

Originally this work arose from the PEPPHER project[1]. The goal of PEPPHER was to improve performance portability and programmability of applications on heterogeneous many-core architectures. The programming model conceptualized in the PEPPHER project was based on task-parallelism with so-called *component tasks*. A component task is a generic version of a task, for which multiple implementation variants can exist for different types of processors. Component tasks are allowed to be implemented in a variety of programming models, and can be parallel as well, allowing them to be run on more than one processor. Programmers have the ability to annotate tasks allowing the scheduling system to make informed decisions when scheduling tasks.

Some of the work presented in this thesis was started in an attempt to tackle some of the challenges posed by PEPPHER [19]. Due to the complexity of those challenges we decided to focus on a simpler model than PEPPHER at first, with an option to later generalize the work. Thus, our work was focused on homogeneous multi-core architectures with a shared memory. Since the challenges posed by this more restricted model already turned out to be extremely interesting, a decision was later made to study this topic in more depth instead of generalizing to a more complex model.

The first problem that we worked on was to support the scheduling of parallel tasks in a scalable scheduler. This resulted in an implementation of the *mixed-mode parallel programming model* [147], which is presented in Section 2.6, as well as the *deterministic team-building* algorithm [146] presented in Section 4.4, which extends work-stealing to support the mixed-mode parallel programming model.

One difficulty that we ran into in our early work was the lack of a methodology for implementing and evaluating schedulers. This sparked the creation of *Pheet*, a framework for writing and evaluating task schedulers, related synchronization primitives and data structures. As a side-effect, Pheet became a fully functional library for writing task-parallel applications. For Pheet to be able to provide efficient performance counters, a wait-free version of *reducer hyperobjects* [141] was implemented. For efficient transitive termination detection of tasks *finisher hyperobjects* [141] were developed. A *wait-free memory reuse scheme* [141] was developed at the same time since it was required for implementing finisher hyperobjects in a wait-free manner.

In later work we were focused on how a scheduler can be informed about properties of tasks to make more informed scheduling decisions, a functionality motivated by the task annotations in PEPPHER. This resulted in the concept of *scheduling strategies* [142,143], which are presented in Section 2.7, an extension to task-parallel programming models, where tasks can be associated with instances of so-called *strategy objects* that can be provided by programmers to the scheduling system to give the scheduler information about the behaviour of tasks.

One aspect that turned out to be important with regard to scheduling strategies was the ability to influence the order in which tasks are executed by the scheduler. This spawned a separate line of work based on the *priority task scheduling* model [144, 148] presented in Section 2.5. Priority task scheduling presents unique challenges with regard to the container data structures used for storing tasks in a task scheduler, challenges that resulted in work on scalable relaxed priority queues, many of which are discussed in Chapter 5.

[1]The research leading to these results was partially funded by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 248481 (PEPPHER Project, `www.peppher.eu`)

1.3 Challenges

The efficiency of a task-parallel application is dependent on many factors. One such factor is the programming model, where the focus lies on *programmer productivity*. Programmers should be able to write and maintain correct and efficient programs effectively. In some cases this results in trade-offs between the efficiency of a program, and the complexity of the programming model. Our philosophy in such cases is to keep the programming model simple, as long as the impact on program efficiency is reasonable. While program transformations by a compiler can help to bridge this gap in many cases, work on compilers is out of the scope of this work and will not be discussed. Programming models are discussed in detail in Chapter 2.

Another factor relevant for efficiency of a task-parallel program is the (run-time) scheduler used to execute the program. The choice of scheduling policies has a large influence on time, space and communication bounds of an application, thus influencing the efficiency of a program. For run-time schedulers there exist trade-offs between the complexity of a scheduler, and the granularity of tasks. The more fine-grained tasks are, the more lightweight a scheduler has to be, to ensure that the cost of a scheduler does not outweigh its gains. Scheduling is discussed in Chapter 3.

Finally, synchronization primitives and container data structures for storing tasks inside a scheduler also have an impact on the performance of a parallel program. Here the goals are to provide low overhead and good scalability while fulfilling all guarantees required by the programming model and scheduling system. Good scalability can often be achieved with the use of non-blocking algorithms with strong progress guarantees. While there are often limits to the overhead and scalability of generic algorithms, *specialization* and *relaxation* can help increase efficiency. Specialized data structures, like *hyperobjects* (Section 2.8) and the *wait-free memory manager* (Section 4.3) allow for higher efficiency by restricting the use of these primitives, thus allowing the implementation to be based on certain assumptions. Relaxed data structures (Section 5.1) relax some of the guarantees provided by a data structure to reduce the amount of synchronization required and to work around bottlenecks. Synchronization algorithms and container data structures used for storing tasks in scheduling systems are discussed in Chapters 4 and 5.

1.4 Pheet

Most of the presented work is practical in nature, and was implemented and tested on shared memory systems. Due to the wide variety of factors that can influence program efficiency as discussed in the previous section, it quickly became clear that it is important to have a structured way by which new extensions to programming models, schedulers and synchronization primitives can be compared.

We developed the *Pheet* framework[2] (Chapter 6) to solve this problem. Pheet is a fully functional task-parallel programming library written in C++ that can be used to implement task-parallel programs. It provides a simple to use programming interface, thus enabling the quick parallelization of common algorithms with only little overhead.

What makes Pheet unique, though, is its plug-in architecture that allows many aspects of Pheet to be exchanged. This, for example, allows an application written in Pheet to be run with different task schedulers by only changing a single line of code. Or, to compare two different implementations of task queues, the task-queue used by a scheduler is replaced, but the scheduler itself stays the same. The scheduler decision is made at compile-time, thus allowing the compiler to perform optimizations based on the structure of a scheduler. As

[2] `www.pheet.org`

an example, Pheet provides a scheduler that will turn a parallel algorithm into a sequential execution by treating task spawns as function calls. Evaluating the scheduler decision at compile-time enables the compiler to perform function inlining and to resolve tail recursions similar to a sequential program.

Even though Pheet provides a generic plug-in interface it provides enough flexibility to extend the programming model. Each scheduler can provide arbitrary extensions to the programming model accessible to all applications that use the scheduler. An application that relies on an extension can also be executed with another scheduler that provides the same extension, but will not compile on a scheduler without support for the extension.

Pheet is accompanied by a set of micro-benchmarks, which are presented in Chapter 7. Each benchmark is implemented in the Pheet plug-in architecture, thus not only allowing to configure Pheet differently for multiple experiments, but also allowing to compare multiple implementation variants of the benchmark. This allows to provide both generic implementations usable on any scheduler, as well as specialized implementations that rely on specific extensions to schedulers.

1.5 Structure

This thesis is structured as follows: Chapter 2 focuses on extensions to the task-parallel programming model that both help to simplify writing parallel programs and help writing efficient task-parallel programs by exploiting their structure. Chapter 3 discusses implications of programming model extensions on the schedulers and their bounds. Chapter 4 presents implementations of programming primitives from Chapter 2, as well as supporting data structures for the implementation of schedulers. The most important type of supporting data structure for schedulers is a container for storing tasks. Chapter 5 presents a wide variety of such containers, each with different trade-offs with regard to scalability and guarantees on the order in which items are returned. The Pheet framework, which was used to implement and evaluate the contributions of this work is presented in Chapter 6. Pheet is accompanied by a set of micro-benchmarks, which are used to evaluate the contributions of this work in Chapter 7. A summary and outlook on future work, as well as work that did not find its way into this thesis, is given in Chapter 8.

2 Programming Models

The focus of this chapter is to present contributions to parallel programming models with a focus on models for task parallelism. It discusses parallel programming languages, programming patterns, interfaces to scheduling systems and synchronization primitives.

2.1 Task Parallelism

In the task parallel model, parallelism is explicitly exposed by the programmer. A program in this model consists of small units of work, the so-called tasks. Given a set of tasks $V = V_1, \ldots, V_n$, a partial order $\prec$ on V determines dependencies between tasks. If, for any $i, j \in 1, \ldots, n$, $V_i \prec V_j$, then V_j must not be executed before V_i finished executing. Tasks with no precedence constraints between each other can be executed in any order as well as in parallel.

Task parallel computations can be modelled as a *directed acyclic graph* (*dag*) of tasks $G = (V, E)$, as shown in Figure 2.1, where the tasks V are represented by vertices and the precedence constraints E by arcs in the dag. A task $x \in V$ can be executed if and only if $\forall p = yx \in E, y \in C$, where $C \subseteq V$ is the set of all tasks that have completed their execution.

We distinguish between *static* and *dynamic* dags of tasks. In a static dag all tasks and their priority ordering are predefined before the execution. Such dags are often found in deterministic applications on data where the structure is known beforehand, like dense linear algebra applications. A dynamic dag, on the other hand, comes from a task-parallel execution, where tasks are created dynamically depending on the data and previous results. Such dags are required for irregular applications, like branch-and-bound algorithms, but can be used for any kind of task-parallel application. In this work we focus on the execution of dynamic dags.

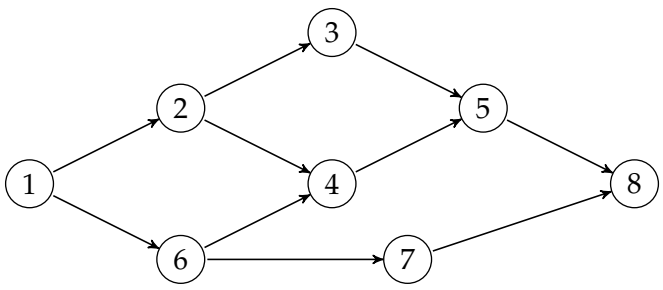


Figure 2.1: A directed acyclic graph.

2.1.1 Scheduling

The task parallel model by itself is independent of the number of processors available in a parallel machine, as well as their capabilities and topology, but instead exposes all potential parallelism to a *scheduler*. A *schedule* is a mapping of tasks to a processor in a specific order that respects the partial order of tasks, thereby determining when the task will be executed. We focus on schedules where each task is executed exactly once (unlike e.g. idempotent work-stealing [107]).

We distinguish between *offline* and *online* scheduling. An offline schedule is a schedule that was statically pre-calculated for a concrete machine. Tasks scheduled to the same processor have to be scheduled so that all precedence constraints are fulfilled. Precedence constraints between tasks scheduled to different processors are maintained based on synchronization mechanisms that ensure that the next scheduled task at a specific processor is only launched when all predecessors have been executed.

Offline schedules can be very efficient, since an optimal schedule can be calculated for a task-parallel application. Also, the only overhead imposed on the application by an offline schedule is the overhead for synchronization between processors. In practice, optimal schedules can only be calculated for small dags, due to scheduling being an NP-complete problem [63]. In addition, offline scheduling requires full information about the dag, including execution times of tasks. This makes offline scheduling impossible for scheduling of dynamic dags, tasks with non-deterministic execution times, as well as for multiprogrammed systems [45], where task execution times can vary depending on other applications running on the same machine.

For this reason, most task-parallel programming models today rely on online scheduling. Greedy online schedules have been shown to yield asymptotically optimal execution time up to a factor of two, and can be efficiently calculated at run-time for task-parallel applications [26, 65]. We concentrate on online scheduling in this work. Details on the scheduling techniques necessary to support our model of task parallelism are discussed in Chapter 3.

2.2 Programming Models based on Task Parallelism

There exists a wide variety of programming models based task parallelism. In this section we classify these models by generality, starting with the most general models.

2.2.1 Constructing a Dag

The most straightforward way for implementing task-parallel applications is to construct a dag of tasks, and to schedule its execution. In the StarPU library [15], for example, a dag is constructed by instantiating tasks and specifying dependencies between them. Dags in StarPU are dynamic, so additional tasks can be added to the dag throughout the execution. In StarPU data is passed between tasks explicitly, and memory transfer is managed by the runtime system, making it suitable for systems with distributed memory. This is mainly used to support heterogeneous systems, where different processing units with separate memories work on a single computation, as is common with GPUs (graphics processing units).

While a specific programming model based on dags may place further restrictions, operating on a dag affords the greatest flexibility and therefore the greatest potential parallelism, since no restrictions are placed on the dag. This comes at the price of being difficult to program and error-prone, however. Cycles are easily introduced into the graph and can only be recognized at run-time for dynamic graphs. Agrawal et al. [6] showed that it is possible to efficiently execute dags of tasks on top of a stricter programming model, making it possible to

use a stricter model in general and resorting to dags only when required to achieve sufficient parallelism.

2.2.2 Futures

A popular model for task-based programming are *futures*. It is used in Intel Threading Building Blocks [93], C++11 tasks [129], Multilisp [72], Mul-T [91] and COOL [37] among others. An example program implemented with futures is shown in Listing 2.1. In this model, a sequential program is annotated with `spawn` directives. Each spawn directive creates a parallel task out of the following expression (typically a function call). Each spawn directive returns a *future*, which is a synchronization variable that stores the result of the spawned expression, as soon as it has been evaluated. Synchronization is performed by waiting on futures.

Listing 2.1 Pseudocode for the quicksort algorithm implemented with futures.

```
1  void quicksort(int* begin, int* end) {
2    if(begin == end) return;
3
4    int* middle = partition(begin, end);
5    future<void> f1 = spawn quicksort(begin, middle);
6    future<void> f2 = spawn quicksort(middle + 1, end);
7
8    f1.wait();
9    f2.wait();
10 }
11
12 void main() {
13   ...
14   quicksort(begin, end);
15   ...
16 }
```

Programming models based on futures can support all task dags that have a single source and sink vertex. Typically, such dags also have in-degree and out-degree of at most two for all vertices, but this is not a strict requirement of the model. As long as the model restricts futures to only be accessed by tasks following in a sequential execution (which is typically the case for programming models with futures), it is not possible to introduce cyclic dependencies, and converting every task spawn into a function call will yield a valid sequential program. Nonetheless, when executed in parallel, the number of deviations from a sequential execution can be expected to be higher than for stricter models [127], thus introducing higher synchronization overhead and making the execution harder to reason about.

2.2.3 Async/Finish

Like programming models based on futures, the *async/finish* model, introduced with the X10 programming language [38], extends sequential programs with statements for parallelism. Tasks are spawned using a `spawn` directive (which is called `async` in X10, hence the name *async/finish*), and synchronization is performed using the `finish` statement. The `finish` statement is a synchronization directive that blocks progress until all (potentially parallel) work of the expression following the `finish` statement has been executed. This includes *transitively spawned tasks*, tasks spawned by other tasks affected by the `finish` statement.

Listing 2.2 Pseudocode for the quicksort algorithm implemented with a `finish` statement.

```
1 void quicksort(int* begin, int* end) {
2   if(begin == end) return;
3
4   int* middle = partition(begin, end);
5   spawn quicksort(begin, middle);
6   spawn quicksort(middle + 1, end);
7 }
8
9 void main() {
10  ...
11  finish quicksort(begin, end);
12  ...
13 }
```

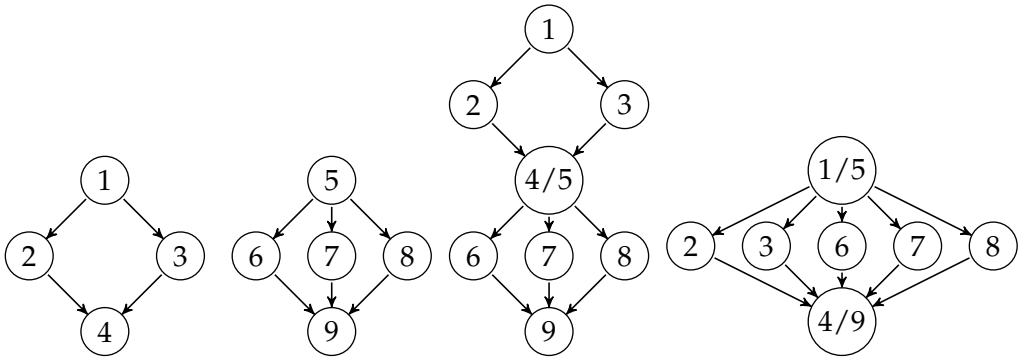
An example is given in Listing 2.2. There, no synchronization is performed at the end of each quicksort task. Instead, the first call to quicksort is preceded by a `finish` statement, which ensures that all tasks transitively created within the quicksort function have finished executing before execution proceeds after the finish statement.

In a parallel program based on the *async/finish* model, a sequential execution of the program, created by ignoring all `spawn` and `finish` directives, is always a valid execution of the parallel program. This also ensures that the dag is acyclic, thereby making the program deadlock-free if no other blocking means of synchronization apart from `finish` are used. Furthermore, this property greatly simplifies the reasoning about the execution, since a parallel execution can be related to the sequential execution. This property is used to calculate bounds on time, space and communication cost of task schedulers relative to to a sequential execution, which is discussed in Chapter 3.

The dags supported by the *async/finish* model fall into the class of *terminally strict* dags [4], which are planar dags where tasks are only allowed to synchronize with ancestors, and where a task is not allowed to synchronize with an ancestor later than its parent, or tasks spawned later by the parent. Typically, dags of an *async/finish* computation have unbounded in-degree, and an out-degree of two.

Listing 2.3 Cilk5 code for the quicksort algorithm.

```
1 cilk void quicksort(int* begin, int* end) {
2   if(begin == end) return;
3
4   int* middle = partition(begin, end);
5   spawn quicksort(begin, middle);
6   spawn quicksort(middle + 1, end);
7
8   // Not really needed due to the implicit sync at end of function
9   sync;
10 }
11
12 void main() {
13  ...
14  quicksort(begin, end);
15  ...
16 }
```



(a) Original (b) Series (c) Parallel

Figure 2.2: Construction of series-parallel dags.

2.2.4 Spawn/Sync

The *spawn/sync* model, which was made popular by Cilk [25], like the *async/finish* model, allows to parallelize a sequential program by adding `spawn` and `sync` statements. A `spawn` statement can be put before a function call, to allow the function call to be executed as a task in parallel with the function that called it. The `sync` statement is used to synchronize with tasks. Unlike futures, the `sync` statement does not synchronize with a specific task, but with all tasks spawned by the function. In addition, there is an implicit `sync` at the end of each function.

An example program is given in Listing 2.3. There, recursive calls to the quicksort function are executed in parallel. The `sync` at the end of the quicksort function synchronizes with the spawned subtasks, so that the function will only terminate if its subtasks terminated. Due to the implicit `sync` at the end of a function, the `sync` statement is purely optional in this case.

The *spawn/sync* model restricts the dags that can be supported to *fully strict* dags, which are restricted in the sense that a spawned task always has to synchronize with its parent. Furthermore, since the `sync` statement synchronizes with all tasks spawned by the parent, the dag also has all properties of a *terminally strict* dag, making the dag *planar*. Similar to the async/finish model, the resulting dags have unbounded in-degree and an out-degree of 2.

2.2.5 Nested parallel

The most restrictive common model for task parallelism is the nested parallel model, which is a *fork/join* model, where a sequential thread of execution forks into multiple parallel tasks, each performing the same operations on different data. As soon as all parallel tasks have been executed, the parent thread can proceed with its execution.

Nested parallel programs always have *series-parallel* dags, which are dags with a single source and a single sink, which can be constructed by combining multiple series-parallel dags in the following manner, as depicted in Figure 2.2: *series*: two dags are combined, by merging the source of one with the sink of the other (Figure 2.2b); *parallel*: two dags are combined by merging their sources and their sinks(Figure 2.2c).

One example of a nested parallel programming language is NESL [23], a functional programming language. An example quicksort implementation in NESL is shown in Listing 2.4. It slightly differs from the other quicksort examples given, in that the partitioning step is also parallelized, but is not performed in-place.

The original OpenMP standard also conformed to the nested parallel model, but since the

Listing 2.4 A nested parallel implementation of the quicksort algorithm in NESL (taken from [23])

```
function Quicksort(S) =
if (#S <= 1) then S
else
  let a = S[#S/2];
    S1 = {e in S| e < a};
    S2 = {e in S| e == a};
    S3 = {e in S| e > a};
    R  = {Quicksort(v): v in [S1,S3]};
  in R[0] ++ S2 ++ R[1];
```

introduction of the `task` directive in OpenMP 3.0 [16] its model of parallelism is close to the *spawn/sync* model.

2.2.6 Other models

One model that is popular for the development of research prototypes of task schedulers is *continuation passing style*. In this model, *continuations* (parts of a task following a blocking synchronization) are explicitly implemented as separate tasks, and communication between tasks is performed using messages. An example written in an old version of Cilk (which was at that point based on continuation passing style) is presented in Listing 2.5. While this style is not very elegant to program, it greatly simplifies the development of a parallel runtime system. One main difficulty that schedulers face if continuation passing style is not used is discussed in Section 3.2.3. Many task-parallel programming languages, like Cilk [25], Cilk++ [97] and Intel CilkPlus, rely on continuation passing style internally, but have a compiler that transforms a program from a model simpler than continuation passing style (e.g. spawn/sync).

Listing 2.5 Fibonacci algorithm written in continuation passing style in an old version of Cilk (taken with slight modifications from [25])

```
thread fib (cont int k, int n) {
  if (n<2) {
    send_argument (k,n);
  } else {
    cont int x, y;
    spawn_next sum (k, ?x, ?y);
    spawn fib (x, n-1);
    spawn fib (y, n-2);
  }
}

thread sum (cont int k, int x, int y) {
  send_argument (k, x+y);
}
```

In the StarSs programming language [113], a sequential program is extended by annotating function implementations as tasks. Whenever the given function is called within a program, the runtime system is allowed to execute it in parallel with the function that called it. Each task is also annotated with its memory access patterns. Whenever two tasks A and B, where A comes before B in a sequential execution need to access the same data, and at least one of

these accesses is a write access, then A comes before B in the partial order of tasks. StarSs allows programs to be parallelized by only modifying functions that are called and making them tasks. There is no need to modify the code that calls those tasks, since there are no explicit spawn operations and synchronization directives. Instead, these are implicitly handled by the runtime system. The main disadvantage of this model is that it is hard to handle irregular memory access patterns, which can easily lead to a dag stricter than required for the program. In addition, due to the lack of loop parallelization, task spawns inside a loop will be performed sequentially, one after the other, thereby reducing potential parallelism. While a heroic compiler might be able to parallelize this in some cases, this is hard to achieve in all cases.

2.3 Memory Models

A memory consistency model defines the order in which changes to shared data become visible to other processors. While sequential consistency [95] is arguably the most intuitive and simple to program model it is also difficult to realize efficiently in hardware, and for this reason most common processor architectures rely on relaxed memory consistency models [2].

While hardware memory consistency models are out of the scope of this work, it has become common for programming models and languages to provide their own, relaxed memory model [9, 16, 24, 93, 101, 129]. These *logical memory models* are independent of the actual hardware they are executed on, and are translated to the hardware memory consistency model by the compiler or the library implementation. They have in common that they all define the notion of a *data race* (a conflicting access to a shared non-synchronization variable, from which at least one is a write) and provide a sequentially consistent execution for *data-race-free* programs. For most models, the behaviour of a program with data races is undefined. One notable exception to this is the new Java memory model [101]. Since the Java programming language was designed to also execute untrusted, and therefore potentially malicious code, it is necessary for Java to preserve causality even for programs with data races. This way, an attacker cannot use data races to set pointers to out-of-thin-air values.

The new C++ memory model [28], on the other hand, was designed with high performance and scalability in mind. For this reason it allows anything to happen for programs with data races. To support fine-grained synchronization, variables can be declared as *atomic* types. Atomic types in C++ are treated as synchronization variables, all accesses are atomic, and their behaviour for concurrent accesses is well defined, so that such variables cannot introduce a data race. Behaviour of concurrent writes to such variables is still non-deterministic though, since depending on the order of these writes a different value might be stored. In addition, the C++ memory model allows programmers to specify the memory ordering requirements for every operation on an atomic type. This allows the programmer to minimize synchronization cost by only enforcing order between operations where it is necessary.

The merit of modern memory consistency models in programming languages is that the programmer is freed from having to think about the properties of the hardware, which might vary. Instead, programmers are required to thing about which ordering constraints on operations on shared variables. This is achieved by *happens-before* relationships that can be established by the programmer. An example of a happens-before relationship is shown in Listing 2.6. In this example, the variable `y` is written, and afterwards the value 1 is written to the atomic variable `x` with the requirement of the write being a *release* operation. A release operation ensures that after the written value is read by another thread using an *acquire* or stronger operation, this thread will also see all writes that the previous thread performed before updating `x`. Thus, the write to `y` *happens before* any thread observes `x == 1`. This greatly

Listing 2.6 Simple C++ example of a happens-before relationship.

```
1 atomic<int> x(0);
2 int y = 0;
3
4 void function1() {
5   y = 42;
6   // Release ensures that if a thread sees x == 1, 42 has been written to y
7   x.store(1, std::memory_order_release);
8 }
9
10 void function2 {
11   // Acquire ensures that all writes released by a store of the observed value
12   // happened before the acquire
13   if(x.load(std::memory_order_acquire) == 1) {
14     // Will be true if x == 1
15     assert(y == 42);
16   }
17 }
```

simplifies reasoning about the correctness of synchronization, while at the same time making the algorithm hardware independent.

Dag consistency [24] is an early logical memory consistency model for task-parallel programming models, which allows reads to return different values if they all correspond to valid serial orders, with the restriction that they must respect the dependencies in the dag of the task-parallel computation. Essentially, dag consistency establishes happens-before relationships between tasks that have dependencies on the dag. The advantage of this model is that, while it allows a relaxed memory model to be used on the hardware level, the program still behaves like a sequentially consistent program for operations with clear dependencies on the dag.

Since dag consistency is a fairly natural fit for task-parallel applications, it is desirable to support this in a task-parallel programming model. Fortunately, this is easily achieved for data-race-free programs when using a modern software memory consistency model as a base. To achieve this, a happens-before relationship must be established between everything that happened before a task was spawned, and the time when a thread starts executing that task. Under the C++ memory model this can be achieved, by storing newly created tasks in the task queue using a *release* operation, and reading tasks from task queues using an *acquire*. All schedulers implemented as part of this work fulfil dag consistency for data-race-free programs.

2.4 Locality Awareness

The shared memory programming model provides the programmer with a single global address space, where all memory accesses are treated equally with regard to the model, and it is assumed that all memory accesses take roughly the same time. While this model was feasible for a long time, since processor speed was the limiting factor for computations, processor speeds have increased at a higher rate than the speed of memory accesses, leading to the so-called *memory wall* [12, 149], where memory access latency becomes the limiting factor for computations. There are two ways, in which this can be counteracted: *caching* and *non-uniform memory access* (*NUMA*).

Caches are small, low-latency memories that exploit *temporal* and *spatial locality* to reduce memory access latency in many cases. This has the effect that many memory accesses become fast, but accesses that cannot take advantage of temporal or spatial locality will take significantly more time, a fact that is not visible in the programming model. Also, to fully explore the memory bandwidth, it is necessary for the programmer to take advantage of spatial locality as much as possible. In addition, in a parallel setting, writes to a memory location will lead to the invalidation of said memory location in all other processor's caches. This will also invalidate adjacent memory locations, leading to *cache thrashing* in *false-sharing* situations, where two processors regularly write to adjacent memory locations thereby invalidating the other processor's cache every time.

In non-uniform memory access (NUMA) systems, the system is split into multiple NUMA nodes, which are groups of processor (cores) that own part of the shared memory. While it is still possible to access memory from another NUMA node, and this memory is still cache coherent, memory accesses to another NUMA node incur much higher costs with regard to memory access latency and bandwidth. Both caches and NUMA require programs to be implemented in a *locality aware* fashion. We understand locality aware programs as programs that take into account differences in memory access latency.

One provably good model for locality aware algorithms is the *cache oblivious* model by Frigo at al. [62]. Cache oblivious algorithms are optimized to reduce the number of memory accesses, similar to the *I/O model* by Aggarwal et al. [5], but with the difference that no assumptions are made about the size of the caches or the depth of the cache hierarchy. Cache oblivious algorithms can be shown to be optimal for any type of cache hierarchy under the assumption that an optimal paging strategy is used. Frigo et al. [62] have shown that it is possible to simulate such an optimal strategy with only a constant factor overhead. The cache oblivious model does not take NUMA costs into account, though.

2.4.1 Task parallelism and locality awareness

The main idea behind task parallelism is that the programmer does not need to be aware about the actual processors a program is executed on, but instead can concentrate on exposing as much parallelism as possible. This bears similarities to the global address space model of shared memory, where the programmer is oblivious of where data is stored, since in the task model the programmer is oblivious of where a task is executed. Task schedulers, like the *parallel depth first* (*PDF*) scheduler [100] have been developed that provide bounds on the amount of cache misses with regard to the cache misses encountered by a sequential execution of the same program.

While we do not intend to enforce any specific model of locality aware scheduling on the programmer, we believe it is important for a task-scheduling system to have a concept of locality, which is also exposed to the programmer. For this, we introduce the concept of *places*. A *place*, by our definition denotes a single worker thread in a scheduling system and its supporting data-structures. A place is bound to a specific processing unit and will never be migrated to another processing unit. A task, which is executed at a specific place will stay in this place until the end of its execution.

Each place is assigned a unique id in the range from 0 to $P-1$, where P is the total number of places in the system. The numbering also reflects the memory hierarchy, so that places assigned to processors close to each other in the memory hierarchy will also have id's close to each other. Places can serve multiple purposes: They can simplify the implementation of parallel algorithms and data structures for task-parallel programming systems, by uniquely identifying processing units. If two memory accesses come from the same place, they are guaranteed to have happened in a specific order, and the later of these memory accesses will

observe the changes made by the previous one. This allows to create place specific local view on shared data, for which no synchronization is needed for accesses, even if those accesses come from unrelated tasks.

Another important aspect of places is that they allow a notion of distance in the memory hierarchy. The distance in the memory hierarchy between two places is related to the number of caches that the processing units those places are assigned to do not share. The larger the memory distance between two places, the higher the cost for one place to read memory last accessed by the other. To allow for NUMA-awareness, in addition to the normal memory distance it should be possible for programmers to query the NUMA distance between two places, with the distance 0 being returned for two places on the same NUMA node.

With the help of task priorities, which are presented in the next section, it is possible for programmers to influence the schedule of a task-parallel program. For data-parallel applications it can be of advantage to query for each block of data the NUMA node at which it is stored, and then to preferably execute tasks operating on each block of data at a place assigned to the same NUMA node as the memory. Unfortunately, on current systems such queries are fairly expensive making them impractical for most applications. We hope that on future systems this will be improved, since this will greatly simplify the creation of NUMA-aware algorithms in the task-parallel programming model.

2.5 Task Priorities

A common programming pattern, both for sequential, as well as for parallel algorithms is the pattern of a work pool. Work stored in such a work pool is processed in some order, and each unit of work may in turn lead to new units of work being added to the pool. One well-known example of an algorithm based on work pools is Dijkstra's algorithm for single-source shortest paths [7].

The task-parallel programming model is an implementation of the work pool pattern for parallel programming systems, where each task represents a unit of work, and requires the programmer to implement all parallel applications in this kind of pattern. While this makes the task-parallel programming model a natural fit for implementing applications based on work pools in parallel, some such applications are surprisingly difficult to implement in a task-parallel programming model. The reason for this is, that some applications based on work pools place requirements on the order in which work is returned from the pool. Dijkstra's algorithm, for example, uses a priority queue ordered by the tentative distance value of each node.

Task-parallel runtime systems, on the other hand, typically have a hard-coded execution order of tasks. The classical *ABP* work-stealing algorithm [11], for example, lets each worker thread execute its own tasks in a LIFO order, and worker threads running out of work will steal the oldest task from a random victim. While this execution order is provably efficient for applications based on a function-level parallelization of a sequential program, as discussed in Chapter 3, such an execution order is insufficient for the parallelization of algorithms based on work pools. Such algorithms can still be implemented in a task-parallel programming model by maintaining a separate ordered work-pool, and feeding the task-scheduler dummy tasks, each of which will then access the actual work pool. This approach is clearly a work-around, and instead of helping, the task-parallel programming model is in the way of the programmer and imposes additional overheads.

We introduce the model of *priority task scheduling* [142–144, 148] to address this shortcoming. With priority task scheduling, the task scheduler is made aware of the prefered execution order of tasks, and uses a concurrent priority queue implementation that is able to fulfil the

ordering constraints required by the algorithm. This enables the parallel implementation of any algorithm relying on work-pools, as long as an efficient concurrent priority queue implementation suitable for this problem is available.

While some algorithms, like Dijkstra's algorithm, rely on specific execution orders in the work pool, others that do not can still profit from priorities. Branch-and-bound algorithms, like the graph bipartitioning algorithm presented in Section 7.4 can use heuristics to explore promising branches first, thereby leading to branches being cut off earlier, and in turn requiring less work. Other algorithms, like prefix-sums (as presented in Section 7.6), can combine two passes on data into one if the tasks are executed in the right order. Other applications can benefit from a locality aware task schedule, which gives priority to tasks that access data already in the cache or stored on the NUMA node the worker thread is executing on [69, 128, 140]. Another common heuristic is to prioritize tasks on the critical path [126]. Resource obliviousness has been achieved with a special priority scheduling scheme [42]. A variety of task-parallel application kernels that profit from prioritization is presented by Lenharth et al. [99]. They postulate that a global priority ordering for tasks is often not beneficial for performance, and that different priority orderings within the same application/system are required.

2.5.1 Types of Priorities

In this section we discuss the types of priorities that can be encountered in an application. In general, priorities only apply to ready tasks. A task for which some dependencies are not satisfied will never be executed, even if it has a priority higher than any other task in the system.

Discrete priorities The simplest way of maintaining priorities for tasks is by assigning a discrete priority value to each task. In the simplest variant, the number of discrete values is very small (e.g. two values: high priority and low priority), which can be realized by maintaining a separate task queue for each priority value. A task queue will only be accessed, if all task queues of higher priority are empty.

If more values for priorities are supported, for example if priorities are 32-bit integers, it is possible to support more complex priority scheduled applications, and in fact most (but not all) of the priority scheduled applications in Chapter 7 can be implemented with discrete priorities, given enough discrete values. While our first priority scheduler prototype was based on discrete priority values, we realized that it is often hard to quantify the priority of a task as an integer without prior knowledge of the other tasks that will be created. A badly chosen task priority can easily lead to future tasks requiring priority values out of the bounds of allowed priority values.

Discrete priority values can, for example, be found in Intel Threading Building Blocks [93], where three priority values (high, medium and low) are supported. StarPU [15], supports a wider range of priority values, but the concrete range of values supported depends on the scheduler that is used.

Comparison based Higher flexibility and better programmability can be achieved by moving to comparison based priorities, an approach common for generic implementations of priority queues, as is found in the standard libraries of popular programming languages like Java and C++. In this case, the programmer provides the scheduling system with a *comparator*, a comparison operator for tasks, that tells the scheduling system, which of two tasks to prioritize. The comparator needs to be transitive, to ensure a total order of tasks.

Most of our work on priority scheduling concentrates on comparison-based priorities, since they provide an elegant abstraction to the programmer, while at the same time allowing for efficient scheduler implementations.

Pareto priorities As a generalization of discrete priority values, *pareto priorities* allow to assign a multi-dimensional priority value to each task. Given a set of ready tasks, pareto priorities ensure that the next task to be executed is a *pareto optimum*, a task for which the partial solution is not dominated by the partial solution of any other task.

The advantage of multi-dimensional priorities is that they only establish a partial ordering on tasks, giving the scheduling system more flexibility as to which task to execute next. In a parallel execution this can greatly reduce the number of processors attempting to execute a specific task in comparison to one-dimensional priorities, thereby leading to better scalability. This requires more complex priority queues, however, which we are currently working on and plan to present in future work. Pareto priorities can be useful for the multi-criteria shortest path problem [102], where exploring pareto optima first has been shown to be beneficial for parallel executions [119].

As a further generalization, to omit problems with discrete priority values, a comparator for each dimension can be used in a pareto set instead of giving the scheduler direct access to the values.

General partial orders Pareto priorities can again be generalized by not providing specific priorities, but instead a general partial ordering on tasks using a comparison operator. We believe that it is hard to develop efficient data structures for this problem, so that the cost of maintaining such a relation can easily outweigh the gains from using a general partial order instead of comparison based priorities or pareto priorities. Also, we are not aware of any applications that can profit from partial orders but are not covered by pareto priorities.

Specialized schemes For some applications, higher efficiency can be achieved by specialized priority queues. One example of an algorithm requiring this is the Δ-stepping algorithm [104], which may hold back the execution of certain work to bound the potentially useless work being performed. While there is no real point in writing a specialized priority task scheduler for a single algorithm, since such an algorithm can also be implemented in a different programming model, the use of strategies, as described in Section 2.7 can enable the use different algorithms relying on different priority queues inside a single task scheduler. This can allow to use the most efficient data structure for an algorithm, while at the same time allowing the task scheduler to distribute available processing units between all algorithms running in the application.

2.5.2 Locality aware priority scheduling

Prioritization of tasks can be used to provide locality optimizations to task-parallel applications. The execution order of tasks in classical *ABP* work-stealing algorithm [11] already performs surprisingly well for this for many applications due to high temporal locality (locally spawned tasks are executed depth-first, tasks spawned by other threads are only executed when own task queue is empty).

While the ABP work-stealing algorithm works well for applications, where a sequential execution is cache efficient, some applications may profit from problem specific locality optimizations. For algorithms with little temporal locality in a sequential execution a completely different execution order for tasks may be better. Also, for an application that relies on priorities, the LIFO/FIFO prioritization is not an option.

To support locality optimization, the prioritization of tasks is allowed to vary for each place. This means that each place would execute tasks in a different order. To achieve this, some of our comparison-based priority queue implementations allow the comparator to return different results depending on the place that calls the comparator. As an example, a strategy may store the NUMA node the data a task operates on is stored on. With such a strategy, tasks can then be ordered by NUMA distance (see Section 2.4), so that tasks with low distance to a given place will be preferably executed by that place. This could, for example be used in the prefix sums benchmark presented in Section 7.6 to let each worker thread preferably execute tasks operating on NUMA-local blocks of data.

2.6 Parallel Tasks/Mixed-mode Parallelism

Complex applications often call for a mixture of approaches to parallelization both for ease of expression as well as for achieving a desired efficiency on the available system architecture. A concrete example is applications that exhibit a mix of parallelism between task parallelism, where tasks that have no dependencies do not need to communicate with each other and *tightly coupled parallelism* between threads (processes) working on different parts of the same data structure with requirements for synchronization and data sharing between these threads (processes). Such a mixture of task and data parallelism is sometimes called *mixed-mode parallelism* (or *mixed data and task parallelism*), and this term will be used in the following.

We proposed an extension to the task parallel model [146,147] by supporting parallelism inside tasks, where a task can be executed by $P \geq 1$ processors (threads). Multi-processor (thread) tasks are used for implementing the tightly coupled, data parallel parts of applications, and require that the requested threads start execution of the task more or less synchronously. In order to facilitate communication within such tasks the allocated, scheduled threads will have a virtual consecutive rank. Furthermore, localized collective synchronization constructs, like barriers are provided that will synchronize between processors working on the same task.

A similar model for mixed-mode parallel tasks was presented by Kessler and Hanson [87]. The *crown scheduling* algorithm by Kessler et al. [88] provides mixed-mode parallel scheduling for streaming task collections a model stricter than the model we support. Like our team-building algorithm presented in Section 4.4, they simplify finding a good schedule, by only allowing team sizes to be powers of two.

Another model to integrate tight synchronization in a task-parallel model was presented by Dummler et al. with *communicating parallel tasks* [51,52]. In this model, communication relations between tasks can be specified, thus allowing such tasks to communicate during their execution. The scheduling system has to ensure that such tasks are co-scheduled at the same time on different worker threads.

In scheduling theory, tasks which by themselves are parallel, are called *malleable* or *moldable* tasks [50]. The difference between these terms is that malleable tasks can gain or lose worker threads while executing, whereas moldable tasks have a fixed number of worker threads while executing. Nonetheless, the term malleable is also often used to describe moldable tasks.

There is a large body of related work concerned scientific applications that mix task and data parallelism [17,36,47,116,130]; other natural, mixed-mode applications include combinatorial searches [46,150] and image processing [92].

A model that allows to analyze benefits and trade-offs in mixed-mode parallel applications was proposed by Chakrabarti et al. [36]. Otherwise, literature is often concerned with efficient scheduling of mixed-mode parallel applications, mostly in centralized, static approaches, see also [114]. Much work has dealt with mixed-mode parallelism as a means to structure and

reduce communication in distributed memory systems [47]. Likewise, there has been considerable work on integrating task and data parallelism in HPF, see for instance [49, 58, 115]. Also the data parallel OpenMP framework has recently been extended with constructs for task parallelism [16].

2.6.1 The programming model

In our model, we distinguish between normal, serial tasks and *mixed-mode parallel* tasks. By default, a task is a sequential task, which allows standard task-parallel applications to be easily migrated to a mixed-mode parallel programming model. When a task is marked as mixed-mode parallel, it can be executed by a *team* of threads instead of a single thread. A *team* is a group of (worker) threads working on processing a single task in an SPMD-like manner. A team is formed by an algorithm that we call *deterministic team building*, which is described in Section 4.4. Teams are built before the execution of a task and all threads stay on the team until the task finishes executing. The size of the team is decided by the programmer at spawn time.

Our mixed-mode parallel model caters mainly to tightly coupled parallelism, which is parallelism with a high amount of synchronization between threads. For this reason it is preferable to minimize the cost of synchronization between threads in a team, which is why teams are chosen, so that memory distance is minimized. As an example, in a multi-socket system with 8-core processors, it is guaranteed that a team consisting of up to 8 threads will have all 8 threads pinned to a different core in the same socket. Threads in a team are locally numbered from 0 to $t - 1$, where t is the team size. The numbering also reflects the memory hierarchy, in that the threads are grouped as to minimize memory distance (see also Section 2.4.1. Essentially, a team is chosen out of a block of subsequently numbered places, and the numbering inside a team corresponds to the place id's minus the offset.

Contrary to standard, serial tasks, the function body of a mixed-mode parallel task is executed multiple times in parallel, once by each worker thread in the team assigned to the task. The only way to distinguish between the worker threads is by the local id of each thread in the team. In addition, the total size of a team can be queried, allowing for static data distribution schemes and collective synchronization.

2.6.2 Extension to the model

While the original model presented in previous work [146,147] would only start execution of a parallel task as soon as all worker threads are ready, we have later relaxed these requirements to reduce idle time. Instead, we now provide an additional blocking statement, after which it is guaranteed that all threads of a team have started executing the current task.

2.6.3 Issues with the model

The main issue with the current model is that the programmer has to explicitly request a specific number of worker threads for a team. This does not fit well with the task model, where the programmer normally is oblivious of the number of processors in a system. Also, to make an informed decision as to what the best team size is, one would require both information about the scheduler load when the task is executed, as well as the scalability of the mixed-mode parallel algorithm. Neither the programmer, nor the scheduling system have enough information to make a good decision. We intend to explore the possibility of using strategies in future work to make a more informed decision about the number of threads working on a team when a task is being executed.

2.7 Scheduling Strategies

Standard task-parallel programming models are oblivious to most properties of individual tasks and treat tasks equally. While this is enough for a scheduler to create an asymptotically optimal schedule for an application, in practice a lot can be gained by giving the runtime system more information about tasks.

We presented *scheduling strategies* [142, 143] as a mechanism to inform a work-stealing scheduling system about properties of individual tasks in order to influence and improve the execution. A strategy can be associated with a task at spawn time. In contrast to scheduling policies that are global in nature, the scope of a scheduling strategy is an individual task. This allows to influence the scheduler behavior for a single task without incurring possibly negative effects for (all) other tasks. Scheduling strategies are *composable*, and different strategies can be used in a single task-parallel execution because there is a well-defined way in which such strategies interact.

2.7.1 Spawn to call

For tasks with small granularity, spawn overhead can significantly influence the total application execution time. On the other hand, too coarse grained tasks may lead to too little parallelism or less than optimal load-balancing. Spawn overhead can be reduced by converting task spawns to function calls at run-time [53]. This should preferably be done dynamically, when the scheduler has a large number of unprocessed tasks in its queues, thereby trading excess parallelism for a lower scheduler overhead. We have noticed that this simple heuristic can lead to a significant performance improvement for applications with either small variance in task granularity (algorithms on same-sized blocks) or decreasing task granularities (divide-and-conquer algorithms).

For other types of algorithms, this heuristic can be problematic since it is oblivious to task granularity. In the worst-case, high granularity tasks would be converted to function calls, and low granularity tasks put into the task queues. Strategies avoid such pathologies by allowing for specifying how and when tasks can be converted to function calls, based on the granularity of a given task and available parallel work.

Strategies make it possible to control the conversion of task spawns to synchronous function calls based on a user-defined function. This function has access to both information about the task, as well as the state of the runtime system, which is the number of tasks in the queues.

With added compiler support strategies can become even more useful, since they can work as a more dynamic replacement to a *cutoff value*, a value, typically hard-coded by programmers, below which an algorithm is executed sequentially. Since work on compilers is out of the focus of this work we have not implemented this. Without compiler support, it is still preferable to keep a cutoff in task implementations, due to the higher overhead of strategies.

2.7.2 Scheduling policy choice

There are two main policies by which task spawns can be handled: *work-first* and *help-first*. In the work-first first policy, a spawned task is executed immediately by the thread that spawned it, leaving the continuation to be executed by other threads, whereas in the help-first policy the continuation is executed first. Both policies are discussed in Chapter 3. While the work-first policy is guaranteed to lead to an asymptotically optimal execution in relation to a sequential execution [27], there are applications which can profit from a help-first scheduling policy [68].

Due to the limitation of our scheduling system to the help-first scheduling policy, for reasons discussed in Section 3.2.3, we do not support policy choices for scheduling strategies.

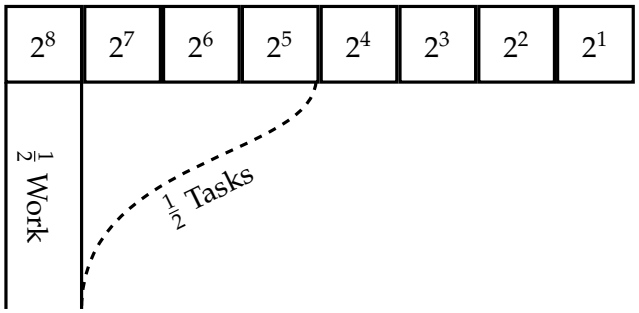


Figure 2.3: Difference between stealing half the tasks and half the work.

Nonetheless we agree with Guo et al [68] who argue that it is a useful extension for scheduling systems to support both policies.

2.7.3 Number of tasks to steal

For work-stealing systems it is well known that it is usually better to steal half the work instead of only a single task [20], the advantage being that work available in one queue quickly disseminates to the whole system. In standard work-stealing systems the amount of work incurred by the tasks is not known (by the scheduler), and stealing half the work is approximated by stealing half the tasks. In many cases, this approximation is highly inaccurate. For example, in many divide-and-conquer algorithms the amount of work is halved at each spawn. To steal half the work in such algorithms it would be sufficient to steal only the task with the largest amount of work, instead of half the tasks.

Our system allows the programmer to specify a *transitive weight* for each task, which is an estimate of the work that will be generated by a task and its descendants. The transitive weight associated with tasks can be used to estimate the work required by each new task and its descendants. This allows the stealing procedure to terminate as soon as half the work has been stolen, irrespectively of the number of tasks in the queues.

An example is depicted in Figure 2.3. There, the first task in the queue requires an estimated work of 2^8, and each following task requires only half the work of its predecessor. This is a common behaviour for divide-and-conquer algorithms. Now for such algorithms it is sufficient to steal the first task. Stealing half the other tasks, on the other hand, would lead to most of the work being stolen, requiring the original owner to steal back work at some point.

2.7.4 Dead tasks

For certain applications, like search-based algorithms, tasks can be speculatively created in order to achieve more parallelism. Newly calculated results can make some of these speculatively created tasks obsolete so that they do not need to be executed any more. Strategies allow the user to expose such *dead tasks* so that they can be removed early and will not be stolen by other threads. Support for speculative executions was added to Cilk [112] and OpenMP [133] using an `abort` command. Our approach is instead based on lazy recognition of dead tasks with the help of strategies. This allows for more efficient implementations of scheduler data structures, since often the cost of removing the task from the data structure can be omitted. This is discussed in more detail in Chapter 4.

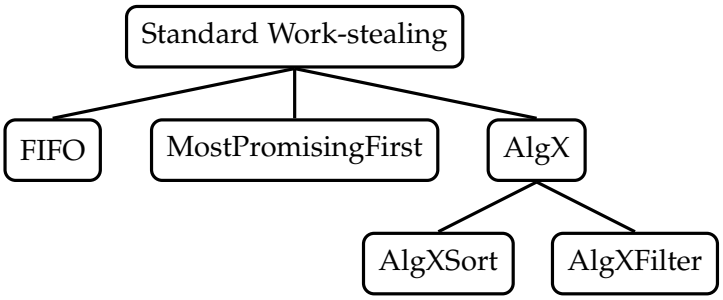


Figure 2.4: A hierarchy of scheduling strategies with LIFO/FIFO as the base strategy.

2.7.5 Specifying parallelism

The mixed-mode parallel model discussed in Section 2.6 allows a task to be parallel in itself. Throughout the development of the mixed-mode parallel model we realized that the decision as to how many processors are used to execute a mixed-mode parallel task cannot easily be made without information about other parallel work available. Strategies can allow to make this decision dynamically at runtime using both information about the application as well as information from the runtime system. So far our work on mixed-mode scheduling and scheduling strategies was performed on separate scheduler prototypes, this functionality is left for future work on a scheduler that combines these techniques.

2.7.6 Task execution order

An application specific execution order of tasks can lead to higher efficiency (performance, memory usage, quality of the results) compared to a fixed execution order like *last-in-first-out*. This is discussed in detail in Section 2.5. Strategies can be used to suggest an execution order to the scheduling system by giving the user a means to prioritize tasks of the same type. In the simplest case, prioritization is implemented by a comparison function that takes two instances of strategies of the same type and determines which should be preferred over the other. Since each instance of a strategy is associated with a single task, the prioritization of an instance leads to the prioritization of a task.

In a more complex setting, a strategy can enforce the use of a *pareto priority queue* to order tasks of the same type. Each such strategy is then associated with a *pareto set*, which is used to determine a partial ordering between tasks.

Each type of strategy can be associated with a different scheduler data structure, allowing for a different kind of priority ordering and different guarantees. While work-stealing based priority queues only guarantee priority order for locally created tasks, ρ-relaxed priority queues can guarantee global ordering within certain bounds (see Section 5.1). Also note that conversion of spawns to function calls may violate the prioritization.

2.7.7 Composability

A major design goal of scheduling strategies is composability. It should be possible for different applications or parts of the same application that use different strategies to run concurrently within the same, single scheduler. While this is simple to achieve for properties that are specific for individual tasks, like for the conversion of task spawns to function calls, it is much more difficult for prioritization.

We solve the composability problem by imposing a hierarchy on strategies as shown in Figure 2.4. Strategies of different types are composed by the strategy of their lowest common ancestor. Since the hierarchy has a single root, any two strategies have at least one common ancestor. A *standard work-stealing strategy* is the default root strategy. The hierarchy allows for different algorithmic kernels in a single application to use different strategies as indicated in Figure 2.4. While some kernels might rely on the base strategy, another kernel might exhibit better performance with a strategy that enforces a FIFO (first-in-first-out) order on tasks. Search algorithms, on the other hand are often faster with strategies where the most promising path is explored first. More complex, real applications often consist of different algorithmic kernels. In Figure 2.4 we included an algorithm, *AlgX*, which calls both sorting (AlgXSort) and filtering (AlgXFilter) kernels inside. Both kernels might require specialized strategies for efficient execution. In addition, *AlgX* might need to reduce its critical path length by ordering different calls to sort and filter. This behavior can be achieved with a common base strategy for both the sorting and the filtering strategy for *AlgX*.

Regarding the priority ordering imposed by strategy, an absolute ordering on tasks with different type of strategies is enforced by always letting child strategies overrule their ancestors. For the strategies in Figure 2.4, this results in the FIFO strategy overruling the standard work-stealing strategy. This is done by first ordering all tasks by the standard work-stealing strategy, and then moving all tasks relying on the FIFO strategy for which the FIFO ordering is violated further to the front to the first position where the FIFO constraints are met.

2.7.8 Syntax of strategies

In our implementation of Strategies in the Pheet framework (see Chapter 6), a scheduling strategy is a class derived from a base strategy class that implements base functionality required by all strategies, and a default behavior. Strategies derived from the base strategy can provide different behavior, for example a different prioritization of tasks, by overriding the default behavior. The constructor of a strategy class is allowed to take any kind of parameter the programmer desires, which allows strategies to act on problem specific information. An instance of the strategy class is created and stored for each spawned task. These objects are then used by the scheduler to make scheduling decisions for the specific task and to determine the execution order of the stored tasks.

Algorithm 2.7 depicts an example implementation of a strategy that provides depth-first execution for locally spawned tasks, and a breadth-first execution for tasks created at other places. It assumes a tree-like algorithm where all tasks in the subtree will be generated. The constructor of the strategy stores the depth of the given task, as well as the place at which the task was spawned.

To enable conversion of task spawns to function calls, which is disabled by default, we provide an implementation of the `can_call` method. This method is called by the scheduler for every task that is spawned with a strategy of this type, and will convert the task spawn into a function call if the method returns `true`. Since, in a depth-first execution the transitive granularity of tasks can be expected to become smaller with increasing depth, it is sufficient to specify a cutoff, after which tasks are executed as function calls. The concrete cutoff chosen depends on the algorithm that uses the strategy.

The `prioritize` method determines the execution order of tasks. It takes a reference to a second strategy object of the same type as parameter and should return `true` if the task associated with the current instance of the strategy should be executed before the other task, and `false` otherwise. Algorithm 2.7 implements different behaviors depending on whether a task was spawned in the same place or not. Tasks spawned in the same place are prioritized for locality reasons and are executed in depth-first order. To facilitate locality-aware

Listing 2.7 Example strategy for a tree-like algorithm with local depth-first local execution and breadth-first stealing.

```
1 class DepthFirstStrategy : public Pheet::Environment::BaseStrategy {
2 public:
3   // Self-reference, by convention
4   typedef DepthFirstStrategy Self;
5
6   // Base class, required by Pheet
7   typedef Pheet::Environment::BaseStrategy BaseStrategy;
8
9   // Priority queue to be used by the scheduler
10  typedef LSMLocalityTaskStorage<Pheet, Self> TaskStorage;
11
12  DepthFirstStrategy(int depth)
13  :place(Pheet::get_place()), depth(depth)
14  {}
15
16  /*
17   * Allow spawns to be converted to calls if number of tasks is enough for all places
18   * (works well, since granularity of tasks is decreasing due to depth-first)
19   */
20  bool can_call(TaskStorage task_storage) const {
21    // CUTOFF is an algorithm dependent tuning parameter
22    return task_storage.size() > CUTOFF;
23  }
24
25  bool prioritize(DepthFirstStrategy& other) {
26    if(this->place == Pheet::get_place()) {
27      // This task has been spawned at this place
28      if(other.place == Pheet::get_place()) {
29        // If both tasks are spawned locally go depth first
30        return depth > other.depth;
31      }
32      // Prefer local task
33      return true;
34    }
35    else if(other.place == Pheet::get_place()) {
36      // Only other task was spawned at this place, so prefer other task
37      return false;
38    }
39
40    // Calculate memory distance
41    int d1 = Pheet::get_distance_to(this->place);
42    int d2 = Pheet::get_distance_to(other.place);
43    if(d1 != d2) {
44      // Take task with smaller memory distance
45      return d1 > d2;
46    }
47
48    // For non-local tasks with same distance go breadth-first
49    return depth < other.depth;
50  }
51 private:
52  Pheet::Place* place;
53  int depth;
54 };
```

scheduling, we provide strategy objects with a way to calculate the memory distance between different places. For tasks spawned by other places, we prefer the execution of tasks with small memory distance between the place that spawned the task, and the place that will execute it. Tasks spawned by places with the same memory distance are executed in breadth-first order to increase the expected amount of locally spawned work.

2.8 Hyperobjects

Hyperobjects are efficient mechanisms to coordinate accesses to shared variables and data-structures in task-parallel programming models, where each thread can operate on its own coordinated local view of the shared data. Synchronization between local views is restricted to occur at well-defined points in the execution, which correspond to synchronization points in the task-parallel application, and can be left to the hyperobject implementation.

Hyperobjects are known from the Cilk++ programming language [97], where they were primarily used to implement *reducers*. Hyperobjects allow implementation of very general reducers that can support any (complex) data type. In addition, the reduce operation does not need to be commutative, associativity suffices. A more efficient implementation of hyperobjects was presented by Lee et al. [96]. Leiserson and Schardl [98] used hyperobjects to implement a work-efficient parallel breadth-first search algorithm. There, a bag reducer is used to collect nodes that should be processed in the next layer of the algorithm. In our own work, we used reducer hyperobjects for the implementation of performance counters and to collect results in search-based algorithms.

In previous work [141], we provided an alternative model for hyperobjects that does not require programming language or runtime support and may therefore be used with any task-parallel programming system. Although less streamlined than the Cilk++ model, it has the advantage of not making any assumptions about the task model, and not requiring runtime system support. With our model hyperobjects can be provided as a standalone library that can be used in any task-based runtime system. Our model allows for efficient wait-free implementations of hyperobjects as shown in Sections 4.5 and 4.6.

Finally, we present the novel family of *finisher hyperobjects* for transitive termination detection based on reference counting. These hyperobjects can be used to efficiently implement task synchronization primitives like *finish*. However, finishers can also be used to manage reference-counted resources, e.g. *shared pointers* and *copy-on-write pointers*.

2.8.1 Related work

While they do not make the connection to hyperobjects, the *finish accumulators* presented by Shirako et al. [122] are a reduction primitive with strong similarities to reducer hyperobjects. As with hyperobjects the structure of a task-parallel computation is used to reduce synchronization. As with our reducer hyperobject implementation, the final result of a reduction can be retrieved at the end of a *finish region*, thus giving the construct the name *finish accumulators*. Even though the name might suggest it, finish accumulators are not finisher hyperobjects.

Hyperqueues [138] are a specific type of hyperobject that is designed to support producer-consumer queues inside task-parallel programs. Like in our model of hyperobjects, hyperqueues need to be explicitly passed on to task, and in addition the programmer needs to specify whether the queue will be used by a producer, a consumer or both in the subtask and all its descendants. Tasks in producer mode can fill a queue in order of a sequential execution with concepts similar to standard hyperobjects. A task in consumer mode will only be executed by the runtime system if all its predecessors in sequential order that also access the queue in consumer mode have completed executing.

2.8.2 Associative reducers

Associative reducer hyperobjects allow to perform associative operations on any variable or data-structure in parallel by multiple tasks. The reducer guarantees that from the user's point of view, the operations are performed in the same order they would have been performed in a sequential execution of the program. The final value of a reducer can be retrieved as soon as all parallel tasks that have access to the reducer hyperobject have finished executing.

2.8.3 Finishers

The novel *finisher* hyperobject provides wait-free reference counting. In parallel executions it maintains local reference counts at each thread. A *uniqueness* check can be performed for a finisher at any time. A finisher is unique if only a single copy of the given finisher exists, meaning that the reference count equals one. In addition, finisher hyperobjects can be configured to call a cleanup function when the last copy of the hyperobject is destroyed.

Shared pointers (resources)

One application of reference counting with finisher hyperobjects is garbage collection for shared resources with reference counting similar to the standard C++ `shared_ptr` class. For this, a shared pointer needs to be stored in conjunction with the finisher. Whenever the last finisher is destroyed, the last reference to the shared data is destroyed as well so that it can be deleted.

Copy-on-write pointer

As a special case of reference counted pointers, we propose the *copy-on-write pointer*, which may be used to pass on references to large data-structures that are rarely modified and where modifications should only be visible to subtasks. It provides shared pointer semantics for read accesses. When a write is requested, exclusive access to a copy of the data-structure has to be established. If there are multiple copies of the hyperobject referencing the same resource, a copy of the referenced data is created.

Finish regions

Finisher hyperobjects can also be used to maintain finish regions for transitive termination detection of tasks in the *async/finish* model (see Section 2.2.3. In practice, we integrated the concepts from the finisher hyperobjects directly into our schedulers, so that the hyperobject is not visible to the user. The findings of our work on hyperobjects allowed us to improve the efficiency of finish regions, and to make them wait-free.

2.8.4 Computation model

To gain an intuition as to how our model of hyperobjects works, and how it is connected to the task-parallel programming model, we need to revisit the way, in which a task-parallel computation can be modelled. For our explanation we will assume the *spawn/sync* (Section 2.2.4) model for simplicity, but our model also works without need for adaptations in the *async/finish* (Section 2.2.3) and *futures* (Section 2.2.2) models.

Figure 2.5 shows the dag a task-parallel computation in the *spawn/sync* model. Each node represents a single statement in the computation. Edges represent computation dependencies. We call nodes with two outgoing edge *spawn nodes* (nodes 3 and 5 in Figure 2.5). These nodes contain a spawn statement, which creates a new task that can be executed in parallel. One of

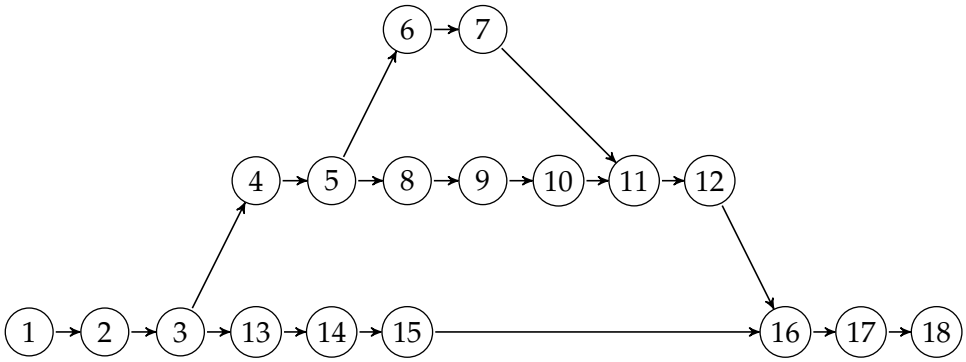


Figure 2.5: A dag in the spawn/sync model.

the outgoing edges, which points to the instructions of the newly spawned task, is called the *spawn edge*. By convention, we draw spawn edges pointing upward. The other edge is called the *continuation edge* and is drawn horizontally by convention.

Sync nodes (nodes 11 and 16 in Figure 2.5) represent a synchronization point between two tasks (a task and its parent in the fully strict spawn/sync model), where one task waits for one or more tasks to finish executing. Sync nodes are represented by nodes with more than one incoming edge. One of these edges is a continuation edge, the others are called *sync edges*. Only when all incoming edges have been reached by the computation (we say in this case that the sync node is *satisfied*), will the outgoing edge of the sync node be followed.

All three models supported by our hyperobjects (spawn/sync, async/finish and futures) have in common that they have clear semantics for a sequential execution, based on the dag of the parallel execution. The execution order of a sequential execution is shown in Figure 2.5 by the numbering of the nodes. At a spawn node a sequential execution will always follow the spawn edge first, until an unsatisfied sync node is reached. Only then will the execution backtrack to the next continuation edge and follow this edge next.

2.8.5 Hyperobject model

Hyperobjects exploit the dag structure of task-parallel computations. Whenever a spawn node is reached, a hyperobject is copied, and the spawned task will operate on a different copy of the hyperobject than the continuation. These copies will then be merged back into a single copy at a sync node. We call these copies *local views* of the hyperobject. The main feature of hyperobjects is that no synchronization is required for accessing a local view, since each local view is only accessed by a single thread. Synchronization is only performed when synchronization is performed in the task-parallel computation as well, thereby intuitively allowing to bound the number of synchronization operations for each hyperobject.

In our model, the spawn and join nodes are made visible to the hyperobject by the programmer. Whenever a new local view might be required, since a new task is spawned, a copy of the hyperobject has to be created by the programmer. At join nodes this copy has to be destroyed, thereby informing the hyperobject of a join.

Although this solution is perhaps less elegant than Cilk++ hyperobjects, which can be used like shared variables, our approach also has a few advantages. First, our model does not make any strictness assumptions about the computation. Second, every access to a Cilk++ hyperobject requires an expensive hash-table lookup, whereas our model relies on fast accesses to local stack variables. Finally, hyperobjects in our model can be implemented independently of the runtime system. Cilk++ hyperobjects require some helper data-structures to be stored with each stack-frame used by a task. These helper data-structures then need to be merged by

the run-time system every time a stack-frame is destroyed. This reliance on run-time system support makes adoption to run-time systems other than Cilk++ difficult, which might be a reason why hyperobjects have not yet been widely adopted for environments other than Cilk.

Our model of hyperobjects, where copies of the hyperobject need to be created and destroyed with new tasks, fits well with an object oriented framework. There, hyperobjects can be passed to subtasks by-value, which essentially leads to a copy being created for each task. As soon as the task finishes executing, the copy automatically runs out of scope, thereby deleting the hyperobject and marking a join node. This object-oriented framework also makes hyperobjects exception-safe. As long as the destruction of (task-)local variables is ensured by the programming language and task-scheduling framework, hyperobjects in our model will behave correctly even in case of exceptions.

Listing 2.8 Pseudocode demonstrating the use of hyperobjects.

```
1 /* With scheduler support, the finisher hyperobject does not
2  * actually need to be passed to subtasks, and is only shown
3  * (in gray) for demonstrative purposes.
4  */
5 void find_smaller(int limit, int* begin, int* end,
6     list_reducer<int> red, finisher f) {
7   if((end - begin) > 1) {
8     int* pivot = begin + (end - begin)/2;
9     spawn find_smaller(limit, begin, pivot, red, f);
10    spawn find_smaller(limit, pivot, end, red, f);
11  }
12  else if(*begin < limit)
13    red.add(*begin);
14  // No synchronization needed inside this task.
15 }
16
17 void main() {
18   int[] data = {3, 5, 8, 4, 2, 7, 6, 1};
19   list_reducer<int> red;
20   finisher f;
21   find_smaller(6, data, data + 8, red, f);
22   // Synchronization with subtasks only occurs here
23   f.finish();
24   // result now contains the list {3, 5, 4, 2, 1}
25   list<int> result = red.reduce();
26 }
```

Algorithm 2.8 shows a parallel array searching algorithm implemented in the *async/finish model* that makes use of hyperobjects. It searches for numbers smaller than `limit` in a given array by recursively splitting the array until one element remains. If the remaining element is smaller than `limit` it is added to the *list reducer*, a hyperobject that builds a list of elements. Note that the reducer preserves the ordering of elements in the array, although the tasks may be executed in any order.

The algorithm also makes use of a *finisher* hyperobject for transitive termination detection of tasks. It provides blocking synchronization until all copies of the finisher are destroyed, which is when all subtasks have terminated. The finisher hyperobject is only explicitly shown for demonstrative purposes. In practice, the finisher will be integrated directly into the scheduler, and hidden from the user.

To illustrate the advantages of hyperobjects more concretely, we provide an implementa-

Listing 2.9 Pseudocode demonstrating the find_smaller algorithm without hyperobjects

```
1 list<int> find_smaller(int limit, int* begin, int* end) {
2   list<int> result;
3   if((end - begin) > 1) {
4     int* pivot = begin + (end - begin)/2;
5     list<int> a =
6       spawn find_smaller(limit, begin, pivot);
7     list<int> b =
8       spawn find_smaller(limit, pivot, end);
9     // Explicit sync needed in this case
10    sync;
11    result.append(a);
12    result.append(b);
13  }
14  else if(*begin < limit)
15    result.append(*begin);
16  return result;
17 }
```

tion of the `find_smaller` algorithm without hyperobjects. This is shown as Algorithm 2.9. While this is a perfectly valid way of implementing the same algorithm, it has some disadvantages. First, it requires a synchronization in each task to make sure that the return value of subtasks becomes available. This synchronization is not necessary with hyperobjects, since different local views are combined as soon as they are available. In addition, hyperobjects can reduce the amount of merging necessary. In Algorithm 2.9 a separate list is created and filled for each task, regardless of whether these tasks are executed in parallel, and those lists are merged afterwards in a parent task. Hyperobjects, on the other hand, automatically reuse lists when they are used in a sequential order, which reduces the total amount of merging operations performed.

3 Task Scheduling

In this chapter we discuss the properties of task schedulers required in order to support the programming models in Chapter 2. This also directly relates to the schedulers used in the Pheet framework, which is presented in Chapter 6.

3.1 Related Work

3.1.1 Early work

Some of the first results on task scheduling for multiprocessing systems were published by Graham in 1966 [65, 66]. The work arose from the observation that execution times can vary for parallel workloads. In some cases an increase in the number of processors would even increase the execution time. Graham created a model of a parallel execution and provided bounds for the possible variations in execution time. The model for parallel executions used by Graham is based on a set of tasks $T = T_1, \ldots, T_m$, for which a partial order is given. These tasks are processed by n identical processing units. Graham introduced the concept of a *list schedule*. A list schedule is based on a list of tasks, which can be any permutation of the set of tasks T. In a list schedule an idle processor scans through the list of tasks, until it finds a task that has not been executed, and for which all precedence constraints are fulfilled. This task is executed next by the given processor without delay. Each task can be executed exactly once. If no task is found by an idle processor, the processor remains idle. As soon as another processor finishes executing a task all idle processors recheck the task list for new work.

Graham was able to obtain bounds for the variation in execution time in his model. The bounds are based on the comparison of two arbitrary list schedules L and L'. The first one is executed in time ω on n processors, and the second one in ω' time on n' processors. It can be shown that for any L and L' the execution time is bounded by : $\frac{\omega'}{\omega} \leq 1 + \frac{n-1}{n'}$ if $\omega' \leq \omega$. For fixed number of processors n, any execution which takes time ω^* fulfils the following bound with regard to to the best possible execution time ω_0: $\frac{\omega^*}{\omega_0} \leq 2 - \frac{2}{n+1}$. If there is no partial ordering (tasks can be executed in any order), this bound can be improved to $\frac{\omega^*}{\omega_0} \leq \frac{4}{3} - \frac{1}{3n}$.

List schedules are a way of modelling greedy schedules, since for any greedy schedule, a corresponding list schedule can be constructed. This means that the bounds by Graham can be applied to any type of greedy task scheduling. Therefore, it was already shown by Graham that any greedy schedule is within a factor two of optimal. What is ignored in the model of Graham is the cost of communication and the cost of scheduling tasks.

Another bound relevant for task scheduling was established by Brent in 1974 [30]. Brent analysed how general arithmetic expressions can be evaluated in parallel. One Lemma in Brent's work (Lemma 2), which was a small building block used to prove the bounds presented in Brent's work, is of particular interest. It states that given a computation with t unit

time operations, which can be executed in q time steps in parallel, this computation can be performed in $t + \frac{q-t}{p}$ time steps. Ironically, even though it was only a side result in Brent's work, this is what is now known under the name *Brent's theorem*. Later results on task scheduling provided similar results based on the *work* and *span* of a task graph.

3.1.2 Functional programming

Functional programming languages are a natural fit for task scheduling. In their purest form function calls in a functional programming language exhibit no side-effects, allowing them to be executed in parallel with other function calls. As an example, many concurrent Lisp languages [41,60,86] were based on the side-effect-free subset of Lisp. In cases where side-effects are allowed, functional programming languages typically provide mechanisms to encapsulate side-effects.

A particularly interesting functional programming language is the Multilisp language by Halstead [71,72]. It is based on the Lisp dialect Scheme [132]. Multilisp supports the relaxations to side-effect freedom provided by Scheme. Multilisp was one of the first systems to use work-stealing. Halstead already observed that, while most parallel executions can lead to an explosion in space usage this is not the case when tasks in the local task queue are executed in last-in first-out (LIFO) order. Halstead even suggested to steal tasks in FIFO order as it is now done in most modern work-stealing systems. The relationship between stack depth of a serial execution and space usage of a parallel depth first execution was also established, but no bounds were given. It is unclear, whether it is possible to give space bounds for Multilisp, since it is possible that the LIFO order is broken whenever the execution of a task is preempted due to a wait condition.

3.1.3 Space bounds

One of the first space bounds for dynamic task scheduling was established by Burton in 1988 [33] for executing *trees of tasks*. A *tree of tasks* is defined by Burton as a DAG where tasks always synchronize with their parent task, which conforms to the definition of a *fully strict computation*. It is shown how a program that requires s units of storage sequentially can be executed in parallel on a machine with s units of storage per processor. Processors are mapped to a virtual tree of processors, where communication can only happen between parents and children. Assuming that the sequential space usage for the transitive (including all children) execution of a task is known, the additional space usage for expanding a task out of order can be calculated. Child tasks are scheduled for execution, and tasks are only stolen if this does not violate the space bounds. This leads to a mixture of a breadth-first and a depth-first execution where depth-first is the common case.

First space bounds for greedy task schedules were provided by Blumofe and Leiserson [26]. They show that there always exists a greedy schedule for strict computations, which only uses P times the space of a corresponding sequential execution, where P is the number of worker threads of the scheduler. They also show that execution time for greedy schedules is always within a factor of two optimal ($T_P \leq \frac{T_1}{P} + T_\infty$), by improving on the time bounds by Graham [65,66] and Brent [30]. This result also implies that given enough parallelism in relation to the depth of the task graph, any greedy execution is close to optimal.

The proof for the space bound by Blumofe and Leiserson is first shown for depth-first computations (a class of computations quite similar to nested parallel computations), and then it is shown that time and space can always be bounded for strict computations. They also show that this is not possible for general computations, where either the time or the space bound has to be sacrificed in favour of the other. Nonetheless, any parallel computation

can be *strictified* by sacrificing some potential parallelism. Blumofe and Leiserson observed that any schedule that, P being the number of processors, only uses space of P times the sequential space can be seen as efficient, since the time-space product is independent of P. Blumofe and Leiserson presented both a centralized and a distributed scheduler for strict parallel computations. The centralized one always executes the P tasks with biggest activation depth, P being the number of processors, while the distributed one is based on a work-sharing like concept with restrictions to allowed activation depth.

Blelloch et al. [22] managed to further improve the space bound for nested parallel programs. They identified a class of task schedules, which can be processed on a PRAM in $O(w/p + d)$, while using $O(s_1 + pd)$ space, w being the size of the DAG, d it's depth, p the number of processors and s_1 the space usage of a space-efficient sequential execution. The bound is achieved by bounding number of *premature nodes* being processed at any point in time to $(p-1)(d-1)$ for this class of schedules. A premature node is a node that is executed even though it is not the next node to be executed in a sequential list schedule. The first proof only works for dags with binary fanout, but converting a dag to binary fanout will increase the depth of the dag by a log-factor. This can be omitted by using a lazy task creation scheme, so that only constant space is used for any nested parallel region, regardless of the fanout. An algorithm is also provided for synchronizing an arbitrary number of parallel tasks.

While the scheduler developed by Blelloch et al. had good asymptotic space bounds, it had too high overheads to be practical. It requires a global rescheduling of threads after every instruction to fulfil the space bounds, and it is possible that a thread is moved to a different processor at every time step, leading to a high cost of communication. Narlikar and Blelloch [108] developed a practical scheduler with similar bounds that is able to operate in a non-preemptive and asynchronous manner. The scheduler performs a depth-first scheduling, where after executing p tasks (p being the number of threads) the first idle thread selects the p deepest tasks to be executed next by all threads. The memory bound is achieved by restricting each node in the DAG to allocate at most K units of memory, and to preempt itself as soon as it needs more. Nodes that need to allocate more than K units of memory can be modelled as a tree of nodes performing no-ops and allocating K units of memory each. Under the assumption that all processors operate in fixed (synchronized) timesteps, it is possible to upper bound the memory used by out of order tasks.

3.1.4 Work-stealing

Work-stealing is a now popular scheduling technique that dates back to work by Burton [34] and Halstead [71]. Blumofe and Leisterson [27] have shown that space and communication bounds can be given for work-stealing schedulers, and that time-bounds can be given for work-stealing under dedicated environments. Tighter bounds for the same setting were presented later by Tchiboukdjian et al. [134]. Arora et al. [11] provided time bounds for work-stealing on multiprogrammed environments, as well as a lock-free implementation of deques for work-stealing schedulers. Most current work on deques for work-stealing builds upon the so-called ABP work-stealing deques (named after the authors Arora, Blumofe and Plaxton). Acar et al. [1] analysed the data locality of work-stealing schedulers, and provided upper bounds on the number of cache misses for nested parallel applications. Spoonhower et al. [127] extended the bounds on work-stealing to task-parallel programs with futures. They do this by bounding the number of *deviations* from a sequential execution.

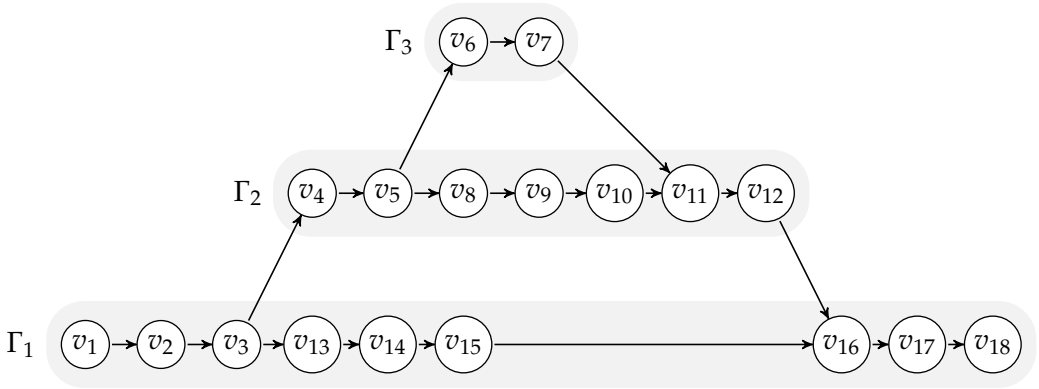


Figure 3.1: The structure of a task-parallel computation.

3.2 Requirements to the Scheduling Model

In this section we describe and motivate the requirements we place on the scheduling model. Some of the requirements differ from what can be found in related work, and where possible we provide bounds on the schedulers in our model.

3.2.1 Structure of a computation

A task-parallel execution in our context can be modelled as a dag of instructions $G = (V, E)$, where V is the set of instructions, and E represent the precedence constraints between instructions. A *task* is a chain of instructions in the dag, starting at the first instruction after a *spawn* edge, and ending at the last instruction before a *synchronization* edge. Edges between instructions in a task are called *continuation* edges. Such a chain of instructions is sometimes also referred to as a *thread* [27] in related work. We will refrain from using the term thread to describe such a chain of instructions in this work to omit confusion with the worker threads, which are used by the scheduler to execute tasks. A *strand* [61] is a subsequence of instructions in a task that contains no parallel control (no spawn and synchronization edges).

We represent task graphs by a formalism similar to the one used by Blumofe and Leiserson [27]. Figure 3.1 shows an example dag, with each Γ_i representing a separate task. Horizontal edges represent continuation edges inside a task, edges pointing upwards are spawn edges, and edges pointing downwards are synchronization edges.

In a sense, such a task graph can be seen as two separate graphs, one describing the relationship between single instructions, and one describing the relationship between tasks. Two tasks connected by a spawn edge are called *parent* and *child* task, where the parent is the task, which is the source of the spawn edge. An *ancestor* of a task is any task from which the given task can be reached only by spawn edges between tasks.

The node v_1 is the root of the computation. In our model, each node may have at most two outgoing edges, one of which has to be a continuation edge. We allow for an arbitrary large number of ingoing synchronization edges to a single node. We restrict our model to be *terminally strict* [4], which means that synchronization between tasks will occur whenever a task terminates, and that a task that terminates is only allowed to synchronize with an ancestor. Figure 3.2 shows an example terminally strict dag. This can be contrasted with *fully strict* (Figure 3.1) and nested parallel computations, where a task always has to synchronize with its parent. Whenever a terminally strict computation branches (due to a spawn operation), there is a single synchronization point at which these branches unite again. All additional branches created between the spawn and its corresponding synchronization point are required to synchro-

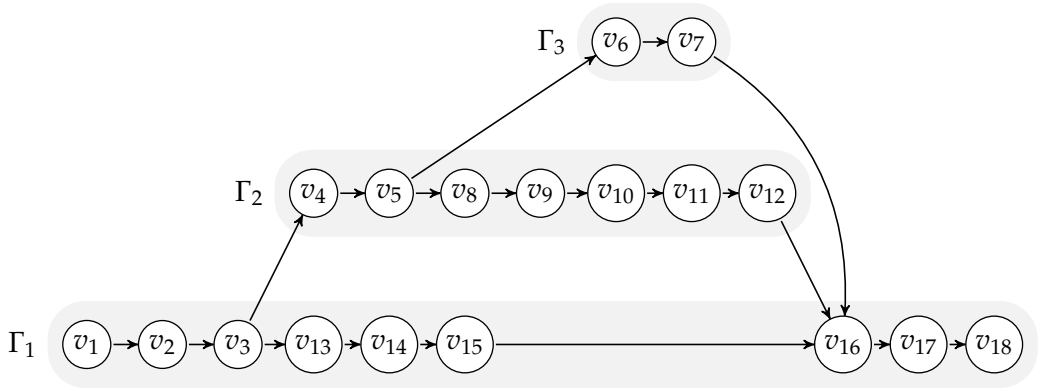


Figure 3.2: A terminally strict dag.

nize no later than the synchronization point. To put it in other words: Whenever a node in the dag has two successors, these represent two branches $B_1 = (V_1, E_1)$ and $B_2 = (V_2, E_2)$, which are disjoint subgraphs of G. In terminally strict computations, both branches have exactly one direct successor node $\{v \in V \setminus (V_1 \cup V_2) | (\exists e_1 = xv, e_2 = yv \in E) \implies x \in V_1 \wedge y \in V_2\}$.

3.2.2 Restrictions to execution

In our model, we also restrict the way in which a task graph is allowed to be executed on a real machine. As is common in fine-grained task scheduling systems, the execution of a task is non-preemptive, meaning that the runtime system is not allowed to migrate a task to another processor or to pause the execution of a task in favor of another at an arbitrary point. Instead, a task can only be descheduled (paused), whenever it spawns another task (to execute the spawned task), or when it has to synchronize with other tasks, and one of these tasks is still executing. The Pheet framework presented in Chapter 6 is even stricter in the sense, that a task that has been descheduled due to a spawned task or due to a synchronization point cannot be migrated to another processor. Even though we do not place this restriction on the general model, we discuss the theoretical implications of this restriction in Section 3.2.3.

There are two main policies that a scheduling system can employ when a new task is spawned. It can either continue executing the current task, and leave the spawned task to be executed later, potentially by another thread, or it can deschedule the current task, and start executing the spawned task, leaving the original task to be executed later, potentially by another thread. The former is called the *help-first* policy, and the latter the *work-first* policy [68]. The work-first policy has the advantage of behaving more similar to a sequential execution, a property useful both for theoretical bounds (see Section 3.1.3), as well as for programming patterns that imitate the behaviour of a sequential program, like reducer hyperobjects (see [61] and Section 2.8.2).

Priority scheduling, on the other hand, which is described in Section 2.5, is a programming pattern that is also useful in a sequential setting, but lends itself well for parallelization. It can be used for applications where programmers previously had to manually maintain a pool of work-units for which a preferred execution order exists, which is not a simple depth-first execution order. In such a priority scheduled application it is desirable, even in its sequential semantics, that each task first generates all its subtasks, before choosing a subtask to execute. Only in this way a priority ordering between subtasks of a task can be established. This necessitates a help-first scheduling policy. Since priority scheduling is an integral part of this work, we focus on help-first scheduling, and provide theoretical bounds, previously only known for work-first scheduling, where possible.

3.2.3 Migration of descheduled tasks

In our model of non-preemptive scheduling there are only two cases in which a running task can be paused and descheduled in favor of another task: (i) when a new task is spawned and the scheduler continues with the execution of the newly spawned task, or (ii) when a task needs to wait for other tasks to finish. For both cases, the bounds presented in this chapter are based on the assumption that an idle thread can pick up such a descheduled task and continue executing it.

In practice, this is not supported in the Pheet framework. To support this, Pheet applications would either need to be implemented in *continuation passing style*, similar to older versions of Cilk [83], or Pheet would need to implement additional synchronization mechanisms specifically for this case, thereby increasing complexity of the framework and the cost of synchronization. While continuation passing style can be hidden from the programmer by having a compiler transform a program into continuation passing style, as it is done in newer versions of Cilk [83, 97], this is not feasible for standalone libraries like Pheet. In addition, forbidding the migration of a task in all cases, allows tasks to take advantage of thread ids to synchronize accesses with other threads, an issue that was also relevant for the design of OpenMP [16], as well as locality information as provided by *places* (see Section 2.4.1) to optimize memory accesses.

There are two main issues arising from forbidding the migration of descheduled tasks. First, it only works with a help-first scheduling policy, since in a work-first scheduling policy all parallelism arises from descheduled tasks being made available to other threads for execution. In our case, since we need to rely on help-first scheduling for priority scheduling, this is not a problem. The second issue is that whenever a task waits for other tasks to finish, the thread may choose to execute another task in the meantime. This can lead to the task being executed blocking progress of the waiting task, potentially increasing the length of the critical path.

In practice, the second issue is not as problematic as it may seem. In most cases, there is either enough parallelism to ensure that progress is not blocked by the additional task that is executed, or the task that blocks the execution is one of the tasks the other task is waiting for anyway. The use of terminally strict computations also allows to reduce the number of synchronization points in comparison to fully strict scheduling. This problem is even smaller when relying on work-stealing, as shown in the following lemma:

Lemma 3.2.1. *As long as a thread only executes locally spawned tasks it cannot block progress of its synchronization condition. (Assuming the standard LIFO execution order for local execution, and FIFO order for steals.)*

Proof. In terminally strict executions, if a task needs to synchronize at a certain point, so need all tasks spawned transitively by it or after it until the synchronization point. Due to the local LIFO execution order, all tasks spawned from the given task, and all transitively spawned tasks will be executed by the local thread before any other tasks. For another thread to steal such a spawned task, there cannot be any older tasks available at the same thread since tasks are stolen in FIFO order. Therefore, if a task relevant for the synchronization point is stolen this means that no tasks irrelevant for the synchronization point are available at that thread.

In the end, if the next task to execute is not relevant for the synchronization point, it is guaranteed that the synchronization condition is met, since all relevant tasks must have been executed locally, and no tasks have been stolen. Otherwise the local task queue of the given thread must be empty. □

Based on this knowledge it is also possible to extend work-stealing schedulers to only allow stealing of tasks, if the stolen tasks are required for the synchronization condition. To

find out whether a stealing victim has tasks relevant for the stealing condition, it is sufficient to check whether an arbitrary task in the victim's deque is relevant. This is shown by the next lemma:

Lemma 3.2.2. *For any thread other than the thread waiting for the synchronization condition either all or none of the tasks in the thread's task queue are required for the condition.*

Proof. In work-stealing, a thread will only steal work if its deque is empty, and it will steal a single task. The deque of that thread will then fill only with descendants of that task. Due to the terminally strict condition these descendants will be required for the synchronization point, if their parent is required. Therefore either all spawned tasks will be required or none. □

While a work-stealing scheduler that is only allowed to steal tasks relevant for its synchronization point could help guarantee that progress of a task with synchronization cannot be blocked by an independent task, forbidding the execution of independent tasks can reduce the utilization of processors in cases of sufficient parallelism. Also, these results do not apply to more complex schedulers like steal-half work-stealing schedulers (see Section 3.6) and priority schedulers.

In our case, due to the negligible practical implications, we decided on forbidding the migration of descheduled tasks. While we do not encourage doing this in general for task schedulers, we believe there are cases where the costs of supporting such a feature outweigh the gains.

3.3 Space Bounds

As described in Section 3.1.3, first space bounds for greedy task schedules were given by Blumofe and Leiserson [26]. They showed that for any strict computation there exist greedy schedules that use $O(S_1P)$, S_1 being the space used by a sequential execution, and P being the number of processors. Such greedy schedules are both time- and space-efficient. In later work [27], they showed that for fully-strict computation, work-stealing fulfils those space bounds. These results for work-stealing were generalized to terminally strict computations by Agarwal et al. [4].

All discussed space bounds have in common that they assume a *work-first* schedule, where a task is immediately executed by the thread that created it, while the continuation is left for other threads to execute. The space bounds by Blelloch et al. [22] do not place this requirement, but only work for nested parallel executions, on task graphs with constant out-degree under a scheduler that is wasteful with regard to communication.

We base the proofs of our space bounds on a dag that we call a *task space graph*, as shown in Figure 3.3. A task space graph is a task graph, where each edge is assigned an edge weight. Positive edge weights represent additional memory that is allocated by the next instruction, and negative edge weights represent memory that is freed.

The space usage of any state in any parallel execution can be represented by a connected sub-graph $G' = (V', E')$ of the task space graph that includes the root node of the task graph. The graph is restricted in the sense that for each node v it will only contain an outgoing edge $e \in E'^{+}_{v}$ if it contains all ingoing edges of the same node: $e \in E'^{+}_{v} \implies E'^{-}_{v} = E^{-}_{v}$ (an instruction will only be executed if all dependencies are satisfied). The sum of all edge weights in G' is the total amount of space usage at this point of the execution. A graph is a valid task space graph if for all valid states the space usage is greater or equal zero for all valid states.

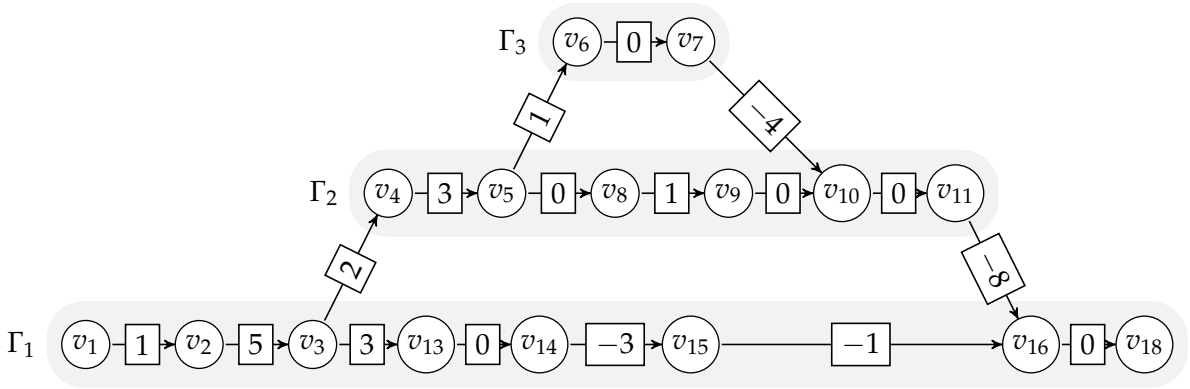


Figure 3.3: A task space graph.

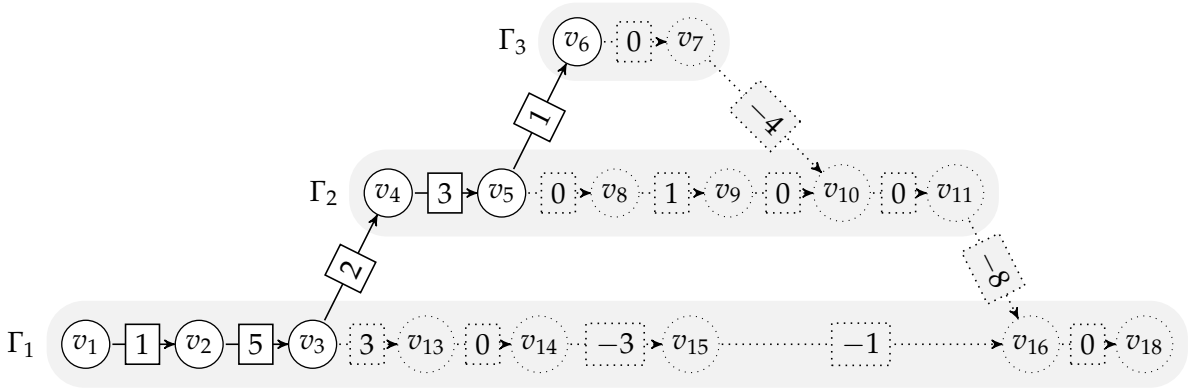


Figure 3.4: The state of a sequential execution that uses the maximum space, represented by a sub-graph of the task space graph.

3.3.1 Sequential execution

A state in a sequential execution can also be represented as a sub-graph of the task space graph. Such a sub-graph represents a state in a sequential execution if and only if on every node that has an outgoing spawn edge, the outgoing continuation edge is only included in this sub-graph if the spawn edge is included, as well as all edges and nodes reachable from the spawn edge up until the corresponding synchronization edge. The maximum space usage of a sequential execution, S_1, can be found by finding the sub-graph with the maximal sum of edge weights, which represents a valid state in a sequential execution, as shown in Figure 3.4.

Lemma 3.3.1. *A sub-graph of the task space graph of a terminally strict task graph represents a state in a sequential execution if and only if on every node that has an outgoing spawn edge, the outgoing continuation edge is only included in this sub-graph if the spawn edge is included, as well as all edges and nodes reachable from the spawn edge up until the corresponding synchronization edge.*

Proof. We prove that any state in a sequential execution can be represented by such a sub-graph by induction on the sub-graph. At the beginning of the execution, the sub-graph only contains the root vertex and no edges. On every step of the sequential execution a single edge rooted at a vertex in the sub-graph is added, as well as the vertex the edge points to, if it is not already in the sub-graph. The graph is constructed in a depth-first manner, and preference is given to spawn edges over continuation edges.

For the first state in the execution, no edges are included in the sub-graph, so the lemma is trivially true. Whenever a spawn edge is included, as well as a continuation edge from a node that has no outgoing spawn edges, the criteria are also fulfilled. A continuation edge from a node that also has an outgoing spawn edge can only be added when the spawn edge has been explored. Due to depth-first execution this requires all edges only reachable from the spawn edge to be included as well before the continuation edge is added. Due to the terminally strict property of the task graph this includes all edges reachable from the spawn edge up until the synchronization edge. □

The inverse, that any such sub-graph represents a sequential state, is trivially true, since the definition of the graph is based on the construction rules.

3.3.2 Work-first schedules

For the sake of completeness we now present space bounds on work-first schedules based on task space graphs. The bounds are well known [27], of course, but were not obtained based on task space graphs, which are a contribution of this thesis.

Similar to sequential executions, a state in a work first execution can be represented by a certain group of sub-graphs of the task space graph. Work-first executions follow the rules of a sequential execution, therefore the same rule as for sequential executions applies to continuation edges from nodes with spawn edges. Due to parallelism, this rule can be violated up to $P - 1$ times for any state as shown in Lemma 3.3.2.

Lemma 3.3.2. *A sub-graph of the task space graph represents a state in a work-first execution, if and only if for all except at most $P - 1$ continuation edges the criteria for states of a sequential execution are fulfilled.*

Proof. Threads in a work-first execution can have two states: *busy* and *idle*. A busy thread is processing some task, and the next instruction it will execute is the instruction following the previous instruction it executed in a sequential execution. Once a busy thread cannot execute the next instruction in sequential order, which can happen if that instruction was executed by another thread, the thread will become idle.

Idle threads look for a continuation that is available to execute and will not be executed by a busy thread next. Such continuations become available whenever a task is spawned, since the thread that spawned it executes the spawned task first. Such a thread will become busy, and will execute the continuation next.

Since any such continuation can be chosen by an idle thread, up to $P - 1$ continuation edges in the task space graph may not fulfil the criteria of a state in a sequential execution. A thread may only become idle again whenever it encounters a continuation that was already executed by another thread, thereby reducing the number of continuation edges in the task space graph that do not fulfil the criteria of a state in a sequential execution by one. □

We will now show that work-first executions will never use more than $S_1 P$ space based on task space graphs.

Theorem 3.3.3. *The space usage of a work-first execution is upper bounded by $S_1 P$ space, where S_1 is the space usage of a sequential execution, and P is the number of processors.*

Proof. We prove this by induction on P, the number of processors. For the base case, $P = 1$ the space bound is trivially fulfilled, since there are no divergent continuation edges. We now show that if another processor is added, the bound still holds. From the task space graph of a

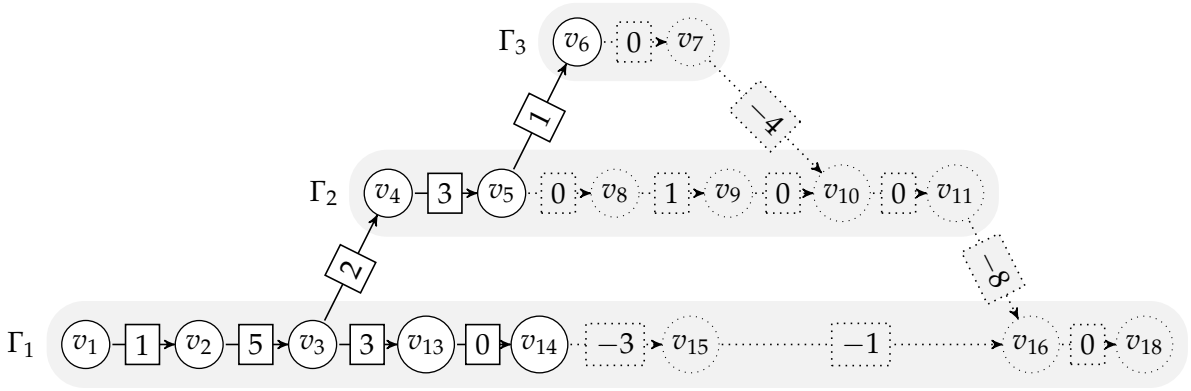


Figure 3.5: A state in a work-first execution represented by a sub-graph of the task space graph.

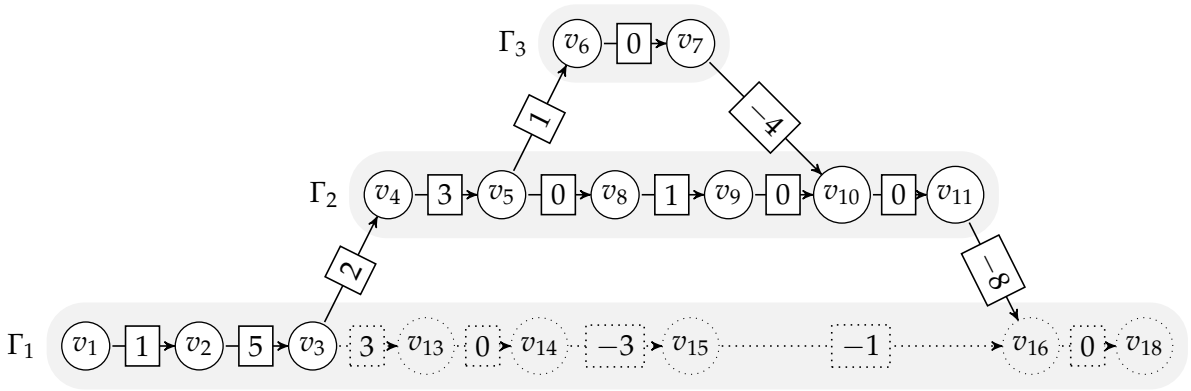


Figure 3.6: A valid sequential state right before the divergent continuation edge between v_3 and v_{13}.

parallel execution we can extract a sub-graph that resembles a state in a sequential execution. We know that the sum of all edges in any such sub-graph cannot be greater than S_1.

We also know that there exist at most $P - 1$ edges that do not fit a sequential execution. As an example, in the graph from Figure 3.5, the edge from v_3 to v_{13} diverges from a sequential execution. For each divergent edge we can construct two sub-graphs, both representing states in a sequential execution of the same task graph. The first, which is shown in Figure 3.6 sub-graph represents the state in a sequential execution right before the divergent continuation edge would be processed. The second sub-graph, shown in Figure 3.7, includes all edges of the first sub-graph as well as all edges reachable from the divergent continuation edge that conform to a sequential execution. The difference between the sums of edge weights of the second and the first sub-graph is the additional space used due to one additional processor. Since both sub-graphs represent states in a sequential execution, both must have a total edge weight w in the range $0 \leq w \leq S_1$. Therefore the difference between the sums of edge weights of two such sub-graphs cannot be greater than S_1. Since there are at most $P - 1$ divergent continuation edges by Lemma 3.3.2, and each divergent continuation edge can contribute at most S_1 additional space, the total space bound is $S_1 P$. □

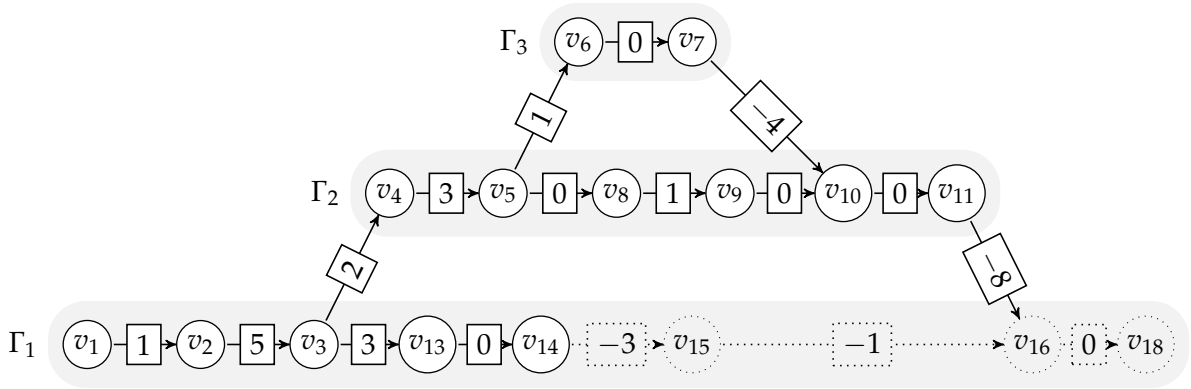


Figure 3.7: A valid sequential state that includes the divergent continuation edge between v_3 and v_{13} and all edges from the state of the work-first execution reachable from it that conform to a sequential state.

3.3.3 Help-first schedules

In our model of help-first scheduling, a task is always executed to the end, before the next task is chosen for execution. This next task is chosen in the same manner as for work-first scheduling. The main difference in the task space graph is that a spawn edge can only be part of a sub-graph representing a state in a help-first execution if the continuation edge starting at the same node, and all following continuation edges in the same strand are also in the sub-graph.

Theorem 3.3.4. *The space usage of a help-first execution for tasks is upper bounded by $S_1P(d-1)o$ space, where S_1 is the space usage of a sequential execution, P is the number of processors, d is the maximum spawn depth of the task graph, and o is the maximum out-degree of tasks in the graph (a task may spawn up to o sub-tasks).*

Proof. Similar to work-first executions (Theorem 3.3.3) we calculate the bound for help-first execution by the number of continuation edges that diverge from a sequential execution. A help-first execution by a single thread can have up to $(d-1)o$ diverging continuation edges, d being the spawn depth of the task graph, and o the maximum out-degree of tasks. This comes from the fact that each task is executed to the end, and contains up to o task spawns. This can happen $(d-1)$ times, since the task at depth d will not spawn any new tasks. All other threads will pick up a spawned task, and by executing the task can create up to $(d-2)o$ diverging continuation edges. In total there can be no more than $P(d-1)o$ diverging continuation edges, giving an upper space bound of $S_1P(d-1)o$. □

Many task-parallel algorithms (e.g. most divide-and-conquer algorithms) have a constant out-degree for tasks, making it possible to improve the bound to $O(S_1Pd)$. Furthermore, for most applications the space used by all continuation edges of a task is $O(os_1)$, where o is the out-degree of the task, and s_1 the space used by the continuation edges until the first spawn edge. This comes from the fact, that memory is mainly allocated before a spawn to accommodate all data required for the spawned task. All memory required directly by a task is typically allocated at the beginning of the task, before the first spawn. If all spawned tasks have memory requirements within a constant factor of each other, the whole task will allocate $O(os_1)$ memory for its subtasks. This enables us to improve the space bound to $O(S_1Po)$ for such applications, and to $O(S_1P)$ if these applications also have constant out-degree for tasks. Such applications are fairly common, in fact this bound applies to all our benchmark

applications in the Pheet benchmark suite (Chapter 7), except for the applications relying on priority scheduling like single-source shortest path.

3.4 Priority Scheduling

Priority scheduling differs from normal task scheduling due to the fact that the order in which tasks are executed is decoupled from the order in which they are created. In a normal task scheduling system, the order is mainly dependent on the policy of the scheduler, of which the most prominent ones are the *work-first* and *help-first* policies [68]. These policies determine the behaviour of the application regarding time, space and communication.

With priority scheduling, on the other hand, the responsibility for the behaviour of the application is put into the hands of the programmer. Depending on the application, they enable the programmer to improve the runtime behaviour, by saving time, reducing space usage and/or reducing the communication required by the application.

Due to the shift of responsibility in favor of the programmer it is hard to provide general bounds on the behaviour of priority schedulers. Instead, the design of priority scheduled applications requires programmers to consider space requirements. Care has to be taken, since the concrete bounds can also vary depending on the guarantees the scheduler gives on the priorities, which depends strongly on the semantics of the data structures used by the scheduler, as discussed in Section 5.1. In the following sections, we will discuss bounds for common classes of priority scheduled applications.

3.4.1 Semantics

In priority scheduling, a (partial) ordering between all *tasks in flight* (tasks that have been spawned, but not yet executed) is established by the programmer. This ordering can vary between threads due to locality considerations. We use the term *prioritize* to describe a relation between two tasks. We say a task is *prioritized* over another task if it precedes that task in the ordering.

To improve scalability, the scheduling system is allowed to break the order according to certain relaxations that reduce the need for synchronization. Depending on the (type of) application, these relaxations may break certain bounds. For each application and class of applications we will discuss the relaxations that can be safely used with them.

3.4.2 Local depth-first schedules

One common class of priority scheduled applications is the class of *local depth-first schedules*. These kinds of schedules are typically used for applications that work well with standard work-stealing schedulers, but where it can be of advantage if subtasks of one task are executed in a specific order rather than the order determined by the scheduling policy. Examples include, but are not limited to, the graph bipartitioning, quicksort and prefix sums applications in Sections 7.4, 7.5 and 7.6.

Main Restrictions

The main restriction placed on local depth-first schedules is that each thread is required to prioritize tasks spawned by itself over tasks spawned by other threads. This means that, for each thread, locally spawned tasks need to be executed before all tasks spawned by other threads. Furthermore, for tasks spawned by the same thread, tasks with higher depth in the dag of tasks are prioritized.

This class of priority schedules does not place any restrictions as to how a thread prioritizes tasks at same depth spawned by the same thread. Also, no restrictions are placed on how a thread prioritizes tasks spawned by other threads.

Lemma 3.4.1. *In local depth-first schedules tasks in flight spawned by the same thread with same depth always have the same parent.*

Proof. We prove this by contradiction. Assume it would be possible for two tasks A and B with different parents spawned by the same thread to be in flight at the same time. We further assume w.l.o.g. that A is spawned before B. For B to be spawned before A is executed, it is required for the parent of B to execute before A. This would either require the parent of B to have a depth greater or equal the depth of A, which contradicts that A and B have the same depth, or that A was spawned by a thread other than the thread executing the parent of B, which contradicts that A and B were spawned by the same thread. □

Space Bounds

It is obvious that the behaviour of a local depth-first priority schedule is very similar to a help-first schedule: Each task is executed to its end, thereby spawning all this task's children. Each thread executes spawned tasks in a strict depth-first manner. Therefore, analogous to help-first schedules, space bounds can be calculated based on the number of continuation edges that diverge from a sequential state in a state represented by a sub-graph of the task space graph. This yields an upper bound on space usage of $S_1P(d-1)o$, where S_1 is the space usage of a sequential application without priorities, P the number of threads, d the maximum spawn depth, and o the maximum out-degree of tasks.

As with help-first schedules, these bounds can be improved for applications with constant out-degree of tasks, as well as for applications where the space used by all continuation edges of a task is $O(os_1)$, o being the out-degree of the task, and s_1 the space used by the continuation edges until the first spawn edge. If both criteria are fulfilled, a priority scheduled application with a local depth-first schedule will use $O(S_1P)$ space. These criteria are fulfilled for all our priority scheduled Pheet benchmarks with local depth-first schedules.

Allowed Relaxations

Local depth-first schedules do not require much synchronization, since, similar to work-stealing, only locally spawned tasks are executed by each thread until there a none left. For this reason it is sufficient if each thread is guaranteed not to skip any locally spawned task when prioritizing tasks, a property that is fulfilled by all data structures presented in this work (but not for the spray-list [10], a concurrent relaxed priority queue by Alistarh et al.).

3.4.3 Global depth-first schedules

In a *global depth-first execution* all threads will prioritize tasks with the highest spawn depth, regardless of the thread that spawned the task.

Space Bounds

While, on average, a global depth-first priority schedule can be expected to use less space than a corresponding local depth-first schedule, the upper bound on space cannot be improved without additional constraints to the scheduler, since theoretically each thread can execute one task at each depth before a deeper task becomes available. Nonetheless, the similarity to

help-first schedules allows to again upper bound the number of continuation edges diverging from a sequential state in a sub-graph of the task space graph to $P(d-1)o$.

To obtain a stricter bound for global depth-first priority schedules a scheduler must be allowed to preempt a task at memory allocations, similar to the scheduler by Narlikar and Blelloch [108].

Allowed Relaxations

As with local depth-first schedules, a global depth-first schedule only requires that no locally spawned tasks are ignored by any thread. With this restriction, the worst case execution coincides with a local depth-first execution, which, as discussed in Section 3.4.2, still has the same space bounds. In practice, priority queues with stronger guarantees will lead to less space usage on average, since threads will execute tasks at higher depths first, but will not improve the worst-case.

3.4.4 Prioritized work pool

There exist applications, like our parallel SSSP benchmark in Pheet (Section 7.8), where the priority scheduler takes the role of a priority queue used to decide which work unit to process next.

Space Bounds

For prioritized work pools, the model of a sequential execution is different than for other schedules, because even in the sequential case, a unit of work (task) is first completely processed, and all its successors put into the priority queue (all subtasks are spawned), before another unit of work is processed. A sequential unit based on a prioritized work-pool uses S_1 space.

In a parallel algorithm it is necessary for more than one unit of work being processed at the same time, thus violating the sequential order of execution. This has to be taken into account for the space bounds. We introduce S_1^*, the space bound of the sequential application if the priority queue is replaced by a bag that returns items in an arbitrary order (for Dijkstra's algorithm for SSSP, $S_1^* = S_1$). We assume for our proofs that the priority queues are space-optimal and use space linear in the number of items stored in them.

Lemma 3.4.2. *An application based on a concurrent prioritized work pool with purely local ordering semantics (as defined in Section 5.1.2), will use at most $O(S_1^* P)$ space, where S_1^* is the space bound for a sequential execution with a bag, and P the number of worker threads (processors).*

Proof. With local ordering semantics, each thread will behave no worse than a sequential execution on a bag. Apart from that, no guarantees can be given on the space, so each thread can use up to $O(S_1^*)$ additional space. □

Lemma 3.4.3. *With a ρ-relaxed priority queue (see Section 5.1.3), the bound can be improved to $O(S_1^* + P + \rho)$, under the assumption that processing a single unit of work uses constant space.*

Proof. Since we are comparing to a sequential execution on a bag, the order in which tasks are executed by multiple threads cannot influence the space bound. The space bound can thus only be influenced by the size of the priority queue, and the additional space used for processing each unit of work. Under the assumption that each unit of work uses constant space to process, a parallel execution will require $O(P)$ additional space for each thread to process a unit of work concurrently.

A ρ-relaxed priority queue can use $O(\rho)$ additional space for storing items compared to a non-relaxed priority queue, since up to ρ items are not synchronized with other threads. □

The bound for a linearizable priority queue is $O(S_1^* + P)$ by Lemma 3.4.3, since a linearizable priority queue is ρ-relaxed priority queue with $\rho = 0$. Our current implementations of ρ-relaxed priority queues presented in Chapter 5 are not space optimal and will thus always use $O(S_1^* P)$ space similar to containers with local ordering semantics. In future work we plan to work on space-optimal ρ-relaxed priority queues.

3.5 Victim Selection

In the original work-stealing algorithm, a thread will steal work from a random *victim* whenever it runs out of work. While this randomized work-stealing algorithm is provably time- and space-efficient [27], concerns exist about its scalability on large-scale systems, particularly distributed memory systems, and NUMA systems, where the cost of communication between different processing units can vary. Previous work on communication bounds for randomized work-stealing suggests good scalability [1, 27] in relation to a sequential execution by bounding the number of deviations from a sequential execution, but assumes that communication costs are constant.

A detailed analysis of victim selection for parallel depth-first search was performed by Sanders [118]. His analysis includes both synchronous and asynchronous algorithms for randomized victim selection.

3.5.1 Deterministic victim selection

The *deterministic victim selection* (*DVS*) algorithm by Varisteas and Brorsson [139] is an alternative approach to victim selection, where a machine is modelled as an n-dimensional grid, and each thread deterministically selects a victim from its neighbours. Varisteas and Brorsson were able to show improved performance compared to a randomized victim selection policy. It is not clear however how this policy influences the communication costs exactly. Furthermore, since in DVS work can only be stolen from direct neighbors, there can be a delay until an idle worker thread gets a chance to steal a task. Since there is a delay between a worker starting to execute the last task in its queue, and its next steal attempt, there does not seem to be a way to bound this delay. Also, the delay can be expected to grow with the number of processors in a system. Thus, the scalability of DVS to large systems is questionable.

A victim selection scheme for parallel depth-first search algorithms that bears some similarities to DVS was presented by Rao and Kumar [94]. Rao and Kumar have shown that a victim selection algorithm that only selects neighbours in a hypercube as victims will require the problem size to grow polynomially with the number of processors to maintain efficiency in certain cases. This issue was resolved by Sanders [118] by regularly randomly permuting work between all processing units. While this can ensure a good load-balancing, the high additional communication costs make it unclear whether this algorithm is useful in practice. Also, the blocking collective nature of permutations makes this algorithm unsuitable for multiprogrammed environments.

3.5.2 Hierarchical victim selection

In our schedulers [142, 143, 145, 146, 148] we rely on a *hierarchical victim selection* policy. This includes non-work-stealing schedulers that rely on *spying* instead of stealing (see Section 3.6). Hierarchical victim selection is a semi-random victim selection policy where randomness is

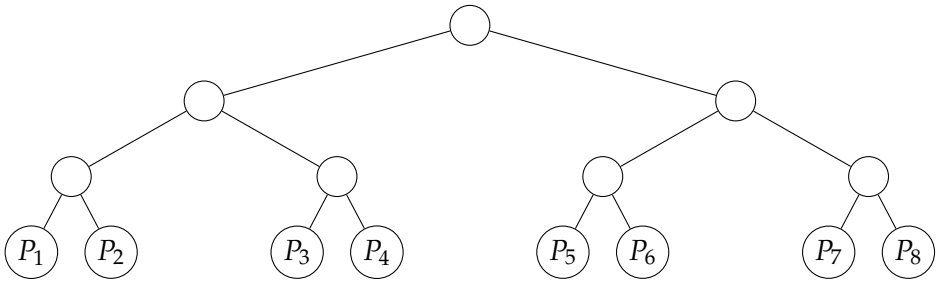


Figure 3.8: An example processor hierarchy.

used to ensure that the chances of a worker thread finding work are similar to randomized work-stealing. At the same time, victim selection is guided by the machine hierarchy, giving preference to victims with smaller communication costs.

For hierarchical victim selection, the scheduling system must be divided into different *places*, as described in Section 2.4.1. Each worker thread of the scheduler is associated with a different place, and is bound to a specific processing unit. The machine is modelled as a binary tree as presented in Figure 3.8, where each place is a leaf in the binary tree. The distance between places in the tree is used as a distance metric for hierarchical victim selection.

Whenever a victim needs to be selected, hierarchical victim selection will select a random victim out of all victims with the same distance d. The victim selection algorithm will first select a victim with $d = 1$, and then increment d every time a steal/spy attempt fails. A successful steal/spy attempt will reset d to 1. Also, d will wrap around to 1 whenever the maximum distance in the hierarchy is reached.

3.6 Stealing Policies

In the classical work-stealing algorithm a thread that runs out of work selects a victim, steals its oldest task, and executes it immediately. For divide-and-conquer applications this strategy is relatively efficient, since the oldest task can be expected to create a large amount of work. As soon as work was spawned from the stolen task, the thread that stole work can also be used as a stealing victim, thus increasing the chances of other threads finding work.

Berenbrink et al. [20] have shown, however, that for general task-based applications, stealing half the work from a victim's deque is an efficient policy. Hendler and Shavit [75] have presented a work-stealing deque that allows to steal half the work from a victim with only a constant number of synchronization operations. One disadvantage of stealing more than one task, however, is that a task can be stolen more than once, and it can even occur that a thread has to steal its own tasks back. This can lead to more deviations from a sequential execution compared to the original work-stealing algorithm resulting in more non-local memory accesses. Also this puts higher stress on the work-stealing deques, since a task can be inserted into and removed from multiple deques during its lifetime. If only a single task is stolen, this task is executed immediately and does not need to be reinserted into a deque. Better load-balancing can be achieved by stealing half the actual work instead of half the tasks, which can be done with strategies (Section 2.7), but this requires the programmer to inform the scheduling system about the granularity of tasks.

An alternative approach to ensure good distribution of work is to let each worker thread store its last victim. Whenever a thread runs out of work, it will first attempt to steal work from its last victim, and only select a new victim if the steal attempt fails. When a victim is

selected using a victim selection policy (see Section 3.5), and this victim is out of work, the last victim of the victim is first checked for work, before selecting another victim based on a victim selection policy. Thus, a thread that steals only a single task will still help to propagate work from its victim, while preserving the properties of work-stealing where only a single task is stolen.

Finally, we present *spying*, an alternative to stealing, where a task can be stored with multiple worker threads. As soon as one thread starts executing a task, it will be invalidated for all other threads to ensure a task is only executed once. Whenever a thread runs out of work, it will select a victim according to its victim selection policy and copy all tasks from its victim. Spying allows for an even better distribution of tasks compared to steal-half work-stealing, since a worker thread has the chance to spy a specific task at more than one victim. However, since tasks can be stored in more than one queue, queues will be longer on average than work-stealing deques.

3.7 Mixed-mode Scheduling

The mixed-mode parallel programming model presented in Section 2.6 allows a task to be parallel, thus requiring more than one processor to execute. For our implementation of the mixed-mode parallel programming model in Pheet we use the team-building algorithm from Section 4.4. The team-building algorithm is allowed to deviate from the execution order of a normal work-stealing applications for tasks that require a different number of threads to execute: A task that requires a smaller amount of threads can be executed before a task with larger threads requirement, regardless of their original execution order. This violates some of the properties necessary to obtain the work-stealing bounds. In this section we discuss what bounds can still be shown for mixed-mode scheduling, and how the algorithm can be improved to achieve better bounds.

3.7.1 Time bounds

We first look at the time bounds of work-stealing schedulers with team-building. Time bounds for greedy schedulers can be obtained as follows [26]: Either all threads are busy processing work, which cannot use more than T_1/P time in total, where T_1 is the time needed by a sequential execution, and P the number of threads, or at least one thread is guaranteed to process the critical path, which takes T_∞ time to process. A parallel execution with a work-stealing scheduler is thus guaranteed to take no more than $O(T_1/P + T_\infty)$ time.

Similar time-bounds can be achieved for a greedy mixed-mode parallel schedule, where at each time step the scheduler will greedily choose a set of tasks to execute that maximizes the utilization of threads. A greedy schedule will guarantee that, given sufficient parallelism, at least $P/2 + 1$ threads will process work at each time-step. Thus a mixed-mode parallel schedule can be executed greedily in $O(2T_1/P + T_\infty)$ time.

Unfortunately, the team-building scheduler cannot fulfil this property, since tasks with smaller thread requirements will be executed before tasks with larger thread requirements. Thus, a single task requiring a single thread to execute can block the execution of a task that requires P threads. Due to this, the only time bound that can be given for a team-building scheduler is that it will execute in $O(T_1)$ time, since at each time step it is guaranteed that at least one thread will execute a task.

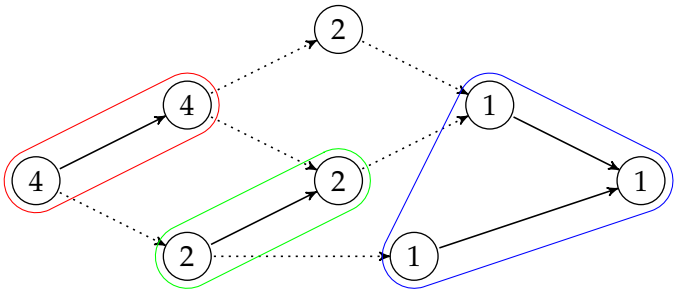


Figure 3.9: Connected components of tasks with same space usage.

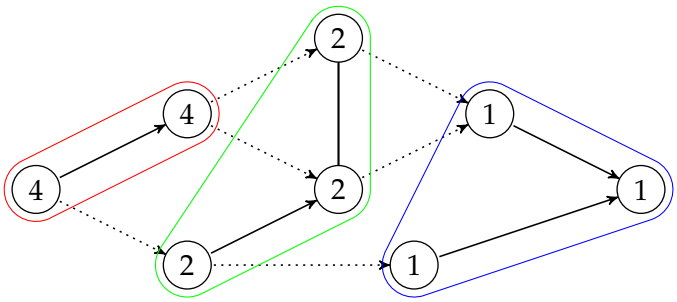


Figure 3.10: Merged components with same predecessor component.

3.7.2 Space bounds

Space bounds for greedy schedules (see Section 3.3) can be obtained by upper bounding the number of deviations from a sequential execution. To obtain space bounds for mixed-mode parallel schedules we upper bound the additional deviations compared to a task-based execution where all tasks require only a single thread to execute.

Assuming that the number of threads required by a task does not follow some regular pattern, there can be an arbitrary number of deviations from a parallel execution with only single-threaded tasks. However, it is still possible to obtain a space bound for applications based on their task graph. To obtain the number of deviations, one needs to delete all edges between tasks with different thread requirements from the task graph, as shown in Figure 3.9. The number of connected components in the graph is an upper bound on the additional deviations compared to an execution without mixed-mode scheduling. A tighter upper bound can be obtained by adding connecting edges between components that have the same predecessors in the original task graph, thus reducing the number of components as shown in Figure 3.10.

For divide-and-conquer applications, like quicksort (Section 7.5.4), where team-sizes are strictly non-increasing, or mergesort, where team-sizes are strictly non-decreasing with the depth of the task-graph, the number of additional deviations is bounded by the number of different team sizes. In practice for our mixed-mode quicksort algorithm no additional space is used, since the deviations that occur due to team-building have no influence on the space used.

3.7.3 Future work

Current time bounds on mixed-mode scheduling with deterministic team-building are weak, thus making the model only useful for well-behaved applications for which stronger bounds can be shown. It might, however, be possible to create a feasible greedy mixed-mode scheduler

that has a greedy policy with a preference to tasks with higher thread requirements, which can obtain good general time bounds for mixed-mode scheduling.

Another approach to obtain feasible time bounds is to restrict the type of applications allowed. Crown scheduling [88], for example, only works on tasks without dependencies, thus giving the scheduler the opportunity create efficient greedy schedules. Similar results should be possible for applications where tasks are only allowed to spawn sub-tasks with non-increasing or non-decreasing thread requirements.

Finally, the use of strategies can help make thread requirements more dynamic, by allowing the scheduler to set the thread requirement before starting to execute a task. This can help to work around pathological cases.

4 Data Structures and Synchronization

This chapter discusses the implementation of data structures and low-level synchronization primitives that are required to support the programming models and patterns presented in Chapter 2 and the schedulers from Chapter 3. The focus lies on non-blocking implementations with strong progress guarantees, and proofs for linearizability and the progress guarantees are provided. Unless otherwise specified, all code examples in this chapter assume *acquire/release* semantics for all operations on shared memory locations. While the actual implementations often contain additional relaxations, we believe this provides the best compromise for readability, while still allowing for efficient implementations on most of the commonly used processor architectures. All code examples are given in C++ with some simplifications to improve readability. Ordered container data structures are discussed separately in Chapter 5.

4.1 Linearizability and Progress Guarantees

Reasoning about the correctness of a synchronization algorithm requires a consensus on what behaviour is seen as correct. *Sequential consistency* [95] is a correctness condition that requires operations on a single concurrent object to take effect in program order. This property is not compositional, however, meaning that for two concurrent objects, sequential consistency for accesses to each object does not imply that the combined history for accesses to both objects is sequentially consistent [81].

Linearizability [82] is a correctness condition stronger than sequential consistency that requires all operations on a concurrent object to take effect instantaneously at some point between invocation and response. The point at which such an operation takes effect is called a *linearization point*. Since linearizable operations take effect instantaneously they are compositional, it can however be argued that this comes at an additional cost [14]. Note, however that this does not imply that an operation composed out of linearizable operations is also linearizable [35].

Progress guarantees can be used to reason about the efficiency and scalability of a synchronization algorithm. A *non-blocking* or *obstruction-free* algorithm is an algorithm, which, if executed in isolation, will finish executing in a bounded number of steps. This means that, contrary to locks, a suspended thread cannot infinitely block the progress of other threads. The *lock-free* progress guarantee is stronger in that it guarantees for at least one thread to progress in a bounded number of steps regardless of actions by other threads. However, lock-free algorithms are not *starvation-free*, since only a single thread is guaranteed to progress, but there are no guarantees as to which thread will progress. Fairness can be achieved with the *wait-free* progress guarantee, which guarantees all threads to achieve progress in a bounded number of steps.

It is always possible to construct a concurrent lock-free [80,81] and wait-free [79] algorithm for access to a concurrent object based on an arbitrary sequential algorithm. Constructing such an algorithm can incur large overheads, however, and it was generally believed that overhead for making any algorithm wait-free was too high to be practical [57, 81, 90]. Recent work by Kogan and Petrank [90] suggests otherwise by presenting a methodology for efficiently implementing a wait-free algorithm out of any lock-free algorithm.

An efficient implementation of lock-free and wait-free algorithms requires the use of operations with *consensus number* [79] greater than one, which are operations which can be used to implement a wait-free consensus protocol for more than one thread. Many algorithms require operations of consensus number P or higher. This necessitates the use of *read-modify-write* operations, which are able to both atomically read and write a value in a single statement. For the algorithms presented in this work we use *fetch-and-op* (mainly *fetch-and-add*) to atomically update a shared variable by applying an associative operation to it. In addition, we use *compare-and-swap* (*CAS*) to reach consensus between threads, since it is the only operation supported by most of the commonly used shared memory architectures with an infinite consensus number, making consensus with an arbitrary number of threads possible. A CAS is a conditional write statement that will only perform the write if the current value of the same memory location equals a specified value.

4.2 Terminology

ABA Problem The term *ABA problem* describes situations where an object in memory appears to be the same as last observed, even though it was changed to some other value in the meantime, and then back to the expected value. As an example, a stack might appear to be unchanged when a thread always finds the same item at the top as on a previous access, while in reality, items might have been removed in the meantime, and the expected object only added back to the stack later.

Contention An access to a memory location is *contended*, whenever more than a single thread attempts to modify the same location. Contention is a limiting factor for scalability, due to memory traffic, and even more for operations that cannot be combined by the hardware like CAS [70].

Happens-before relation We say an operation A *happens-before* an operation B if for any execution it is guaranteed that B will observe the effects of A. Happens-before relations are transitive.

Local An object in memory is *local* to a specific thread if it was allocated by that thread. Some algorithms allow passing ownership of an object in memory to another thread. In that case, an object is local to its current owner.

Observation A thread *observes* a specific state of a shared object by reading members of said object. Since an observation can consist of multiple reads and is not necessarily atomic, inconsistent states can be observed.

Spurious failure If an operation is allowed to *spuriously fail*, it is allowed to fail even if no external condition for failure is met. In non-blocking synchronization it is common to allow operations to spuriously fail as long as another thread is making progress. Such operations can then be used as parts of a lock-free algorithm.

4.3 Wait-free Memory Reuse

Memory management for concurrent data structures can be non-trivial, since it is often hard to guarantee that no thread is still accessing an object that is ready for deletion. For algorithms that provide lock-free or wait-free progress guarantees, the same progress guarantees need to be provided by the memory management scheme.

A wide variety of concurrent *reference counting* schemes exist [48,106,131,136,137], but the costs are prohibitive for many applications, since a read-modify-write operation is required every time a reference is created or freed. *Hazard pointers* [105] remove the necessity of using a read-modify-write operation. Instead they require a thread to read a list of hazard pointers from each thread before an object can be freed. This might lead to scalability problems on future architectures with large numbers of threads. *Quiescent-state-based reclamation* [74], *epoch-based reclamation* [59] and *drop the anchor* [29] all divide an execution into different epochs, and only allow an object to be freed once all threads have entered epochs after the point where an item became inaccessible.

In this section we present a wait-free memory reuse scheme that is based on the idea of garbage collection. The scheme is very lightweight, requiring only $O(1)$ amortized overhead for finding an item to reuse. The low cost comes at the cost of flexibility, since this scheme is restricted to the management of uniform items, which are ABA-safe, and for which there exists a way to recognize when they can be reused. Also, while the pool of items to reuse can grow, if necessary, it can never shrink.

4.3.1 The algorithm

The idea behind the wait-free memory reuse algorithm is simple: The user specifies the type of item that is stored in the memory pool, as well as a function that can be used by the memory reuse scheme to check whether an item in the pool can be reused. Each place (thread, see also Section 2.4.1) maintains its own memory pool that stores items in a circular linked list. The pool contains both reusable items and items that are in use. Every time an item is requested, the algorithm traverses the linked list, and checks for an item that can be reused. As soon as a reusable item is encountered this item is returned. If no reusable item is found, a new item is spliced into the circular linked list and returned to the user.

Listing 4.1 Our wait-free memory reuse algorithm.

```
 1: while head.next ≠ tail do
 2:   if reusable(head.next.item) then {Item is reusable}
 3:     head, tail ← head.next, tail.next {Advance both head and tail}
 4:     if head ≠ tail then {Make sure not to overtake head}
 5:       tail ← tail.next {Advance tail a second time}
 6:     end if
 7:     return head {We found a reusable item}
 8:   else {View is still in use}
 9:     head ← head.next
10:   end if
11: end while
12: {No reusable item found, allocate and splice in new item}
13: r ←new item()
14: r.next, head.next ← head.next, r
15: head, tail ← head.next, tail.next
16: return head
```

The algorithm is presented in Listing 4.1. Each thread has its own pool of items, therefore the algorithm is not required to be thread-safe. To ensure that the algorithm does not check the whole pool every time an item is requested, two pointers into the pool are maintained: *head* and *tail*. The *head* pointer always points to the last item that was checked for reuse. The *tail* pointer ensures that only two items are checked for reusability on average. The *head* pointer is never allowed to reach the *tail* pointer, therefore whenever the next element to check for reusability is the element pointed to by *tail*, the algorithm is required to allocate a new item, and to splice it into the pool. The *tail* pointer is only advanced in two cases: whenever a reusable item was found, or when a new item is spliced in. To ensure that the pool does not grow too large, *tail* needs to be advanced more than once on average whenever a reusable item is found. In our implementation we advance it twice, unless $head = tail$.

4.3.2 Bounds and progress guarantees

Lemma 4.3.1. *The memory management is wait-free assuming both the reuse check, and the memory allocation routine are wait-free.*

Proof. Each check performed by the memory reuse algorithm is followed by an advancement of *head*. The advancement of *head* is limited by *tail*, which in turn can only advance after either a reusable item was found, or a new item was allocated, which both in turn lead to a termination of the algorithm. Thus, since there is a bound on advancements regardless of what other threads do, the algorithm is trivially wait-free. □

Lemma 4.3.2. *The memory reuse algorithm has amortized complexity* $O(1)$.

Proof. We use an amortization argument, where for every reusable item that is found no more than one non-reusable item can be traversed. Since the function terminates whenever a reusable item is encountered, the number of non-reusable items that are traversed will not exceed the number of function calls.

For this, we define a potential function Φ. The potential function is defined by the number of times *head* is allowed to advance before its successor is *tail*. The pool is initialized to $head.next = tail$ in the beginning, resulting in $\Phi = 0$.

We argue that each non-reusable item that is encountered will reduce the potential by 1, since in this case *head* is advanced, but not *tail*. Every reusable item that is encountered decreases the potential by at most 1, since *head* is advanced once, and *tail* at most twice.

Whenever $head.next = tail$ all the potential has been used up, resulting in $\Phi = 0$. In this case a new item is allocated and spliced into the linked list. Since the splicing in happens between *head* and *tail*, this increases the potential by 1. Since the newly created item is guaranteed to be reusable, and is returned by the function, the increase in potential is justified. □

Lemma 4.3.3. *The number of items i that are allocated for the memory pool of a given thread will never exceed* $3n_{\max}$, *where* $n_{\max} > 0$ *is the maximum number of items in use at the same time throughout the execution.*

Proof. For our argument we reuse the potential function Φ from Lemma 4.3.2. We take an arbitrary point in time during the execution and show that it is impossible for the memory pool to contain more than $3n_{\max}$ items. The pool is initialized with a single item. We distinguish between two situations: $\Phi_c \geq n_{\max}$ and $\Phi_c < n_{\max}$, where Φ_c is the potential of the memory pool at the beginning of a *cycle*. We say the memory manager has completed a cycle whenever *head* returns to its starting position.

For the first case, $\Phi_c \geq n_{\max}$, we know that no new items will be allocated on a single cycle, since Φ cannot decrease by more than $n_{\max}$ items on a single cycle. If the total number of items in the pool $i \geq 2n_{\max} + 1$, it is guaranteed that Φ will not go below $n_{\max}$ after any number of cycles, thus the pool will not grow from this point on. The reason for this is that on each cycle $n_{\max} + 1$ items are encountered that will increase the potential. While the potential may not grow in all cases, since the potential is capped at $\Phi \leq i - 1$, when the cap is reached, the potential can only decrease by at most $n_{\max}$.

We now look at the other case $\Phi_c < n_{\max}$. Here we argue that for any cycle c, $i_c + g_c \leq 3n_{\max}$, where i_c is the size of the pool at the beginning of the cycle, and g_c the growth of the pool during the cycle. Since growth is always coupled with a decrease in potential, we can trivially infer that $\forall c, g_c \leq n_{max} - \Phi_c$. For $\forall c, i_c \geq 2n_{\max} + 1$ it is trivial to show that $\forall c, \Phi_c \geq i_c - 2n_{\max}$, leading to:

$$\forall c, i_c + g_c \leq i_c + n_{\max} - \Phi_c \leq i_c + n_{\max} - i_c + n_{\max} = 3n_{\max} \qquad \square$$

4.4 Deterministic Team-building

In this section we present the *team building algorithm* [146] which extends a (work-stealing) scheduler to support the mixed-mode parallel model presented in Section 2.6. The mixed-mode parallel model allows a task to require more than one worker thread to execute, thus necessitating for worker threads to *build a team* of worker threads, which will all execute the given task at the same time. Our team building algorithm builds upon the philosophy of work-stealing, where each worker thread operates on its own deque of tasks and only communicates with other threads when absolutely necessary. In the original work-stealing algorithm this is only the case whenever a thread runs out of work. In our team-building algorithm communication also becomes necessary once a team needs to be built to execute a task.

We call the team building algorithm presented in this section the *deterministic team-building algorithm*, since it builds upon the premise that once a worker thread decides to execute a parallel task, and starts building a team, the structure of the team is deterministically decided. Due to this, each thread that encounters a team being built can immediately determine which worker threads are supposed to be part of the team, and which id will be assigned to each thread in the team. If two overlapping teams are being built by different threads, a deterministic tie-breaking scheme is used to decide which team will be built first, and all threads that are part of both teams are guaranteed to eventually join the correct team.

4.4.1 Algorithm

We extend work-stealing to cater for mixed-mode parallelism, where tasks requiring a certain, determined number of processors or threads for their execution can be dynamically spawned. This thread requirement is denoted by r. In the standard task-parallel work-stealing setting $r = 1$ for all tasks, whereas we want to allow for any $1 \leq r \leq P$ number of required threads, P being the number of processors in the system (requirements $r > P$ are of course infeasible). Thread requirements are fulfilled by building *teams* of worker threads for tasks with $r > 1$. When a team of r threads has been formed for some task, the task can be executed. Threads in a team are numbered consecutively, such that a thread can identify the other threads in the team and communicate with them. Since in our approach each worker thread is pinned to a specific processing unit in the system, we use the terms *thread* and *processor* interchangeably.

The *deterministic team-building* algorithm is based on a machine model, where the machine is represented as a tree. The leaves in this machine model are individual processing units,

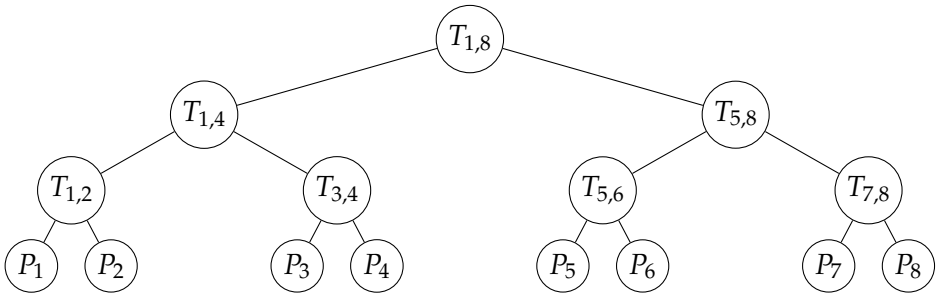


Figure 4.1: The team-building hierarchy hierarchy.

each of which is associated with a single worker thread of the scheduler. The distance between leaves can be used to estimate relative communication costs between leaves. A pair of two leaves with a distance in the tree greater than some other pair of leaves, will not have lower communication cost in the real machine. Due to the requirements of the team-building algorithm, this hierarchy is further refined into a binary tree. A similar hierarchy is used in Pheet (Chapter 6) to support *places* (Section 2.4.1) and to implement victim selection policies for schedulers (Section 3.5).

In the simplest case, which we will assume to simplify the presentation of the team-building algorithm, we assume that P, the number of processors (worker threads), is a power of two and thus the machine model a full binary tree. In Section 4.4.9 we explain how this restriction can be lifted. This binary tree can be used to group processors into teams of different sizes, as shown in Figure 4.1, where team sizes are always a power of two. The binary tree is divided into different levels starting at the leaves. At level 0 (leaves) there exist P individual tree nodes, each representing a team containing a single worker thread. At level 1, there are $\frac{P}{2}$ teams of size 2 each. At level i, there exist $\frac{P}{2^i}$ teams of size 2^i each. The topmost level is $\log P - 1$, which consists of a single team containing all worker threads.

To execute a task with a thread requirement r, a thread tries to build up a team of t processors, where $1 \leq t \leq r$. The decision to execute such a task is made by a single worker thread, which we call the *coordinator* of the team. After making the decision to execute a task, the coordinator determines the team size t by choosing the largest possible $t \leq r$. The threads in a team are numbered by the leaves in the binary tree from left to right starting at 0. Given only r and the global id of the coordinator, other threads can infer the exact structure of the team, including the id of each individual thread. As an example, for the hierarchy in Figure 4.1, for $r = 5$ and thread P_6 being the coordinator of the team it can be determined that the team will consist of the four threads $P_4 - P_7$, where the id of each thread inside the team will be its global id minus 4.

As presented here, the model is restricted to processor hierarchies where the number of processors is a power of two, and team sizes are also restricted to be powers of two. We will later show that both restrictions can be lifted, although lifting the second restriction can lead to less efficient schedules in many cases.

4.4.2 The team building scheduler

As with our other schedulers we divide our scheduling system into different *places* (see Section 2.4.1), where each place is associated with a single worker thread, and is pinned to a specific processing unit. In the following we present the data structure storing all information relevant for a specific place. Each place has a fixed (integer) id I, $0 \leq I < P$, which is used to determine the victim for work-stealing and team-building attempts as well as to compute the

local thread ids in a team. We assume that the place data structure can be accessed by other places with the help of a lookup table indexed by I, the place id. In the following we describe all members stored in the place data structure:

- The unique thread id I, $0 \leq I < P$.
- A double ended queue Q of spawned tasks. Local accesses always happen at the bottom, while stealing is done from the top of the queue. The queue needs to be able to only return tasks for certain levels in the processor hierarchy. A simple way to do this is to use $\log P - 1$ queues instead of a single queue, one queue for each level in the team-building hierarchy.
- The id c of the coordinator of the team the place is currently part of. If the place is a coordinator, $c = I$. This is always the case when scheduling tasks with $r = 1$, or when attempting to steal work from other places. An invariant of the team-building work-stealer is that a coordinator is always set.
- An integer r storing the thread requirement for the next task to be executed.
- A reference to a ready task T that can be executed by the current team. As soon as the coordinator of a team sets T, all threads in the team can start execution of the task.
- A countdown G for the ready task that is initialized to $t - 1$ before T is set ($t \leq r$ is the team size of T). Each non-coordinator thread must atomically decrement this field when it starts executing the task. As soon as $G = 0$, the coordinator can be sure that execution has started by all threads in the team, and can then reset T.

The following methods are needed for team-building. In Section 4.4.3 we show how to implement them in a lock-free manner.

- `registerThread(`I`)` is called by threads trying to join a team. It takes the id of the calling thread as parameter. The method first checks whether the given thread is eligible for the team, and if this is the case registers the thread for the team.
- When a thread wants to drop out of a team to start working on something else, it must call `dropThread(`I`)`. This tries to remove the thread from the team, and returns true on success. A call to this method is only successful if the team is not yet built. If a team has been built, only the coordinator can free threads from it.
- The coordinator can check if the team-building process has finished by `teamBuilt()`. No thread may by its own leave a team after it has been built. The coordinator can reuse and if needed downsize an already built team.
- The coordinator (only) can disband an already built team by `disbandTeam()`.
- `inTeam(`I`)` is used by threads to check whether they are still members of a team. This may not be the case if the coordinator has disbanded the team, or if the team has been resized.

Listing 4.2 The modified stealing procedure for team building (`stealTasks()`)

```
1: ℓ ← 1
2: while 2^ℓ ≤ p do
3:    x ←getVictimForLevel(ℓ) {Victim thread at level ℓ; random or deterministic depending on policy}
4:    xc ← x.c {The victim's coordinator}
5:    xcr ← xc.r {The thread requirement of the team being built}
6:    if xcr ≥ 2^ℓ then {Victim's coordinator requires this thread for execution of its task}
7:       xc.registerThread(I) {Join this team}
8:       c ← xc {and set coordinator}
9:       return
10:   else
11:      {Steal from victim instead}
12:      if Q.popappend(x.Q, 2^(ℓ−1)) > 0 then
13:         {At least one task stolen}
14:         return
15:      end if
16:      {Nothing to steal, next level in the hierarchy}
17:      ℓ ← ℓ + 1
18:   end if
19: end while
20: {No success in stealing procedure}
21: backoff()
```

4.4.3 Implementation

A key property of work-stealing is that threads do local work as long as work is available, and only resort to stealing when they run out of work. Deterministic team-building preserves this property, since each team will work on the coordinator's local queue of tasks for the current level in the processor hierarchy. Only when a team runs out of work, it is disbanded, and the modified stealing procedure started. Pseudocode for stealing procedure is shown in Listing 4.2.

A worker thread will only start to steal whenever it is not being coordinated by another thread, and it has run out of tasks to process, including tasks requiring teams. For team-building, the victim selection policy (see also Section 3.5) is required to be hierarchical in nature. The policy used in Listing 4.4 iterates through the levels in the team-building hierarchy, and selects a single victim at each level. Only if the victim at level ℓ neither has work to steal, nor is part of a team which also requires the stealing thread to join, the stealing procedure will proceed to the next level in the hierarchy. This scheme ensures that a thread in a stealing procedure will encounter a team being built with a certain probability. The probability is higher the more threads have already encountered the same team.

If the stealing procedure encounters a thread that is part of a team that the stealing thread is also supposed to be a part of, it will join the team and exit the stealing procedure. Otherwise it will attempt to steal at least one task. This is done with the `popappend(`q`, maxTeam)` method of the queue structure. It tries to pop a number of tasks from the queue q and append them to its own queue. This call is only allowed to steal tasks with a thread requirement that is at most `maxTeam`, which is $2^{\ell-1}$ in the stealing procedure. The reason for this is that there is no point in stealing a task that requires both the stealing thread and the victim to execute.

The team-building scheme is lazy, which means that a thread will only find out about

Listing 4.3 The polling procedure for threads building a team (`pollPartners(r)`)

```
1: ℓ ← 1
2: while 2^ℓ ≤ r do
3:   x ← getVictimForLevel(ℓ) {Partner thread at level ℓ; random or deterministic depending on policy}
4:   xc ← x.c {The victim's coordinator}
5:   xcr ← xc.r {The thread requirement of the team being built}
6:   if xc ≠ c then {Victim has a different coordinator}
7:     if xcr ≥ 2^ℓ then {Victim's coordinator requires this thread for execution of its task}
8:       {Conflicting teams being built, choose}
9:       if chooseTeam(c, xc) = xc then
10:        {Other team wins, switch}
11:        if c.dropThread(I) then
12:          {May fail if team at c has already been successfully built}
13:          xc.registerThread(I) {Join other team}
14:          c ← xc
15:          return
16:        end if
17:      end if
18:      {Our team wins, other threads will eventually switch}
19:    else
20:      {Steal from victim instead}
21:      if Q.popappend(x.Q, 2^(ℓ−1)) > 0 then
22:        {At least one task stolen}
23:        if c.dropThread(I) then
24:          {Try to drop from team being built}
25:          c ← I {Thread becomes coordinator and will attempt to execute task}
26:        end if
27:        return
28:      end if
29:      {Nothing to steal, next level in the hierarchy}
30:    end if
31:  end if
32:  ℓ ← ℓ + 1
33: end while
34: if ¬ c.taskIsReady(I) then
35:   backoff()
36: end if
```

a team being built whenever it selects a victim that is part of this team during the stealing procedure. This fits well with the philosophy of work-stealing, where a thread only communicates when it has nothing better to do, and no extra coordination overhead is introduced. Nonetheless, this also means that a team will only be built when each thread required for the team has finished processing its local work. To speed up this process, a helping scheme is introduced, where threads that join a team steal work from threads needed for the team. In this way, threads attempting to build a team will help other threads in this queue finish their work, so they can proceed to join the team.

The `pollPartners()` procedure shown in Listing 4.3 implements this helping scheme. Each thread that is part of an unfinished team will repeatedly call this procedure in an at-

Listing 4.4 The procedure called by a thread to get the next task to execute (`getTask()`)

```
 1: task ←⊥
 2: repeat
 3:   if c ≠ I then
 4:     {Thread is in team coordinated by another thread}
 5:     if c.T ≠⊥ then
 6:       {The coordinators task is ready, and thread is in team}
 7:       return c.T
 8:     else if c.inTeam(I) then
 9:       pollPartners(c.r)
10:     else
11:       c ← I
12:     end if
13:   else if Q.isEmpty() then
14:     if teamBuilt() then
15:       {Out of work, so team is not needed anymore}
16:       disbandTeam()
17:     end if
18:     stealTasks()
19:   else
20:     {Thread is coordinator}
21:     if teamBuilt() then
22:       T ← Q.popBottom(r)
23:       return T
24:     else
25:       pollPartners(r)
26:     end if
27:   end if
28: until task≠⊥
```

tempt to steal work requiring smaller teams from other threads. The procedure bears a lot of similarities to the `stealTask()` procedure presented in Algorithm 4.2. The main differences are that the only victims that will be selected are threads required for the team. Also, if another team being built is encountered, which also requires the participation of the stealing thread, a deterministic tie-breaking scheme needs to be employed to ensure that eventually all threads will join the same team. In our polling procedure, the tie-breaking scheme is encapsulated in the procedure `chooseTeam(`I`,` J`)`, which must be deterministic, commutative and transitive (e.g. by always choosing the thread with the lower id). We also require that `chooseTeam(`I`,` J`)` always chooses smaller teams over larger ones.

We now put the pieces together in the `getTask()` procedure shown as Listing 4.4. This procedure is called whenever a thread is ready to process the next task. It distinguishes three different situations. If the thread is not the coordinator of the team it is in it calls the `pollPartners()` procedure until either the team is ready, a task has been stolen, or the team-building cancelled due to a conflict. Otherwise, if the thread is a coordinator, it is either out of work, which means that the team can be disbanded and stealing can start using the `stealTasks` procedure, or team-building is still in progress, which means that the `pollPartners()` procedure is called, or the team is ready. In case the team is ready, the next task with thread requirement r is retrieved from the queue using the `popBottom(`r`)` method and then prepared for execution by the team.

4.4.4 Basic properties

Teams are always built out of consecutive threads. The threads that are allowed to join a team of a certain size at a certain coordinator are static and deterministic as they have to be members of the same group of processors at a certain level in the processor hierarchy. Due to the requirement that threads in a single group in the processor hierarchy are numbered consecutively, the threads in a team are numbered consecutively as well. By subtracting the (known) lowest id of a thread in the group from the thread ids, team-local thread ids in the range $[0, t-1]$ can easily be computed. If the processor hierarchy can be represented as a full binary tree, finding the id of the first thread in the team is particularly easy and requires finding only the most significant bit in the team size t. This can often be done by a hardware instruction, or in $O(\log \log t)$ steps.

An important property of work-stealing is that as long as a thread can execute tasks it does not have to communicate with other threads. With the *cohorting* optimization presented in Section 4.4.7 this property can be extended to teams, where threads in a team will never need to communicate outside the team, until no more tasks for the given team size are available in the coordinator's queue.

4.4.5 Correctness

Lemma 4.4.1. *Assume the computation is finite. A thread I has spawned a task requiring $r \geq 1$ threads. This task will eventually be executed.*

Proof. For $r = 1$ the case is clear. Tasks requiring a single thread will be popped or stolen and executed before tasks using more threads. No coordination is required before execution, so the task will eventually be executed, as in standard work-stealing.

Let $r > 1$ and assume the task is coordinated by I (it might have been stolen from some other thread). Eventually the other threads will join the team for the task as they run out of tasks requiring less than r threads. These will be executed because threads waiting for the formation of the large team help smaller teams to empty their task queues. Threads joining the team set their coordinator to I such that other threads that have to join the team eventually see that the team is coordinated by I. When all threads have joined the team the task will be executed. □

Lemma 4.4.2. *If two or more threads trying to build teams compete for threads to join their team, the conflict is resolved deterministically.*

Proof. Assume that thread x and thread y both try to build a team and compete for the same threads to join the team. Over time each of the threads will join one of the competing teams. As soon as a thread has joined a team, it will poll the other threads required for the team, and eventually see the other team. In this case, the `chooseTeam(`I`, `J`)` function is called to resolve the conflict, which is required to be deterministic and commutative. Assume the team of thread x is deterministically chosen over the team for thread y. Each thread in team y will then switch to team x as soon as it meets a thread in team x. Over time all threads will have joined team x. As `chooseTeam(`I`, `J`)` is also required to be be transitive, the argument extends to more than two conflicting teams. □

In our current implementation, `chooseTeam` selects the team with the smaller thread requirement, and selects the team with the smaller coordinator id on a tie. This fulfils both the transitivity and commutativity requirements.

Lemma 4.4.3. *Each task is executed exactly once by each of the threads in a team.*

Proof. A task is always managed by only one thread (the coordinator) and cannot occur in two queues at the same time. When a task is stolen, it is first removed from one queue, before being added to the other one. The start of task execution is managed by the coordinator by storing a reference to the task in T. Each thread will remember the last task it executed for the team. A task may only be executed by a thread in a team if a reference to it is stored in T at the coordinator and the task is not the same as the previously executed task. Thus, a task will never be executed twice.

Before starting to execute a task, each thread atomically decrements the countdown variable G at the coordinator. The coordinator is only allowed to modify T when $G = 0$, which is only the case when all threads in the team have started execution of the task. Since no thread will execute T again if it was the last task it executed, each thread will decrement G exactly once. Also, the coordinator may only disband a team after announcing T when $G = 0$. This shows that the task will eventually be executed by all threads of the team. □

4.4.6 Lock-free implementation of the registration mechanism

A central aspect of the deterministic team-building algorithm is the registration mechanism which we now show how to implement in a lock-free fashion. Each place maintains a registration structure R that is modified by a *compare-and-swap* (CAS) operation when a thread registers or deregisters from a team. The coordinating thread does not need to use compare-and-swap operations. The registration structure is used for keeping track of a team being built for a task currently at the bottom of the thread's queue, and contains the following fields:

- The number of *required* threads r for the task at the bottom of the queue. This is modified every time a new task is added to the bottom of the local queue.
- The number of *acquired* (or *registered*) threads a, which is the number of threads currently registered for the team. Only threads that are required for a team of size r can be registered. If a new task is added to bottom that requires more threads, this number can stay. If it requires less threads, we have to reset it to the number of teamed threads and increment the new counter N (see below) to ensure that no invalid thread has registered.
- The number of *teamed* threads t is set to the size of the team by the coordinator after all threads have registered, therefore fixing the team. By default t is set to 1, which means that the team consists of a single thread (the coordinator). Teams can be shrunk by setting t to the new team-size. Disbanding a team means shrinking a team to size 1.
- The *new counter* N is incremented every time the coordinator decides to reset the number of acquired threads to the current team size t, in order to signal to all acquired threads that team-building has to start over again. This happens every time the coordinator calls the `disbandTeam()` method. It is also incremented in some cases where teams are resized, but as team resizing is an optimization not required for the basic algorithm, we do not cover it here.

The full registration structure can be packed into a 64-bit integer, and thus all fields updated by a single 64-bit CAS instruction by assigning 16 bits to each field. For smaller numbers of hardware threads a 32-bit CAS suffices. In theory N would have to be unbounded, but in practice a sufficiently large, finite N with wrap around suffices.

Now we describe how the registration structure is accessed and updated:

- `registerThread(I)` atomically increments the number of acquired threads a using a *compare-and-swap* operation. The thread that registers for the team locally stores the current value of N at the time of incrementing a.
- `dropThread(I)` tries to decrement a, therefore reversing the registration. If N has changed since the last call to `registerThread(I)`, the thread has already been dropped by the coordinator, and therefore decrementing is not required. If $a = r$, or the given thread id is part of the already built team of size t, dropping out is forbidden, and therefore fails.
- `teamBuilt()` first checks whether the thread requirement changed since the last call. If this is the case, the team is disbanded as described below, and team-building restarted by setting r to the new thread requirement. (In our actual implementation teams are resized when possible to reduce team-building overhead.) Otherwise, the algorithm checks whether $a = r$. If this is the case, the team is fixed by setting $t = r$. This does not require atomic operations, as the registration structure may not be modified by other threads after all threads have registered for a team.
- `disbandTeam()` atomically overwrites the registration structure with a new version. In the new version $r = 1$, $t = 1$, $a = 1$ and N is incremented. This does not require a compare-and-swap, but only an atomic write to the integer containing the registration structure.

As currently implemented we estimate the extra overheads in deterministic team-building as follows: an extra CAS used when a thread registers to or deregisters from a team. If all tasks require $r = 1$ the algorithm coincides with a locality-aware work-stealing scheduler where $\log P - 1$ victims are checked before the `backoff()`. The additional CAS is never executed in this case.

4.4.7 Optimizations

The following optimizations have been omitted in the algorithm description to simplify presentation:

- When stealing tasks, the last stolen task is not appended to the deque but instead returned immediately from the `stealTasks()` function. This is necessary to prevent situations where a task is stolen back and forth with no thread being able to execute it.
- We have observed that stealing the largest (wrt. team size) possible task from a victim will often result in more efficient schedules. This comes from the fact that a thread only steals from a thread at a certain level if all victims at lower levels had empty queues. Therefore, the chances are high that the stealing thread will be able to build up a team.
- To reduce the team-building overhead, we allow *cohorting* of tasks which require the same team size. This means that after executing a task, the coordinator of a team will give the team another task requiring a team of same size to execute. If no such task exists, the coordinator will attempt to shrink the team to a size for which it has tasks, or if no such tasks exist it will disband the team.

4.4.8 Arbitrary team sizes

We now indicate how to cope with the case where each task is assigned a team of size $t = r, \forall r \leq P$. To support this, it is first necessary to determine how a team of arbitrary size is deterministically built. This can be easily done by taking the next-bigger team-size in the

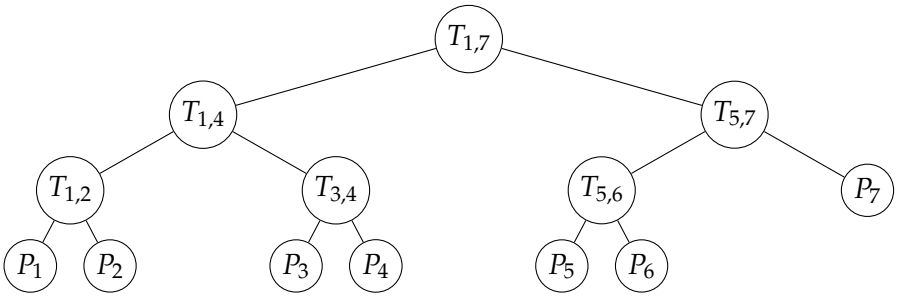


Figure 4.2: The team-building hierarchy hierarchy.

team-building hierarchy $2r > t' \geq r$ and leaving out $t' - r$ threads. The threads being left out are either the $t' - r$ leftmost or rightmost threads, depending on which are further away from the coordinator.

The polling procedure of the team-building algorithm also has to be modified to ensure that none of the left out threads are ever selected as stealing victims. Thus, the exact range of processor ids needs to be used as parameter for the victim selection algorithm used by the polling procedure.

In general, we expect that allowing arbitrary team sizes will often lead to the threads left out of the next-bigger team being idle, thus leading to a less efficient schedule. So unless the application requires the number of threads in a team to be exactly the requested number, we recommend allowing the scheduler to choose smaller team-sizes aligned to the hierarchy for higher efficiency.

4.4.9 Arbitrary number of hardware threads

We finally extend to the general situation where the number of processors in a system is not a power of two. To support this, the algorithm needs to be extended to work with binary trees that are not full binary trees. Most parts of the algorithm will work for such binary trees without modification. Care only has to be taken when determining team sizes at a specific level in the hierarchy, which is not 2^ℓ and can even vary depending on the processor. Instead, each node in the binary tree representing the hierarchy needs to store the first and last processor id contained in the team, as shown in Figure 4.2, to allow the scheduler to calculate the team size for each team individually.

4.4.10 Notes on the current implementation in Pheet

The team-building algorithm presented in this section has been implemented in a precursor to Pheet, and performance results for this implementation have been shown in previous work [146, 147]. A scheduler for mixed-mode parallelism has been implemented for Pheet as well and while it is true to the original philosophy some modifications and optimizations were made.

The registration structure used for the lock-free implementation of team-building has been updated and now only requires two of the original four members of the registration structure (r and a) to be stored in a single integer that is updated atomically.

A more controversial modification is that the Pheet implementation does not allow a task to be stolen, once the team-building process was started for it, even if that process is later cancelled in favor of a competing team. This allows interleaving some of the execution of the task with the team-building process, thus reducing the cost of team-building, but can lead to

load imbalance. While in practice this does not seem to be a problem for the applications we tested, we expect that this can lead to worse bounds for the scheduler.

4.5 Reducer Hyperobjects

Associative reducer hyperobjects (see Section 2.8), allow to perform associative operations on any variable or data-structure in parallel by multiple tasks. The reducer guarantees that from the user's point of view, the operations are performed in the same order they would have been performed in a sequential execution of the program.

4.5.1 Algorithm

As defined in our general model of hyperobjects in Section 2.8.5, each task carries around its own copy of the hyperobject. This copy is associated with a single structure called the *local view*. A local view contains part of the data, which can be updated by the task without the need for synchronization, and will be combined with other pieces of data after the task has terminated. In addition, the local view contains information for synchronization with other local views.

Data for reducer hyperobjects is represented by an object-oriented data type, which we call a *monoid*. In mathematics, a monoid is a triple consisting of a set, a binary operation over the set, and the *identity* value. The monoid class used for hyperobjects captures the properties of a specific monoid and allows to apply it to specific data. An instance of the monoid class stores a single value out of the set, and provides a function that allows to `reduce` two values into a single value using the binary operation over the set, as well as a function to `reset` the value back to the identity of the set. An example monoid for constructing lists is shown in Listing 4.5. In the remainder of this section we use *monoid* to denote instances of the monoid class, not the algebraic structure.

Listing 4.5 Pseudocode for a list-reducer monoid, which can be used to construct lists of integers.

```
1 class ListReducerMonoid {
2   void reduce(list<int>& other) {
3     data.append(other);
4   }
5
6   void reset() {
7     data.clear();
8   }
9
10  list<int> data;
11 }
```

For associative reducers, there is a one-on-one relationship between a copy of a reducer hyperobject and the local view it is associated with. This does not mean that a reducer always references the same view during its lifetime. Whenever a task is spawned that receives its own copy of the hyperobject, the new copy will receive the original local view, while a new local view is created for the original copy of the hyperobject used in the continuation. This is necessary to maintain the illusion of a sequential execution for this hyperobject. As shown in Figure 4.3, a sequential execution (represented by the numbers on the nodes) is depth-first and will always first follow spawn edges before exploring continuation edges. To ensure the

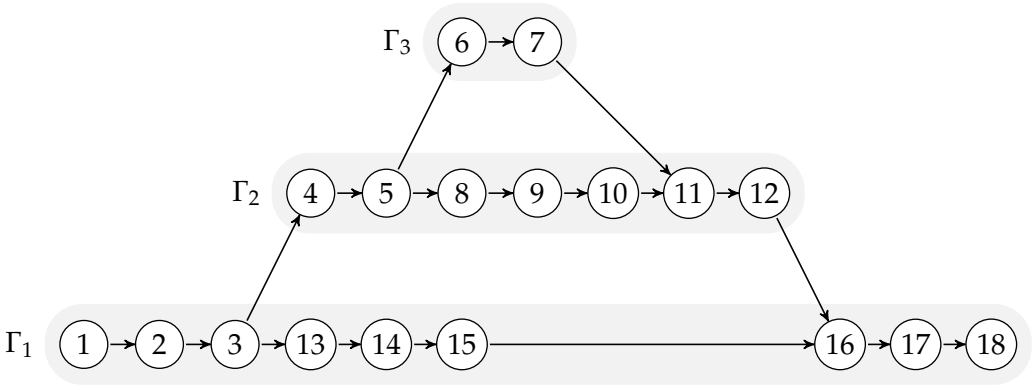


Figure 4.3: A task-parallel execution.

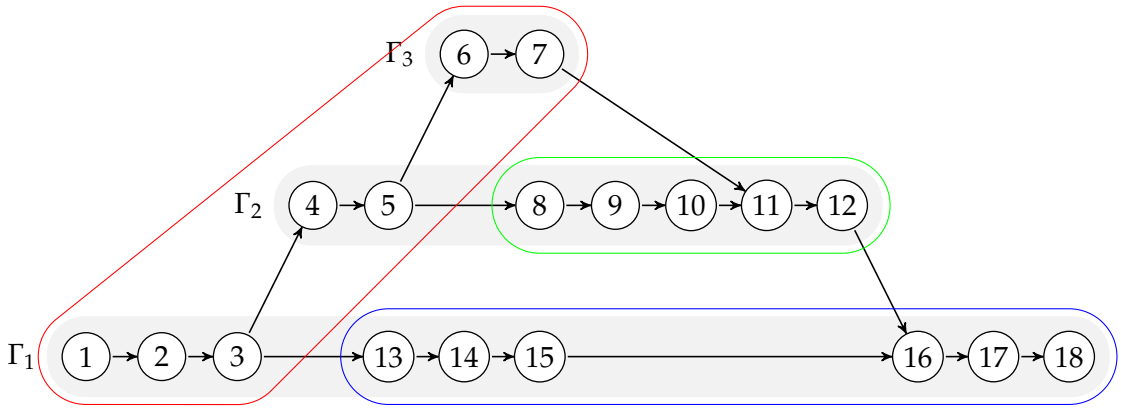


Figure 4.4: The local views used by a hyperobject in a task-parallel execution.

values stored in local views of reducer hyperobjects are constructed in sequential order as well, local views always need to be passed on to the spawned task.

The new copy of the hyperobject, which is created at a task spawn, will also store a pointer to the local view that was created for the continuation as *successor view*. While the local view of a copy of the hyperobject can change throughout the execution, its successor view will never change. This pointer can then be used to reduce both views back into a single view, thus maintaining the illusion of a sequential execution order. Figure 4.4 shows all local views used for the dag from Figure 4.3. Reducing all local views with their predecessors (in any order) until only a single local view remains will yield the same results as a sequential execution.

4.5.2 Implementation

An associative reducer hyperobject is implemented as a small data-structure that is passed on to subtasks by value. A reducer hyperobject has the following fields:

- A reference to a local view v. Changed on every copy operation.
- A reference to the successor view s. Initialized when a copy is created. Constant for this copy.

The local view data-structure consists of the following fields:

- The data m, which is an instance of the monoid class.

- A reference to the predecessor view p, initialized to $\perp$.

Algorithm 4.6 shows how the copy operation for reducers is implemented. The newly created reducer is linked to the original view, and a new view is created for the original reducer. The newly created view is also stored as successor view in the newly created reducer. The intuition for this is that the original hyperobject is used in the continuation ($\rightarrow$ successor) of the task.

Listing 4.6 `copy(r)()`

```
1: nr.v ← r.v {New reducer gets old view}
2: r.v ←new view() {Old red. gets new view}
3: nr.s ← r.v {Store successor view}
4: return nr
```

For reducer hyperobjects the main objective of the destructor (see Algorithm 4.7) is to perform a reduction. It reduces its own monoid with its predecessor's monoid, and makes the predecessor view its active view. This can be performed multiple times. As soon as no predecessor is available ($p = \perp$), reduction terminates, and the reducer makes its own value available to another hyperobject. This is done by setting the p field of its successor view to point to its own local view.

Listing 4.7 `destruct(r)()`

```
1: while r.v.p ≠⊥ do {has predecessor}
2:    p ← r.v.p {Store predecessor}
3:    p.m ←reduce(p.m, r.v.m)
4:    delete r.v {delete current view}
5:    r.v ← p {set view to predecessor}
6: end while
7: r.s.p ← r.v {Notify successor view}
```

For the sake of simplicity, the algorithm presented here does not contain any means for termination detection. Instead we rely on the programmer to provide proper termination detection using a finisher hyperobject, a finish region or other synchronization primitives, similar to finish accumulators [122]. If all tasks using a certain reducer have been created inside a finish region, it is guaranteed that at the end of the region all predecessors are available. The programmer can then request the reduced value, which is obtained by a final reduction of all remaining values, similar to the reduction performed in the destructor.

4.5.3 Optimization

For sequential executions it is desirable that all operations are performed on a single instance of a monoid, removing the need to perform (potentially expensive) reductions on monoids. It can be observed that for a sequential execution the first operation on a monoid is always performed after all predecessor pointers have been set. One way to exploit this is to perform a check for potential predecessors before adding a value to a previously unused monoid. If a predecessor exists a reduction may be performed immediately, but no need exists to reduce the monoids as one monoid still contains the identity. This means that a merge operation can be omitted. We call this type of reduction *folding*.

With folding it is also possible to reuse local views. Every time a local view with an unused monoid is folded it is put into a list for reuse. Only views folded as part of a sequential

execution by a single thread can be reused. In a sequential execution, the number of views created per hyperobject will never exceed the stack-size, therefore the list of reusable nodes is bounded.

Finally, the destruction and reduction algorithms require a strong memory consistency model to enforce an ordering between operations. Whenever it is clear that a view and its successor/predecessor are both owned by the same thread, the memory model can be relaxed and fences omitted.

4.5.4 Correctness

Lemma 4.5.1. *Reduction operations on local views preserve associativity with respect to a sequential execution.*

Proof. We define a sequential ordering of operations in a task-parallel application to be the ordering occurring when all task spawns are converted into synchronous function calls. By definition, the monoid used by a reducer has to be associative, therefore the order in which reductions are performed does not matter as long as the sequence of operands is not changed.

On each spawn, two separate execution paths are created, and each builds its own linked list of local views, which contains the operands of the associative operation. The original local view is passed on to the spawned task. A new local view is created for the continuation. In a sequential execution, the spawned task would be executed before its continuation, therefore all reduce operations in the spawned task need to occur before operations in the continuation. Since the continuation has no access to the original local view, this is obviously the case.

In the continuation a new local view is used. Due to associativity it is allowed to perform reductions inside. After the spawned task has completed executing, the original local view will become available as predecessor of the new local view. Both views can now be reduced to a single one, which again preserves order. □

Lemma 4.5.2. *With exception for the predecessor field p all fields in a local view can only be accessed by a single thread at any point in time.*

Proof. There is a one-on-one relationship between reducers and local views. Since a single reducer cannot be shared between threads, only a single thread can access the local view through a reducer. The linearization point of the destructor of a reducer hyperobject is when its local view is passed on as predecessor to another local view. It is the last operation in the destructor, so that no members of the hyperobject are accessed after it. The view that gets access to the view of the deleted hyperobject is again only accessible by a single thread, therefore its list of predecessors is only accessible by this thread as well. □

Lemma 4.5.3. *The p field is written at most once for each local view.*

Proof. A reducer hyperobject contains a pointer to a successor view, which is only used once to modify the p field in the successor view. The pointer to the successor is always set to point to a view created in the previous statement, therefore a view is only accessible through a single successor pointer. As there is at most one successor pointer pointing to each local view, and it is used exactly once for writing the p field, the p field is written at most once for each local view. □

Theorem 4.5.4. *Reducer hyperobjects are wait-free, assuming a wait-free memory allocator.*

Proof. The copy operation only consists of wait-free operations and is therefore trivially wait-free. The destruction operation consists of a reduction and a single assignment. If reduction

is wait-free so is destruction. Reduction iterates through all currently available predecessors of a local view. During the iteration more predecessors may become available, but the local views that are predecessors of a given view are bounded by the application regardless of whether they are already available or not. Therefore the iteration is bounded by the number of predecessors.

Exclusive access is guaranteed for both monoids used in a single reduction by Lemma 4.5.2 since the monoids are stored in local views. Due to the exclusive access, no synchronization is required inside the reduce operation, which makes it wait-free. All other operations used in a reduction are trivially wait-free as well, which makes the whole reduction operation wait-free. □

4.6 Finisher Hyperobjects

In this section we present a high-level overview of the wait-free reference counting algorithm used to implement finisher hyperobjects.

As described in our model for hyperobjects in Section 2.8.5, a hyperobject is a lightweight data-structure that mainly consists of a pointer to its *local view*. The local view is where the reference counting is performed. Contrary to associative reducers, where the sequential order of execution needs to be preserved, a local view can be shared by multiple copies of the hyperobject. For finisher hyperobjects each local view is associated with a operating system thread. As long as all copies of a hyperobject are owned by the same thread, all will reference the same local view.

Certain fields of a local view are only allowed to be modified by the thread the local view is associated with. Whenever a copy of a finisher hyperobject is passed on to another thread, this thread will encounter a local view where it is not allowed to modify all fields. If it needs to modify one of those fields, it has to create a new local view to do that. The new local view will store the previous local view as its `parent`.

The local view of a finisher hyperobject is used for reference counting. It has a local counter, which counts all copies of the hyperobject that use this local view, as well as all local views that have the given local view as parent. At the end of the lifetime of a copy of a finisher hyperobject, it decrements the reference count of its own view. If the finisher was the last one to reference it, the local view can be cleaned up. If the local view has a parent, this requires to decrement the reference count of the parent in a thread-safe manner. If no parent exists, this means that no more copies of the finisher hyperobject exist, allowing for clean-up.

4.6.1 Implementation

We now provide details of the finisher hyperobject implementation. Central to a hyperobject is the local view, which is a data-structure containing the following fields:

- The thread id t.
- A pointer to the parent view p. For the initial view this is a null-reference.
- The local reference count l. Incremented when a finisher is copied and decremented when a local copy is destroyed. Initialized to 1.
- Number of copies finished by another thread r. Incremented atomically whenever a view referencing this view is cleaned up. Initialized to 0.

- A flag f used to atomically decide which thread is responsible for clean-up. To avoid an ABA-problem an integer is used. Even values stand for views in use, odd for cleaned up views. Initialized to 1.

The true reference count for each local view is the difference $l - r$. As long as at least one reference to a view exists, the invariant $l > r$ holds. If $l = r$, the last reference to the view has been removed and the view can be cleaned up. We say a hyperobject is *unique*, if for all its views $l = r + 1$.

As mentioned before, the finisher hyperobject only consists of a reference (called v) to the local view data-structure described above. Finisher hyperobjects are used with value semantics in an object-oriented sense, so that each time a finisher is passed on (e.g. to a subtask) or stored, a copy constructor is called.

Listing 4.8 `copy(f)()`

```
1: f.v ←active_view(f.v) {Get own view}
2: nf ← f {Clone finisher}
3: f.v.l ← f.v.l + 1 {Increment local reference count}
4: return nf
```

Algorithm 4.8 shows how a finisher hyperobject is copied. After copying a finisher, both copies reference the same local view. The copy constructor increments the local reference count l. Our model requires that a hyperobject may only be copied by the thread that owns it. Therefore, no synchronization is required when accessing l.

Listing 4.9 `destruct(f)()`

```
1: v ←active_view(f.v) {Get own view}
2: vf ← v.f {Store flag}
3: v.l ← v.l − 1 {decrement local reference counter}
4: if v.l = v.r ∧ CAS(v.f, vf, vf + 1) then
5:    p ← v.p {Store parent pointer}
6:    while p ≠⊥ ∧ p.l = p.r do {Predecessor can also be cleaned up}
7:       v, p ← p, p.p {Move to parent}
8:       vf ← v.f {Store flag}
9:       vr ←get_and_increment(v.r)
10:      if ¬(v.l = vr + 1 ∧ CAS(v.f, vf, vf + 1)) then {No need to clean up next view}
11:         return
12:      end if
13:   end while
14: end if
```

When a copy of the finisher hyperobject reaches the end of its lifetime, its destructor is called, which is shown in Algorithm 4.9. The destructor decrements the local reference count l, and checks whether $l = r$. In this case, no more references to the view exist. The last thread accessing the local view is required to perform some clean-up. It is possible that more than one thread sees $l = r$. A *compare-and-swap* (CAS) setting the flag field to an odd value ensures that only one thread performs clean-up.

On clean-up, if the local view has a parent p, the thread that is responsible for clean-up has to signal that a remote reference has been finished by atomically incrementing $p.r$. Again, if $l = r$ for the predecessor view, the view needs to be cleaned up. As before, a *compare-and-swap* on the flag field is needed to ensure that clean-up is only performed once.

A hyperobject may change hands after creation since the copy operation is performed in the context of the parent task. To ensure that the view referenced by the finisher hyperobject is owned by the current thread, a check is performed at the beginning of both the copy constructor and the destructor (Algorithms 4.8 and 4.9), which is every time the l field of the local view needs to be modified accessed. This ensures that a new local view is created any time a hyperobject changes hands. Algorithm 4.10 depicts this check. If the t field of the local view differs from the thread id the hyperobject must have changed hands in the meantime. In this case, a new local view is created for the given thread, and a reference to the previous view stored in p. The previous local view is not modified. All future accesses to the currently accessed copy of the hyperobject use the newly created local view. Since views cannot be safely deleted, we use our wait-free memory manager from Section 4.3 to manage views. A view can be reused by the memory manager, as soon as f is set to an odd value.

Listing 4.10 `active_view(v)()`

```
1: {Check for view ownership}
2: if v.t =thread_id() then {Owner of this view}
3:    return v
4: else {Not the owner - Clone the view}
5:    c ←acquire_view() {Get new view}
6:    c.t ←thread_id()
7:    c.p, c.l, c.r ← v, 1, 0
8:    c.f ← c.f + 1 {Mark view as in use}
9:    return c
10: end if
```

4.6.2 Shared pointer

Finisher hyperobjects can be used to manage a shared pointer, so that referenced data is automatically deleted when the last pointer is destroyed. If the shared pointer is stored with the finisher, the data can be deleted when the last finisher is deleted ($l = r$ in the root local view).

To allow shared pointers that store null-pointers, and to allow reuse of shared pointers, we make the following modifications to the finisher algorithm:

- No local view is assigned to shared pointer hyperobjects storing null pointers.
- Whenever new data is assigned to the pointer, a new root local view is created.
- Whenever a shared pointer containing data is assigned to new data or null, l is decremented, and the local view cleaned up if necessary.
- Whenever a shared pointer is assigned the value of another shared pointer, this is treated like a copy operation.

4.6.3 Copy-on-write pointer

To implement a copy-on-write pointer based on a shared pointer hyperobject, the normal dereferencing operation of the pointer has to return a reference to an immutable value. In C++ this can be done using the `const` keyword. In addition, a copy-on-write pointer has an access method, which we call `get_writable()`.

If $l = r + 1$ in the local view and all its predecessors (uniqueness), it is guaranteed that the data referenced by the pointer is not shared. Therefore `get_writable()` can return the pointer. Otherwise, the data is copied, and a new local view created for the hyperobject (with $p \leftarrow \perp$). The old local view is dereferenced by decrementing l and clean-up performed if necessary.

4.6.4 Finish region

Finish regions may easily be implemented using finishers, by passing a copy of a finisher to each subtask. For nested finish regions, only the innermost finisher needs to be passed on to subtasks. The finish condition is met if the finisher is unique, which is when for all local views $l = r + 1$. In this case the only copy of the finisher left is the original copy. The functionality of finisher hyperobjects can be embedded directly into a scheduling system, so it is not required to pass around finisher hyperobjects. The implementation of finish regions in Pheet is based on finishers.

4.6.5 Optimizations

With finishers one can observe that in most computations many local views in finishers are actually *leaves*. We define that a local view is a leaf when no finisher referencing it is passed on to another thread. In the destructor it is possible to recognize a leaf when the last finisher referencing it is destructed. If $l = 0$ after decrementing, all copies of the finisher have been destructed locally. In this case, no synchronization is needed for the clean-up operation and the atomic *compare-and-swap* on f variable can be omitted.

If a local view is unique ($l = r + 1$), it is guaranteed that only one thread can access it. In this case, a thread is allowed to change the owner of a local view if necessary to omit creating a new local view. Also, if a local view is unique, r does not need to be incremented atomically.

4.6.6 Correctness

We now prove that our finisher implementation is correct, linearizable and wait-free.

Lemma 4.6.1. *At any point in the computation, the invariant $l \geq r$ always holds. If $l = r$ at some point, it stays this way.*

Proof. Each local view is initialized with $l = 1$ and $r = 0$, because in the beginning a single finisher references it. We now look at some local view, which we will call vv. Each time a finisher with $v \rightarrow vv$ is created, $vv.l$ is incremented. No other finisher may reference vv, and since l is only decremented in the destructor of a finisher, the invariant $l \geq 0$ holds. For l to be incremented, at least one hyperobject must exist to be copied, which is only when $l > 0$.

To increment $vv.r$, a local view lv ($lv \neq vv$) has to have vv as its parent ($lv.p \rightarrow vv$) and enter clean-up. When a new local view with $lv.p \rightarrow vv$ is created, it takes over a single finisher from vv by changing its local view pointer from $v \rightarrow vv$ to $v \rightarrow lv$. Therefore, $vv.l$ can be bounded to be at least the number of new local views with $p \rightarrow vv$, since for each such view one finisher does not reference vv any more, but did not decrement $vv.l$. For n views with $p \rightarrow vv$, we will therefore have $l \geq n$ in vv. At the same time, each clean-up of a view results in only one increment of r in its parent, which is ensured by the atomic flag variable. Therefore $r \leq n$. This implies $l \geq n \geq r$. If $l = r$, both l and r will never change. For l to be incremented, $l > n$ is required, which contradicts $l = r$. Since r is never decremented, and all other changes would violate $l \geq r$, both l and r will not be modified after $l = r$. □

Lemma 4.6.2. *A local view can never be cleaned up before all finishers referencing it or its successors have been destructed.*

Proof. Assume it would be possible to clean up a local view before all finishers referencing it have been destructed. This would require some thread to observe $l = r$ without this being the case. A thread may observe a value of a variable different to its actual value, if the value has changed since it was read out of memory. By Lemma 4.6.1 we know that $l \geq r$. Assume that $l > r$, but some thread observes $l = r$. For this to happen, either the observed l must be lower than its actual value, or the observed r higher.

Since r is never decremented, the observed value can never be higher than its actual value. Therefore, the observed l must be lower than its actual value. The owner of the local view will always observe the correct value for l, since it is the only thread modifying it. Other threads only read l during clean-up. There, the value for l is read after the value for r. This ensures that the observed value for l is not older than the observed value for r. Since r is never decremented, if $l = r$ is observed it must have been true at some point. By Lemma 4.6.1, l and r will not change after $l = r$, so if $l = r$ is observed this must be true, which contradicts $l > r$. □

Lemma 4.6.3. *Clean-up of local views is always started.*

Proof. Assume that clean-up is not started for a local view. In this case, all threads must observe $l > r$. As both the destruction algorithm as well as the clean-up algorithm for local views are linearizable by Theorem 4.6.4, an ordering between decrements to l and increments to r is established. The last operation will observe the changes to l and r by all other operations, which contradicts that all threads must observe $l > r$. □

Theorem 4.6.4. *All operations on finishers are linearizable and wait-free.*

Proof. The copy operation for finishers reads shared variables, and writes to a single variable that may only be read by other threads. In addition it may acquire a new view from the memory manager, which is wait-free by Lemma 4.3.1. Therefore it is trivially wait-free. The linearization point for copy is the write to l.

The linearization point for the destructor is the decrement of l. After this point, threads are allowed to clean up the local view if no more references to it exist. If clean-up is necessary, the thread performs wait-free two-thread consensus using an atomic *compare-and-swap* operation. On failure, the thread may exit the destructor, and as there are no loops or recursions it is trivially wait-free.

clean-up of a local view is similar to the destruction of a finisher, only that instead of a decrement to l, the variable r of the predecessor view is atomically incremented. The increment of r is the linearization point of the clean-up. The only exception is if there is no parent view, then the *compare-and-swap* to the flag variable of the previous view is the linearization point.

We have shown that all parts of the destructor are wait-free. For the destructor as a whole to be wait-free it is necessary that there is only a bounded number of clean-ups to perform. This is the case, since the parent pointers of a local view are initialized when the local view is initialized. The list of parents never changes for a local view during the view's lifetime, so its length bounds the number of clean-ups performed by a destructor. □

5 Ordered Containers

In this chapter we look at *containers*, bag-like data structures that offer the operations `push` and `pop`. With `push`, an item can be added to the container, and `pop` returns a previously added item and removes it from the container. In addition, some ordered containers may support the operation `peek`, which operates similarly to `pop` but does not delete the returned item from the container.

We use the term *ordered container* to describe containers for which there exists a specific order in which items are returned by `pop`. For some ordered containers, the order is determined by the order of item insertions (e.g. in queues and stacks), whereas for others the order is determined by properties of the inserted items (e.g. heaps).

Concurrent implementations of ordered containers are an important part of task scheduling systems, where they are used to store tasks. The main focus of this chapter is to look at concurrent ordered containers that are useful for task scheduling systems, but some of the presented containers can be useful outside the context of task scheduling as well.

5.1 Semantics for Concurrent Ordered Containers

Except for the Log-structured merge-tree presented in Section 5.7, all ordered containers presented in this chapter are concurrent implementations that allow multiple threads to `push` and `pop` items at any point in time. Unless otherwise specified, we allow `pop` operations to spuriously fail, as long as at least one thread is guaranteed not to fail on its next `pop` operation. Depending on the requirements of the application, different semantics can be used for an ordered container.

5.1.1 Global ordering semantics

The strictest possible semantics for concurrent ordered containers is to linearize all `push` and `pop` operations by all threads with regard to each other. This will often lead to high congestion on concurrent `pop` operations, since all threads will attempt to remove the same item, and only one can succeed. For stacks, congestion can be reduced with *elimination* [76], but this is not applicable to all types of orderings.

5.1.2 Purely local ordering semantics

In a purely local setting, each thread maintains its own ordered container, which it accesses using `push` and `pop` operations. When a thread accesses the ordered container of another thread, its operations are linearized with the operations of other threads on the same ordered container. Operations on ordered containers owned by different threads are not linearized

with regard to each other. Purely local semantics can for example be found in work-stealing deques [11, 39].

Shared items

Some concurrent ordered containers maintain one ordered container per thread, but allow an item to be accessible from more than one local ordered container. While there is still no global order on operations, `pop` operations on shared items need to be linearized with regard to all local containers from which the item is accessible. This does not imply that the popped item needs to be the next item in order for every local ordered container it is contained in. Ordering constraints are still only maintained locally for each container. However it implies that if an item is the next item in order in a local container then that container will return the item, unless another thread is faster.

5.1.3 Relaxed semantics

Implementing a correct, linearizable globally ordered container often has inherent costs associated with it. Some ordered containers have inherent sequential bottlenecks where ordering constraints limit `push` and/or `pop` operations to only allow one thread to succeed at any point in time.

For stacks it is possible to work around this bottleneck in a correctly linearizable manner using *elimination* [76], where corresponding `push` and `pop` operations are matched and thus never directly access the stack. For other concurrent ordered containers a slightly weaker version can be achieved by weakening the correctness conditions. *Quiescent consistency* [13, 121] is a correctness condition weaker than linearizability and sequential consistency, which requires operations to occur one at a time. Only operations separated by a period of quiescence (a period with no accesses to the container) are required to occur in their real-time order. Quiescent consistency allows the relaxed version of the skiplist-based priority queue by Lotan and Shavit [81, 120] to match `pop` operations to concurrent `push` operations, as long as the item inserted by `push` has a higher priority than any item that existed in a previous quiescent state.

Both elimination and quiescently consistent priority queues require that the number of `push` operations that can be matched to `pop` operations is roughly the same as the number of `pop` operations. If these numbers diverge, bottlenecks will again occur for the more frequent operation.

Quantitative or ρ-relaxation

Afek et al. [3] have presented an alternative consistency model called *quasi linearizability*. Their model builds on the notion of linearizability, but allows operations to occur out of a correct linearizable order. While operations may appear to be out of order, quasi linearizability puts a upper bound on the *distance* each operation is allowed to have from a correct linearizable order of operations. Quasi linearizability has been mainly used as a consistency condition for FIFO queues [3, 18, 89], but is not restricted to these. Later work by Henzinger et al. [77] has provided a different model closely related to quasi linearizability, which is called *quantitative relaxation*.

We use the term *ρ-relaxation* [144, 148], to describe quantitative relaxation on concurrent ordered containers, where an upper bound, ρ, can be given on the number of items that can be *skipped* on data structure accesses (`pop` and `peek`). We say an item is *skipped* whenever an item is not returned on an access, even though the item that was returned would have to be returned after the skipped item according to global ordering semantics. For our containers this include cases where a null-value is returned, signalling an empty container.

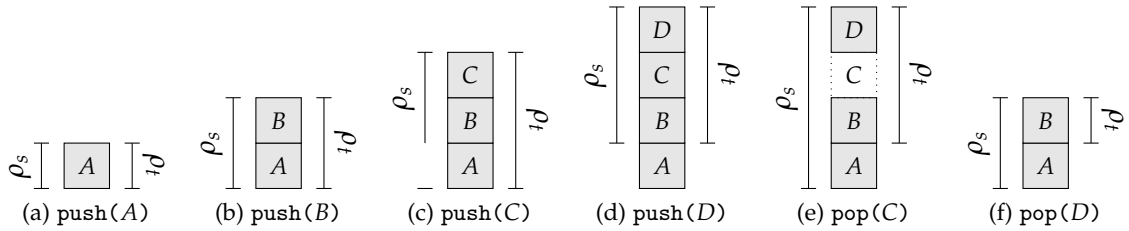


Figure 5.1: Temporal vs. structural ρ relaxation on a stack for $\rho = 2$. The items that can be relaxed by structural ρ-relaxation are marked with ρ_s, temporal with ρ_t.

We distinguish between *temporal* and *structural* ρ-relaxation. Temporal ρ-relaxation is a property based on the recency of items, closely related to quasi linearizability. A temporally ρ-relaxed ordered container is only allowed to skip the ρ most recently added items. Temporal ρ-relaxation can be applied *globally*, so that only the ρ most recent items added by *any* thread can be skipped, or *locally*, where the ρ most recent items by *each* thread can be skipped.

Structural ρ-relaxation, unlike temporal ρ-relaxation is not concerned with when an item was added. It is only concerned with the amount of items that are allowed to be skipped at any point in time regardless of their recency.

Figure 5.1 helps to illustrate the differences between temporal and structural ρ-relaxation based on a stack for $\rho = 2$. While for the freshly initialized stack both types of ρ-relaxation allow the stack to skip the two items on the top of the stack on pop, temporal ρ-relaxation will not allow a thread to skip B after two more items have been added, regardless of how many items have been removed in the meantime. This makes temporal ρ-relaxation stricter than structural ρ-relaxation.

5.1.4 Combining ρ-relaxation with local ordering

In practice, many scalable implementations of ρ-relaxed data structures automatically exhibit local ordering semantics, where each thread will never skip items that it created. This ensures that a thread will always return a locally added item next if it has the highest priority, unless another thread takes it first.

5.2 Priority Work-stealing Queue

To support task-parallel programming model with priorities, as described in Section 2.5, it is necessary for a task scheduler to use a concurrent priority queue as a task queue. Our first approach was to stay within the work-stealing model [11], where each worker thread operates on its own task queue, and steals work from another thread's task queue whenever it runs out of work locally.

5.2.1 Internal structure

To separate between the concerns of maintaining an efficient priority queue, and allowing other threads to steal items, our local priority task queues are split into two parts: A linked list containing items (tasks) in order of creation, which we call the *stream*, and a sequential priority queue, which operates on references to items in the stream.

5.2.2 Implementation of the stream

The stream is implemented as a linked list of items where items are stored in order by age from oldest to newest item, which can be traversed by all threads. When an item is removed either due to a `pop` or due to a steal operation, it is marked as *taken* in the stream. Each item in the stream has the following members:

- `id`: The id of the item. Items are numbered in order of creation, and it is guaranteed that each item has a lower id than its successor (unless a wraparound occurs, but this does not change the correctness of the algorithm).
- `next`: A pointer pointing to the successor in the stream. Can be modified by the owner, but read by all threads.
- `taken`: A flag that can be atomically modified by any thread to mark an item as logically removed. To avoid an ABA problem, `taken` is an integer, initialized to the id of the item. An item is taken if its value does not equal the id.
- `references`: An integer representing the number of threads currently processing the given item. It is atomically incremented and decremented by other threads.
- `num_pred`: The number of items for which the `next` pointer points to this item. This includes items that have been unlinked from the stream but might still be accessed by other threads.
- `key`: The key used to order items by priority.
- `data`: The actual data stored with the item. (For task queues, this is the task.)

Items are managed by the wait-free memory reuse scheme presented in Section 4.3 and will be reused as soon as they have been taken and it is safe to reuse them. Items may be reused even if there are still priority queues with references to an item, and for this reason the `taken` flag has to be implemented in an ABA-safe manner. An item cannot be reused as long as other threads are processing the given item (`references`$\neq 0$). Also, we guarantee that regardless of how long a thread holds a reference to an item in the stream, it will always be able to continue processing the stream without encountering an item twice or missing an active (non-taken) item. This requires that following the `next` pointer will eventually lead to the next active item, even for items that have already been unlinked from the stream. An item that still has a next pointer pointing to it cannot be reused even if no thread holds a reference to it at the moment. What can be done is to create shortcuts whenever `next` pointer points to an item that has already been taken, thus making an in-tree out of a linked list. This has two effects: First, all leaves of the in-tree are safe to be reused if they are taken and no more references to them exist. We use the field `num_pred` to track how many items still point to a given item, thus allowing us to recognize leaves. When they are reused, their successor might become a leaf as well. Second, shortcutting also reduces the number of taken items a thread has to process until it finds the next active item.

To minimize the number of taken items a thread has to go through in the stream to find an active item in the worst case, and to reduce the number of shortcutting operations necessary, we only allow shortcutting operations to be performed that will result in a specific binomial tree. The shape of the binomial tree is predetermined by the id's of items. Possible shortcuts for items with id's $0-8$ are shown in Figure 5.2. The resulting binomial tree after shortcutting is shown in Figure 5.3. The algorithm for creating shortcuts is presented in Listing 5.1. Item id's are numbered continuously and there are no id's missing in the stream before a shortcut

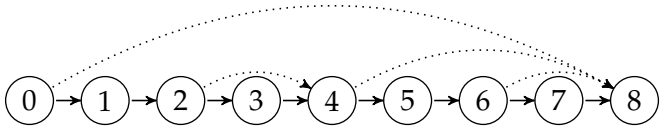


Figure 5.2: Binomial shortcutting.

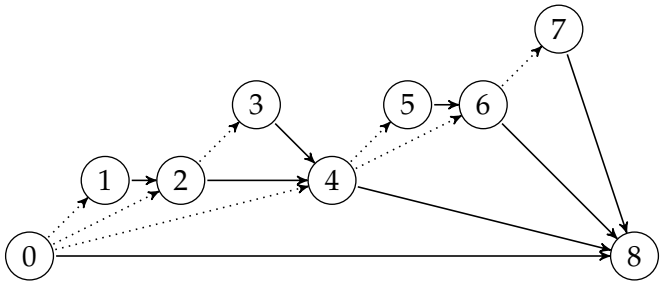


Figure 5.3: Binomial shortcut tree. (Dotted edges represent edges that were shortcut)

is made. Under this assumption, it is possible to determine whether the successor of a node is supposed to be in the same tree branch by performing a *bitwise and* between the node's id and the successor's id. If the *bitwise and* returns the node's id, both nodes are guaranteed to be in different branches. As an optimization we also allow further shortcuts to be created whenever an item is a leaf. To ensure thread safety exclusive access needs to be guaranteed to a node for shortcutting (but not necessarily its successor). For priority work-stealing we achieve this by only letting the owner of a node perform the shortcutting algorithm.

Listing 5.1 The binomial-tree shortcutting algorithm for items.

```
1  StreamNode* next = node->next;
2  if(next == nullptr)
3    return;
4  StreamNode* nnext = next->next;
5  while(nnext != nullptr && // There exists a successor to create a shortcut to
6    next->taken != next->id && // Item was already taken
7    (next->id & nnext->id) != next->id) && // No shortcutting possible for successor
8    (node->num_pred == 0 || // Node is a leaf
9    // or shortcut corresponds to an edge in the binomial tree
10   ((node->id & next->id) == node->id) {
11
12   // Shortcut
13   node->next = nnext;
14
15   // Next node got rid of one predecessor
16   --next->num_pred;
17   // Shortcut node got a new predecessor
18   ++nnext->num_pred;
19
20   // Try creating shortcut for successor node as well
21   next = nnext;
22   nnext = next->next;
23 }
```

5.2.3 The `push` operation

A `push` operation is fairly straightforward. It first retrieves a reusable `StreamNode` from the memory pool, stores the item in it, and adds it to the stream. Then it puts a reference to the `StreamNode` into its local priority queue. The code is shown in Listing 5.2. To simplify the algorithm the stream is initialized with a single sentinel node, which is marked as taken, so we can assume that the stream always contains a node.

Listing 5.2 The `push` operation in priority work-stealing.

```
1  void push(Key key, Data data) {
2    StreamNode* node = node_memory_pool.get_item();
3    node->id = stream->tail->id + 1;
4    node->next = nullptr;
5    // Taken is initialized to the id, to avoid an ABA problem
6    node->taken = node->id;
7    // Number of other threads referencing the node
8    node->references = 0;
9    // In the beginning only one node will have this node as successor
10   node->num_pred = 1;
11
12   // Store key and data in the node
13   node->key = key;
14   node->data = data;
15
16   // Add node to stream
17   stream->tail->next = node;
18
19   // Add reference to item to local priority queue
20   // The node id needs to be stored separately for ABA safety
21   priority_queue->push(key, tuple(node->id, node));
22 }
```

5.2.4 The `pop` operation

The `pop` operation removes the highest priority item from the priority queue and then attempts to atomically mark the item in the stream as taken using a compare-and-swap. On success, this item is returned, otherwise the operation has to be repeated until either an item is returned or the queue is empty. When a thread runs out of local work, it will attempt to `steal` items from another thread. Pseudocode for the `pop` operation is presented in Listing 5.3.

5.2.5 The `steal` operation

A `steal` is performed whenever the local priority queue of a worker thread is empty on a `pop` operation. The `steal` operation will select a victim thread according to its victim selection policy (see Section 3.5), and then scan the stream of a victim thread, and sort the encountered items locally by priority. It will then attempt to steal the highest priority items from the other thread. We allow steals to use different priorities than are used in the local priority queues. This reduces congestion, and also allows the priority queue to emulate the behaviour of a work-stealing deque.

Listing 5.4 shows a largely simplified version of the `steal` operation. After a victim is selected, the stream of the victim is processed and all items stored in a temporary priority

Listing 5.3 The pop operation in priority work-stealing.

```
1 Data pop() {
2   while(!priority_queue.empty()) {
3     // Pop item from priority queue
4     auto node_tuple = priority_queue.pop();
5
6     // Decompose tuple
7     int id = node_tuple.first;
8     StreamNode* node = node_tuple.second;
9
10    // Node was not yet taken and we succeed in marking it as taken
11    if(node->taken == id && node->taken.cas(id, id+1)) {
12      return node->data;
13    }
14  }
15  return steal();
16 }
```

Listing 5.4 A simplified version `steal` operation for a given victim in priority work-stealing.

```
1 Data steal() {
2   // Select a victim and store its stream in stream_node
3   StreamNode* stream_node = ...
4
5   // Use temporary priority queue to sort items
6   PriorityQueue pq;
7
8   // The first steam_node is always a sentinel node so we can skip over it
9   while(stream_node->next != nullptr) {
10    StreamNode* next = stream_node->next;
11    // Register with next node
12    next->registered.atomic_increment(1);
13    // Deregister from old node
14    stream_node->registered.atomic_decrement(1);
15
16    // Transition to next node
17    stream_node = next;
18
19    // Add node to temporary priority queue
20    pq.push(stream_node->key, tuple(stream_node->id, stream_node));
21  }
22
23  while(!pq.empty()) {
24    // Try to take and return highest priority item, similar to pop
25    ...
26  }
27 }
```

queue. Finally, the stealing thread will pop the highest priority item from the priority queue and attempt to mark it as taken, similar to the `pop` operation.

5.2.6 Optimizations

To simplify the presentation of the algorithm, it contains some inefficiencies. One obvious cost is associated with the reference counting necessary for processing the stream, which can be easily reduced by grouping items into *blocks*, and performing reference counting per block instead of individual items. Also, blocks will increase efficiency of memory accesses, since they allow turning the linked list into a linked list of arrays.

The biggest problem with the presented algorithm is the `steal` operation, where each steal requires scanning the whole stream of the victim and storing all items from the stream in a priority queue. Our current implementation in Pheet resolves this by caching both the pointer to the stream and the temporary priority queue, so that subsequent steal operations between the same two threads are only required to scan items added in the meantime. In addition, stealing half the work (see also Section 2.7.3) instead of a single item allows to amortize the cost of scanning the stream over the stolen work.

Both optimizations are not without costs: A disadvantage of stealing half the work is that, depending on how items are prioritized exactly, a thief might steal all high priority items from its victim. Caching streams will create a memory overhead of $O(P^2)$, where P is the number of worker threads (places) in the system, a cost that can become prohibitive in future architectures. Limiting the size of the cache, on the other hand, would reduce the effect of caching for more or less random victim selection policies. In addition, even if the memory overhead is not an issue, the overhead of parsing a stream is only reduced whenever the same victim is visited twice within a time-frame where many of the previously encountered items still exist. For larger numbers of threads the chances of choosing the same victim twice within a certain time-frame decrease.

From today's point of view we would solve stealing in priority work-stealing differently. Instead of stealing half the items, we would steal a single item, but cache the victim from the last steal operation, as well as the `StreamNode` and the temporary priority queue. Whenever we would run out of work, we would attempt to steal from the same victim again, unless the victim is also out of work, in which case we could delete both the cached `StreamNode` and priority queue and look for a new victim. Since we believe that *spying*, as used by our newer concurrent priority queues (e.g. the *concurrent LSM* in Section 5.8), provides even better results than can be expected from this approach, these improvements were never implemented in Pheet.

5.2.7 Correctness

We will now show that our priority work-stealing queues are both lock-free and linearizable. For linearizability we require purely local semantics as described in Section 5.1.2. Thus, all operations (`push`, `pop` and `steal`) that modify items owned by a specific thread (place) need to be linearizable with regard to each other, but do not need to be linearizable with regard to operations on items owned by another thread.

Lemma 5.2.1. *The `push` operation is wait-free.*

Proof. A new `StreamNode` is retrieved from our memory pool using our wait-free memory manager from Section 4.3. Apart from this, the only synchronization performed by `push` is adding the node to the stream, which is performed in a wait-free manner using an atomic write operation. □

Lemma 5.2.2. *The `push` operation is linearizable.*

Proof. The linearization point for the `push` operation is when the node is linked into the stream. From this point on an item is visible to all threads, and can be taken by any thread. □

Lemma 5.2.3. *The `steal` operation is lock-free.*

Proof. A `steal` will progress in the stream on each iteration, which means that it will eventually reach the end of the stream, unless the owner makes progress and adds new items. Once the priority queue is filled, `steal` will attempt to mark items as taken. If it succeeds, the algorithm terminates, otherwise another thread has made progress by marking that item as taken first. □

Lemma 5.2.4. *The `steal` operation can be made linearizable.*

Proof. The linearization point for `steal` needs to be in-between the time when the last node in the stream is encountered, and when a new node is added by the owner. The presented simplified implementation is not linearizable, since the stealing thread might try to mark an item as taken, which was marked as taken *after* a new item was added to the stream. Since the newly added item might have a higher priority than the item that will finally be returned (or `steal` might not return an item at all as if the queue were empty), this will violate linearizability. This can be easily fixed, though, by always rechecking the stream for new items, whenever an attempt to mark an item as taken fails. □

It is worth noting that stealing more than one item cannot be linearized that easily, and thus our implementation in Pheet does not have a linearizable `steal` operation for priority work-stealing.

Lemma 5.2.5. *The `pop` operation is lock-free.*

Proof. The first part of `pop` is wait-free, since each iteration will remove a single item from the local priority queue, which will never be updated by other threads. Thus, the number of iterations is bounded, regardless of whether it fails to return an item. The second part of `pop` resorts to `steal`, which is lock-free by Lemma 5.2.3. □

Lemma 5.2.6. *The `pop` operation is linearizable.*

Proof. The `pop` operation is linearized with regard to `steal` operations, either when an already modified `taken` flag is encountered, or when it attempts to mark an item as taken. □

5.3 Centralized k-priority Queue

Purely local ordering semantics, like the semantics provided by priority work-stealing are insufficient for an efficient execution of applications like our parallelized single source shortest path algorithm presented in Section 7.8. The centralized k-priority queue [142–144, 148] was our first attempt to implement a concurrent priority queue with stronger ordering guarantees. It is based on a centralized data structure design that is capable of supporting global ordering semantics. Since it has been shown that global priority queues exhibit a lot of congestion when used as part of a scheduling system [99], we appl¡ a temporal ρ-relaxation scheme (see Section 5.1.3) to the priority queue, which allows each thread to skip up to ρ of the most recent items. The priority queue also provides local ordering semantics in addition to the ρ-relaxed semantics (see also Section 5.1.4).

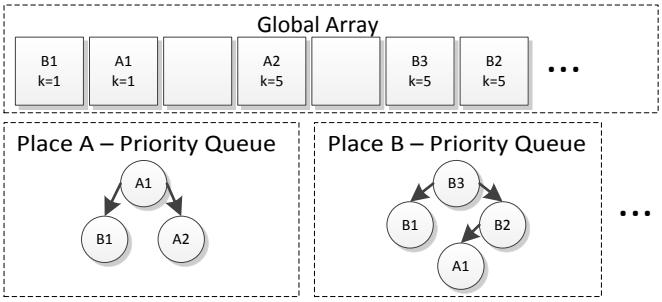


Figure 5.4: Centralized k-priority queue. Each place maintains its own priority queue with references to items in the global array. The newest (rightmost) items in the global array are only visible to the place that created them.

5.3.1 Internal structure

The basic idea of the centralized k-priority data structure is to create a global priority ordering between all the items stored in the priority queue, while allowing each thread to miss up to k of the newest items. As with the priority work-stealing queue from Section 5.2 we separate the concerns of synchronization and maintaining priorities. Synchronization is performed using a global, shared array, which is used to share items between all threads and to maintain information about which items must be globally visible to ensure that only the k most recent items are missed. Randomization is used to improve scalability when adding elements to the global array. Priorities are maintained using local priority queues for each place (thread). This is depicted in Figure 5.4. Any sequential implementation of a priority queue can be used for the local priority queues, since each priority queue is only accessed in the context of a single place, and therefore only by a single thread.

5.3.2 The `push` operation

The `push` operation stores the item, together with some additional information. For an item to be visible to all threads, it needs to be added to the global array. The items in the global array are stored in an order close to sequential. A task may only be placed up to k positions away from its correct sequentially consistent position.

Pseudocode for the `push` operation is shown in Listing 5.5. There, we choose a random position in the range from `tail` to `tail` $+ k$ and try to put the item into the array at the chosen position, if the position has not yet been taken by another item. In case the position is taken, a linear search is performed inside the `tail` to `tail` $+ k$ range until a free position is found or all positions have been checked. If all positions are taken, `tail` can be updated to `tail` $+ k$ and the search restarted. This scheme for adding items to an array was inspired by the k-fifo queues of Kirsch et al. [89].

As soon as the item has been added to the global array, a reference to it is added to the priority queue of the place at which it was created. This ensures that the centralized k-priority queue will exhibit local ordering semantics in addition to ρ-relaxed semantics as described in Section 5.1.4.

5.3.3 The `pop` operation

Pseudocode for the `pop` operation can be found in Listing 5.6. The `pop` operation checks whether `tail` has changed since the last time it was checked, and if so adds all the newly added items to the local priority queue. Each place maintains its own `head` index into the

Listing 5.5 Pseudocode for `push` in the centralized k-priority queue.

```
1 void push(Place place, Key key, Data data) {
2   Item it = new Item(place, key, data);
3
4   // Attempt until successful
5   while(true) {
6     int t = tail;
7
8     // Choose a random offset at which to put item
9     int offset = rand(0, k - 1);
10
11    // try all indices in k-range starting at offset
12    for(int i = offset; i < offset + k; ++i) {
13      int pos = t + (i % k);
14      // A tag of -1 refers to a taken item. We store pos
15      // in the tag field to omit the ABA problem
16      it.tag = pos;
17      // Try to put item into global array
18      if(global_array[pos].cas(null, it)) {
19        // Item was succesfully put into array
20        // Now put a reference into local priority queue
21        ItemRef ref = new ItemRef(pos, it);
22        place.prio_queue.push(key, ref);
23        return;
24    }}
25
26    // No more free slot found, try updating tail
27    // One thread will succeed, no need for checking which
28    tail.cas(t, t + k);
29  }
30 }
```

global array, to track which items have already been seen. Tasks that have been created by the same place can be omitted, since they were already added to the priority queue at the push operation. Next, the highest priority task is removed from the priority queue, and an attempt is made to mark the item as taken, by atomically setting the `tag` of the item to -1 using a *compare-and-swap* operation (CAS). Only one thread can succeed in updating the tag. In case of failure, the global array is rechecked for new tasks before trying again.

If the priority queue is empty, there can be up to k tasks stored after `tail` waiting for their execution, stored in the `tail` of the global array and its k subsequent positions. Since there are at most k tasks stored after `tail`, no priority ordering needs to be guaranteed if there are no tasks before the `tail`, and a random position can be checked for a task to execute. Since we allow for spurious failures on `pop` as long as another thread is making progress (executing a task), it is not necessary to exhaustively search for all tasks stored after tail, a random attempt to find an item suffices.

5.3.4 Additional implementation details

So far we have assumed that the global array used for storing items is unbounded. In practice, we implemented the global array as a linked list of arrays. Whenever an index is requested that is outside the bounds of the existing arrays, a new array is allocated and added to the end of the linked list using a single *compare-and-swap* operation.

Listing 5.6 Pseudocode for pop in the centralized k-priority queue.

```
1  Data pop(Place place) {
2    // Check for new items in global array
3    while(place.head < tail) {
4      if(global_array[place.head].place ≠ place) {
5        ItemRef ref = new ItemRef(place.head, global_array[place.head]);
6        place.prio_queue.push(ref);
7    }}
8
9    ItemRef ref;
10   while(ref = place.prio_queue.pop()) {
11     Data data = ref.it.data;
12     // Take item atomically by setting tag to -1
13     if(ref.it.tag.cas(ref.tag, -1)) {
14       // Success, return data
15       return data;
16     }
17     // Recheck for new items in global array again
18     ... // (not shown)
19   }
20
21   // Priority queue is empty, try to find random item
22   int offset = rand(0, k - 1);
23   if(global_array[tail + offset] ≠ null) {
24     Item it = global_array[tail + offset];
25     Data data = it.data
26     // Take item atomically by setting tag to -1
27     if(it.tag.cas(tail + offset, -1))
28       return data;
29   }
30   return null;
31 }
```

Each array in the linked list can be deleted as soon as all tasks stored in the array have been executed and the `head` indices of all places point to positions in arrays that are successors of the given array. The first condition can be lazily checked using our wait-free memory manager from Section 4.3. The second condition can be checked by atomically decrementing a reference counter whenever a head index moves on to the next array. If the reference counter was initialized to the number of places in the beginning, it is guaranteed that no place will scan the array for new tasks once the counter reaches zero.

It is also necessary to clean up the structure for storing *items*. Again we use our wait-free memory manager, and allow such a structure to be reused for a new item as soon as the previous item has been marked as taken. The use of a tag for each item, which is initialized to the item's position in the global array, guards against the ABA-problem, since positions for items are strictly increasing. Also, since items may be reused directly after the compare-and-swap, the data to return by a pop has to be read out before the compare-and-swap.

Both the `head` and `tail` indices in the data structure are strictly growing, therefore it is necessary to take a possible wraparound into account. We use 64-bit values, which ensures that wraparounds will only occur after a long time. Due to the long timespan between wraparounds, we consider it unlikely that an ABA problem will occur due to colliding indices.

While not shown here, we allow the parameter k used for quantitative relaxation to be

specified per item. The semantics for this are, that an item with a given value k can only be returned by a pop operation if no more than k items with a k less or equal the k-value of the returned item are skipped. To support this, the pop operation has to be modified, so that k is a parameter that is passed to the function instead of a global constant. Also, the value for k needs to be stored with the item. For the pop operation, the case where a random item behind tail is selected needs to be modified. First, a maximum value for k needs to be known that can be used to limit the random selection. In addition, if an item is found due to random selection it may only be returned if its k value does not exceed its distance to the observed tail. Otherwise we might skip more than k items, which can be the case if tail was updated in the meantime.

5.3.5 Correctness

In this section we argue that the centralized k-priority queue fulfils temporal ρ-relaxed semantics, is lock-free and linearizable.

Lemma 5.3.1. *The centralized k-priority queue fulfils temporally ρ-relaxed semantics for $\rho = k$.*

Proof. The execution can be divided into *epochs*, where an epoch ends whenever tail is updated to a new value. In each epoch, up to $\rho = k$ items can be added to the data structure without becoming immediately visible to all threads. After k items have been added, tail has to be updated before another item can be inserted, thus starting a new epoch.

Both push and pop operations need to be linearized within the epoch associated with the observed value of tail. As long as pop operations can be correctly linearized with regard to each other, the actual linearization order of push operations in the same epoch does not play a role, since the number of push operations in an epoch is bounded by ρ and only items added in the current epoch may be skipped. □

Lemma 5.3.2. *Even when allowing k to be specified per item, the centralized k-priority queue fulfils temporally ρ-relaxed semantics, where an item may only be returned if no more than $\rho = 2k - 1$ items with similar or stricter ρ-requirements are skipped ($\rho = 2k - 1$). Structural ρ-relaxation is fulfilled with $\rho = k$. A* `push` *operation may be linearized to a later epoch than the epoch of the observed value of* `tail`*, as long as the* `tail` *of the epoch it is linearized to has not passed the position of* `tail`*.*

Proof. We follow the argument from Lemma 5.3.1 where an execution is divided into epochs. The difference here is that an item added in a given epoch will not necessarily be visible in the next epoch, if it has a value for k larger than the item that triggered the epoch transition.

Both push and pop operations still need to be linearized within the epoch associated with the observed value of tail. Structural ρ-relaxation is still fulfilled with $\rho = k$, since the array after tail only allows storing at most k items with a smaller or equal value for k. Thus, a pop operation for an item with any k will skip at most k higher priority items for which k is less or equal the k of the returned item.

The bound for temporal ρ-relaxation is slightly weaker, since after an item with any value for k has been added, up to $2k - 2$ additional items with less or equal k can be added in the worst case, before tail is guaranteed to be moved past the item, thus guaranteeing it to be globally visible. □

Lemma 5.3.3. *The* `push` *operation is lock-free*

Proof. All memory allocation inside push is done using the wait-free memory manager from Section 4.3. The push operation tries to find an empty slot in the global array to insert its item. It searches k positions from a local copy of the tail index. If no empty slot is found, the tail

is moved forward at least one step, either by the current thread or a concurrent thread. This is repeated until an empty slot is encountered. If no empty slot is found, or the CAS used to insert the item fails, another thread must have succeeded in inserting at least one item. This is in accordance with the lock-free property. □

Lemma 5.3.4. *The `push` operation is linearizable according to temporally ρ-relaxed semantics.*

Proof. The linearization point of a `push` operation is the time at which a reference to the item is successfully written to the global array. It is guaranteed that the linearization point of `push` will not fall into an epoch where `tail` has advanced past the item, as required by Lemma 5.3.2, since `tail` can only be advanced past positions in the global array that contain items. □

Lemma 5.3.5. *The `pop` operation is lock-free.*

Proof. The `pop` operation has to check the global array for new items. This can be done in a bounded number of steps if no other thread is making progress or adds new items. After reading the global array, the operation tries to acquire one of the tasks referenced in the priority queue. The size of the priority queue only grows when another thread is making progress and adds new items to the global array. □

Lemma 5.3.6. *The `pop` operation is linearizable according to temporally ρ-relaxed semantics.*

Proof. According to Lemma 5.3.1, `pop` operations need to be linearized within the epoch of the last observation of `tail`, and also with regard to each other. Therefore, we choose to use the last observation of `tail` as linearization point. The only time when `pop` operations conflict, requiring a specific linearization order, is when two `pop` operations attempt to `pop` the same item, or if a `pop` operation observes an item that was already marked as taken by another thread. In these cases, the thread that fails to take the item will re-read `tail` afterwards, thus ensuring its linearization point is after the linearization point of the successful thread. □

5.4 Hybrid *k*-priority Queue

The hybrid *k*-priority queue attempts to combine the advantages of both priority work-stealing queues, as presented in Section 5.2 and the centralized *k*-priority queue from Section 5.3. The main idea is that each place maintains its own, local priority queue, and that synchronization is only performed if either a place runs out of work, or if the guarantees provided by *ρ*-relaxation are violated. To achieve this, we build upon the local variant of temporal *ρ*-relaxation: A `pop` operation is allowed to skip at most the $\rho = k$ most recent items by *each* thread (see also Section 5.1.3). The priority queue also provides local ordering semantics as described in Section 5.1.4, so that no locally created items will ever be skipped by a thread. For structural *ρ*-relaxation this leads to a bound of up to $\rho = k(P-1)$ items that can be skipped in total, where P is the number of places (threads) in the system.

5.4.1 Internal structure

The hybrid *k*-priority data structure consists of three components: (a) a global list storing items visible to all places, (b) one local item list per place, containing up to k items that are not guaranteed to be visible to all places, and (c) one priority queue per place storing references to items in the global and local lists, ordered by priority. As with both the priority work-stealing queue in Section 5.2 and the centralized *k*-priority queue from Section 5.3, we separate the concerns of synchronization and maintaining priorities. Synchronization is performed using both the global and the local lists of items. After more than k items have been added to a local

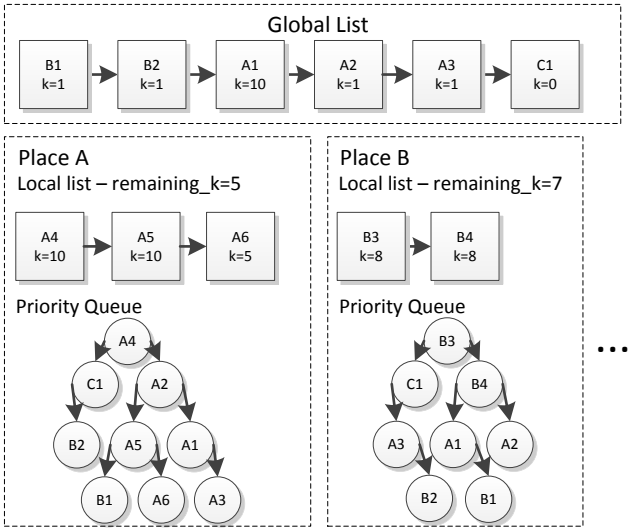


Figure 5.5: Hybrid *k*-priority data structure. Each place maintains its own priority queue with references to items. Each place adds new items to its local list as long as the ρ-relaxation guarantees are not violated. If adding a new item would violate these guarantees, the local list is appended to the global list, and a new local list is created.

list of items, the place that owns this list makes all local items globally visible by moving them to the global list. All places are required to regularly check the global list for new items.

Priorities are maintained using the local priority queues per place. Such a priority queue stores references to all items in the local linked list of items, as well as to all items in the global list. An item can be referenced by multiple priority queues at the same time, which is required to guarantee that no more than the k most recent items at each place are skipped by other places.

5.4.2 The `push` operation

The `push` operation (see Listing 5.7) adds a new item into the data structure. The `push` operation proceeds as follows: first, the task is inserted into the local list of the given place, and a reference is stored in the local priority queue of the place. Afterwards, a check is performed whether more tasks can be added without needing to publish any of the locally stored tasks, which is the case if the size of the local list is smaller than k. If any of the locally stored tasks needs to be made available globally, the local list of tasks is atomically appended to the global list. A new, empty local list is then created, which will be used in the next push operations.

5.4.3 The `pop` operation

The `pop` operation (see Listing 5.8) pops a reference to the highest-priority item from the local priority queue and tries to mark the task as taken by setting the `taken` flag with an atomic *test-and-set* operation. If it succeeds the item is returned. To make sure that no more than k item per place are skipped, the local priority queue has to be regularly updated with the newest additions to the global list. This is always done before a task is popped from the priority queue.

If the priority queue is empty after processing the global list, an attempt is made to find tasks stored locally at another place. This is called *spying*. Spying is related to stealing in the work-stealing algorithm in that a victim is selected and scanned for items that have not

Listing 5.7 Pseudocode for `push` in the hybrid k-priority queue.

```
1 void push(Place place, Key key, Data data) {
2   Item it = new Item(place, key, data);
3   // Place task in local list and priority queue
4   place.local_list.add(it);
5   place.prio_queue.push(new ItemRef(it));
6
7   // All items need to be made globally visible
8   // to not violate the ρ-relaxation requirement
9   if(place.local_list.size() == k) {
10    // Add local list to global list
11    do {
12      processGlobalList(place)
13    } while(¬global_list.tail.next.cas(null, local_list.head));
14    // Create a new local list
15    place.local_list = new List();
16  }
17 }
18
19 // Add references to unread items from
20 // the global list to the local priority queue
21 void processGlobalList(Place place) {
22   while(place.iterator≠global_list.tail) {
23     Item it = place.iterator.item()
24     // Do not add local or already taken tasks
25     if(it.place ≠ place and ¬it.taken)
26       place.prio_queue.push(new ItemRef(it));
27     place.iterator = place.iterator.next;
28   }
29 }
```

yet been taken. The main difference is that items that are encountered during spying are not removed from the owner's local list of items. Instead, references to all encountered items are stored in the priority queue of the spy. This is necessary to avoid breaking the ordering guarantees for the victim, but also greatly simplifies synchronization.

Spying is only required when the global list of items is empty, which also implies that no more than k items from each place are available. Due to this ρ-relaxation cannot be violated regardless of which item is returned next. While spying still makes an effort to return higher priority items before low priority items, the semantics of spying are designed to reduce synchronization costs to a minimum. Due to this, the spy is allowed to skip any number of items while scanning whenever an inconsistent state is encountered, and it will never retry reading an item. Due to spying it is possible for a place to store up to two references to a single item in its priority queue, one copy coming from spying, one from the global list. This does not affect the correctness however, since an item can only be marked as taken once.

5.4.4 Additional implementation details

For efficiency reasons, our implementation of the hybrid k-priority data structure does not use linked lists, but instead uses a linked list of arrays, which can be implemented in a manner similar to priority work-stealing, as described in Section 5.2.

Again, our wait-free memory manager from Section 4.3 is used to manage the data struc-

Listing 5.8 Pseudocode for `pop` in the hybrid k-priority queue.

```
1  Task pop(Place place) {
2    do {
3      processGlobalList(place);
4      // Try to take the highest priority task
5      while(¬place.prio_queue.empty()) {
6        Ref r = prio_queue.pop();
7        if(¬r.item.taken) {
8          Task ret = r.item.task;
9          if(r.item.taken.test_and_set()))
10           return ret;
11       }
12       processGlobalList(place);
13     }
14
15     // If the priority queue is empty, add references
16     // to remote tasks from a pseudo-random place
17     List vl = getRandVictim().local_list;
18     foreach(Item it in vl) {
19       if(it.place ≠ place and ¬it.taken)
20         place.prio_queue.push(new ItemRef(it));
21     }
22   } while(¬place.prio_queue.empty());
23   return null;
24 }
```

ture used to store all information about an item. Although we have not implemented this, we believe that alternatively, items can also be stored in-place in the linked list of arrays for higher efficiency. The data structure for items can be reused as soon as an item was taken, thus opening up the possibility of an ABA problem. To guard against this, items are associated with an `id`, and the `taken` flag is an integer of same size as the `id`. Similar to our priority work-stealing algorithm from Section 5.2, an item is seen as taken, whenever the `taken` flag does not match the `id` of the item. The `id` is stored alongside the item in the priority queue of each place to allow checking whether an item has been taken without running into ABA problems.

Spying does not put any tasks into the local task list of the spy (contrary to steal-half work-stealing), which makes the spy appear as being out of work when selected as a victim by other spies. To ensure a proper distribution of tasks throughout the whole system, each place stores a reference to its last successful spying victim. In case a victim is encountered with no local work, its last successful spying victim is checked instead.

As with the centralized k-priority queue, we allow k to be specified per item. To achieve this, we keep track of the minimum k encountered in the local list of a place, and require the local list to be published whenever the size of the local list equals the minimal k or if a new item added by `push` has a k smaller or equal the list size.

5.4.5 Correctness

In this section we argue that the hybrid k-priority data structure fulfils temporal ρ-relaxed semantics, is lock-free and linearizable.

Lemma 5.4.1. *Assuming $k > 0$, the hybrid k-priority queue fulfils local temporally ρ-relaxed semantics where the $\rho = k$ most recent items of each place are allowed to be skipped, except for locally created items. The hybrid k-priority also fulfils structurally ρ-relaxed semantics for $\rho = k(P-1)$.*

Proof. The execution can be divided into *epochs*, where a new epoch begins whenever a local list of items is appended to the global list. In each epoch, up to $\rho = kP$ active items can exist outside the global list. Since each thread is guaranteed not to skip its own items, no more than $\rho = k(P-1)$ items can be skipped by a `pop` operation under the assumption that it is linearized in the epoch associated with the state of the global list the `pop` operation observed. Thus, the hybrid k-priority queue fulfils structurally ρ-relaxed semantics for $\rho = k(P-1)$.

Since each place will publish its local list of items before more than k items are added, and since this will trigger an epoch transition, no more than the $\rho = k$ most recent items per thread can be skipped, thus fulfilling temporal ρ-relaxation semantics with $\rho = k$ per place. □

Lemma 5.4.2. *Both the temporal and the structural ρ-relaxation bounds from Lemma 5.4.1 still apply even when allowing k to be specified per item. Whenever an item with any k is returned by a `pop` operation, no more than (the most recent) ρ items with less or equal k will be skipped.*

Proof. All items added to the same local list of items are treated as if they had the same value of k, which is the minimum of all k values of items in the local list. The list is published whenever a new item would violate the requirements of ρ-relaxation. Thus, temporal ρ-relaxation per place is still fulfilled when allowing varying values for k.

For structural ρ-relaxation, it is guaranteed that when a `pop` operation returns an item associated with any value for k, each other place will either have at most k items in its local list, or only items with a k larger than the k of the returned item. Therefore still at most $\rho = k(P-1)$ items with less or equal k can be skipped by a `pop` operation. □

Lemma 5.4.3. *The `push` operation is lock-free.*

Proof. All memory allocation in `push` relies on the wait-free memory manager from Section 4.3. When there are less than k items in the local list, the entire push operation is done locally and is thus wait-free. When the local list has k items, it is added to the global list. This step requires making sure that the entire global list has been read and then adding the local list to the end. Adding the local list to the global list can fail if another place adds its list first, but this means another place made progress. Reading and adding to the global list is thus lock-free. □

Lemma 5.4.4. *The `push` operation is linearizable according to ρ-relaxed semantics.*

Proof. All `push` operations are linearized when the item is added to the local list. Before this point the task is not visible to any other thread, while after the point it can be spied and taken by any thread. Since publishing the local list is performed by the same thread as `push` it is guaranteed that a `push` operation will fall into an epoch between the last time the same thread published a local list, and the time the list containing the item is published (regardless of whether the item is taken in the meantime). Within this time-frame no more than k items can be added, so that ρ-relaxation guarantees are fulfilled. □

Lemma 5.4.5. *The `pop` operation is lock-free.*

Proof. At certain points the `pop` operation needs to make sure it has read the entire global list. The global list can only grow if another place is making progress, which makes reading the list lock-free. Multiple places may try to acquire the same item, but only one will successfully take it. The number of already taken items can only grow if another place is making progress.

If the priority queue is empty and the global list has been read, an attempt is made to spy on the local list of another place. The length of the remote local list is bounded by k, making spying wait-free. □

Theorem 5.4.6. *The pop operation is linearizable according to ρ-relaxed semantics.*

Proof. A pop operation is linearized at the point when the global list was checked for new items. When two pop operations conflict on an item, the pop operation that did not take the item has to recheck the global list to ensure its linearization point is after the linearization point of the conflicting pop operation, since the pop operation might already have been linearized in a later epoch. It is also necessary to recheck the global list directly after spying, since spying might have found an item added in a later epoch. □

5.5 Two-level Concurrent Ordered Container

Due to the separation between synchronization and priority queue implementation, the concurrent priority queues presented in the previous sections can be modified to implement concurrent versions of any type of ordered container. This can be used to support specialized priority queues for specific applications (see also Section 2.5.1), or even to compose different priority queues inside a single data structure as required by strategies (see Section 2.7.7).

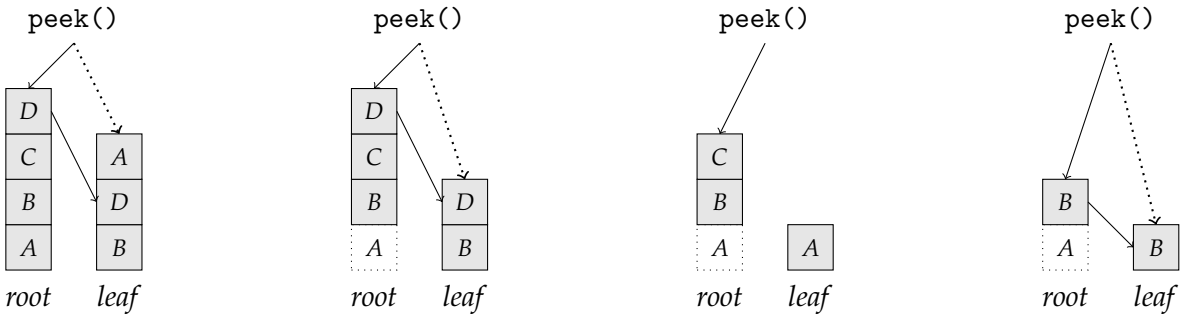
Nonetheless, the synchronization mechanism being oblivious of the type of ordered container in use can also be a disadvantage, since it is impossible for the synchronization algorithm to take advantage of the ordered container's structure. This leads to double bookkeeping for items, and maintenance work being performed once for the synchronization mechanism and once for the local ordered container. In addition, if some of the items stored in the data structure require a strong synchronization mechanism to maintain ordering guarantees, this mechanism needs to be used for the whole application. This can reduce scalability when used as part of a task scheduling system, where not all tasks require strong guarantees on their execution order.

In this section, we present a concurrent ordered container that can be used to compose different types of concurrent ordered containers into a single data structure. Such a data structure helps to support composable scheduling strategies as presented in Section 2.7.7, by allowing tasks with specific strategies to use a concurrent ordered container specialized to the problem, while at the same time providing a clear set of rules for ordering items with different strategies. This is made possible by a two-level approach to the data structure design, where at the *root level* a single concurrent ordered container determines the order between all items stored in the container, and at the *leaf level* items of same type can be reordered with regard to each other depending on their specific ordering requirements.

5.5.1 Internal structure

The two-level container consists of a two-level hierarchy of individual ordered containers, laid out as a tree. We call the container at the root of the tree the *root container*, and the other containers the *leaf containers*. The root container stores all items added to the two-level container, whereas leaf containers only store disjoint subsets of items. Interaction only occurs between the root container and a leaf container, for items stored in the leaf, but never between two leaves.

The root container determines the order in which items are returned by a pop operation. Leaves can override this order for its own items, but never for items of other types. To allow for more flexibility, a leaf is allowed to store additional information, not accessible to the



(a) `peek()` → A (b) `peek()` → D (c) `peek()` → C (d) `peek()` → B

Figure 5.6: Ordering semantics in the two-level concurrent ordered container: The leaf overrules the ordering of items stored in it, but cannot influence the ordering of items not in the leaf.

root, alongside all its items. To make this possible, the leaf is responsible for the memory management of the data structure storing an item. The memory management scheme used by the leaf needs to be at least as strict as required for the root. For items that are not stored in any leaf, the root container also takes over the role of the leaf.

5.5.2 Ordering semantics

Each container enforces ordering constraints on the items it owns as well as the items that were shared with it. Since ordering constraints by two containers may contradict each other, disambiguation is necessary. Generally, ordering constraints of leaves overrule ordering constraints in the root. So two items that both occur in the same leaf container will be ordered according to this container and not the root.

To give a better understanding of how the ordering constraints are enforced, we now give a small example, which is presented in Figure 5.6. For this we assume a two-level container with a single root and a single leaf container. Four items, A, B, C and D are stored in the two-level container. From the items stored in the two-level container, only C is not stored in the leaf. We assume that the root container would return its items in the order D, C, B, A, whereas the leaf would return its items in the order A, D, B.

Now, as explained before, a leaf container overrules the ordering of the root when contradictions occur. There is a contradiction on A, which is supposed to come before D according to the leaf. Since the leaf overrules the root, the final order will therefore be A, D, C, B. It is important that B is still not allowed to be executed before C since executing it after C does not contradict the ordering rules of the leaf.

For efficiency reasons, if there are multiple valid orderings in a leaf, any of these orderings can be chosen and used to overrule the ordering of the parent container. This is even allowed in cases where there exists an ordering that does not contradict the root, and another one that does. So if, in the previous example, the order of B and D in the leaf is irrelevant (e.g. because both of them have an equal key, if the leaf is a priority queue), then both A, B, D, as well as A, D, B are valid orderings in the leaf, resulting in either A, B, D, C or A, D, C, B for the total order depending on which ordering is chosen by the leaf.

5.5.3 The base class for stored items

Each container in the two-level data structure can have its own implementation for the data structure for storing items owned by it, optimized for its requirements. Since items are shared between containers, a common way to access functions for all containers is necessary. In the description of this data structure we use an object-oriented interface to make this possible.

Listing 5.9 Pseudocode for the base class for items stored in the multi-level container.

```
1 class StoredItem {
2 public:
3   // Check to see whether an item is still active
4   // (Has not yet been taken by any thread)
5   virtual bool active(int expected_version);
6
7   // Atomically take item from container. Only one thread may succeed.
8   // Item will be inactive afterwards
9   virtual bool take(int expected_version);
10
11  // The container responsible for the management of the item
12  Container* leaf_container;
13
14  // The actual data
15  Data data;
16
17  // A version number used to get around the ABA problem
18  int version;
19 }
```

The common interface for items stored in the multi-level data structure is shown in Listing 5.9. It provides the method `active` to check whether an item is still logically part of the data structure, and the method `take`, which atomically tries to (logically) remove the item from the container. Only one thread can succeed in taking an item, and after a successful `take` all threads will see the item as inactive. In addition a pointer to the leaf container is stored alongside the item. A version number is also stored with items and is used to get around the ABA problem when an instance of the `StoredItem` class is reused. Calls to `active` and `take` both require an expected version to be specified and will only succeed if the version matches the current version of the item. All implementations of `take` are required to increment the version number on success. Items may only be reused after `version` has been incremented.

5.5.4 The `push` operation

The `push` operation, for which pseudocode is shown in Listing 5.16, first determines which data structure needs to be used when the leaf container for the item calls the `allocate_item` function of the leaf to allocate an instance of the `StoredItem` class for storing the item. Afterwards, the item is first stored in the leaf container using `push`, and then in the root.

5.5.5 The `pop` operation

Listing 5.11 shows how the `pop` operation is implemented. First the next item in order according to the root container is determined using its member function `peek`. Next, the leaf container for the returned item is determined. The leaf is then checked for the next item according to its own ordering using `peek`. The item returned from the root container is passed

Listing 5.10 Pseudocode for `push` in the two-level container.

```
1 void push(Item item) {
2   // Get leaf data structure for this type of item
3   Container* leaf = leaves[item.type];
4
5   // Let leaf allocate the data structure for storing the item
6   StoredItem* leaf->allocate_item(item);
7
8   // Push item to the leaf
9   leaf->push(item);
10
11  if(leaf != root) {
12    // Push item to the root (makes item visible to all threads)
13    root->push(item);
14  }
15 }
```

on to `peek` as a *boundary item*, to ensure that only items are returned that come before the boundary item according to the ordering semantics of the leaf. Finally, an attempt is made to mark the item returned by the leaf container as taken, and the item is returned if this succeeds.

To ensure linearizability of `pop` operations it is necessary to guarantee that the result of `peek` is still the same when `peek` is called for the leaf container. To ensure this, `peek` returns an ABA-safe snapshot, which contains the item, the version number of the item, as well as additional, implementation-specific information that is used to guarantee that if the same snapshot is returned later, no higher priority item existed in that container at any time between the two calls to `peek`. This is used as a kind of transaction to ensure that the result is only returned if no changes potentially invalidating the result occurred in the meantime.

5.5.6 Correctness

In this section we argue that the two-level concurrent ordered container is lock-free and linearizable under the assumption that the specialized data structures used inside are lock-free and linearizable.

Lemma 5.5.1. *The `push` operation is lock-free.*

Proof. The `push` operation does not perform any synchronization by itself, and instead only calls functions of the root and leaf containers. Since each function is called only once, if these functions are lock-free, so is `push` of the two-level container. □

Lemma 5.5.2. *The `push` operation is linearizable.*

Proof. The linearization point for the `push` operation of an item depends on how a corresponding successful `pop` finds the item.

- The linearization point of the `push` operation on the root container, if the item is already returned by the `peek` on the root container, or when the boundary item returned by the root container was added after the item.
- The linearization point of `push` on the leaf container, if the boundary item returned by `peek` on the root container was pushed to the root container before this item was added to the leaf container.

Listing 5.11 Pseudocode for pop in the two-level container.

```
1 Data pop() {
2   while(true) {
3     // ABA-safe snapshot of the result of peek.
4     auto root_p = root->peek();
5
6     // Item that should be returned next according to the root container
7     StoredItem* si = root_p->item;
8
9     if(si == nullptr)
10      return nullptr;
11
12    Container* leaf = si->leaf_container;
13
14    // Use the item from the root container as boundary item for leaf->peek
15    auto leaf_p = leaf->peek(si, root_p->version);
16    StoredItem* si2 = leaf_p->item;
17
18    Data data = si2->data;
19    // Return data, if boundary was valid all the time,
20    // and we manage to mark item as taken
21    if(root_p == root->peek() && si2->take(leaf_p->version)) {
22      return data;
23    }
24  }
25 }
```

- The linearization point of when the boundary item was added to the root container if this happened between the two push operations on leaf and root. □

Lemma 5.5.3. *The pop operation is lock-free.*

Proof. The pop operation consists of a loop, which is repeated until either the root container does not return an item (either because it is empty, or due to a spurious failure), or when a valid item is found and marked as taken. An item will not be seen as valid if it cannot be guaranteed that the result from peek on root was still the same while calling peek on the leaf. This can only happen if another thread made progress by either adding an item to, or removing an item from the root container. An unsuccessful attempt to mark an item as taken can only occur if another thread succeeded in marking the item as taken. □

Lemma 5.5.4. *The pop operation is linearizable.*

Proof. A successful pop call is linearized at the linearization point of the last call to peek on the leaf container. At this point it is guaranteed that the given item fulfils the composed ordering constraints from the root container and the leaf. A failed pop is linearized at the linearization point of peek on the root container. □

5.5.7 Generalization to more than two levels

Scheduling strategies, as presented in Section 2.7, allow complex hierarchies of strategies, where each type of strategy brings with it different ordering constraints. This requires a generalization of the two-level concurrent ordered container to arbitrary levels.

Fortunately, this is fairly straightforward to do: The `push` operation needs to be modified to traverse the hierarchy of containers starting at the leaf container and moving on through the parents until the root container, calling `push` on each container on the way. The `pop` operation needs to traverse the hierarchy in the other direction, from the root to the leaf, requesting an ABA-safe `peek` snapshot at each level, and using the result from `peek` as a boundary item at the next level.

5.6 Root Container based on Work-stealing Deques

In this section we present a data structure based on work-stealing deques that can be used as a root container for the two-level concurrent ordered container. The behaviour is similar to ABP work-stealing [11]: It provides purely local semantics with stack-like behaviour for local accesses. When a thread runs out of work it will attempt to steal the oldest task from a victim thread.

5.6.1 Internal structure

The root container consists of a centralized component, shared by all threads, and multiple *places*. A place is associated with a single worker thread, which is called the *owner* of the place. Each place contains a deque for items. The owner of a place accesses its deque like a stack. To be consistent with other literature on work-stealing deques, we have these stack-like local accesses occur at the *bottom* end of the deque. The *top* end is reserved for remote accesses, and is only accessed by the local thread if top and bottom refer to the same element.

Each deque consists of a (virtual) array of infinite size, where *top* and *bottom* are marked by indices t and b. The bottom index b can only be accessed by the owner of the deque, and can be both incremented and decremented. The top index t, on the other hand, is accessed by all threads using atomic *compare-and-swap* operations, but will never be decremented. An item stored at index i in the array, which is owned by the root container, is in the deque iff $t \leq i < b$.

A main difference between this and other implementations of work-stealing deques is that the root container is also responsible for storing items that are managed by other containers, which is required for it to be part of the two-level concurrent ordered container. This requires for `pop` operations to be split into `peek` and `take` operations, and for the root container to work correctly and efficiently even for items stored at a leaf container, for which the leaf container's implementation for `take` is used. The root container will only later see that an item has been taken, and has to clean out such items after they have been removed. With a few modifications to the standard work-stealing algorithms it is possible to skip inactive items without need for complex synchronization. It can happen that an item has been marked as taken by a leaf, but has not been removed from the root container so far. If such an item is then reused, this can lead to an ABA problem in the root container. This is avoided in our implementation by storing the version number of the stored item alongside the reference to the item and skipping all items with a version mismatch.

5.6.2 Implementation of the centralized component

The centralized component of the root container is mainly used to relay calls to member functions to the place associated with the current thread. This is shown for the `push` method in Listing 5.12. We assume that each thread in the system is associated with a unique *place-id* and that these id's are subsequently numbered starting from 0, a standard feature in the *Pheet*

framework presented in Chapter 6. This place-id is then used as an index in an array of places to gain access to the place-specific component of the data structure.

Listing 5.12 Pseudocode for the `push` method of the root container

```
1 void push(StoredItem* item) {
2   // Get the id of the place associated with the current thread
3   int place_id = get_place_id();
4
5   // Call push for the given places
6   this->places[place_id]->push(item);
7 }
```

5.6.3 Implementation of stored items

Items owned by the root container do not need much auxiliary data to be implemented. The only additional information that is required is the place at which the item is stored, as well as the index at which it is stored in the local deque of the place. This can be seen in Listing 5.13. To check, whether an item at index i is `active`, it suffices to check whether $t \leq i < b$. After these checks, the version needs to be checked to ensure ABA-safety.

We say an item is *managed* by the root container, if the root container is also the leaf container for the given item. This means that the root container is responsible for the memory management of the item, but also that the item will be stored in no other container than the root. Items managed by the root container will always only be stored in a single deque, which is the deque of the place that called `push` for the given item. Also, items cannot occur in any deque more than once. Since all accesses to items are made ABA-safe by the version number, an item managed by the root container can be reused as soon as it has been marked as taken, and for this we use our wait-free memory manager from Section 4.3.

Listing 5.13 Pseudocode for the implementation of items owned by the root container.

```
1 class RootStoredItem : public StoredItem {
2   // The place the item is stored in
3   RootContainerPlace* place;
4
5   // The index at which it is stored
6   int i;
7
8 public:
9   // An item is active if t ≤ i < b
10  bool active(int expected_version) {
11    return i >= place->t && index < place->b
12      && version == expected_version;
13  }
14
15  // Omitted, will be described later
16  bool take(int version);
17 }
```

The member function `take` is shown separately in Listing 5.14. This function is called both for local, as well as for steal accesses. Only one thread may succeed in taking an item, and afterwards the item will not be active any more. Similarly to other work-stealing algorithms,

a `take` by the owner first starts off by decrementing b. The decrement needs to be sequentially consistent with updates to t by other threads. This ensures that the value of t that will be read next is the highest value for t any thread that misses the decrement to b will see. If after the update, b is still greater than t, it can safely be processed. If b equals t, the local thread needs to synchronize with stealing threads with a *compare-and-swap* to t. The thread that succeeds to increment t will successfully take the item. In case $b < t$, the queue is already empty, and a take will fail. In this case, b will be set to the value of t. This is safe since even before the last update to t is guaranteed to have been the last update, since it removed the last element.

Listing 5.14 Pseudocode for the implementation of `take` for items owned by the root container.

```
1  bool take(int expected_version) {
2    // Read out local values
3    int i = this->i;
4    auto p = this->place;
5
6    // ABA safety check
7    if(this->version != expected_version)
8      return false;
9
10   if(p->id == get_place_id()) {
11     // Local access, take by decrementing b
12
13     // Decrement b (requires sequential consistency)
14     --p->b;
15
16     if(p->b > p->t) {
17       // There was more than one item in-between, item can be safely taken
18       return true;
19     } else if(p->b == p->t &&
20       p->t.cas(p->b, p->b + 1) {
21       // Needed to update t and succeeded
22       ++p->b;
23       return true;
24     } else {
25       // Safe, since t is guaranteed not to change at this point
26       p->b = p->t;
27     }
28   } else {
29     // Steal access, take by incrementing t
30
31     // Check whether queue is not empty or fail
32     if(i >= p->b) return false;
33
34     // Try taking item (requires sequential consistency)
35     if(p->t.cas(i, i+1)) return true;
36   }
37   return false;
38 }
```

Stealing accesses in `take` always work on the top end of the deque. It is only safe to be performed if $t < b$. Due to sequential consistency between writes to t and b, the value read for b for this check is guaranteed to be no older than the value for t. Steal accesses only occur for items at the bottom of the deque, therefore we can assume that $t = i$ for the item being

stolen, where i is the index stored with the item. A *compare-and-swap* on t that only succeeds if $t = i$ ensures that only one thread can succeed in taking the item. No ABA problem can occur for stealing accesses since t is never decremented.

5.6.4 The `allocate_item` method

The `allocate_item` method, which is shown in Listing 5.15, is only called for items owned by this container, and is responsible to allocate memory for storing an item and to return a reference to it. We use the wait-free memory manager from Section 4.3 for memory management.

Listing 5.15 Pseudocode for the `allocate_item` method of the root container

```
1  StoredItem* store_item(Data data) {
2    // Create an object storing the item along with auxiliary data
3    RootStoredItem* ref = RootStoredItemPool.get_item();
4
5    // Increment version number for item to omit ABA problem
6    ++ref->version;
7
8    // Initialize
9    ref->data = data;
10   ref->leaf_container = this;
11   ref->place = this->places[get_place_id()];
12
13   // Store index at which item will be stored
14   ref->i = b;
15
16   return ref;
17 }
```

5.6.5 The `push` method

The `push` method of the root container is fairly straightforward, and is comparable to the `push` operation in the work-stealing deques by Arora et al. [11]. This can be seen in Listing 5.16. An item is stored at position b in the array, and b is incremented afterwards, making the item visible. The version number of the item is stored alongside the item, to avoid an ABA problem where threads attempt to steal already reused items.

Due to the fact that items managed by leaf containers can be marked as taken in any order, it can occur that some items in the range between t and b are marked as taken. For efficiency reasons such items can be skipped without synchronization. This can lead to stealers encountering an inconsistent state due to an ABA problem whenever an item is removed from the bottom, and a new item is pushed into the same position. To omit this, we increment the `epoch` of our deque whenever such a case occurs, thus resulting in a spurious failure on steals.

5.6.6 The `peek` method

The `peek` method is presented in Listing 5.17. It returns the item stored at the bottom of the deque. If it encounters inactive items on the way, it will physically remove those items by either updating b, if the queue is not empty, or by updating t for an empty queue. It is sufficient to update t using an atomic write, because any write to t by other threads that might be overwritten due to this is guaranteed to write a value less or equal to b.

Listing 5.16 Pseudocode `push` method in the place specific part of the root container

```
1 void push(StoredItem* ref, int item_version) {
2   // Omit a problem where stealers may encounter an inconsistent state
3   // by incrementing epoch whenever a push follows a pop
4   if(decremented_b) {
5     ++epoch;
6     decremented_b = false;
7   }
8
9   // Store reference in the virtual array
10  items[b].ref = ref;
11  // Store version number of item along with reference to avoid an ABA problem
12  items[b].version = item_version;
13
14  // Increment b, thus making item visible
15  ++b;
16 }
```

The return value of `peek` is an ABA-safe snapshot of the result, which contains the version number of the item, and the epoch of the queue in addition to a reference to the item.

5.6.7 The `spy` method

When a thread runs out of work in its local deque, it tries to *steal* work from another thread. Due to the structure of the two-level container, which splits a `pop` into a `peek` and a `take` operation, it is necessary to do the same for stealing. Therefore, we split a steal into a `spy` and a `take` operation. In the root container, a `spy` operation will only be called if the local deque is empty, and only a single item will be spied. This item will not be stored locally.

To give deterministic results, thus allowing the two-level container to call `peek` twice and get the same results as long as the previously returned item has not been taken, `spy` will cache the result. If the item stored in the cached result is still active, it will be returned. Otherwise, `spy` selects a potentially different victim according to its victim selection policy (see Section 3.5). For this victim, the `spy_from` method is called, which is shown in Listing 5.18.

The `spy_from` method works similar to `pop` but starts at the top of the queue instead of the bottom. It will first take a snapshot of `epoch`, t and b, and will then search the queue for an active item. If an active item was found, the epoch is rechecked, to ensure a consistent state was encountered. This might not be the case, if b was decremented and later incremented again, thus making it possible that some of the scanned positions were modified while scanning.

After ensuring that a consistent state was encountered, an attempt is made to update t to the position before the spied item, thus permanently skipping all encountered inactive items. For items managed by the root container, this is necessary for the correctness of the `take` operations, for all other items this is not necessary, but reduces the amount of items other threads will need to scan on their `spy` operations. Finally, after t was updated, the validity of the item is rechecked, and the item then returned if it is still valid.

5.6.8 Implementing a virtual array of infinite size

So far we have worked on the assumption that there exist an infinite global array of items. For the root container, this infinite array can be replaced by a doubly linked list of fixed-size arrays. Both ends of the deque are marked by pointers to the *top* and the *bottom* block. Each

Listing 5.17 Pseudocode of the `peek` method in the place specific part of the root container

```
1  struct RootPeekState {
2    StoredItem* item;
3    int version;
4    int epoch;
5  }
6
7  RootPeekState peek(StoredItem* boundary) {
8    // Take snapshot of t and b
9    int t' = t;
10   int b' = b;
11
12   while(b' > t') {
13     RootPeekState state;
14     state->item = items[b' − 1].ref;
15     state->version = items[b' − 1].version;
16
17     // Skip inactive items (only happens for items managed by other containers)
18     if(!state->item->active(state->version)) {
19       --b';
20       continue;
21     }
22     state->epoch = epoch;
23
24     // Item is active, it is safe to update b
25     b = b'
26
27     // Found an item, return it
28     return state;
29   }
30
31   // No item found, update t to the value of b to signal queue is empty
32   t = b;
33
34   // Try to find an item to steal
35   return spy();
36 }
```

block is assigned an offset, which is used to translate global indices for t and b into indices in a block. Offsets for blocks are strictly increasing, and a newly added block has the offset of its predecessor incremented by its block size.

The bottom block pointer is only accessed by the owner of the deque, and always points to the block that stores the item at position b. The pointer to the top block can be updated by all threads. Whenever a block only contains indices smaller than t, stealing threads will try to update the top block pointer to the block that contains the index t and to unlink all previous blocks from the linked list. To avoid conflicts, a lock is used to protect this update operation. Threads that fail to acquire the lock can still perform a steal operation, therefore progress is not blocked.

To support infinitely growing indices on finite integer types, wraparounds for indices need to be supported. To support this, all comparisons between indices need to take potential wraparounds into account. This can be done by comparing the difference between two indices to 0 instead of doing a direct comparison. This is safe, as long as the length of a deque ($b - t$)

Listing 5.18 Pseudocode for the `spy_from` method in the place specific part of the root container.

```
1  RootPeekState spy_from() {
2    // Return value
3    RootPeekState state;
4
5    // Take a snapshot of the epoch, t and b (exactly in this order)
6    state->epoch = epoch;
7    int t' = t;
8    int b' = b;
9
10   // Iterator
11   int i = t';
12
13   while(b' > i && !items[i].ref->active()) {
14     // Skip inactive items
15     ++i;
16   }
17
18   if(b' ≤ i || b ≤ i) {
19     // No more items stored in original range, fail
20     return null;
21   }
22
23   state->item = items[i].ref;
24   state->version = items[i].version;
25
26   if(e != epoch) {
27     // Encountered state might be inconsistent, therefore fail
28     return null;
29   }
30
31   // Try to correct t if items were skipped. (Sequential consistency needed)
32   if(t' ≠ i && !t.cas(t', i)) {
33     // Some other thread was faster, may still progress if item is not managed
34     // by root container, in this case the CAS to t is just maintenance work
35     if(state->item->leaf_container == this) return null;
36   }
37
38   // Ensure that item is still valid after t was updated
39   if(b ≤ i || state->epoch ≠ epoch) return null;
40
41   return item;
42 }
```

never exceeds half the range of the (unsigned) data type used for indices. An index type with a size similar to the system pointer type should therefore suffice, since the system will run out of memory before a wraparound leads to a problem.

5.6.9 Memory management

For a correct implementation of the root container in systems without garbage collection, it is necessary to perform memory management for items stored in the container. For this, the wait-free memory manager presented in Section 4.3 is used. An item can be reused as soon as it is not active any more, meaning that another thread succeeded in taking it. To avoid ABA problems, we add a version number for each item to the interface for stored items. The version number is stored alongside the pointer to the item in the deque, and whenever a check is performed whether an item is active, it also checks whether the version number of the item is still the same as the version number stored alongside the pointer to the item. All containers used in the multi-level ordered container need to update the version number of their items, before they are reused.

The blocks that are used as a replacement for the virtual global array also require memory management, which is again done by a reuse scheme for each block. Empty blocks found at the bottom end can simply be left in the list, since the next time a new block is needed, it will be added at the bottom end. So whenever a bottom block becomes empty, the bottom block pointer is moved to the predecessor, but the predecessor still has the previous bottom block as its successor. When a new bottom block is needed, the current bottom block is first checked for a successor. Blocks becoming empty at the top end, on the other hand, will never be needed and therefore need to be reused. When they are unlinked from the linked list of blocks, a flag is set marking them ready for reuse, and again the wait-free memory manager from Section 4.3 is used to find a new block to use when required. To avoid an ABA problem with reused blocks, before an item is successfully spied a consistency check needs to be performed on the expected index of the item, and the indices serviced by the block.

5.6.10 Correctness

In this section we argue that the root container is lock-free and linearizable. The methods `active`, `take`, `allocate_item` and `push` are trivially wait-free and proofs are omitted.

Lemma 5.6.1. *The `active` check for items is linearizable, unless it is part of an inconsistent state, which will always lead to a spurious failure of `spy`.*

Proof. A successful `active` check has its linearization point when t is read. This is trivial for the owner, since it is the only thread that can change b and `version`. For stealing threads, if the given item is in the deque, it must have been in the deque at some point before the call to `active` or it would not have been found. Since t cannot be decremented, and the last item in the deque is always taken by an increment to t, if t was read and the item stored at position t then it must have been active when t was read.

An exception to this is when a `spy` operation scans items in the deque, while the owner removes an item, only to `push` a new item in place of the old one. This can lead to an inconsistent state. While no guarantees can be given on the results of `active` on an inconsistent state, Lemma 5.6.7 guarantees that `spy` will fail when an inconsistent state is encountered.

A failed `active` check is linearized either at the read of t, if the item was stored at a position smaller than t, at the read of b if it was stored at a position greater or equal to b, or at the read of `version` if the item was already reused in the meantime. □

Lemma 5.6.2. *The `push` operation is linearizable.*

Proof. The `push` operation has its linearization point when b is incremented. The increment to `epoch` can be linearized separately, since a `spy` operation with its linearization point after the increment to `epoch`, but before the linearization point of `push` either spuriously fails or observes a correct state. □

Lemma 5.6.3. *The `take` method is linearizable.*

Proof. A successful `take` by the owner of the deque is linearized when b is decremented. If t needs to be updated as well, it is linearized with regard to other threads trying to update t at the successful update to t. A failed `take` by the owner is linearized when t is read, or on a failed update of t.

Successful `take` operations by threads other than the owner are linearized when t is updated, or when b was updated by the owner to equal t, whichever comes first. The update to b will only occur if a single element remains in the queue and the owner wants to `take` it as well. For this to be linearizable, the updates to b and t that attempt to `take` an item need to be sequentially consistent. Failed `take` operations by threads other than the owner have their linearization point either at the read of b or at a failed update of t. □

Lemma 5.6.4. *The `peek` method is wait-free if `spy` is wait-free.*

Proof. The `peek` method may skip items already processed by other threads, but the amount of items that can be skipped is bounded by the difference between b' and t'. This comes from the fact t' is a snapshot that does not change, and b' is decremented at each iteration. Either `peek` succeeds, or at some point $t' \geq b'$ and the deque is empty, leading to a `spy` operation. The owner of the deque is the only thread that can call the `peek` method. It is also the only thread that will `push` new items to the deque, which would increment b. Therefore it is guaranteed that `peek` will skip at most $b' - t'$ items, or no items at all if $b \leq t'$. Since all checks in the `peek` method are wait-free, and the number of items to be checked is bounded, `peek` is wait-free, if `spy` is wait-free. □

Lemma 5.6.5. *The `peek` method is linearizable.*

Proof. If the deque is found to be non-empty, the linearization point of a `peek` is at the linearization point of the `active` check for the returned item. Otherwise the linearization point is the linearization point of the `spy` operation. □

Lemma 5.6.6. *The `spy` method is wait-free.*

Proof. The `spy` method selects a victim and calls `spy_from` for the victim. The `spy_from` method only checks the range of items available at the beginning of the call, and ignores any items added afterwards. Each item is looked at exactly once until either an active item is found, or no more items exist in this range of items. Since all checks performed on items are wait-free and the number of checks is bounded, `spy_from` is wait-free. □

Lemma 5.6.7. *A spurious failure will occur for `spy` whenever an inconsistent state is encountered.*

Proof. An inconsistent state occurs in `spy` if the version number read is the version number of another item than the item read, or if an inactive item is skipped, which is replaced by an active item later. In both cases b is decremented while `spy` is scanning the deque, so that `spy` scans past the bottom of the deque. This will either be recognized if after the scan $b \leq i$ or if the `epoch` is updated in the meantime, which is always done before b is incremented after a decrement, guaranteeing that `spy` will see an epoch change if it missed out on the decrement because of the later increment of b. □

The semantics of spy allow spurious failures as long as another thread makes progress. If an item is returned, the only restrictions are that it has to be active at the linearization point of spy and that there are no more active items before it. For items owned by the root container, another restriction is that the item has to be stored at t in the deque at the linearization point of spy.

Lemma 5.6.8. *The spy method is linearizable.*

Proof. A successful spy has its linearization point in the successful spy_from call. A failed spy can be linearized at any point in the call, since spurious failures are allowed.

A successful spy_from is linearized at the linearization point of the call to active for the item that is returned. Since spurious failures are allowed, an unsuccessful call can be linearized at any point during the call. □

Lemma 5.6.9. *The spy_from method always returns the first active item after t.*

Proof. The spy_from method looks at items in order starting from the item at position t. It is only allowed to skip an item and move on to the next if the item is inactive. For an active item to occur before the returned item, this item has to be added after the active check to the item previously stored in the same position. For an item to be added at a certain position there cannot be any active items stored at a position after the item, due to the stack-like behaviour of accesses by the owner. If an active item is observed at a position after the first active item by the spying thread, this item must either be observed before the push of the new item, or the observed item must have been added after the other item has been added.

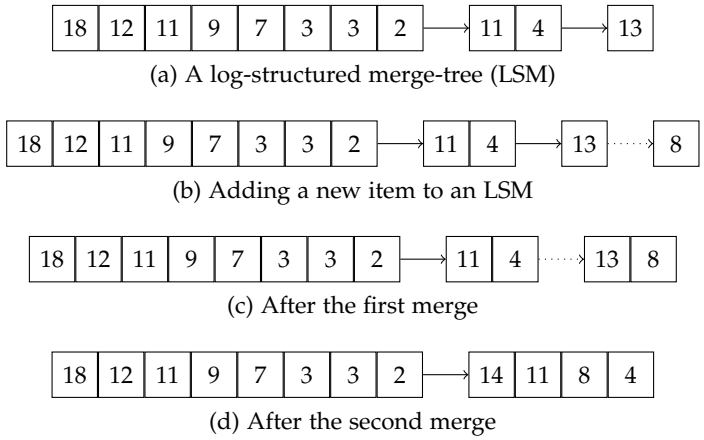
If the item was observed before the push this means that at the time of observation there were no active items before the observed item, and that b was greater than the position at which the item was stored. For a push of an item stored before the observed item to occur after this was observed, b would need to be decremented to a position before the item. This cannot happen, since b is only decremented to the position behind an active item, if an active item is found, or by marking an active item at this position as taken. It can never go to a position with smaller index unless there is another active item there. Since there are no active items at smaller indices, this is not possible.

If the observed item was added after the skipped item, this means that the spying thread observed a value for b greater than the position of the item at the beginning of the iteration. For an item to be added at a position before the observed item, b must have first been decremented, leading to an increment of the epoch if b is incremented again later, leading to a guaranteed failure as shown in Lemma 5.6.7. □

Lemma 5.6.10. *For quiescent states of the owner (when the owner does not make any calls to the queue), the invariant $b \geq t$ holds under the assumption that take operations and active checks by leaf containers also exhibit sequentially consistent behaviour.*

Proof. The invariant can only be broken when either the owner decrements b, or any thread increments t. The owner will only decrement b if it encounters an active item. On a peek operation, b is decremented to the position above the active item if there are no active items above the encountered item. Since the item is still active, it is guaranteed that at the linearization point of the call to active $t < b$ for the new value of b. Also, since at this point there are no active items above the item, it is guaranteed that t cannot be incremented above b.

On take, b will be decremented to the position below the active item. This might conflict with an increment to t. Since this will only occur if before $t = b - 1$, this is guaranteed to lead to a later increment of b, thus protecting the invariant $b \geq t$ for quiescent states. □



(a) A log-structured merge-tree (LSM)

(b) Adding a new item to an LSM

(c) After the first merge

(d) After the second merge

Figure 5.7: A log-structured merge-tree (LSM)

If `take` operations by leaf containers do not provide sequential consistency, additional protective measures need to be made to ensure the invariant $b \geq t$ in quiescent states. The reliance on sequential consistency on work-stealing deques is a general issue of the classical work-stealing algorithm, and may lead to scalability issues on architectures with a strongly relaxed memory consistency model.

5.7 Log-structured Merge-tree (LSM)

In this section, we present a priority queue based on *log-structured merge-trees*. We present this priority queue here in spite of it being purely sequential, since it is the base for the concurrent log-structured merge-tree presented in the next section.

Log-structured merge-trees [110] originally come from the database community, where they are used as a disk-based index structure. They consist of a logarithmic number of sorted arrays. They provide high efficiency for applications with a large amounts of item retrievals and removals, and rare lookups. A fact that makes this data structure very useful for priority queue implementations, is that finding the minimum or maximum item can be done much faster than other lookups. The fact that the data structure is optimized to reduce the number of disk accesses and has a low non-contiguous disk accesses shows that it has the potential for implementing a cache-efficient data structure for shared memory as well.

The LSM presented in this section was thought up independently, based on requirements for concurrent relaxed priority queues, and inspired by merge-sort. The connection to the data structure used in the database community was only made later. Therefore it may differ from related work in many aspects.

A log-structured merge-tree (LSM) is a type of priority queue that operates on a logarithmic number of sorted arrays as depicted in Figure 5.7a. New elements that are added to the LSM can either be added to the end of one of the existing arrays, if they fit the sorting order, or added as a new array. The element with the highest priority in the LSM can be found by looking at the highest-priority element in each array. In order to guarantee logarithmic lookup time for retrieving the maxima and minima, the number of sorted arrays has to be kept logarithmic. This is achieved by merging arrays of similar size like in the merge-sort algorithm.

5.7.1 The algorithm

Our LSM variant operates on sorted arrays of fixed size, where sizes are powers of two. We will assume that the arrays are sorted in ascending order by priority. A linked list is used to store all arrays in the LSM. No two arrays of same size are allowed in this linked list, and each array in the list needs to be smaller than its predecessor.

The `push` operation creates a new array of size 1, which is filled with the item passed on to the `push`. This item can then be added to the linked list as long as there is no other array of size 1 in the list. If the list already contains a same-size array, a new array with double the size is created, and both arrays merged into the new array and then deleted. The new array will then be added to the list, unless the list already contains a same-size array, which will trigger another *merge*.

Figure 5.7a shows an example LSM, with arrays of sizes 8, 2 and 1. When a new item is added, it is put into a new array of size 1, which cannot be added to the list, since an array of size 1 already exists in the list (b). To resolve this, both arrays of size 1 are merged into an array of size 2, which again cannot be put into the list, due to another array of size 2 (c). Both arrays of size 2 are then merged into an array of size 4. Since there is no array of size 4 in the list, it can be added to the list (d). Pseudocode for the `push` operation is presented in Listing 5.19.

Listing 5.19 Pseudocode for the `push` method of an LSM.

```
1  void push(Item item) {
2    // Create new array of size 1 and store item in it
3    ItemArray* new_array = new ItemArray(item);
4
5    // Check whether tail of list has same size
6    while(!list.empty() && list.back()->n == new_array->n) {
7      // Create new array of double size and merge the existing arrays into it
8      ItemArray* merged = new ItemArray(new_array->n * 2);
9      merge(list.back(), new_array, merged);
10
11     // Delete old arrays
12     delete list.back();
13     delete new_array;
14     list.pop_back();
15
16     // Continue on with the merged array
17     new_array = merged;
18   }
19
20   // Add new array to end of list
21   list.push_back(new_array);
22 }
```

To find the item with the highest priority in an LSM, as required by the `peek` operation (shown in Listing 5.20), it needs to perform a linear search through the linked list. It compares the highest priority items from each (sorted) array against each other and returns the item with the highest priority of all items. The implementation shown in Listing 5.20 assumes `peek` is only called when the LSM is not empty, semantics common for serial containers in C++.

The `pop` operation, which is shown in Listing 5.21, first searches for the array containing the highest priority item, similar to `peek`. It then removes the highest priority item from the array. Even if an item is removed from an array, this does not change the size of the array, only

Listing 5.20 Pseudocode for the peek method of an LSM.

```
Item peek() {
  ItemArray* best = null;

  // Iterate through list (C++11 style)
  for(auto i = list.begin(); i != list.end(); ++i) {
    // Compare priorities of best and current array and replace best array if needed
    if(best == null || best->peek() < *i->peek()) best = *i;
  }

  return best->peek();
}
```

how full the array is. As soon as the array is half full (or half empty, for pessimists), the array needs to be shrunk to half the size. This may trigger a single merge with the array following the given array.

Listing 5.21 Pseudocode for the pop method of an LSM.

```
Item pop() {
  list::iterator best = list.end();

  // Iterate through list (C++11 style)
  for(auto i = list.begin(); i != list.end(); ++i) {
    // Compare priorities of best and current array and replace best array if needed
    if(best == list.end() || *best->peek() < *i->peek()) {
      best = i;
    }
  }
  // Pop best item from its array
  Item ret = *best->pop();

  // Check whether array needs to be resized
  if(*best->filled <= *best->n / 2) {
    // Array is only half filled, shrink to half the size
    *best->shrink(*best->n / 2);

    // Check whether a merge is required
    if((best + 1) != list.end() && *best->n == *(best + 1)->n) {
      ItemArray* merged = new ItemArray(*best->n * 2);
      merge(*best, *(best + 1), merged);

      // Delete previous arrays from list and add new array instead
      delete *best;
      delete *(best + 1);
      list.erase(best, best + 2);
      list.insert(best, merged);
    }
  }
}
```

5.7.2 Extensions

This section presents some extensions that are not required to make the LSM work, but help improve on its properties. One such extension allows the reuse of sorted arrays, which reduces the need for memory allocation to cases where the LSM runs out of space. For this, the LSM pre-allocates two arrays of each size, up until the size of the biggest array currently in use. Each array has a flag associated with it to tell whether it is currently used in the linked list. Since only one array of each size is allowed to be in use in the linked list, there will always be one array of each size available that can be used as a target for a merge. Whenever a merge requires an array of a size bigger than all preallocated arrays, two new arrays of the given size will be allocated. When the largest block is shrunk, all arrays of same size can be deleted.

In an LSM, merges can be omitted whenever presorted data is added. To omit such merges, whenever an item is added to an LSM, the array at the tail is checked to see whether putting the new item at its end would violate the sorting order of the array. If the new item fits in, it is added to the end of the array. If the array is full, it can be grown to double the size to fit the new item, unless an array of the new size already exists in the linked list. If the item cannot be added to the linked list, a new array of size 1 is created as in the normal case. Care has to be taken for cases where a `push` that grows an array is followed by a `pop` that removes the same item, since such a `pop` would normally shrink an array. This can be resolved by only virtually shrinking the array the first time, and only physically shrinking an array if it is only filled by 25% or less. This can even be done with preallocated sorted arrays, since it is guaranteed that there will be no other array with the same physical size in the linked list after the shrink until the given array is consumed by a `merge` operation.

Sometimes it is desirable to remove items from the LSM even if they are not the highest priority items. This can be easily done for items that are the last items in one of the sorted arrays. For all other items, a lazy removal scheme can be used, where an item is marked as removed, and will be physically removed whenever an array is shrunk, grown or merged with another array. Each sorted array can keep track of the number of active items in it. Whenever the number of active items falls below the threshold for shrinking, the array can be shrunk, and all items that have been marked as removed can be removed.

5.7.3 Time and space bounds

Lemma 5.7.1. *The upper bound on the number of sorted arrays in the linked list is* $1 + \log_2 n$*, where* n *is the number of items stored in the LSM.*

Proof. Array sizes are powers of two, and only one array of each size is allowed to exist in the linked list. If all arrays are fully filled with items, it is clear that no more than $1 + \log_2 n$ arrays will be used to store these items. Now we assume that if the arrays are not fully filled, an additional array is needed. Since each array has to be more than half filled with items, the number of items available for an additional array is less than $\frac{n}{2}$. Since $1 + \log_2 n$ arrays are already in use, and each array can only be used once, an additional array would have a size greater than n. Since there are less than $\frac{n}{2}$ items to store, and the array has a size greater than n, this contradicts the property that an array has to be more than half filled.

With the extension for presorted data, we allow an array to be virtually shrunk once without being physically shrunk. This conforms with the bound, since still only one array of each virtual size is allowed in the linked list. □

Lemma 5.7.2. *An LSM uses* $O(n)$ *space to store* n *items.*

Proof. Arrays that are used to store items have to be more than half filled, therefore a constant amount of space is used per item. With the extension for presorted data, arrays have to be

filled by more than 25%, which is only a constant factor increase in space usage. Merging will allocate an additional array smaller than $2n$, before deleting the merged arrays, which is also only a constant factor increase in space usage.

With the use of preallocated arrays, two arrays of each size are allocated. Since no arrays larger than the largest array in use are kept and an array has to be half filled to be in use, the biggest available array has to be smaller than $2n$ (or $4n$ with the extension for presorted data). Since array sizes are all powers of two, and only two of each size are kept, the sum of all array sizes will stay below $8n$, resulting in a constant space usage per item. □

Lemma 5.7.3. *The* `push` *operation has amortized complexity* $O(\log n)$.

Proof. Creating an array of size 1 and storing an item in it can be done in constant time. If no items are removed in the meantime, one merge operation on two items has to be performed every second time, another merge operation on four items needs to be performed every fourth time, and so on. After adding n items, at most $n - 1$ merge operations have been performed that processed at most $n \log_2 n$ items, resulting in an amortized complexity per `push` of $O(\log n)$.

If some items are removed in the meantime, but no array needs to be shrunk because of that, the merges will still occur at the same time. Not more than $\frac{n}{2} - \log n$ items can be removed without an array being shrunk, which is less than a factor of 2 difference and results in the same amortized complexity for `push`. If a `pop` operation shrinks an array in the meantime, and this triggers a merge, one merge operation is taken away from `push` resulting in less work.

If, on the other hand, an array is shrunk and no merge is required this will lead to an additional merge on a future `push` operation that would not have happened otherwise. We amortize these merges over the `push` operations that added the items that were removed to create the additional merge. To create an additional merge operation for n items, n items must have been pushed originally to create an array of size n. For the array of size n to be shrunk to size $\frac{n}{2}$ half the items must have been removed from the array before the shrink. Even if the array was never fully filled, this only means that the missing items were already removed in one of the arrays that were merged to create this array, without resulting in a shrunk array. Same argument counts if one of these arrays was never fully filled. Therefore, for a premature merge of n items to occur, $\frac{n}{2}$ items must have been removed from the array before the merge. Since each item is only counted for a single shrink operation, and each such item must have been added by a previous `push` operation, there is a constant amortized overhead from premature merges. □

Lemma 5.7.4. *With the extension that supports presorted data, the* `push` *operation has amortized complexity* $O(1)$ *if only presorted data is added.*

Proof. With presorted data, only a single array is used that is doubled in size whenever it is filled up with items. Adding an item at the end of such an array can be done in constant time. Doubling the array in size can be amortized over all `push` operations since the last growth. Each such `push` operation pays for two items being copied to the array of double size.

When items are removed, an array might be shrunk again, but it is only physically shrunk if it is filled by at most 25%. To reach a fill state of 25%, at least 25% of the items previously stored in the array need to be removed first. These items must have been added by a previous `push` operation, and each such item pays for one item being copied when the shrunk array needs to be grown again. □

Theorem 5.7.5. *The `peek` operation has complexity $O(\log n)$.*

Proof. Finding the highest priority item in a single sorted array can be done in constant time. Since the number of arrays in the linked list is bounded by Lemma 5.7.1 to $1 + \log_2 n$, the `peek` operation will use $O(\log n)$ time. □

Theorem 5.7.6. *The `pop` operation has amortized complexity $O(\log n)$.*

Proof. Removing the highest priority item from a sorted array can be done in constant time. A `pop` may trigger the shrinking of the array, which requires all remaining items in the array to be copied. Since, at least half the items have to be removed from an array before it is shrunk, each of the items removed by a previous `pop` pays for an item being copied.

If an array is shrunk due to a `pop` operation, this may also trigger a single merge operation with another array of same size as the newly shrunk array. Again, since the shrinking operation required half the items to be removed first, each previously removed item pays for one item in each of the arrays that are being merged.

Both shrinking, and merging take time linear in the number of items, and each item taking part in these operation is paid for by a previous `pop` operation, the amortized complexity for removing an item from an array is constant.

A `pop` operation is required to first find an item to remove, which has complexity $O(\log n)$, as described in Lemma 5.7.5. Therefore the total amortized complexity of `pop` is $O(\log n)$. □

5.8 Concurrent LSM Priority Queue

In this Section we present a concurrent priority queue based on on the serial LSM presented in Section 5.7. It provides local semantics with shared items as defined in Section 5.1. The implementation presented in this section is intended to be used as a container in the two-level concurrent ordered container from Section 5.5. The changes necessary to convert the presented implementation into a standalone priority queue are described where it is required.

5.8.1 Internal structure

Our concurrent LSM uses a decentralized scheme, where each thread maintains its own local LSM. We call all data owned by a single thread, including the local LSM a *place*. The concurrent LSM also has a centralized component, which is mainly used to relay method calls to a specific place, similar to the centralized component of the root container described in Section 5.6.2. It will not be described separately for the concurrent LSM. All the methods described below are for the place-specific components.

5.8.2 Benefits of using LSM for concurrency

Log-structured merge-trees have many properties that make them desirable to be used in a decentralized concurrent priority queue. One such property is the simplicity of creating a consistent snapshot. If merge operations are performed in the right order, then the LSM will have a consistent state at any time. Since a merge is not performed in-place, the state of an array is unchanged until the merged array is put into the linked list instead of the arrays it consists of. Storing the items in sorted arrays also leads to good cache efficiency.

Another benefit of LSMs for concurrency is the lazy sorting of data. This means that all data is regularly touched, newer data more often than older data, allowing the LSM to recognize and remove data processed by other threads. This can also be used to allow for items being removed from the outside, as required for the *elimination of dead tasks* feature

presented in Section 2.7.4. Also, if data is added in a presorted or partially presorted manner, most items will be removed without ever being sorted before. But even for random data and random `push` and `pop` operations, newly added items are more likely to be removed first, since older data will have a tendency to have lower priorities or would have been popped earlier otherwise. This reduces the number of items that are merged in practice.

In a concurrent setting presorted data is very common, since whenever a thread runs out of local items, it will copy items from another place. Since items that are copied are partially presorted this will reduce the amount of merges other threads will perform on data coming from another thread.

5.8.3 Implementation of stored items

An item that is stored in the concurrent LSM is stored in a separate object that stores auxiliary data along with the item. The implementations for these items conforms to the interface for items in the two-level container described in Section 5.5.3. Pseudocode for the implementation of items is shown in Listing 5.22.

Listing 5.22 Pseudocode for the implementation of items owned by the concurrent LSM.

```
1  class LSMStoredItem : public RCStoredItem {
2    // Flag to mark item as taken, and to tell when it was taken
3    int taken;
4    // Key used for sorting items
5    Key key;
6    // Owner of item
7    LSMPlace* place;
8
9  public:
10   // An item is active if taken == expected_version
11   bool active(int expected_version) {
12     return taken == expected_version;
13   }
14
15   // To take an item change taken to not equal expected_version
16   bool take(int expected_version) {
17     return taken.cas(expected_version, expected_version + 1);
18   }
19 }
```

To mark an item as taken, a single integer field named `taken` is updated using a *compare-and-swap* operation. To omit an ABA problem, an item is seen as active, whenever `taken` equals the version number, and inactive otherwise.

5.8.4 The sorted arrays

One main difference between the concurrent and the serial LSM is how sorted arrays are handled. To reduce the amount of synchronization needed in systems without garbage collection, we forbid sorted arrays to ever be deleted. In this way, reference counting for sorted arrays can be completely omitted. Instead, sorted arrays are reused as soon as they have been removed from the linked list. This also means that the concurrent LSM can never shrink, when the number of items is reduced, but will never grow beyond the bounds for the highest number of items that were stored in the LSM at a single point in time.

Since sorted arrays are never removed, we can allow for arrays to always only be shrunk logically, never physically. This can be done since the number of arrays of logical size bigger than the shrunk array will not grow before the shrunk array has been merged with another array. We place the restriction for all arrays in the linked list that an array of bigger logical size than another array may never have a smaller physical size. It is allowed for both arrays to have the same physical size, though. To ensure this, a merge operation may use an array of a bigger physical size than necessary to store a merged array. It is guaranteed that no more than two arrays of each physical size will occur in the linked list of such an LSM. For this reason, no more than three arrays of each size need to be allocated. This is shown in Lemma 5.8.11. With a few tricks this can be reduced to two as for the serial LSM, but since this can lead to one additional array size being allocated in some cases, without being needed, the gains of this are questionable and will not be discussed here.

Each position in a sorted array will store a reference to a stored item, alongside with the version number of the item to ensure ABA safety.

5.8.5 The `allocate_item` method

The `allocate_item` method is shown in Listing 5.23. Items are allocated using the wait-free memory manager from Section 4.3, and afterwards the item is initialized. When implemented as a standalone priority queue, the `allocate_item` method is not part of the external interface, but only called internally by `push`.

Listing 5.23 Pseudocode for the `allocate_item` method of the root container

```
1  LSMStoredItem* allocate_item(Data data) {
2    // Create an object storing the item along with auxiliary data
3    LSMStoredItem* ref = LSMStoredItemPool.get_item(item);
4
5    // Increment version number to omit ABA problem
6    ++ref->version;
7
8    // Initialize
9    ref->data = data;
10   ref->leaf_container = this;
11   ref->taken = ref->version;
12   ref->place = this->places[get_place_id()];
13
14   // Retrieve key
15   ref->key = data.key;
16
17   return ref;
18 }
```

5.8.6 The `push` method

The `push` operation for the concurrent LSM, which is presented in Listing 5.24 already takes into account presorted data. Whenever the array at the tail of the linked list has space for an additional item, and the new item has higher priority than all previously stored items, it can be added to the array at the `tail`. In addition, since adding the item may increase the logical size of the array at `tail`, the item is only added if there is no predecessor array, or the predecessor array has a logical size more than double the number of items in `tail`.

In case that the item cannot be added to the array at `tail`, a new array is created, and filled with the item. In the concurrent LSM arrays are never physically grown to minimize the number of modifications to the linked list. Therefore, to allow for constant insertion times for pre-sorted data, the biggest available array that satisfies all invariants is used. If the linked list is empty, this is the biggest array that has been allocated by the LSM. Otherwise it is an array of a size smaller than the current `tail`. Only if `tail` has size 1, an array of size 1 is used, which will trigger a merge when linking in the array. After the item has been put into the selected array, it is atomically linked to the linked list using the `link_in` method described in the next section.

Listing 5.24 Pseudocode for the place-specific `push` method of a concurrent LSM.

```
1  void push(StoredItem* item, int item_version) {
2    Key key = item->key;
3
4    // Check whether item can be added to existing array
5    if(tail != null && tail->peek()->key <= key
6        && tail->filled < tail->physical_n
7        // And that there are no two arrays of same logical size after adding the item
8        && (tail->prev == null || tail->prev->logical_n > tail->filled * 2)) {
9      // Put item into existing array and return
10     tail->items[tail->filled]->ref = item;
11     tail->items[tail->filled]->version = item_version;
12
13     ++tail->filled;
14     // Update logical size if necessary
15     if(tail->logical_n < tail->filled) tail->logical_n *= 2;
16     return;
17   }
18
19   // We need to create a new array. Make it as big as possible for presorted data
20   ItemArray* new_array;
21   if(tail == null) {
22     // If no tail is allocated take the biggest available array size
23     new_array = get_biggest_allocated_array();
24   } else if(tail->logical_n > 1) {
25     // Get array of size smaller than the logical size of tail
26     new_array = get_array_of_size(tail->logical_n / 2);
27   } else {
28     // Get smallest possible array size (will lead to a merge)
29     new_array = get_array_of_size(1);
30   }
31
32   // Put item into new array
33   new_array->items[0]->ref = item;
34   new_array->items[0]->version = item_version;
35   new_array->filled = 1;
36
37   // Link array into list
38   link_in(new_array);
39
40   return;
41 }
```

5.8.7 The `link_in` method

The `link_in` method is responsible for atomically linking a new array into the linked list of sorted arrays, while keeping all invariants intact. It is called by the `push` operation presented in the previous section. To make this possible all necessary merges with other arrays are performed before the new data becomes visible. Since the linked list is accessed concurrently by all threads, it needs to be maintained directly by the container, contrary to the serial LSM, which can use the linked list through an abstract interface. In the concurrent LSM an item is in the linked list iff it is reachable from the `head` of the list. The `tail` and predecessor pointers in the linked list are only used as shortcuts by the owner and will never by accessed by other threads.

Listing 5.25 Pseudocode for the `link_in` method of a concurrent LSM.

```
1  void link_in(ItemArray* new_array) {
2    // Merge new array in if necessary
3    ItemArray* other = tail;
4    ItemArray* to_free = null;
5    while(other != null && new_array->logical_n == other->logical_n) {
6      // Create new array of double size and merge it in
7      ItemArray* merged = new ItemArray(other->logical_n * 2);
8      merge(other, new_array, merged);
9
10     // Free previous temporary array and continue with merged array
11     clear_array(new_array);
12     new_array = merged;
13     to_free = other;
14     other = other->prev;
15   }
16
17   // Since new item has not been published, merged array cannot be empty
18   // Connect new array to linked list
19   new_array->prev = other;
20   if(other != null) other->next = new_array;
21   else head = new_array;
22   tail = new_array;
23
24   // Free arrays that have been merged
25   while(to_free != null) {
26     ItemArray* tmp = to_free;
27     to_free = to_free->next;
28     clear_array(to_free);
29   }
30 }
```

Pseudocode for the `link_in` method is shown in Listing 5.25. It is guaranteed that the array being linked into the list has a physical size smaller than the physical size of the current tail, and that it has a logical size smaller or equal the logical size of the tail. If the tail has the same logical size as the new array, this means that a `merge` operation is required. For this, the tail and the new array are merged into an array of physical size double the logical size of the source arrays. The merged array then takes the place of the new array and the previous new array is freed. The tail is not modified by the merge and is kept in the linked list.

If the predecessor of the tail has the same logical size as the merged array, another merge operation is performed, which creates a new merged array and frees the previous merged

array. The merging process then continues through the predecessors of the last array from the linked list that has been merged, until an array is reached with logical size bigger than the logical size of the merged array, or all arrays in the linked list have been merged in.

After the merging is completed, the resulting new array is put into the linked list atomically, thereby atomically replacing all arrays that were merged into it. As soon as either the `head` pointer directly points to the array, or the next pointer of another array has been set to point to the array, the new array becomes visible.

A `merge` operation is allowed to clean out all inactive items during the merge, which means that the resulting array can contain less items than the arrays that were merged into it. Nonetheless it is guaranteed that the merged array will not be empty, since it contains a single new item that will only become visible to other threads when the result of the merge becomes visible. Care has to be taken, if items can become inactive due to an external reason, as with the *elimination of dead tasks* feature of our scheduling system, which is described in Section 2.7.4. In this case, `link_in` needs to check whether the new array is empty after all merges have been performed. If this is the case, all items that were merged will be removed from the linked list, and no new array will be added.

5.8.8 The `merge` method

The `merge` method of the LSM performs a classical two-way merge of two arrays and stores the result in a third array. The two arrays being merged are not modified by the `merge`. Due to the simplicity of this scheme, no code is shown for the merge. The only difference to a normal merge operation is that inactive items are ignored during the merge, and the resulting array can be smaller than the sum of its parts. A `merge` is allowed to ignore duplicates when recognized. It is not required to recognize all duplicates. When copying items, `merge` needs to increment the reference count for each copied item.

5.8.9 The `clear_array` method

The `clear_array` method is called whenever an array is not in use any more, and can be prepared for reuse. This is done by incrementing the version counter of the array before setting the variable `filled` to zero.

5.8.10 The `peek` method

The core of the `peek` method is similar to the `peek` method of its serial counterpart. The highest priority items in each array are compared against each other, and the item with the highest priority returned. This can be seen in Listing 5.26. The main difference to the serial version is that a boundary item is passed on to the `peek` method. Due to concurrency, `peek` may encounter inactive items. Therefore, the `peek` method of the sorted arrays will automatically clean out any inactive items before returning the highest priority item. If the number of items in an array falls below half the logical size of the array, the logical size of the array is reduced, and it is merged with its successor if necessary using the `merge_shrunk` method.

The `peek` method is required to only return items with a priority at least as high as the priority of the boundary item or the boundary item itself. If the boundary item is stored in the local LSM, this is guaranteed. If it is not, it can happen that the `peek` operation will only find items with lower priorities, or no item at all if the local LSM is empty. In this case, the `spy` method is called, which will copy items from the owner of the boundary item to the local LSM. The boundary item will also be added separately, in case `spy` fails to copy the boundary item. Afterwards, if the boundary item is still active, the `peek` operation is repeated, until

Listing 5.26 Pseudocode for the peek method of a concurrent LSM.

```
struct LSMPeekState {
  StoredItem* item;
  Key key;
  int version;
}

LSMPeekState peek(StoredItem* boundary, int boundary_version) {
  while(true) {
    // Fail if boundary item is not active spying
    if(!boundary->active(boundary_version)) {
      return null;
    }

    LSMPeekState best = null;

    // Iterate through list
    for(ItemArray* i = head; i != null; i = i->next) {
      // Get highest priority item out of the sorted array
      // Automatically cleans out inactive items
      LSMPeekState item = i->peek();

      // Check whether resizes and merges are needed
      while(i->filled <= i->logical_n / 2) {
        // Logical size needs to be reduced
        i->logical_n /= 2;

        // Merge with successor if necessary
        i = merge_shrunk(i);

        // Merged array is empty
        if(i == null)
          break;

        // Rerun peek
        item = i->peek();
      }
      if(best == null ||
        (peek != null && best->key < item->key) best = item;
    }

    // Check whether found item is the boundary item
    // Or an item with greater or equal priority
    if(best != null && best->key >= boundary->key) {
      return best;
    }

    // Boundary was not spied yet, spy items from owner of boundary
    spy(boundary->place);

    // Add boundary to LSM
    push(boundary, boundary_version);
  }
}
```

either an item with priority greater or equal the boundary item is found, or the boundary item becomes inactive. An inactive boundary item leads to a spurious failure of `peek`.

When implemented as a standalone priority queue, the `peek` method of the concurrent LSM will be implemented differently, since there is no boundary item. In this case, the `peek` method will call `spy` on a victim selected according to the victim selection policy of the LSM, whenever the local LSM is empty.

5.8.11 The `spy` method

Listing 5.27 Pseudocode for the `spy` method of a concurrent LSM.

```
1  void spy(LSMPlace* place) {
2    // Read head from place and retrieve version number
3    ItemArray* i = place->head;
4
5    // Loop through ItemArrays
6    while(i != nullptr) {
7      int f = i->filled;
8      // Copy items
9      for(int j = 0; j < f; ++j) {
10       StoredItem* si = i->items[j]->ref;
11       int v = i->items[j]->version;
12
13       if(si != null && si->active(v)) {
14         // Push reference to spied item into the local LSM
15         push(si);
16       }
17     }
18     // Find next array to look at
19     i = i->next;
20   }
21 }
```

The `spy` method, which is shown in Listing 5.27, is called by a place whenever it has a boundary item that is better than all items stored in the local LSM. The goal of `spy` is to copy as many items stored in the LSM of a victim as possible in a wait-free manner. There is no guarantee that `spy` will copy a reference to a certain item. We allow for `spy` to discontinue spying, when an inconsistent state is encountered for arrays.

The `spy` method proceeds by iterating through the linked list of sorted arrays and copying all active items found in these arrays. Since the linked list might change in the meantime, and blocks can be unlinked from the list or even reused, there is no guarantee that all items will be encountered.

5.8.12 The `merge_shrunk` method

The `merge_shrunk` method, which is shown in Listing 5.28, is called whenever an array is logically shrunk. It performs merges on arrays if necessary. The `merge_shrunk` method starts with the array that was shrunk, and checks whether the logical size of its successor is greater or equal its own logical size. If this is the case, both arrays are merged. An array of the same physical size as the physically smaller array is chosen as the merging target, unless it does not fit all data, then an array of double the physical size of the physically smaller array is used. Since a merge operation automatically cleans out all inactive items, the logical size of the

Listing 5.28 Pseudocode for the `merge_shrunk` method in the concurrent LSM.

```
1  ItemArray* merge_shrunk(ItemArray* i) {
2    ItemArray* begin = i;
3    ItemArray* next = i->next;
4    int size = i->physical_n;
5
6    // Loop as long as logical size of successor is greater or equal i
7    while(next != null && next->logical_n >= i->logical_n) {
8      ItemArray* merged = get_array_of_size(size);
9      merge(i, next, merged);
10
11     // Clear i if it is the result of a previous merge
12     if(i != begin) clear_array(i);
13     // Prepare for next iteration
14     i = merged;
15     next = next->next;
16     // Next merged array will have half the size
17     size /= 2;
18   }
19
20   // Check if merge occured
21   if(i != begin) {
22     // If merged array is empty, cut the list and clear merged array
23     if(i->filled == 0) {
24       if(begin->prev == null) {
25         tail = null;
26         head = null;
27       } else {
28         tail = begin->prev;
29         tail->next = null;
30       }
31       clear_array(i);
32     } else {
33       // Link in i
34       if(next == null) tail = i;
35       else next->prev = i;
36       i->next = next;
37       i->prev = begin->prev;
38
39       // Becomes visible here
40       if(begin->prev == null) head = i;
41       else begin->prev->next = i;
42     }
43     // Clear old arrays
44     while(begin != next) {
45       ItemArray* tmp = begin->next;
46       clear_array(begin);
47       begin = tmp;
48     }
49   }
50   return i;
51 }
```

resulting array may be smaller than originally expected. This can trigger a subsequent merge with the successor of the previously merged arrays in the linked list. As with the `link_in` method, all merges become visible at a single point in time, which is when the merged array becomes reachable from the head of the linked list.

Care has to be taken not to put an empty array into the list. A merge operation may result in an empty array if all items in the merged arrays are inactive. In case this happens, all arrays that took part in the merge are removed from the list and no new array is added to the list.

The physical size for the first merged array is chosen to be equal to the physical size of the shrunk array. Every subsequent array will have half the size of the previous array. This guarantees that only one array of each size is used for merges in `merge_shrunk`, while at the same time making sure that no physical size is used that is smaller than the physical sizes of the merged arrays, as shown in Lemma 5.8.15.

5.8.13 Supporting the *elimination of dead tasks* feature

To support *elimination of dead tasks*, a feature required provided by some of our task scheduling systems, help is required from the data structure used for storing tasks. In a concurrent LSM this feature can be easily supported, by adapting the `active` check for items stored in the LSM. In addition to checking, whether an item has been processed by another thread, the `active` check can also check for dead items (tasks), allowing the `item_taken` method and merge operations to automatically clean out these items.

5.8.14 Optimizations

There are corner cases where a `spy` operation mostly encounters inactive items. This can occur in cases where the thread being spied at rarely accesses its local LSM, thereby making maintenance work that cleans out inactive items rare. In applications where the priority ordering varies between threads this effect can also occur for certain pathological cases. This is not a problem for the complexity of the concurrent LSM as long as the number of inactive items is within a constant factor of the total amount of items. In cases where delays between accesses by threads can be arbitrarily long this is not the case any more. This can be resolved by limiting the number of inactive items that will be spied from each array in relation to the spied active items.

To do this, `spy` counts both the number of active and inactive items encountered while spying at an array. When the ratio of active to inactive items falls below a certain threshold, the spying thread moves on to the next array. For higher robustness, a constant number of inactive items can be encountered before they are counted. To ensure that each item has a chance to be encountered by a `spy` operation, the offset in the array at which spying starts is chosen randomly, and `spy` wraps around the array.

5.8.15 Correctness

In this section we argue that the concurrent LSM is lock-free and linearizable. The methods `active`, `take`, `store_item` and `verify` are trivially wait-free and linearizable and proofs are omitted.

Lemma 5.8.1. *The `push` operation is wait-free if the `link_in` operation is wait-free.*

Proof. Adding an additional item to the tail array is trivially wait-free, since no thread can interfere with it. Each thread has its own pool for arrays of all sizes, therefore getting a new array and filling it is also wait-free. So if `link_in` is wait-free, so is the `push` operation. □

Lemma 5.8.2. *The `push` operation is linearizable.*

Proof. If the item is added to the tail array, the `push` operation is linearized when the counter `filled` is incremented for the tail array, which makes the item visible to all threads. In case a new array is used, the item does not become visible to other threads, until the new array becomes visible in the linked list. Therefore in this case the linearization point of `link_in` is also the linearization point of `push` □

Lemma 5.8.3. *The `link_in` operation is wait-free.*

Proof. The `link_in` operation merges the new array with arrays in the list, until it has either been merged with all arrays in the linked list, or the merged array has a logical size smaller than all remaining arrays in the linked list. The number of arrays in the linked list is fixed at call-time, and no new arrays can be added by other threads. Progress cannot be hindered by other threads, therefore `link_in` is wait-free. □

Lemma 5.8.4. *The `link_in` operation is linearizable.*

Proof. The linked list is not modified up until the merge operations are finished. An array, and all the items contained within is in the LSM when it is reachable from the head of the linked list. The linearization point of `link_in` is when the new array is made visible by an update to the `head` or `next` pointers. The update to the `head` or `next` pointers also atomically removes all arrays that were merged into the new array from the LSM. The active items contained within those arrays are still reachable in the LSM through the new array. □

Lemma 5.8.5. *The `peek` method is wait-free if `spy` is wait-free.*

Proof. The `peek` method iterates through the local linked list of arrays, which is wait-free since the linked list cannot be modified by other threads. It might be required to remove inactive items from sorted arrays, which is bounded by the number of items in an array, and to merge arrays, which is bounded by the number of arrays. If the boundary item was not in the linked list, a `spy` operation will try to copy all items stored at the place owning the boundary item. The boundary item will be added to guarantee that on the next iteration either the boundary item or a higher priority item is returned, or that the boundary item is inactive, allowing `peek` to fail, thus making `peek` wait-free if `spy` is wait-free. □

Lemma 5.8.6. *The `peek` method is linearizable.*

Proof. A successful `peek` method is allowed to return items after they have become inactive, as long as they are the highest priority items in the local LSM. A successful `peek` can therefore be linearized at any point throughout the call of `peek` after the last change to the local LSM. Since the local LSM cannot be modified by other threads, the local LSM can only be changed during the call to `peek` when a `spy` operation is performed. Therefore, a successful call to `peek` can be linearized at any point throughout its execution after the last call to `spy`.

Since `peek` is allowed to spuriously fail, a failed call to `peek` can be linearized at any point in time throughout the call to `peek`. □

Lemma 5.8.7. *The `merge_shrunk` method is wait-free.*

Proof. Each iteration of the merging process in `merge_shrunk` consumes two arrays to create one new array. The progress of the merging operation cannot be hindered by other threads. No new arrays can be added by other threads, therefore the number of merge operations is bounded by the number of arrays in the local linked list. This makes `merge_shrunk` wait-free. □

Lemma 5.8.8. *The `merge_shrunk` method is linearizable.*

Proof. The state of the linked list is not changed throughout the merging process. A call to `merge_shrunk` is linearized when the merged array is made visible in the linked list instead of the arrays that were merged into it. Since an array is in the linked list iff it is reachable from the head, the linearization point is when either the head or the next pointers are updated. If the merged array is empty after the merge, the new array is not put into the linked list. In this case, the linearization point of `merge_shrunk` is when all arrays that were merged into the now empty array are cut off from the linked list by an update to the head or next pointer. □

Lemma 5.8.9. *Logical sizes of arrays in a local LSM are strictly decreasing outside of `peek` method. While `peek` is being executed, this order will only be violated for a single array.*

Proof. New arrays are only added to the tail of the local linked list using the `link_in` method. Sizes of arrays are chosen so that the newly added array is smaller than its predecessor, unless the predecessor has size 1. On size 1 the new array is merged with the current `tail`. Since both arrays have the same logical size, the merged array will have at most double the logical size of the arrays being merged. Since logical sizes are strictly decreasing, and logical sizes are always powers of two, the predecessor of the last merged array in the list will have at least double the size of the arrays from the last merge. This means that either the merged array has exactly the same size, which will trigger another merge with similar results, or the merged array will be smaller.

An array in a local LSM may never grow logically, but it may be logically shrunk when `peek` is called. After it has been shrunk, this single array may violate the ordering for logical sizes. This is fixed by a call to the `merge_shrunk` method directly after the array has been logically shrunk. Since `peek` is only called by the owner of the local LSM, and `merge_shrunk` is called before `peek` terminates or another array is shrunk, only a single array in the local LSM may violate the order of logical sizes.

The `merge_shrunk` method merges successor arrays with the shrunk array until either no more successor exists, or no successor with logical size greater or equal the merged array exists, thereby enforcing strictly decreasing logical sizes. Since logical sizes were strictly decreasing before the shrinking of the array, and since logical sizes are powers of two, the successor of the shrunk array is guaranteed to have a logical size of at most a quarter of the shrunk array's predecessor. For this reason it is guaranteed, that the merge operation will not yield an array of logical size greater or equal its predecessor. □

Lemma 5.8.10. *The merge targets chosen by `merge_shrunk` will always fit the data being merged in.*

Proof. The first array used as merge target by `merge_shrunk` has the same physical size as the shrunk array. The physical size of an array can never be less than its logical size. A merge operation will only occur if the array was logically shrunk, which means that the logical size of the shrunk array is guaranteed to be smaller than the physical size. Also, since logical sizes are strictly decreasing (Lemma 5.8.9), an array can never have a logical size greater or equal the physical size of its predecessor. Both logical and physical sizes are powers of two, therefore both the shrunk array and its successor will have a logical size of at most half the physical size of the shrunk array. Therefore, since the merge target has the same physical size as the shrunk array, it is guaranteed to fit the data.

A subsequent merge may occur in `merge_shrunk` if some items are cleaned out during the merge, making the logical size of the merged array less or equal the logical size of its successor. Since logical sizes are strictly decreasing, the successor will have a logical size less or equal half the size of the previous array that was merged in, which is a quarter of the physical size of the merged array. A merge will only occur if the merged array has a logical

size less or equal the logical size of the successor, which means that the merged data from both arrays will fit an array of half the physical size of the previous merged array. □

5.8.16 Time and space bounds

Lemma 5.8.11. *The upper bound on the number of sorted arrays in the local LSM is* $2 + \log_2 n$ *where* n *is the number of items stored in the local LSM, including inactive items.*

Proof. The proof is similar to the proof of Lemma 5.7.1. Array sizes are powers of two, and only one array of each logical size is allowed to exist in the linked list. There is a single exception to this rule, which is that an array that is being processed by the `peek` method can have the same logical size as another array in the list. As shown in Lemma 5.7.1, if no two arrays of same (logical) size can exist in the list, the number of sorted arrays in the list cannot be higher than $1 + \log_2 n$. Since we allow for a single array to violate this rule in the concurrent LSM (see Lemma 5.8.9), the bound increases to $2 + \log_2 n$. □

Lemma 5.8.12. *If no items are ever cleaned out by* `peek`*, the arrays in a local linked list of an LSM will have strictly non-increasing physical sizes.*

Proof. New items are always added to the tail of a local LSM, and arrays in the local LSM can never grow physically. Whenever the tail runs out of space a new array is added to the tail, which has a physical size of half the logical size of the previous tail. This leads to strictly decreasing physical sizes in the local LSM. If the tail array has the physical size 1, no smaller array can be put into the local LSM. Instead, another array of size 1 is created for the new item, and merged into the existing list by the `link_in` method. All merges performed by `link_in` operate on two arrays of same logical size. To be able to fit all merged data, all data is merged into an array of physical size double the logical size of one of the arrays being merged in.

Logical sizes of arrays are strictly decreasing as shown in Lemma 5.8.9. Since only powers of two are used for array sizes, it is guaranteed that each array has at least double the logical size of its successor. The physical size of an array can never be smaller than its logical size, therefore a merged array cannot have a physical size greater than the physical size of its predecessor in the list. □

Lemma 5.8.13. *The physical size of an array added by* `link_in` *will never exceed the logical size of its predecessor. It will only have the same physical size as its predecessor, if for the predecessor the physical size equals the logical size.*

Proof. As shown in Lemma 5.8.9, logical sizes of arrays in a local LSM are strictly decreasing. The physical size of a merged array is chosen to be double the logical size of one of the arrays being merged (both have the same logical size). For a merged array to have physical size n, each of the arrays being merged needs to have the logical size $\frac{n}{2}$. Since logical sizes are powers of two and strictly decreasing, this requires the predecessor to have a logical size of at least n.

Since the physical size of an array cannot be smaller than its logical size, the predecessor must have a physical size of at least n as well. The only possible case for the merged array to have the same physical size as its predecessor is if the predecessor has both physical and logical size n. □

Lemma 5.8.14. *If no items are ever cleaned out by* `peek`*, the physical size* n_A *of any array* A *in a local LSM is* $n_A \geq n_B 2^{\delta_{AB}}$*, where* n_B *is the size of a successor of* A *in the local LSM, and* δ_{AB} *denotes the number of arrays in-between* A *and* B*.*

Proof. The proof is by induction on δ_{AB}. The base case if B is the direct successor of A, so that $\delta_{AB} = 0$. In this case $n_A \geq n_B$, which means that array sizes are strictly non-increasing. This

has already been shown to be true for Theorem 5.8.12. For the case $\delta_{AB} = 1$, it needs to be shown that $n_A \geq 2n_B$. We know that the logical size of A is less or equal n_A, and the successor of A will have at most half that logical size. From Theorem 5.8.13 it can be deduced that n_B cannot exceed the logical size of its predecessor, which is at most $\frac{n_A}{2}$. This confirms $n_A \geq 2n_B$.

In the general case, the logical size of the predecessor of B will be less or equal $\frac{n_a}{2^{\delta_{AB}}}$, which also restricts the physical size of B. □

Lemma 5.8.15. *An array merged by `merge_shrunk` will always have a physical size greater or equal the smallest and less or equal the largest array that was merged in.*

Proof. The first array used as merge target by `merge_shrunk` has the same physical size as the first array, and any subsequent merge will use a smaller array. Therefore it is trivial to see that the merged array will never be larger than the largest array that was merged in.

Both Lemma 5.8.12 and Lemma 5.8.14 work under the assumption that `peek` does not remove any items. The method `merge_shrunk` is the only point in `peek` where arrays are merged and therefore the only method that could lead to a violation of the given theorems. If the merged array from `merge_shrunk` stays within the bounds provided by the physical sizes of the arrays that it replaces, both theorems must still be true after the merge.

A `merge_shrunk` that performs a single merge, will consume two arrays, to produce one merged array of same physical size as the array that was shrunk, which is the first of the merged arrays. Any additional array that is merged in will halve the physical size of the merged array. We call the shrunk array A, and the last array that was merged in B. The physical size of the merged array M can be calculated as $n_M = \frac{n_A}{2^{\delta_{AB}}}$, where δ_{AB} is the number of arrays between A and B in the local LSM, which are all merged into M. Since physical sizes of arrays are strictly non-increasing, it can be safely assumed that B has the smallest physical size of all arrays that were merged in. With the bound from Lemma 5.8.14, n_M can be expressed with n_B as $n_M \geq \frac{n_B 2^{\delta_{AB}}}{2^{\delta_{AB}}} = n_B$. □

Lemma 5.8.16. *The physical size n_A of any array A in a local LSM is $n_A \geq n_B 2^{\delta_{AB}}$, where n_B is the size of a successor of A in the local LSM, and δ_{AB} denotes the number of arrays in-between A and B.*

Proof. Lemma 5.8.14 shows that this is true if no items are removed by `peek`. The only place in `peek`, where physical sizes in a local LSM may change is the merge operation in the `merge_shrunk` method. This merge operation is guaranteed to yield a merged array with physical size no bigger than the biggest and no smaller than the smallest array that was merged in as shown for Lemma 5.8.15.

To show that $n_A \geq n_B 2^{\delta_{AB}}$ still holds after a call to `merge_shrunk`, three cases need to be looked at. The first case is the case, where A is a predecessor of the merged arrays and B is a successor. We say that δ_{AB} is the number of arrays between A and B before and δ'_{AB} after the merge. Since a merge consumes two or more arrays to create a single new array we know that $\delta'_{AB} < \delta_{AB}$. Based on this we can show that $n_A \geq n_B 2^{\delta_{AB}} > n_B 2^{\delta'_{AB}}$.

The second case that needs to be looked at is when B is the first of the arrays being merged by `merge_shrunk`, and A is one of its predecessors. We say that M is the merged array. Since physical sizes are strictly non-increasing, B is the biggest of the merged arrays. From Lemma 5.8.15 we therefore know that $n_M \leq n_B$. Since $\delta_{AB} = \delta_{AM}$ we therefore have $n_A \geq n_B 2^{\delta_{AB}} \geq n_M 2^{\delta_{AM}}$.

The last case to be looked at is when A is the last of the arrays being merged and B one of its successors. The merged array is called M. Physical sizes of arrays are strictly non-increasing, therefore A is the smallest of the arrays being merged. Using Lemma 5.8.15 we can show that $n_M \geq n_A \geq n_B 2^{\delta_{AB}}$. □

Lemma 5.8.17. *Physical sizes of arrays in a local LSM are strictly non-increasing.*

Proof. This follows from Lemma 5.8.16. For any two arrays A and B, where B is the direct successor of A, we can show that $n_A \geq n_B$. □

Lemma 5.8.18. *No more than two arrays of same physical size follow each other in the local linked list of a thread.*

Proof. This also follows from Lemma 5.8.16. For any two arrays A and B, where there is at least one array in-between A and B, we can show that $n_a \geq 2n_B$. □

Theorem 5.8.19. *No more than three arrays of each physical size need to be allocated per thread in a concurrent LSM.*

Proof. Since each thread maintains its own local LSM and uses its own arrays for it, it suffices to show that no more than three arrays are in use for a single local LSM. Physical sizes in the local LSM of a thread are strictly non-increasing as shown in Theorem 5.8.17. No more than two arrays of same physical size can follow each other as shown in Theorem 5.8.18. Therefore, at most two arrays of each physical size will be used in a local LSM. Merge operations performed in the methods `link_in` and `merge_shrunk` may require up to two additional arrays. One array, which holds the result of a previous merge, and another array that is used as merge target. It remains to show that either the arrays used for the merge have different physical sizes, or the local LSM only contains a single array of the given physical size if both arrays used by the merge have the same size.

In the `link_in` method, merges start at the tail of the list, and are only continued if the merged array has a logical size equal to the logical size of the predecessor of the last array that was merged in. Since logical sizes are strictly decreasing, this requires logical sizes of the merged arrays to be strictly increasing throughout the execution of `link_in` for the algorithm to continue. The physical size of the merged arrays is always chosen to be equal to the expected logical size. The expectation is that the logical size will be double the logical size of its predecessor, and the algorithm only continues merging if the expectation was fulfilled. Therefore, no two additional arrays of same physical sizes are used by `link_in`.

The `merge_shrunk` method halves the physical size of the array used for the merge on each merge, therefore no two arrays of same physical size are used throughout the execution of `merge_shrunk`. □

Lemma 5.8.20. *The `merge_shrunk` method will never allocate arrays of a size bigger than any previously used array.*

Proof. The `merge_shrunk` method uses an array of same physical size as one of the arrays being merged as its first merge target. Since one of the arrays being merged already has the same physical size, this size cannot be previously unused. All subsequent merges will always use smaller arrays than the array of the initial merge. □

Theorem 5.8.21. *A new array of size n is only allocated if at least $\frac{n}{2} + 2$ items are stored in the local LSM of which at least $\frac{n}{4} + 1$ must have been active at a single point in time.*

Proof. A new array of size n is only allocated if required by a merge operation. In `link_in` a merge is performed for two arrays of same logical size, and the merged array will have a physical size that fits both arrays, which is double the logical size of one of the arrays being merged. To require a merged array of size n, both arrays must have a logical size of $\frac{n}{2}$. The physical size of these arrays must be $\frac{n}{2}$ as well, or arrays of size n would have already been allocated. An array must have at least $\frac{n}{4} + 1$ items stored in it to have a logical size of $\frac{n}{2}$.

Therefore there must be at least $\frac{n}{2} + 2$ elements stored in the local LSM before an array of size n is allocated by `link_in`.

While the first array used by `link_in` in a merge operation comes from the local LSM, this is not the case for the second array. The second array is either a result of a previous merge in `link_in`, or an array of size 1 containing a new item. In both cases all items in the second array must have been active at a single point in time. If the array was the result of a merge, all items must have been active when observed by merge, or some of these items would not have been copied to the merged array. Since no new items can be added to an array during a merge, all these items must have therefore been active at the beginning of merge. In the other case, where the second array is an array of size 1 containing a new item, the item in the array is guaranteed to be active, since it has not been added to the local LSM at this point, making it impossible for another thread to take it. In both cases, the second array must have at least $\frac{n}{4} + 1$ items stored in it, which must have all been active at a single point in time.

As shown in Lemma 5.8.20, `merge_shrunk` will never allocate a new array, so except for arrays of size 1 all arrays will be allocated by `link_in`. □

5.8.17 Notes on the current implementation in Pheet

The concurrent LSM was implemented in Pheet for use in task schedulers for strategy scheduling (see Section 2.7). Both task priorities (Section 2.5) and recognizing dead tasks (Section 2.7.4), which are standard features of our scheduling system, require for the data structure to directly access an instance of the strategy object. Due to the generality of our model it was necessary to provide stronger protection for accesses to strategy objects, so that we decided to protect accesses to items by other threads than the owner with a lock-free reference counting scheme. While we make use of some optimizations to reduce the need for communication in the reference counting scheme, the current implementation still suffers from scalability problems due to the high cost of reference counting. In future work we plan to improve scalability of the implementation by either providing a more efficient protection scheme, or making it possible to avoid the protection altogether.

5.8.18 Supporting ρ-relaxed ordering guarantees

In the following we will describe the modifications that need to be made to the concurrent LSM priority queue to implement a k-LSM priority queue, which is a priority queue that provides ρ-relaxation guarantees. It is possible to provide support for both temporal and structural ρ-relaxation, similar to the hybrid k-priority queue from Section 5.4. In this section we restrict ourselves to supporting structural ρ-relaxation, since it allows for higher scalability.

In the concurrent LSM, each thread maintains its own local LSM, which consists of arrays of items, sorted by priority. To maintain structurally ρ-relaxed semantics we allow a thread to globally announce an array, thus requiring all threads to scan it, and to copy all items into their local LSM. Announcements are maintained as a globally accessible linked list with pointers to sorted arrays. Announcements are stored in chronological order, and as soon as all threads have seen an announcement it can be reused.

During a `push` operation, A thread will globally announce all previously unannounced arrays in its local LSM whenever the number of items in its unannounced arrays reaches k. Since each merge of arrays cleans out inactive items, this does not give any temporal ρ-relaxation guarantees. Due to this, a thread may never announce an array, if the number of unannounced items never exceeds k, regardless of the number of `push` operations. It does provide structural ρ-relaxation guarantees, however, with $\rho = k(P-1)$, since no thread can have more than k unannounced items.

5.9 Conclusions and Future Work

Quantitative relaxation is a fairly new technique for improving the scalability of concurrent ordered containers, and we expect many new containers to be developed in the next few years. To our knowledge, the centralized *k*-priority queue and the hybrid *k*-priority [144, 148] were the first published implementations of quantitatively relaxed priority queues. The *spraylist* by Alistarh et al. [10] is a skiplist-based, structurally relaxed, priority queue, for which probabilistic upper bounds on the relaxation can be given. While the spraylist provides good throughput for large amounts of items, the upper bounds provided by the relaxation are too weak to work well with small amounts of items, and thus will lead to a more or less random pop operation in such cases.

We see high potential in LSM for the implementation of relaxed concurrent priority queues, due to its efficient use of memory bandwidth (both merges and spying require reading and writing whole sorted arrays, not just single elements), and due to the regular maintenance provided by the frequent merge operations. Also the maintenance of a local LSM per thread reduces communication cost on NUMA architectures. One main issue with our current approaches is that they mainly focus on reducing the overhead of insertions. While contention is also reduced on pop as a side-effect, this is not always the case, thereby limiting scalability in general.

In future work we plan to explore alternative implementations of concurrent LSM priority queues that, in addition to using relaxations to reduce insertion overhead, will use further relaxations to reduce pop overhead as well. In addition we plan to work on concurrent pareto priority queues, which reduce contention on pop for cases where only a partial ordering between items exists.

6 The Pheet Framework

Pheet is a C++ template library for implementing task parallel applications in the tradition of Intel Threading Building Blocks [93] and C++11 tasks [129]. It provides an easy to program interface, while maintaining a good performance and a low overhead of scheduling tasks.

Even though Pheet can be used as a standalone library for writing task-parallel applications, its real value lies in its configurability. Almost every performance relevant aspect in the system can be replaced in a modular manner at compile time, which allows to quantify the contribution of different scheduler components and data-structures to application performance. The system also allows to compile multiple configurations of a component into a single program.

In addition, Pheet provides various fine-grained performance counters that can be enabled individually at compile time, which gives an insight into the scheduler and its supporting data-structures. When disabled, each performance counter is optimized away by the compiler. Pheet is accompanied by a set of microbenchmarks, which are presented in Chapter 7. These benchmarks enable us to quickly evaluate new prototype schedulers, data structures and algorithmic primitives.

For a long time Pheet has been an internal research tool, and in fact most of the contributions presented in this work have been developed with the help of Pheet. Pheet has also been used as a tool for teaching in a lecture on advanced multiprocessor programming, where students learn how to implement efficient concurrent data structures. In this lecture, Pheet allows students to focus on the implementation by providing an interface into which the data structures implemented by the students can be plugged in. The concept of *places* (Section 2.4.1), the integrated wait-free memory manager (Section 4.3), and the large library of synchronization primitives also greatly simplify implementation of such data structures. The students can evaluate their implementations by running Pheet benchmarks that use their data structures, and can compare their implementations to state-of-the-art implementations and implementations by other students.

In the year 2013 Pheet reached the maturity that allowed us to release it as an open source project[1] under the Boost Public Licence. This allows Pheet to reach a more general audience, allowing it to be used both by application developers that require a library for task parallelism, as well as researchers working on schedulers and concurrent data structures. While the main development platform for Pheet was Linux on x86 processors, the code is written in an operating-system and hardware independent manner that should allow it to run on different operating systems, as well as, different general purpose shared memory processors. The only restrictions that we need to place on the platform is that a C++ compiler and standard library that support most of the C++11 standard needs to be available, as well as the *hwloc* library [32], which we use to configure Pheet to the topology of the hardware it is run on.

[1] `http://www.pheet.org`

6.1 Design goals

Pheet was designed around the idea that almost every performance-relevant aspect of a task-scheduling framework should be configurable. The programmer should be able to take a program written for the framework and run it under various configurations. (e.g., run with different schedulers but otherwise identical configuration). The following considerations have been taken into account when deciding how to provide this configurability:

- The programmer should be able to configure each single performance-relevant aspect in a test program separately.
- Preferably, it should be possible to run tests with different configurations side-by-side inside a single program.
- There should be no significant runtime overhead incurred by the possibility of choosing different variants compared to an implementation where the choice is hard-coded.
- Each implementation should come with its own detailed performance counters, and while implementations should be able to use the same performance counters, it should also be possible to define implementation-specific performance counters to better understand the performance of a specific implementation.
- It should be possible to enable and disable performance counters individually, and disabled performance counters should not incur any performance overhead over code written without performance counter.
- The interface should not be rigid. Each implementation should provide some basic feature set to ensure compatibility, but other than that it should be possible to provide additional functionality in some implementations (e.g., spawn methods that have a specific scheduling strategy as parameter). It should be possible to mix and match implementation variants. If a non-standard feature is used on an implementation variant that does not support it, a compile-time error should be thrown.
- Type-safety, even on aspects that are configurable.

In addition to the configurability goals, Pheet should nonetheless be an easy to use standalone library for task-parallelism. With a simple and efficient interface it should be possible to get a larger number of kernels and applications implemented in the framework. This will allow us to run more tests and comparisons with the framework.

Because of those requirements, the choice was made to use C++ templates as a meta-programming mechanism to allow configurability of performance-relevant aspects at compile-time. The term *Modern C++ Design* [8] has been coined for this type of application design. We rely on the newest C++ standard C++11 [129], which greatly simplifies creating a programming interface that fulfils our requirements due to new templating features, and also provides a new memory model [9], which allows for efficient hardware and operating-system independent implementations of fine-grained synchronization mechanisms.

6.2 Interface

The interface for the library is designed to be as simple as possible, without sacrificing flexibility or performance. A typical task-parallel application/kernel, only requires the programmer to remember a small set of basic commands: `spawn`, `call` and `finish`. The complexity of the high configurability is hidden from the user with default configurations.

As an example, we present a simple parallelization of the quicksort algorithm. Listing 6.1 gives an example of a sequential quicksort implementation that we will parallelize with Pheet throughout this section.

Listing 6.1 Simple implementation of quicksort.

```
#include <functional>
#include <algorithm>

void quicksort(int* begin, int* end) {
  if(end - begin <= 1)
    return;

  int* middle = std::partition(begin, end - 1,
    std::bind2nd(std::less<int>(), *(end - 1)));
  std::swap(*(end - 1), *middle); // move pivot to middle

  quicksort(begin, middle);
  quicksort(middle + 1, end);
}

int main(...) {
  [...]
  // start quicksort
  quicksort(begin, end);
}
```

6.2.1 Simple parallel quicksort

A simple parallel implementation of quicksort in Pheet is shown in Listing 6.2. To parallelize quicksort using Pheet, first Pheet has to be configured by including the Pheet header, and defining a type for Pheet. This single `typedef` hides all the configuration of Pheet in a single statement, and is initialized with the recommended default values. The trick here is, that `pheet::Pheet` is, in fact, based on a templated class with a large amount of template parameters for which reasonable default values have been chosen. The `typedef` then gives the configuration a name, which is used by the application when accessing Pheet. Essentially, the name `Pheet` used by the application is an alias for a specific configuration of Pheet. To reconfigure Pheet, only the `typedef` needs to be changed as shown in the next section.

In addition, the first recursive function call to quicksort is replaced by a `Pheet::spawn` statement. Code executing Pheet calls has to be enclosed in a Pheet environment, which maintains all the parallelism in the background. It is initialized using the RAII (Resource acquisition is initialization) pattern [129]. As long as the variable of the type `Pheet::Environment` is in scope, all Pheet statements can be used. As soon as it runs out of scope, all parallel executions in Pheet are finished and Pheet is cleaned up. The next statement after `Pheet::Environment` runs out of scope will only be executed after all tasks spawned inside the Pheet environment have finished executing.

In practice, due to the overhead incurred by spawning tasks, an efficient parallel implementation of quicksort should provide a cutoff value at which it switches to the sequential algorithm. While we do not show such a cutoff here for simplicity of the code, such a cutoff is used in the quicksort benchmark provided as microbenchmark with Pheet and presented in Section 7.5.

Listing 6.2 Simple parallel implementation of quicksort in Pheet.

```
#include <functional>
#include <algorithm>

// Configure Pheet
#include <pheet/pheet.h>
typedef pheet::Pheet Pheet; // Default configuration

void quicksort(int* begin, int* end) {
  if(end - begin <= 1)
    return;

  int* middle = std::partition(begin, end - 1,
    std::bind2nd(std::less<int>(), *(end - 1)));
  std::swap(*(end - 1), *middle); // move pivot to middle

  Pheet::spawn(quicksort, begin, middle);
  quicksort(middle + 1, end);
}

int main(...) {
  [...]

  // Initialize pheet environment
  {Pheet::Environment p;
      // start quicksort
    quicksort(begin, end);
  } // At the end of the Pheet scope, quicksort is guaranteed to be finished
}
```

6.2.2 Configuring Pheet to use a different scheduler

One of the main features of Pheet is its configurability. Most components of Pheet can easily be replaced. As an example, if none of the advanced features are required, the standard scheduler of Pheet can be replaced by the lightweight `BasicScheduler`. This can be done by using the `::WithScheduler` member of Pheet as shown in Listing 6.3. The member `::WithScheduler` of `Pheet` is a template alias that refers to a templated type that is identical to the type it is contained in, except for the scheduler that is replaced by the scheduler specified as template parameter. Many types in Pheet contain such members starting with `::With` that allow to reconfigure types without the need to specify a whole list of template arguments.

6.2.3 Finish regions

While the Pheet environment contains an explicit finish, it might be necessary to explicitly synchronize between tasks throughout a parallel execution. Pheet relies on the *async/finish* model (Section 2.2.3) for task synchronization. In Pheet there are two flavors of finish: *finish calls*, and *finish regions*. A finish call, which is shown in Listing 6.4, calls a single function (or task object), and ensures that all transitively spawned tasks in this function have finished executing before proceeding to the next step.

Finish regions, on the other hand, allow to encapsulate multiple lines of code into a single region. At the end of the region, Pheet implicitly waits for all transitively spawned tasks to finish executing. Finish regions are declared using the RAII pattern similar to Pheet envi-

Listing 6.3 Reconfiguration of Pheet to use the `BasicScheduler`.

```
#include <functional>
#include <algorithm>

// Configure Pheet
#include <pheet/pheet.h>
#include <pheet/sched/Basic/BasicScheduler.h>
typedef pheet::Pheet::WithScheduler<pheet::BasicScheduler> Pheet;

void quicksort(int* begin, int* end) {
  if(end - begin <= 1)
    return;

  int* middle = std::partition(begin, end - 1,
    std::bind2nd(std::less<int>(), *(end - 1)));
  std::swap(*(end - 1), *middle); // move pivot to middle

  Pheet::spawn(quicksort, begin, middle);
  quicksort(middle + 1, end);
}

int main(...) {
  [...]

  // Initialize pheet environment
  {Pheet::Environment p;
    // start quicksort
    quicksort(begin, end);
  } // At the end of the Pheet scope, quicksort is guaranteed to be finished
}
```

ronments. For this, we create an instance of the Finish class in the given scope as shown in Listing 6.5. As soon as the Finish object runs out of scope, the Finish region ends, and the next statement is only executed if all tasks transitively spawned inside the finish region have finished executing.

6.2.4 Task class

While the use of functions in task spawns allows for a simple parallelization of sequential calls, it has one limitation: There is no separation between task initialization work, which should be performed in the context of the spawning thread, and the actual task execution. Also, advanced schedulers may delete tasks without them ever being executed, thereby making it necessary to separate clean-up from the task execution. For this reason, Pheet also allows the use of *task classes* instead of function pointers.

Like task functions, task classes can be used with `spawn`, `call` and `finish` statements. Since a task class is a type, it is not passed to those statements as a normal parameter, but as a template parameter instead, as shown in Listing 6.6. The constructor of the task class will be called in the context of the spawning thread immediately when the task is spawned. The `operator()` method represents the actual function body of the task, and will be executed when the task is executed. Finally, the destructor will be called whenever a task is deleted from the scheduling system, regardless of whether it was executed or not.

Listing 6.4 Quicksort example in Pheet with a `finish` call.

```
1 #include <functional>
2 #include <algorithm>
3
4 // Configure Pheet
5 #include <pheet/pheet.h>
6 typedef pheet::Pheet Pheet; // Default configuration
7
8 void quicksort(int* begin, int* end) {
9   if(end - begin <= 1)
10    return;
11
12  int* middle = std::partition(begin, end - 1,
13    std::bind2nd(std::less<int>(), *(end - 1)));
14  std::swap(*(end - 1), *middle); // move pivot to middle
15
16  Pheet::spawn(quicksort, begin, middle);
17  quicksort(middle + 1, end);
18 }
19
20 int main(...) {
21  Pheet::Environment p;
22  [...]
23
24  // start quicksort
25  Pheet::finish(quicksort, begin, end);
26 }
```

6.3 Framework Structure

Pheet has a modular structure, where each basic building block of the framework is realized as a *concept* for which there can be multiple implementations. Concepts are a common technique in generic programming and are well-known since the introduction of the *C++ Standard Template Library (STL)* [84]. In general, a concept describes a set of syntactic and semantic requirements for a type. Any type conforming to a certain concept can be used as input to an algorithm or data-structure requiring this concept. There has been an effort to provide support for concepts in the C++ standard [67], but as of now the proposal has not been accepted for inclusion. In our framework, a concept is provided as a textual description of the interface, and its semantics.

One example of a concept used in the framework is a Mutex. A Mutex should provide a `lock()` and an `unlock()` method. The `lock()` method is a blocking method that only returns after a lock has been acquired. Different lock implementations all conform to the same interface, but perform differently depending on lock congestion, and the system the lock is used on. Optionally, an implementation can also provide a `try_lock(long int time_ms)` method, which tries to acquire a lock. If it does not succeed in a given time-frame, the method returns false. This method is optional and not provided by implementations, since it is not required in many applications, and it makes some mutex implementations more complex, or even more expensive.

On a higher level, the framework is built from different parts, namely: *Primitives*, *Memory management*, *Data-structures*, *Execution models* and *Schedulers*, which will be described in the following sections.

Listing 6.5 Quicksort example in Pheet with a finish region.

```
#include <functional>
#include <algorithm>

// Configure Pheet
#include <pheet/pheet.h>
typedef pheet::Pheet Pheet; // Default configuration

void quicksort(int* begin, int* end) {
  if(end - begin <= 1)
    return;

  int* middle = std::partition(begin, end - 1,
    std::bind2nd(std::less<int>(), *(end - 1)));
  std::swap(*(end - 1), *middle); // move pivot to middle

  Pheet::spawn(quicksort, begin, middle);
  quicksort(middle + 1, end);
}

int main(...) {
  Pheet::Environment p;
  [...]

  {Pheet::Finish f; // declare finish region
    // start quicksort
    quicksort(begin, end);
    quicksort(begin2, end2);
  } // End of finish region, both lists will be sorted after this point
}
```

6.3.1 Primitives

The primitives in our work-stealing framework are basic building blocks for parallel applications that are needed by other parts of the framework, but are also made available to the users. Currently, the following primitives are provided:

- **Mutexes**: Simple mutex implementations. Are rarely used, as the framework generally tries to use lock-free techniques. They are provided nonetheless to allow for simple comparison between lock-based and lock-free data-structure implementations. Also, they are a valuable research and teaching tool for testing out new lock implementations.

- **Backoff**: Generally used in lock implementations and lock-free data-structures to reduce congestion. Different implementations may use different backoff strategies. By default we use an exponential backoff, which is capped at a certain maximum backoff.

- **Barriers**: A simple low-level barrier that also provides signal/wait semantics. (Inspired by Phasers [123], although not as powerful)

- **Finishers** and **Shared Pointers**: Finisher hyperobjects used for transitive termination detection (e.g. for pointers).

Listing 6.6 Task classes in Pheet.

```
1 class TaskImplementation : public Pheet::Task {
2 public:
3   RecursiveParallelPrefixSum2OffsetTask(/* task parameters */)
4   {
5     // Constructor - perform initialization here (in context of spawning thread)
6   }
7
8   virtual ~RecursiveParallelPrefixSum2OffsetTask() {
9     // Destructor - perform clean-up here
10  }
11
12  virtual void operator()() {
13    // Task body - This will be executed when task is executed
14  }
15 };
16
17 int main(...) {
18   Pheet::Environment p;
19   [...]
20
21   Pheet::finish<TaskImplementation>(/* task parameters */);
22 }
```

- **Reducers**: Contains both ready-to-use reducer hyperobjects (max-, min- and sum-reducers for scalar types), as well as interfaces that allow to implement reducer hyperobjects for different types and operators.
- **Performance counters**: Using reducers, they provide a simple way to measure different performance relevant aspects inside the framework and in applications. Each performance counter can be activated and deactivated separately, and deactivated performance counters should not lead to any performance overhead.

6.3.2 Memory management

A collection of memory management methods that simplify the implementation of concurrent data structures, including the wait-free memory manager described in Section 4.3. Unfortunately, hazard pointers [105] cannot be added to this collection, since they are protected by US Patent US 20040107227 A1. As an open source framework, Pheet therefore cannot make use of any concurrent data structures that require hazard pointers.

6.3.3 Data structures

Pheet contains a wide variety of (mostly concurrent) data structures, many of which are presented in Chapter 4. While it would be useful for the user, the focus of Pheet is not on providing a full library of concurrent data structures. Instead, it contains mainly specialized data structures required by one of the schedulers.

6.3.4 Execution models

An *execution model* is an abstraction of the concrete hardware the system is running on. It provides a scheduler with information about processing resources in the system, as well as,

information on the memory hierarchy. The execution model is modelled as a tree, where the leaves represent the processing resources, and the other nodes represent levels in the memory hierarchy that are shared between processing resources. A similar approach for modelling the machine is employed in [56].

Pheet uses the hardware model to initialize the *places* (Section 2.4.1) of a scheduler, and to bind each place to a specific processing unit or group of processing units. In addition, Pheet contains a virtual distance metric that is related to the communication costs between two places. It is guaranteed that for two places with a certain distance, the communication costs will not be lower than if one of these places communicates with a place with smaller distance. In addition, Pheet is NUMA-aware, allowing an application to see whether two places are located on different NUMA nodes. We also plan to allow applications in Pheet to check whether a certain memory location is NUMA-local in the future, as soon as this is efficiently supported by common operating systems and hardware.

An execution model does not necessarily have to exactly represent the system it is running on. One may simulate a system with more processing resources by over-subscription, or make only a part of the machine accessible to the scheduler. By default, Pheet uses a machine model that wraps functionality from the *hwloc* [32] library, and which automatically initializes Pheet to take advantage of all (shared memory) processor cores in a machine.

6.3.5 Schedulers

Schedulers are the central part of Pheet and tie together all components. There is a common subset of functionality each scheduler has to support, like `spawn` and `finish`. There is no restriction as to what additional features a scheduler is allowed to provide. As a baseline, Pheet provides a scheduler called the `SynchroneousScheduler`, which will sequentially execute a parallel algorithm written in Pheet, by treating any task spawn as a function call. This is both useful to debug the sequential semantics of a program, as well as to provide baseline sequential performance of the parallel algorithm without scheduler overhead. As another baseline, Pheet provides the `BasicScheduler`, a lightweight work-stealing scheduler with no additional functionality.

The advanced schedulers provided by Pheet include a mixed-mode scheduler, which supports the mixed-mode parallel programming model presented in Section 2.6, and various flavours of schedulers with support for strategies. The first strategy scheduler based on priority work-stealing (Section 5.2) is called the `StrategyScheduler`. The `BStrategyScheduler` (the B stands for *bounded*) allows for bounds on the priority inversions of tasks by using the centralized k-priority queue (Section 5.3) or the hybrid k-priority queue (Section 5.4). Finally, the latest model of strategies that allows each strategy to use a different data structure for storing tasks is used by the `StrategyScheduler2`. This scheduler is based on the two-level ordered container presented in Section 5.5. Due to its flexibility, allowing both bounds on priority inversions where needed and low overhead work-stealing deques for other tasks, `StrategyScheduler2` will be used as the default scheduler in Pheet in the future.

7 The Pheet Benchmarks

To allow for an experimental evaluation of each aspect of the Pheet framework, Pheet is accompanied by a set of microbenchmarks, each aimed at evaluating specific aspects of the framework. In this chapter we describe each benchmark application separately, motivate its inclusion in the Pheet framework, and discuss the insights gained through this benchmark.

7.1 Setup

We used two shared memory systems for the evaluation. One is a Intel Xeon based system comprising four Intel Xeon E7-8850 processors with 20 cores each, leading to a total of 80 cores. Additional parallelism can be achieved with simultaneous multi-threading, but since many applications are memory bandwidth bound on this architecture, Pheet is not configured to take advantage of this by default. This system has 1TB of main memory. To simplify the discussion throughout this chapter, this system will be referred to by the name *Mars* throughout this chapter. A chart, showing the processor and memory hierarchy of *Mars* is shown in Figure 7.1.

The other system that we use for evaluation is an AMD Opteron based system consisting of four AMD Opteron 6168 processors with 12 cores each, leading to a total of 48 cores. This system has 128GB of main memory. We will refer to this system by the name *Saturn*. The processor and memory hierarchy is shown in Figure 7.2.

We rely on two compilers for our experiments: *GCC 4.8.2* and the *Intel C++ compiler 14.0.1*. All programs are compiled using the `-O3` flag to enable standard compiler optimizations.

7.2 Methodology

Experiments were performed for both machines and on both compilers. Results are always shown for both machines, but only for one compiler (typically the compiler with better sequential performance). In most cases, the code created by both compilers behaves similarly and only the absolute performance differs. In cases where code shows different behaviour on another compiler this is mentioned in the text.

All results shown are mean values over at least 20 experiments. Confidence intervals are shown for all values. If no confidence interval is visible for a value this means that the interval is too small to be visible and thus confidence is high. For benchmarks with high variance in results we use more than 20 experiments to achieve reasonable confidence intervals. Where applicable we use different (random) inputs of same input size for each of the experiments and each implementation variant is executed on exactly the same inputs. For random inputs this is achieved by fixing the random seed for each of the experiment.

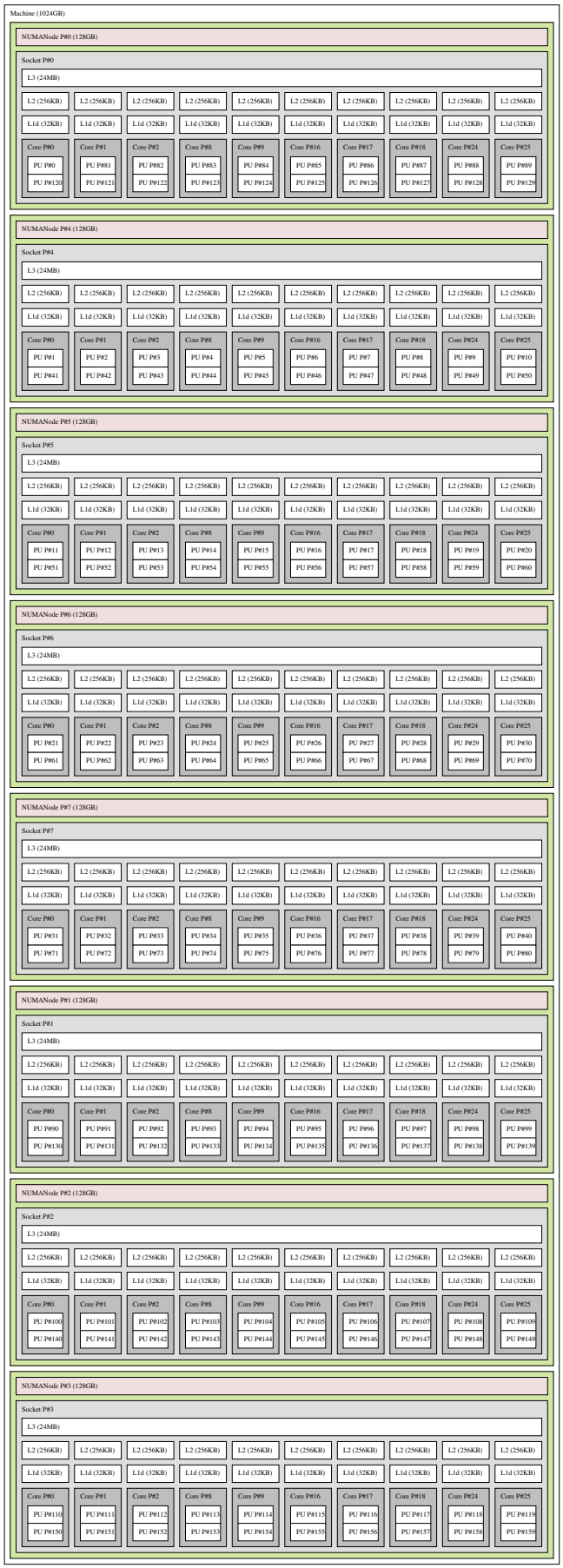


Figure 7.1: Processor and memory hierarchy of the *Mars* system as presented by the `lstopo` utility of the *hwloc* library [32].

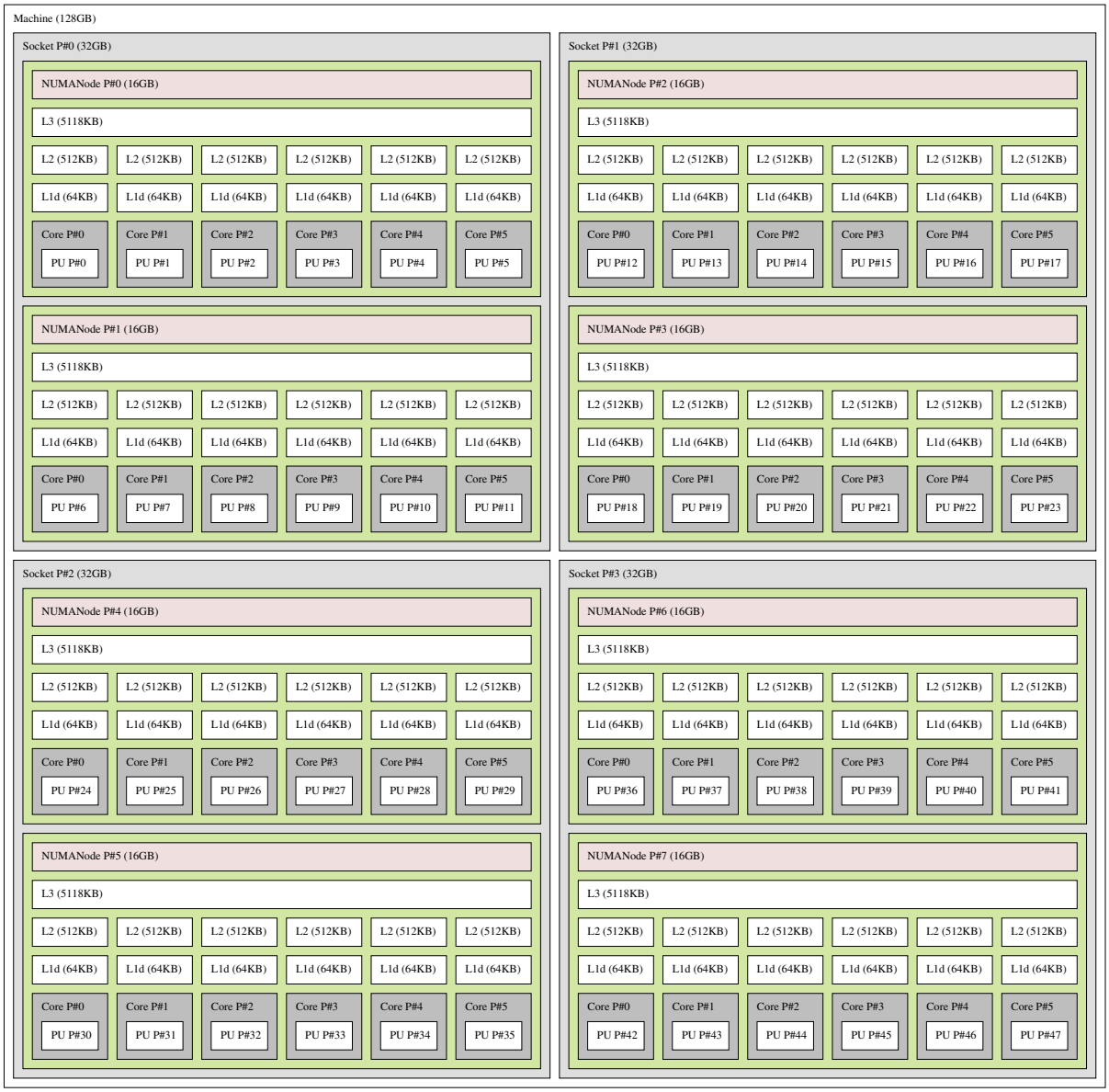


Figure 7.2: Processor and memory hierarchy of the *Saturn* system as presented by the `lstopo` utility of the *hwloc* library [32].

To have the same starting conditions for all experiments, the input data is freshly allocated and generated the same way before each experiment. Also, task schedulers are freshly initialized before each experiment and shut down and destructed afterwards.

Our primary metric of evaluation is execution time of the benchmark. The execution time is measured inside the application and does not include the time for generating/loading the input data, initializing/shutting down the scheduler or verifying the result. Depending on the application we use additional performance counters for our evaluation, which are explained in the application specific sections.

7.3 Unbalanced Tree Search

We start off with the Unbalanced Tree Search (UTS) benchmark by Olivier et al. [109]. This synthetic benchmark spawns a large number of small tasks, corresponding to nodes in a unbalanced search tree, according to a given distribution. The decision on how many subtasks to spawn from a given task is made with the help of a hash of the parent descriptor and the child index. This makes it possible to get the exact same tree every time, based on the parameters used to initialize the tree.

UTS is a useful benchmark to evaluate dynamic load balancing techniques. Each task in UTS is extremely lightweight, thereby exposing the overhead of the scheduling system. Since tasks are computation bound and do not share any memory with other tasks, memory bandwidth is not an issue with UTS, which allows to measure the raw scalability of a scheduler on a given system.

The UTS benchmark provides two types of search trees that can be used: *binomial trees* and *geometric trees*. In the *binomial tree*, each node has m children with probability q, and no children with probability $1 - q$, where m and q are parameters of the class of binomial trees. The depth of the tree follows a power law. Information about where in the tree a node is located cannot be used to infer the number of its ancestors, thus making the binomial tree the perfect adversary for dynamic load balancing techniques. The *geometric tree*, on the other hand, follows a geometric distribution. Due to the long tail of geometric distributions, some nodes will have significantly more descendants than others, resulting in extremely unbalanced trees. Contrary to binomial trees, sub-trees closer to the root have a larger expected size.

7.3.1 Aim of this Benchmark

The main purpose of the UTS benchmark in Pheet is to measure the overhead and scalability of different schedulers in the Pheet framework. Tasks in UTS are extremely lightweight, both concerning execution time, and memory usage, and require only local information to compute. Thus, a large part of the execution time is used by calls to the scheduler. We use this fact to compare the overheads of different schedulers.

With the UTS benchmark we also evaluate the potential performance improvements due to the spawn to call conversion feature of our strategy schedulers (see Section 2.7.1). This feature is used to decrease the scheduler overhead by converting task spawns into function calls whenever sufficient parallelism is available. For geometric trees, the expected size of a sub-tree rooted at a specific node is dependent on the depth of the node, with the expected size decreasing with higher depth. To take this into account, the task spawns of higher depth have a lower threshold for number of concurrently available tasks necessary to achieve this result. For binomial trees this is not the case, but the fact that the expected size of the subtree is the same for every node can be used to convert task spawns to function calls whenever there is sufficient parallelism, which is the case whenever the number of tasks available for execution exceeds a certain threshold.

7.3.2 Implementation

The UTS benchmark in Pheet is based on the original UTS benchmark code, which has been adapted for the use in Pheet. We provide two implementations, a baseline task-parallel implementation that does not use any Pheet specifics, and an implementation that takes advantage of strategies.

Listing 7.1 shows the implementation of UTS tasks with strategies. The baseline implementation is not shown, since it only differs when spawning a child task where a normal `spawn` statement is used without a strategy. A task in UTS expands a single node by creating nodes for all its children and then spawning a new task for each child. Each node is initialized with its own random seeds by the `rng_spawn` method from UTS. The parameter `computeGranularity` can be used to increase the granularity of the computation by calling `rng_spawn` multiple times. Since we were mainly interested in exposing the overhead of our schedulers by keeping granularity low, we kept this parameter at 1 for all our measurements.

Listing 7.1 Implementation of the UTS task with strategies in Pheet.

```
1  virtual void operator()()
2  {
3    Node child;
4    int parentHeight = parent.height;
5    int numChildren;
6
7    numChildren = uts_numChildren(&parent);
8    auto childType  = uts_childType(&parent);
9
10   // record number of children in parent
11   parent.numChildren = numChildren;
12
13   // construct children and push onto stack
14   if (numChildren > 0)
15   {
16     int i, j;
17     child.type = childType;
18     child.height = parentHeight + 1;
19
20     for (i = 0; i < numChildren; i++)
21     {
22       for (j = 0; j < computeGranularity; j++)
23       {
24         // computeGranularity controls number of rng_spawn calls per node
25         rng_spawn(parent.state.state, child.state.state, i);
26       }
27       Pheet::spawn_s<Self>(UTSStrategy(child.height), child);
28     }
29   }
30 }
```

The implementation of the UTS strategy is shown in Listing 7.2. It implements the `can_call` method, which can be used by the scheduler to determine whether a task spawn can be converted into a function call. The strategy allows the scheduler to convert task spawns to function calls, whenever the number of tasks in the task queue is at least 4. For the first tasks being spawned, the threshold is slightly higher, both to spread work to other places more quickly, as well as to make tasks close to the root more likely to be stolen, which is relevant

for geometrically distributed UTS trees.

Listing 7.2 Implementation of the UTS strategy in Pheet. (simplified)

```
1  class UTSStrategy : public BaseStrategy {
2  public:
3    UTSStrategy(int height)
4    :height(height){}
5
6    /*
7     * Checks whether spawn can be converted to a function call
8     */
9    inline bool can_call(TaskStoragePlace* p) {
10     if(height >= 8)
11       return p->size() >= 4;
12     else
13       return p->size() >= 12 - (unsigned)height;
14   }
15
16 private:
17   int height;
18 };
```

7.3.3 Results

We have performed our experiments on various trees generated by UTS. Since trees generated by UTS have an extremely high variance in size and structure, it is not sensible to aggregate performance measurements on different trees. Instead, each experiment was performed on a specific tree, and repeated 20 times for each implementation. We ran the experiments on both the Intel compiler and GCC. The behaviour and scalability of UTS is the same on both compilers. Since the code generated by the Intel compiler is faster, we only show the results obtained with the Intel compiler. An exception to this is the experiment on the large tree shown in Figure 7.4, which, due to its long running time, was only compiled and run with GCC.

In our experiments we compare the following variants of UTS:

Sequential A sequential execution of UTS without a task scheduler. Each task spawn is converted into a function call at compile time. This sequential execution is a good baseline for measuring the task scheduling overhead.

Strategies This implementation uses scheduling strategies as described in Section 7.3.2 to convert task spawns into function calls at run-time whenever sufficient parallelism is available to reduce scheduler overhead. (`StrategyScheduler2` in Pheet)

Strategy Scheduler A UTS implementation that does not make use of strategies, but is executed on a scheduler that is capable of supporting strategies. This implementation is used to measure the overhead of supporting strategies in a scheduler. It uses the `StrategyScheduler2` from the Pheet framework. Since UTS does not require task priorities, the root container of the two-level concurrent ordered container (see Section 5.6) is used to store UTS tasks.

Work-Stealing A standard work-stealing scheduler provided as a baseline for task-parallel executions. It uses the `BasicScheduler` from the Pheet framework.

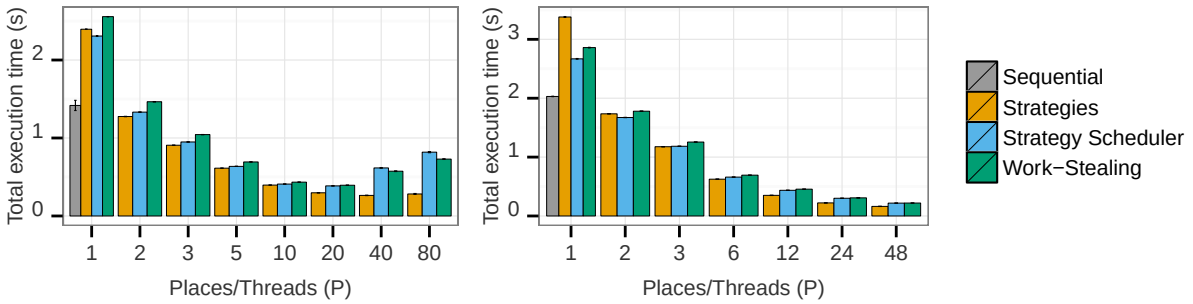


Figure 7.3: Average execution time of UTS on a geometric tree with 4130071 nodes on *Mars* (left) and *Saturn* (right). (UTS parameters: `T1 -t 1 -a 3 -d 10 -b 4 -r 19`, Intel compiler)

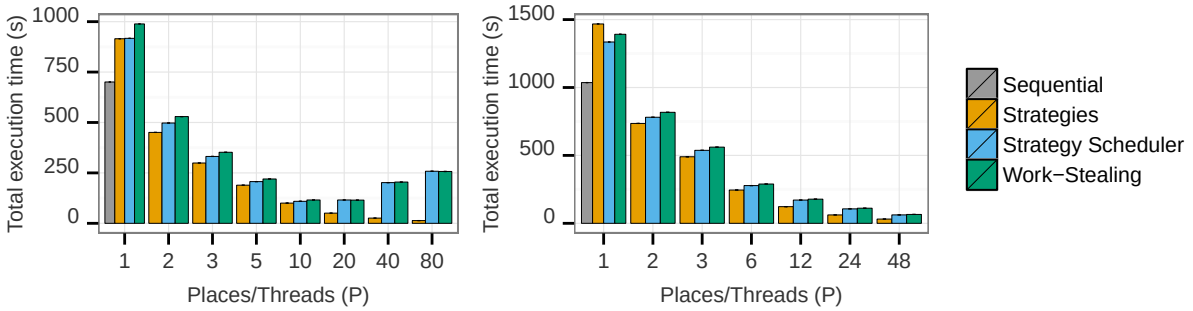


Figure 7.4: Average execution time of UTS on a geometric tree with 1635119272 nodes on *Mars* (left) and *Saturn* (right). (UTS parameters: `T1XL -t 1 -a 3 -d 15 -b 4 -r 29`, GCC)

Results for running the UTS benchmark for a small tree with 4130071 nodes based on a geometric distribution for both *Mars* (left Figure) and *Saturn* (right Figure) are shown in Figure 7.3. As described before, we kept the granularity of tasks in UTS as small as possible to expose the overhead of task schedulers. The overhead of task scheduling compared to sequential UTS is clearly visible on both machines, but already with two threads some speedup can be achieved. On *Saturn*, the algorithm scales well, achieving a speedup of 10.9 on 48 cores for the standard work-stealing algorithm with regard to the sequential baseline. Even better scalability is achieved when using strategies, with a speedup of 15.7.

The results of this experiment also show that the strategy scheduler does not have any visible overhead compared to a standard work-stealing scheduler if no strategies are used. In fact it is even faster for smaller numbers of threads, which means that the deques used by the strategy scheduler are more efficient than their counterparts in the original Pheet work-stealing scheduler.

On *Mars* it was not possible to achieve the same scalability with all implementations. For both the strategy scheduler and the standard work-stealing scheduler the costs of stealing tasks from other NUMA nodes outweighed the gains on average, leading to a slowdown when using more than 20 threads. The use of strategies to convert task spawns into function calls helps to get rid of this bottleneck, however, resulting in a speedup of 6.6 on 80 threads.

Results for a large tree with otherwise similar parameters are shown in Figure 7.4. Similar results can be observed as for a smaller tree, but higher scalability is achieved with a speedup of 51.3 on *Mars* and a speedup of 32.9 on *Saturn* compared to the sequential implementation.

Figures 7.5 and 7.6 show results for a medium sized binomial and a hybrid tree. In both cases the trends are similar to what we have shown for the geometric tree.

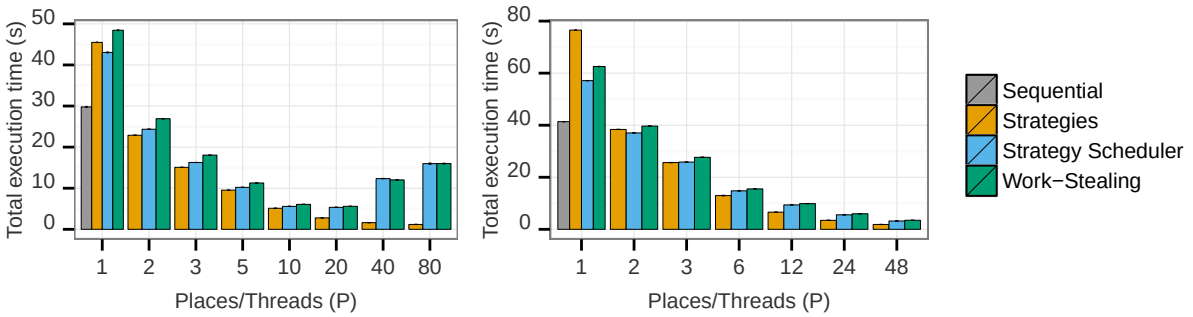


Figure 7.5: Average execution time of UTS on a binomial tree with 111345631 nodes on *Mars* (left) and *Saturn* (right). (UTS parameters: `T3L -t 0 -b 2000 -q 0.200014 -m 5 -r 7`, Intel compiler)

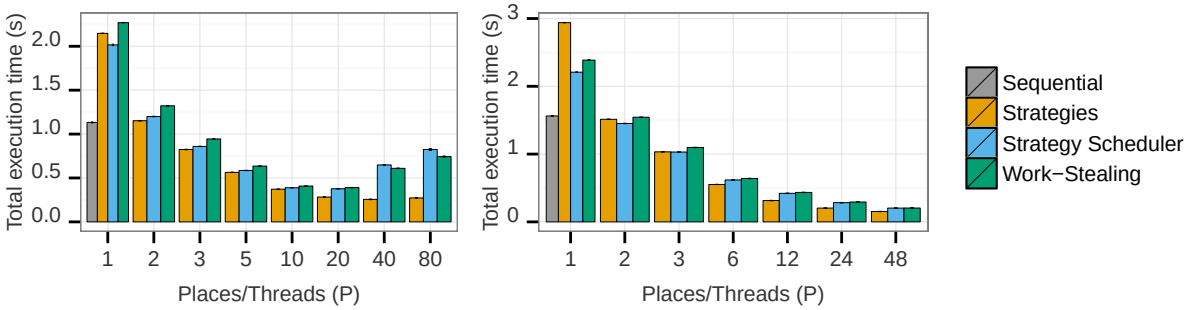


Figure 7.6: Average execution time of UTS on a hybrid tree with 4132453 nodes on *Mars* (left) and *Saturn* (right). (UTS parameters: `T4 -t 2 -a 0 -d 16 -b 6 -r 1 -q 0.234375 -m 4 -r 1`, Intel compiler)

7.4 Graph Bipartitioning

The branch-and-bound paradigm is generally well suited to parallelization [43]. Efficient parallel branch-and-bound implementations rely on a concurrent data structure for storing unexplored subproblems [78, 85, 117].

We focus on the well-known, NP-hard *graph bipartitioning problem* [111] where the vertices of an undirected, weighted, n-node, m-edge graph are to be partitioned into two sets with given sizes with minimum total cut weight. The branch-and-bound algorithm for graph bipartitioning fixes a single node at each step, by assigning it to one of the sets for each branch. For bounding (elimination) of sub-problems we use a simple, easily computable lower bound [40]. Incrementally updating the lower bound for each new node subproblem takes $O(n \log n + m/n)$ (amortized) steps.

7.4.1 Aim of this benchmark

Performance of graph bipartitioning depends on the order in which sub-problems are explored. Finding close to optimal solutions early on allows to cut off branches that might otherwise be explored, thus greatly reducing the execution time. We use simple heuristics to determine promising branches, which are then prioritized in a priority scheduler.

Branch-and-bound algorithms are well suited for task parallelism, due to the high granularity per task for many tasks. Nonetheless, the granularity of tasks greatly varies for different

branches, with granularity strictly decreasing with each node that is assigned to a set. For tasks with small granularity, the overhead of the scheduling system becomes visible, and due to a large number of fine-grained tasks in each execution this has a visible impact on total execution time. We use the spawn2call feature of our strategy schedulers (see Section 2.7.1) to reduce this overhead. Since spawn2call overrides prioritization of tasks, parameters have to be chosen carefully to achieve a reasonable balance between early exploration of promising branches and reduction of scheduler overheads.

7.4.2 Implementation

We encapsulated the actual logic for maintaining sub-problems, branching and calculating the lower bounds into the class `SubProblem`, which allows to keep the parallel implementation separate from the logic for branching and bounding. This fits well with the Pheet philosophy, and allows us to mix and match parallel implementations with bounding implementations. The branching operation is performed by the method `split`, which splits the given sub-problem into two sub-problems by assigning a single, previously unassigned node to one of the two subsets.

We encapsulated the actual logic for maintaining sub-problems, branching and calculating the lower bounds into the class `SubProblem`, which allows to keep the parallel implementation separate from the logic for branching and bounding. This fits well with the Pheet philosophy, and allows us to mix and match parallel implementations with bounding implementations.

Whenever a subproblem represents a unique solution (with no more branching possible), this can be determined using the `can_complete` method. The method `complete_solution` is then used to calculate the actual solution. The value of the currently best known feasible solution is kept in a global variable that is updated atomically by `complete_solution` whenever a better solution is encountered. Due to the atomicity requirement, it is not efficient to store the whole solution in this global variable, but instead we only store the value. This value can be retrieved using the `get_global_upper_bound` method of a sub-problem. The full solution is stored in a reducer hyperobject, which is updated by the `update_solution` method.

The parallel algorithm

Our actual parallel implementation of graph bipartitioning with strategies is shown in Listing 7.3. For better readability we omitted the memory management for sub problems in this code. Again, we do not show the code for the baseline implementation without strategies, since it only differs at the `spawn` statements.

In our implementation of graph bipartitioning a task is responsible for a single branching operation, where a single node of the graph is assigned to one of the two subsets. This is achieved using the `split` operation of the sub-problem. For each of the resulting two sub-problems the task then checks whether it represents a unique solution, and otherwise recursively calls itself to further expand the sub-problem.

Strategies

We experimented with different strategies for an efficient execution of graph bipartitioning. Listing 7.4 presents a strategy, which we call the *estimate strategy* that resulted in comparably good performance. The main intuition behind this strategy is that the execution order of tasks is guided by a heuristic that is guided by an estimate of the actual solution value. Tasks spawned at a single place will always be executed in order of lowest estimate. While the actual estimate is dependent on the implementation of `SubProblem`, we require this estimate to be

Listing 7.3 Implementation of the graph bipartitioning task with strategies in Pheet. (simplified)

```
virtual void operator()()
{
  // Split into two sub problems. One will be stored in the original sub problem,
  // the other in sub_problem2
  SubProblem* sub_problem2 =
      sub_problem->split();

  if(sub_problem->can_complete()) {
    sub_problem->complete_solution();
    sub_problem->update_solution(best);
  }
  else if(sub_problem->get_lower_bound() < sub_problem->get_global_upper_bound()) {
    Pheet::spawn_s<Self>(Strategy(sub_problem),
        sub_problem, best);
  }

  if(sub_problem2->can_complete()) {
    sub_problem2->complete_solution();
    sub_problem2->update_solution(best);
  }
  else if(sub_problem2->get_lower_bound() <
      sub_problem2->get_global_upper_bound()) {
    Pheet::spawn_s<Self>(Strategy(sub_problem2),
        sub_problem2, best);
  }
}
```

fairly pessimistic, so that in most cases at least one of the sub-problems created by a `split` will have a lower estimate than its parent, resulting in a depth-first execution in most cases.

To reduce communication required between places, tasks spawned by other places will not be executed ordered by estimate, but instead the task with the highest uncertainty will be executed first. The uncertainty is defined as the difference between the estimate and its lower bound. We base this strategy on the assumption that tasks with higher uncertainty will tend to generate more work, thereby reducing the need to steal work from other threads. In this case we let all places attempt to expand this node, and take the additional congestion into account in the hopes of finding a good upper bound faster.

In addition to manipulating the execution order of tasks, graph bipartitioning also takes advantage of the *elimination of dead tasks* and *spawn-to-call* features of strategies. The elimination of dead tasks feature is used as an additional bounding measure whenever a new, tighter upper bound was found after a task was spawned.

The spawn-to-call feature can reduce the overhead of the task scheduler by reducing parallelism if enough parallelism is available. Since converting task spawns to function calls may work against the heuristics that ensure that the best solution is found faster, it is only performed whenever the gains of these heuristics are expected to be negligible. The best compromise we found (shown in Listing 7.4) was to enable spawn-to-call conversion for tasks that are expected to only yield little additional work, which are tasks for which the lower bound is close to the global upper bound. The exact boundary at which task spawns are converted to function calls depends on the number of tasks stored in the queues, to ensure that sufficient parallelism is available at all times.

Listing 7.4 Implementation of the graph bipartitioning strategy in Pheet. (simplified)

```
1  class EstimateStrategy : public BaseStrategy {
2 public:
3    // Inform scheduling system about the preferred priority queue implementation
4  typedef PriorityQueueImplementation TaskStorage;
5
6    EstimateStrategy(SubProblem* sub_problem)
7    :place(Pheet::get_place()),
8     sub_problem(sub_problem),
9     estimate(sub_problem->get_estimate()),
10    uncertainty(sub_problem->get_estimate() - sub_problem->get_lower_bound()),
11    lower_bound(sub_problem->get_lower_bound()),
12   global_upper_bound(sub_problem->global_upper_bound)
13    {}
14
15 bool prioritize(Self& other) {
16   Place* p = Pheet::get_place();
17   if(this->place == p) {
18      if(other.place == p)
19       return estimate < other.estimate;
20      else
21        return true;
22   }
23   else if(other.place == p) {
24     return false;
25   }
26   return uncertainty > other.uncertainty;
27 }
28
29 bool dead_task() {
30   return lower_bound == *global_upper_bound;
31 }
32
33 /*
34  * Checks whether spawn can be converted to a function call
35  */
36 inline bool can_call(TaskStoragePlace* p) {
37   // Retrieve newest value for the global upper bound
38   size_t ub = *global_upper_bound;
39   // Do not bother creating a task if this branch will be cut off
40   if(lower_bound >= ub)
41     return true;
42   size_t diff = ub - lower_bound;
43   size_t open = (diff / (1 + (ub/sub_problem->size))) >> 1;
44
45   return open < 28 && (p->size()*p->size() >> open) > 0;
46 }
47
48 private:
49 Place* place;
50 SubProblem* sub_problem;
51   size_t estimate;
52   size_t uncertainty;
53 size_t lower_bound;
54 size_t* global_upper_bound;
55 };
```

7.4.3 Results

We have performed our experiments on Erdős-Rényi random graphs [54, 64]. All measurements shown in this section are averages over 100 experiments on the same type of graph, but with different random seeds. For each implementation variant of the algorithm the same 100 random graphs were used for better comparability.

In our experiments we compare the following variants of graph bipartitioning:

Sequential A sequential execution of the branch and bound algorithm, where all task spawns are replaced by function calls.

Work-Stealing Baseline implementation based on work-stealing that does not use strategies.

Priority Work-Stealing Strategy-based implementation that uses priority work-stealing (Section 5.2).

Centralized k Strategy-based implementation that uses the centralized k-priority queue from Section 5.3 for storing tasks.

Hybrid k Strategy-based implementation that uses the hybrid k-priority queue from Section 5.4 for storing tasks.

CLSM Strategy-based implementation that uses the concurrent LSM priority queue from Section 5.8 for storing tasks.

Some of the presented schedulers, name centralized k, hybrid k and priority work-stealing, rely on an older version of the interface for strategies, and therefore use a strategy different from the strategy presented in the previous section, which is the best strategy for graph bipartitioning found in previous work [142, 143]. The new strategy for graph bipartitioning used for the CLSM implementation is still a work in progress, and while it tries to imitate the behaviour of the old strategy it has not yet been tuned to the same extent, thus better performance results can be expected for the older implementation.

Results for dense graphs of size $n = 35$ and edge probability $p = 90\%$ are shown in Figure 7.7. The shown results were obtained using the Intel compiler. Results obtained with GCC show similar behaviour on both machines, but provide worse absolute performance, and will therefore not be shown separately. It can be seen that task priorities can even help improve the performance in the sequential case, with a 9% improvement over the sequential execution for priority work-stealing. The algorithm provides good scalability on both machines. On *Mars*, a speed-up of 18.5 over the sequential implementation can be achieved with the hybrid k-priority queue on 40 threads (there is some slowdown when moving from 40 to 80 threads, though), and on *Saturn* the speed-up is 21.3 on 48 threads.

What can be seen is that, if well optimized, strategies can help to improve both performance and scalability. The implementation of graph bipartitioning that uses our new model (CLSM) for strategies is still a work in progress, and thus it is visibly slower than the well-tuned old implementations based on strategies for small numbers of threads. It nonetheless already provides a significant performance advantage over the parallel implementation without strategies. With higher numbers of threads this implementation is even able to match the performance of the other strategy.

In Section 7.8.5, we show that there are large differences in the overheads of the data structures used by schedulers to store tasks, with centralized k having the highest overhead. What is interesting to see in Figure 7.7 is that central k, hybrid k and priority work-stealing, which are all based on the same strategy, have roughly the same performance. This shows that due to conversion of task spawns to function calls, the actual number of tasks stored in

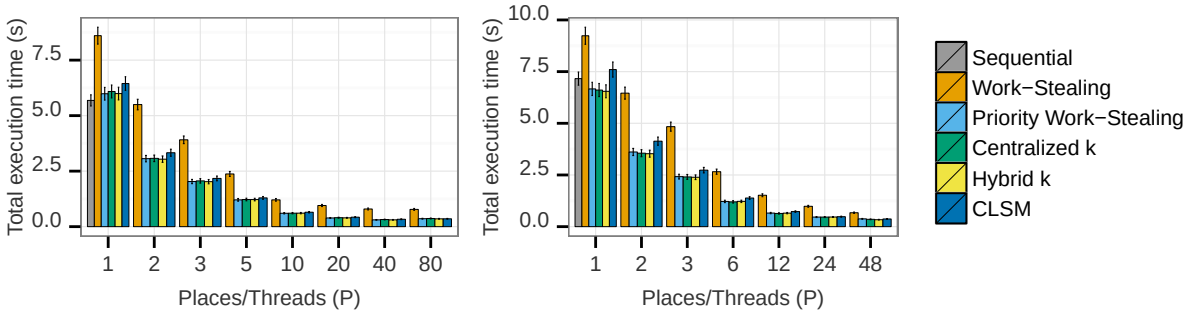


Figure 7.7: Average execution time of graph bipartitioning on dense random graphs *Mars* (left) and *Saturn* (right). ($n = 35$, $p = 90\%$, Intel compiler)

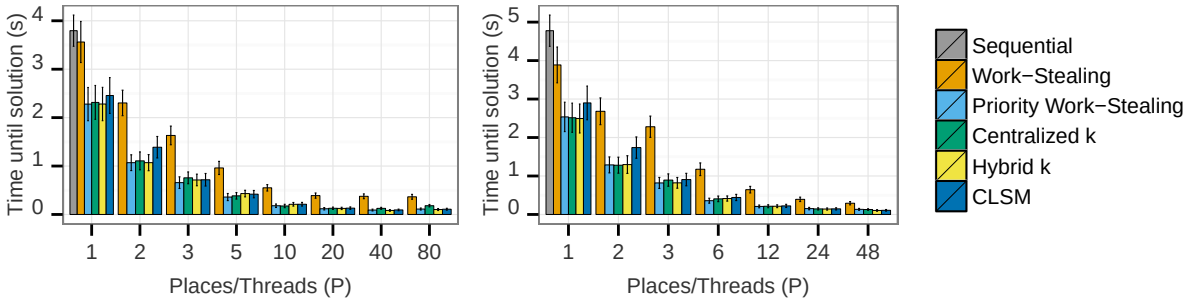


Figure 7.8: Average time until the best solution was found for graph bipartitioning on dense random graphs *Mars* (left) and *Saturn* (right). ($n = 35$, $p = 90\%$, Intel compiler)

any of the task queues is small enough so that the overhead becomes negligible. The good scalability also shows that this optimization does not have a visible impact on parallelism.

While converting task spawns to function calls to reduce scheduler overhead is an optimization that can help throughout the execution of the algorithm, prioritization of tasks can help reduce total work, by finding good solutions faster. As soon as the best solution was found, nothing can be gained by prioritization of tasks. To better show the gains from prioritization, in addition to the total execution time we also measured the last time the global solution was updated, which is the time at which the best solution was found. This metric has a high variance, since there is an element of chance involved, but it gives an insight into the usefulness of a heuristic for prioritization nonetheless.

The results of these measurements are shown in Figure 7.8. What can be seen is that the heuristics provide significant gains due to strategies. Since these gains are higher than for the total execution time, this effect can not be explained by the conversion of task spawns to function calls, and must therefore come from the prioritization of tasks.

For comparison, we now also show results for sparse graphs. Total execution time is shown in Figure 7.9, the time until the optimal solution was found is shown in Figure 7.10. The results are fairly similar to the results for dense graphs. We tested graph bipartitioning for several other graph sizes and densities, with similar results (not shown here).

7.5 Quicksort

Quicksort is a textbook algorithm commonly used to demonstrate divide-and-conquer algorithms. As with many divide-and-conquer algorithms parallelization of this algorithm is

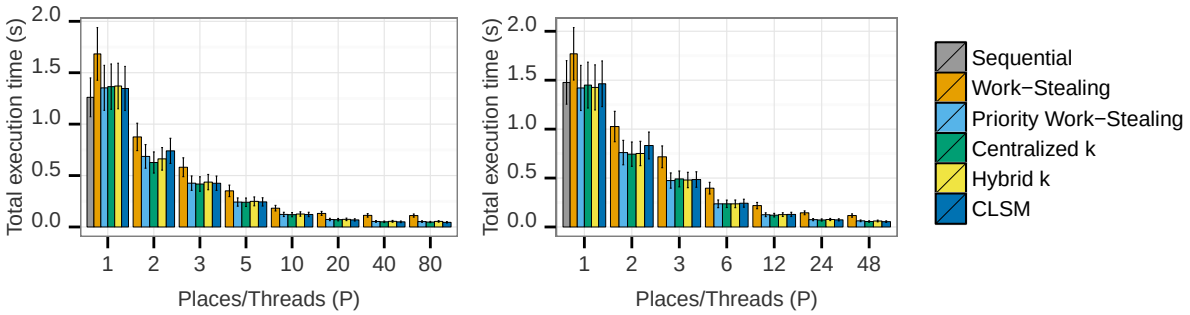


Figure 7.9: Average execution time of graph bipartitioning on sparse random graphs *Mars* (left) and *Saturn* (right). ($n = 59$, $p = 10\%$, Intel compiler)

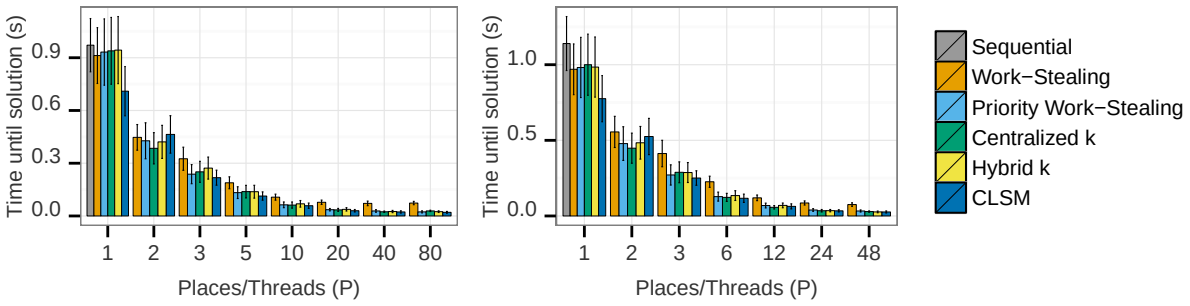


Figure 7.10: Average time until the best solution was found for graph bipartitioning on sparse random graphs *Mars* (left) and *Saturn* (right). ($n = 59$, $p = 10\%$, Intel compiler)

straightforward: recursive calls to quicksort are independent of each other and can therefore be executed concurrently. This makes quicksort a convenient textbook example for task parallel programming as well, as seen in Section 2.2.

In spite of its popularity, the straightforward parallelization of quicksort has limited scalability, a fact which is often overlooked. This comes from the fact that the partitioning step is not parallelized, thus creating an expected critical path of length $O(n)$. Since a sequential Quicksort execution has expected time $O(n \log n)$, the maximum achievable speedup is $O(\log n)$, which is not sufficient for weak scaling. Halstead [72] first presented a task parallel implementation of quicksort that also parallelizes the partitioning step, but with an algorithm that does not work in place. Tsigas and Zhang [135] provided an in-place algorithm for quicksort with a parallel partitioning step, but the algorithm assumes a data parallel programming model, and cannot be easily combined with standard task-parallel programming models. Using mixed-mode scheduling, we were able to implement a variant of the Tsigas and Zhang quicksort algorithm for task parallel programming models [146, 147].

7.5.1 Aim of this benchmark

Due to its popularity as a textbook example, quicksort implementations are available for most task-parallel programming models. This allows for easy performance comparisons between different such programming models.

We use quicksort to demonstrate the usefulness of mixed-mode scheduling for this application. While the improved performance is not due to the scheduler, but due to the algorithm that exposes more parallelism, the lack of task-parallel in-place algorithms for parallel partitioning gives a good argument for mixed-mode scheduling.

In addition, we have implemented an algorithm that uses strategies to always explore the smaller sub-problem first and leaving the larger sub-problem to be stolen.

7.5.2 Implementation

Our parallel implementation of quicksort, which is shown in Listing 7.5, is a typical textbook implementation, which partitions the data and recursively spawns more work until a cutoff, at which it switches to a sequential algorithm.

Listing 7.5 Implementation of the quicksort task in Pheet. (simplified)

```
virtual void operator()()
{
  if(length <= CUTOFF)
    sequential_quicksort(data, length);

  unsigned int * middle = std::partition(data, data + length - 1,
    std::bind2nd(std::less<unsigned int>(), *(data + length - 1)));
  size_t pivot = middle - data;
  std::swap(*(data + length - 1), *middle); // move pivot to middle

  Pheet::spawn<Self>(data, pivot);
  Pheet::call<Self>(data + pivot + 1, length - pivot - 1);
}
```

7.5.3 Strategies

We have also implemented an algorithm based on strategies to see whether it is possible for a simple algorithm like quicksort to profit from strategies. Since the algorithm is executed sequentially after a cutoff there is not much to be gained from converting task spawns to function calls, therefore we focused on priority scheduling. Prioritization uses a simple scheme shown in Listing 7.6, where locally spawned tasks are always prioritized, with the task on the smallest subsequence always being executed first, and other threads stealing tasks operating on the longest subsequence.

Listing 7.6 Implementation of the prioritization function in the quicksort strategy in Pheet. (simplified)

```
bool prioritize(QuicksortStrategy& other)
{
  Place* p = Pheet::get_place();
  if(this->place == p) {
    if(other.place == p) {
      return length < other.length;
    }
    return true;
  } else if(other.place == p) {
    return false;
  }
  return length > other.length;
}
```

7.5.4 Mixed-mode parallel algorithm

The mixed-mode parallel exposes additional parallelism compared to the quicksort algorithm presented in the previous section, by executing the partitioning step in parallel in case of insufficient parallelism in the algorithm. The array is divided into N blocks of size B, and the root task is allowed to be executed by a team of size $T \leq \frac{N}{8} + 1$. After partitioning, the team is divided up between the sub-tasks as shown in Listing 7.7. Since the current Pheet scheduler is allowed to use more than the requested number of threads to execute a mixed-mode parallel task whenever these threads would otherwise be idle, an additional check is made to ensure team sizes do not grow beyond the number of blocks in a sub problem.

Listing 7.7 Implementation of the mixed-mode quicksort task in Pheet. (simplified)

```
virtual void operator()()
{
  procs_t team_size = Pheet::Environment::get_place()->get_team_size();
  if(team_size == 1) {
    // Switch to non mixed-mode algorithm
    Pheet::template call<Quicksort>(data, length);
    return;
  }

  // Run parallel partitioning algorithm for whole team
  partition();

  // Make sure all threads have finished partitioning
  barrier.barrier(0, team_size);

  // Maximum number of processors used for sub-teams
  procs_t team_max = std::min(team_size, ((length / BLOCK_SIZE) / 8) + 2);
  // Divide up team for the subtasks
  procs_t team_size1 = std::max((pivot * team_max) / length, 1);
  procs_t team_size2 = team_size - team_size1;

  // Spawn mixed-mode sub-tasks processors
  Pheet::spawn_nt<MMQuicksort>(team_size1, data, len);
  Pheet::spawn_nt<MMQuicksort>(team_size2, data + pivot + 1, length - pivot - 1);
}
```

For the partitioning step, a modified version of the parallel partitioning algorithm by Tsigas and Zhang is used [135]. In this algorithm, the array is divided into blocks, and each thread will grab one block from each side of the array and swap items between those blocks similar to a standard partitioning algorithm. After finishing to process a block, a thread will attempt to get another block from the same side. Our algorithm [147] differs from Tsigas and Zhang's algorithm in what happens once all blocks from one side have been grabbed. While in their algorithm one thread will process all remaining blocks sequentially, our algorithm has threads with larger thread-ids yield their blocks to threads with lower thread-ids.

7.5.5 Results

We run our sorting algorithms on arrays of 10^7 (32-bit) integers. We base our experiments on a uniformly random data distributions, and a gauss distribution for the arrays. Experi-

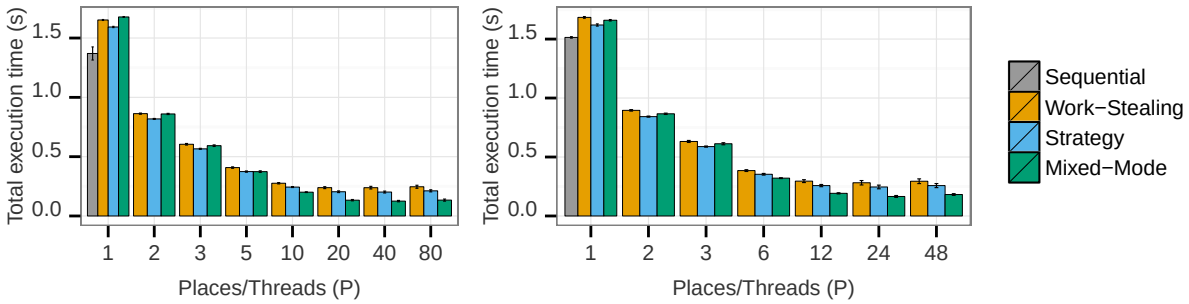


Figure 7.11: Average execution time of quicksort on the *random* data-set on *Mars* (left) and *Saturn* (right). ($n = 10^7$, Intel compiler)

ments for mixed-mode scheduling with additional data distributions can be found in previous work [146].

In our experiments we compare the following variants of graph bipartitioning:

Sequential A sequential implementation of quicksort.

Work-Stealing The parallel quicksort algorithm from Section 7.5.2 executed on a work-stealing scheduler.

Strategy The priority scheduled quicksort algorithm from Section 7.5.3, run on a scheduler that uses the concurrent LSM priority queue for maintaining the task execution order.

Mixed-Mode The mixed-mode parallel algorithm that uses relies on the parallel partitioning step.

Results on random arrays are shown in Figure 7.11. We have measured the performance both for the Intel compiler and GCC, but since the behaviour of the application is the same on both compilers, with only a small performance difference, we only show the results for the Intel compiler.

What can be seen is that the all parallel algorithms provide reasonable scalability on a single NUMA node, but cannot scale beyond that. This is expected, since this benchmark currently is not optimized for NUMA architectures, resulting in the whole array being located on a single NUMA node. Both the standard work-stealing algorithm and the algorithm based on strategies provide reasonable scalability, given the scalability bottleneck due to the sequential partitioning step. The prioritization of smaller sub-arrays due to strategies provides a small performance advantage over the standard work-stealing algorithm, due to better cache locality, regardless of the number of threads.

On small numbers of threads the mixed-mode parallel algorithm provides performance similar to the work-stealing algorithm, with a small overhead due to the higher scheduler overhead. The advantages of using a parallel partitioning step become apparent when scaling up the algorithm.

Performance for the gauss data distribution is shown in Figure 7.12. There is no visible difference in behaviour for to the random data distribution.

7.6 Prefix Sums

Prefix sums or *scan* is a commonly used algorithmic primitive, which is the base for many (parallel) algorithms [21]. The problem is for an array of numbers to calculate the prefix sum

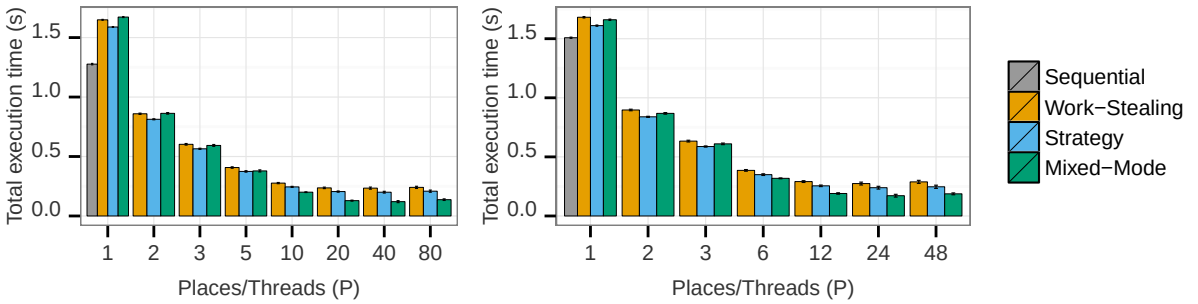
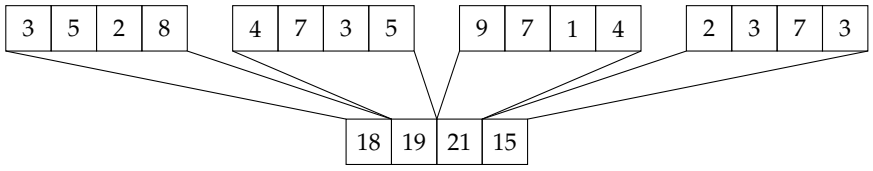
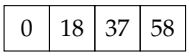


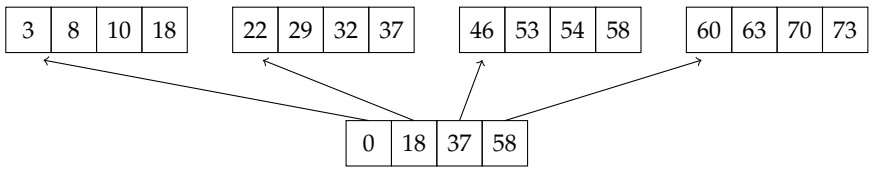
Figure 7.12: Average execution time of quicksort on the *gauss* data-set on *Mars* (left) and *Saturn* (right). ($n = 10^7$, Intel compiler)



(a) First pass: Calculating block sums



(b) Calculating exclusive prefix sums for auxiliary array



(c) Second phase: Calculating prefix sums per block with offset

Figure 7.13: A parallel prefix sums algorithm.

(the sum of all items before a certain item) for each item in the array. This can be easily done sequentially in a single pass, by linearly scanning through all items and updating the array (hence the synonym *scan*).

The sequential algorithm cannot be parallelized, though, due to data dependencies between iterations of the algorithm. Instead, the standard algorithm for parallel prefix sums [21] requires two passes on the data as shown in Figure 7.13. In the first pass (a), the data is split into multiple blocks, and the sum of all elements in a block is calculated and stored in an auxiliary array with one element per block. Next, the prefix sums are recursively calculated for the auxiliary array (b). Finally, in the second pass (c), the prefix sums are calculated for each block, with the offset being taken from the auxiliary array.

7.6.1 Aim of this benchmark

Prefix sums is a commonly used algorithm, and a basic primitive used for the implementation of many other algorithms. Prefix sums has a high ratio of data accesses with regard to computation, making it memory bandwidth bound on many current shared memory architectures. Vectorization allows to further improve this algorithm's performance, but puts an even higher strain on the memory bandwidth. This allows to explore the limits on scalability

due to memory bandwidth for different architectures.

Another point that makes prefix sums an interesting benchmark is that any parallel algorithm requires more work than the sequential algorithm [125]. This opens up possibilities for adaptive algorithms that will smoothly blend between the sequential and parallel algorithms depending on the available parallelism.

7.6.2 Basic implementation

Listing 7.8 Implementation of the prefix sums main task in Pheet.

```
virtual void operator()() {
  if(length > BlockSize) {
    size_t num_blocks = ((length - 1) / BlockSize) + 1;
    aligned_data<unsigned int, 64> auxiliary_data(num_blocks);

    // Calculate offsets
    Pheet::finish<OffsetTask>(data, auxiliary_data.ptr(), num_blocks, length);
    // Prefix sum on offsets
    Pheet::finish<ExclusivePrefixSums>(auxiliary_data.ptr(), num_blocks, pc);
    // Calculate local prefix sums based on offset
    Pheet::finish<LocalSumTask>(data, auxiliary_data.ptr(), num_blocks, length);
  } else {
    // Perform sequential prefix sums on block if only one block exists
    ...
  }
}
```

Listing 7.9 Implementation of the prefix sums `OffsetTask` in Pheet.

```
virtual void operator()() {
  if(blocks == 1) {
    // Sum up block
    unsigned int sum = 0;
    for(size_t i = 0; i < length; ++i) {
      sum += data[i];
    }
    auxiliary_data[0] = sum;
  } else {
    // Recursively spawn tasks
    size_t half = blocks >> 1;
    size_t half_l = half * BlockSize;

    Pheet::spawn<Self>(data + half_l, auxiliary_data + half, blocks - half,
      length - half_l);
    Pheet::call<Self>(data, auxiliary_data, half, half_l);
  }
}
```

Our basic algorithm for prefix sums is based on the two-pass algorithm described above. Listing 7.8 shows the function body of the main task in our implementation in Pheet. In the first phase, the array is split into equally sized blocks (which should be aligned to cache lines), and an auxiliary array is created, which will store the sums of elements in each block.

Then, the `OffsetTask` is called, which recursively spawns tasks, where each task sums up a single block and stores the value in the auxiliary array as shown in Listing 7.9. As soon as all `OffsetTasks` finished executing, exclusive prefix sums need to be created for the auxiliary array, which is done by a recursive call to the prefix sums algorithm.

Finally, as soon as the prefix sums of the block offsets are available, the prefix sums for each block can be calculated independently, since the auxiliary array contains the sums of all items before the first element of each block. This is done in the `LocalSumTask`, for which the function body of the task implementation is shown in Listing 7.10. Similar to the `OffsetTask`, the `LocalSumTask` recursively spawns more tasks until there is a single task available for each block.

Listing 7.10 Implementation of the prefix sums `LocalSumTask` in Pheet.

```
1 virtual void operator()() {
2   if(blocks == 1) {
3     // Sum is initialised to the sum of all items before the first element
4     // in the block
5     unsigned int sum = auxiliary_data[0];
6     // Sequentially calculate prefix sums for block
7     ...
8   }
9   else {
10    // Recursively spawn tasks
11    size_t half = blocks >> 1;
12    size_t half_l = half * BlockSize;
13
14    Pheet::spawn<Self>(data + half_l, auxiliary_data + half, blocks - half,
15      length - half_l);
16    Pheet::call<Self>(data, auxiliary_data, half, half_l);
17  }
18 }
```

7.6.3 Adaptive algorithm

The main drawback of the basic algorithm for prefix sums is that it always has to pay the overhead for parallelism by requiring two passes on the data, even if it is executed only by a single thread. We developed an adaptive algorithm that will smoothly blend the sequential and the parallel algorithm together to ensure that the overhead for parallelism is only paid in case there exists actual parallelism. The main idea behind the algorithm is to have a single thread process the blocks in sequential order as in the sequential algorithm (while still putting the sums of the blocks into the auxiliary array), until a block is encountered that has been processed by another thread. This way, a second pass is only required for all blocks that have been processed in parallel.

Our basic prefix sums algorithm can be easily adapted to this adaptive algorithm by modifying the main task and the `OffsetTask` of the original algorithm. The implementation of the main task is shown in Listing 7.11. The main difference to the original algorithm is that a counter variable is used to keep track of the sequentially processed blocks. Prefix sums for the auxiliary array are then only calculated for the last sequentially processed block and all blocks that were processed in parallel. The `LocalSumTask` is only called for all blocks that were processed in parallel and uses the same implementation as the original algorithm.

Listing 7.11 Implementation of the adaptive prefix sums main task in Pheet.

```
virtual void operator()() {
  if(length > BlockSize) {
    size_t num_blocks = ((length - 1) / BlockSize) + 1;
    aligned_data<unsigned int, 64> auxiliary_data(num_blocks);
    // Counts how many blocks have been processed sequentially
    std::atomic<size_t> sequential(0);

    // Calculate offsets
    Pheet::template finish<OffsetTask>(data, auxiliary_data.ptr(), num_blocks,
      length, 0, sequential);
    size_t seq = sequential.load(std::memory_order_relaxed);
    // Were any blocks executed in parallel?
    if(seq < num_blocks) {
      // Prefix sum on offsets
      Pheet::finish<ExclusivePrefixSums>(auxiliary_data.ptr() + seq - 1,
        num_blocks + 1 - seq, pc);

      // Calculate local prefix sums based on offset
      Pheet::finish<LocalSumTask>(data + (seq * BlockSize),
        auxiliary_data.ptr() + seq, num_blocks - seq, length - (seq * BlockSize));
    }
  }
  else {
    // Perform sequential prefix sums on block if only one block exists
    ...
  }
}
```

The modified version of the `OffsetTask` is shown in Listing 7.12. As in the basic implementation, this task will recursively spawn more instances of itself, until a task is available for each block. When processing a single block, the adaptive algorithm distinguishes between the sequential and the parallel case. If the `block_id` equals the number of sequentially processed block this means that the given block is also processed sequentially. In this case, the algorithm will retrieve the offset from the previous block out of the auxiliary array, and then calculate the prefix sums for the given block. Finally, when the prefix sums are calculated, the element in the auxiliary array corresponding to the given block is filled with the sum of all elements until the end of the given block. This includes all elements in the preceding blocks as well. As a last step, the counter for the sequentially processed blocks is incremented to signal that another block has been processed sequentially.

7.6.4 Strategies

One thing that makes the adaptive algorithm so useful is that it will even reduce the total amount of work if there is some parallelism available, since for all blocks up until the first block processed in parallel only a single pass is required. To maximize this effect, it is best if all threads that do not process blocks in a sequential order preferably process blocks as far away as possible from the sequentially processed blocks. The adaptive algorithm has already been written such that it can achieve such a behaviour up to a certain point on a work-stealing scheduler. When recursively splitting blocks, the algorithm will always first create the task for the right half of the array before recursively calling itself for the left half. This way blocks will

Listing 7.12 Implementation of the adaptive prefix sums `OffsetTask` in Pheet.

```
virtual void operator()() {
  if(blocks == 1) {
    // Have all blocks before this block been sequentially processed
    if(block_id == sequential.load(std::memory_order_relaxed)) {
      // Calculate offset
      unsigned int sum = (block_id == 0)?0:auxiliary_data[block_id - 1];
      // Sequential prefix sums
      ...
      // Store sum in auxiliary array
      auxiliary_data[block_id] = sum;
      // This block has been processed sequentially as well
      sequential.store(block_id + 1, std::memory_order_release);
    }
    else {
      // Sum up blocks, this is a block that is processed in parallel
      unsigned int sum = 0;
      for(size_t i = 0; i < length; ++i) {
        sum += data[i];
      }
      auxiliary_data[block_id] = sum;
    }
  }
  else {
    // Recursively spawn tasks
    size_t half = blocks >> 1;
    size_t half_l = half * BlockSize;

    Pheet::spawn<Self>(data + half_l, auxiliary_data, blocks - half,
      length - half_l, block_id + half, sequential);

    Pheet::call<Self>(data, auxiliary_data, half, half_l, block_id, sequential);
  }
}
```

be stored in a work-stealing deque in order from the rightmost range of blocks to the leftmost. Due to the first-in first-out behaviour of typical *steal* operations, a work-stealing scheduler will steal the rightmost range of blocks first, and only steal more blocks when all blocks have been processed. Nonetheless, the more threads are in the system, the more random the choice of block to process becomes, making the gains from the adaptive algorithm small.

In previous work [142, 143] we used scheduling strategies to ensure such an adaptive behaviour. For this work we were able to implement an adaptive algorithm without the use of strategies. To find out whether more performance can be gained by using strategies, we have also implemented a variant of the adaptive algorithm that relies on strategies.

7.6.5 Vectorization

In cases, where the memory bandwidth is not exhausted by the conventional algorithm, vectorization can help to further accelerate the algorithm. This can be easily done for the first phase, and this is done automatically by some compilers. The second phase, where prefix sums are calculated separately for each block, requires replacing the sequential prefix sums algorithm with a parallel one to make it vectorizable. In contrast to the coarse-grained parallel

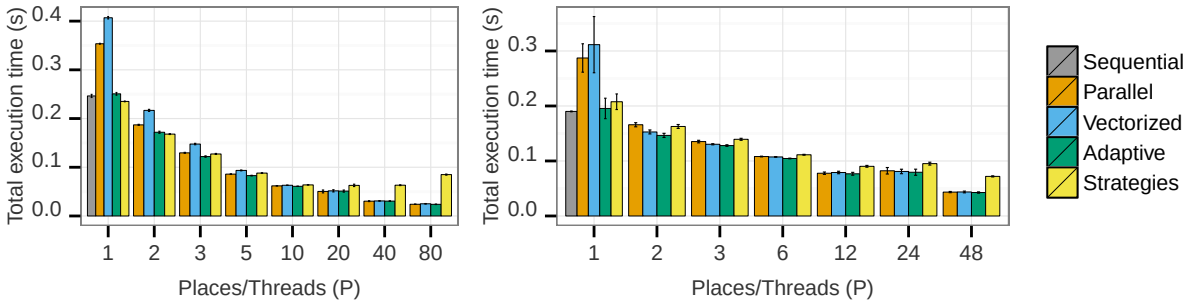


Figure 7.14: Average execution time of prefix sums on *Mars* (left) and *Saturn* (right). ($n = 10^8$, GCC)

algorithm, this fine-grained parallelization for vectorization works on subsequent elements in an array. For this, our algorithm performs element-wise shifts on the given vector and adds the shifted vector to the original one. The amount the vector is shifted is doubled on every step. We manually unrolled the loop performing those shifts for higher performance.

7.6.6 Results

We run our prefix sums algorithms on arrays of 10^8 (32-bit) integers. The numbers shown are mean values from 20 experiments. Since the Intel compiler generated significantly slower code on *Saturn* for the parallel algorithms, we show results for GCC.

In our experiments we compare the following variants of graph bipartitioning:

Sequential A sequential implementation of prefix sums.

Parallel Algorithm The basic parallel algorithm, run on a work-stealing scheduler

Adaptive The adaptive algorithm run on a work-stealing scheduler

Strategies The algorithm with strategies, run on a scheduler with a k-LSM priority queue and $k = 8$.

Results are shown in Figure 7.14. The overhead of the parallel algorithm over the sequential, due to the required second pass over the data, is clearly visible. Nonetheless, the overhead is not a factor of two, since the compiler is able to automatically vectorize the first phase of the algorithm. The gains from manually vectorizing the algorithm are small, and in some cases the vectorized code is even slower than the original code. This is different from our observations in previous work [145], where we achieved significant performance improvements due to vectorization. The main reason for this is that we now rely on a different parallel algorithm for parallel prefix sums, which is easier for compilers to automatically vectorize, thus achieving better performance by default.

The adaptive algorithm comes close to sequential performance, since it only performs a single pass over the data. On more than one thread, both the parallel and the adaptive algorithm scale well and are able to beat sequential performance. On *Saturn* some additional NUMA performance overhead can be seen for 24 threads. The algorithm is scalable to NUMA architectures nonetheless, and on 48 threads a significant speed-up can be seen. To achieve NUMA scalability it is necessary for the input data to be distributed over all NUMA nodes.

Even though the algorithm based on strategies tends to perform less work compared to the adaptive algorithm, the additional overhead for priority scheduling outweigh the gains.

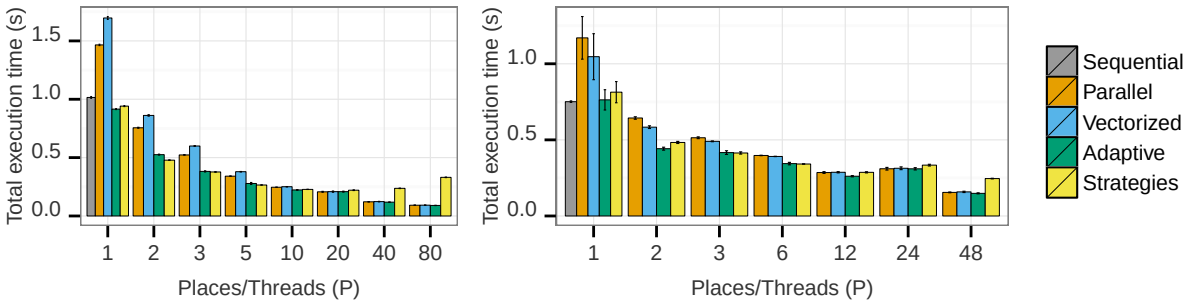


Figure 7.15: Average execution time of four instances of prefix sums on *Mars* (left) and *Saturn* (right). ($n = 10^8$, GCC)

Furthermore, scalability limits of the k-LSM priority queue result in significantly worse scalability of the algorithm. In previous work [142,143] we used our algorithm based on strategies as an argument for using strategies. Our new adaptive algorithm, makes the algorithm based on strategies obsolete, however.

The performance advantage of the adaptive algorithm over the parallel algorithm quickly decreases with higher numbers of threads, since the amount of blocks on which only a single pass is required becomes smaller. The good thing about the adaptive algorithm is that it does not provide a slowdown compared to the parallel algorithm, and thus can always be used as a replacement for it. Whenever the prefix sums algorithm cannot utilize the full parallelism of the system, either due to other algorithms running on the same scheduler, or due to other applications running in a multiprogrammed environment, the adaptive algorithm can automatically reduce the amount of work. This is shown in Figure 7.15, where four instances of the prefix sums algorithm are run at the same time. As can be seen, the performance advantage of using the adaptive algorithm persists to larger numbers of threads.

7.6.7 Future work

Our current algorithm implementations do not take into account NUMA effects, where communication costs may differ depending on whether a processor accesses a block that is stored locally or at a different NUMA node. By actively querying such NUMA locality information for each block in the array, it should be possible to achieve a NUMA-aware schedule, which reduces communication costs in-between NUMA nodes and therefore increases scalability of the algorithm.

7.7 Triangle Strip Generation

The generation of triangle strips to represent 3D models is a common optimization to improve rendering performance. Instead of providing individual triangles to the rendering hardware, adjacent triangles are combined into strips, where vertices appearing in two adjacent triangles only have to be transmitted once. This lowers the number of vertices from $3n$ to $n + 2$. In the optimal case one would need only one triangle strip to represent the entire model. Finding such a strip can however be reduced to the NP-complete problem of finding an Hamiltonian path in a dual graph of the model, where nodes represent triangles and edges represent adjacent triangles. This problem is thus best solved using heuristics.

7.7.1 Aim of this benchmark

The main reason for including this benchmark into Pheet is to show that strategies can lead to qualitatively better results, as well as performance improvements. In this case, the quality can be measured by the number of generated triangle strips (the less the better) and the length of these strips. To achieve a higher quality, strategies employ a heuristic for prioritization that preferably picks nodes with a low number of neighbours.

7.7.2 Implementation

For this benchmark we have used a version of the so called SGI algorithm [55]. A node on the graph is randomly picked from the set of nodes with the lowest degree. This is done to minimize the number of single triangle strips. A strip is then built by adding neighbouring nodes to the strip at both ends. Priority is given to nodes with a low number of neighbours. When no more nodes can be added to the strip, a new node is randomly picked and a new strip is started. This is repeated until all nodes are part of a strip.

The benchmark uses two types of tasks. The first is the `StartTask`, which at spawn-time is assigned a pointer to a possible node to start a triangle strip from. The strategy for this type of task stores the number of neighbouring nodes that are not part of a strip, and it uses that to prioritize tasks. Generating a strip is a relatively quick operation, so it is suitable for spawn-to-call transformation and is thus given a low transitive weight. Spawning a `StartTask` for every node in the graph would be wasteful, as many will be part of other triangle strips and thus not eligible to start a new strip from. Instead we provide a second type of task, the `SpawnTask`, to gradually spawn new `StartTasks` when needed and only for eligible start nodes. This task spawns new `StartTasks` for a given interval of nodes. It does not allow the task to be transformed into a call. The two strategies are composed by a common parent strategy that gives priority to `SpawnTasks` when stealing and to `StartTasks` when working locally.

7.7.3 Results

The data used for the benchmark is the 3D model Lucy from the Stanford 3D Scanning Repository[1]. The model consists of around 28 million triangles.

In our experiments we compare the following variants of triangle strip generation:

Standard A parallel algorithm without strategies.

Strategies A variant with strategies based on priority work-stealing.

The triangle strip implementation in Pheet is currently not maintained, and thus no implementations on newer schedulers are available. We decided to show the results nonetheless, since it is an interesting example that shows that strategies can be used to improve the quality of results as well.

Execution times for both *Mars* and *Saturn* are shown in Figure 7.16. For *Saturn*, the implementation based on strategies is consistently faster compared to the standard parallel implementation. This confirms results from previous work [142, 143]. On *Mars*, a scalability issue reproducibly occurs for less than 20 threads. So far, we were not able track the cause of this issue, but we assume that this comes from a bug in the implementation of the algorithm with strategies.

[1] `http://graphics.stanford.edu/data/3Dscanrep`

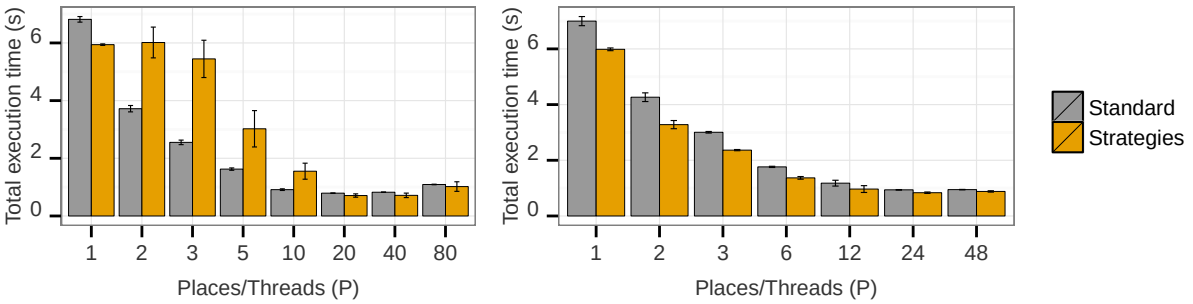


Figure 7.16: Average execution time of triangle strip generation on *Mars* (left) and *Saturn* (right). (Model: Lucy, GCC)

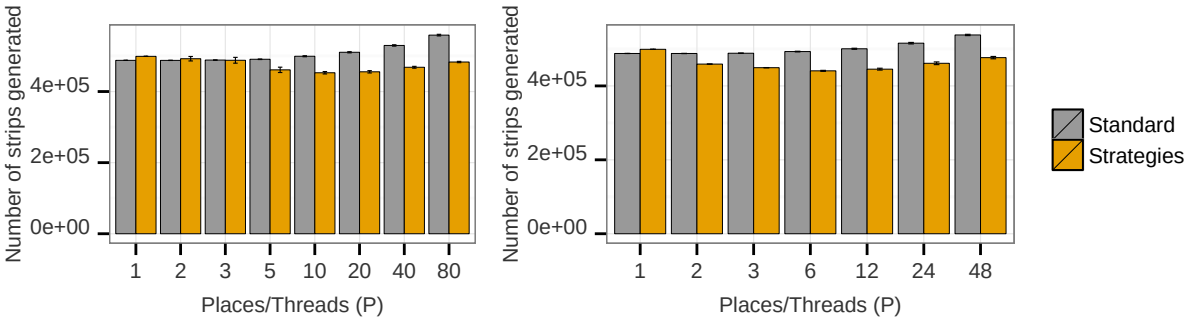


Figure 7.17: Number of triangle strips generation (lower is better) on *Mars* (left) and *Saturn* (right). (Model: Lucy, GCC)

The objective of a good heuristic for triangle strip generation is to reduce the number of triangle strips generated. Thus, the number of triangle strips generated can be used as a metric for the quality of the result. We show the number of strips generated in Figure 7.17. What can be seen is that for the standard parallel algorithm the number of strips generated increases with the number of threads. For the algorithm based on strategies, on the other hand, less triangle strips are generated for higher numbers of threads. It is interesting to note, that the number of strips generated in parallel is even less than for the sequential execution in many cases.

7.8 Single-source Shortest Paths

Single-source shortest paths (SSSP) is another classic algorithmic problem that has been well studied both in the sequential and the parallel context [7, 31, 44, 73, 104]. To our knowledge, the first publicly known work-efficient (in the average case) parallel algorithm for the single source shortest path problem was the Δ-stepping algorithm by Meyer and Sanders [104].

7.8.1 Aim of this benchmark

We use the single-source shortest paths problem to demonstrate how the priority task scheduling programming model can enable programmers to implement a parallel algorithm for single-source shortest paths with only a few lines of code. In addition, we made use of the fact that SSSP has been extensively studied and is well understood to analyse what guarantees concerning the execution order of tasks a scheduling system needs to provide in order

to be useful for the implementation of the SSSP algorithm. While our parallel algorithm for SSSP is certainly not the fastest or most scalable, it provides valuable insights about ordering semantics of priority queues. The knowledge gained from this analysis was used for the implementation of the k-LSM priority queue presented in Section 5.8.18, and we expect to gain additional insights leading to even more efficient priority queues in future work.

7.8.2 The parallel algorithm

We base our implementation on a simple parallelization of Dijkstra's algorithm. Dijkstra's algorithm maintains a tentative distance value for each node in the graph. At each iteration, a node relaxation is performed, where the tentative distance values of the neighboring nodes are updated if the path through the relaxed node is shorter. At termination, the distance from the source node is available for each node. A priority queue is used to decide the order in which nodes are relaxed; the priority ordering guarantees that each node is relaxed exactly once.

Our parallel version relaxes multiple nodes in parallel. Due to the parallelization some node relaxations might be performed prematurely, when a node is not yet *settled*, which means that its distance value is not final. These nodes will have to be re-relaxed when their distance values are updated. Premature relaxations are therefore *useless work*.

7.8.3 Implementation

Listing 7.13 Implementation of the SSSP main task in Pheet.

```
1  virtual void operator()() {
2    // Retrieve distance value
3    size_t d = graph[node].distance.load(std::memory_order_relaxed);
4
5    // Explore all outgoing edges
6    for(size_t i = 0; i < graph[node].num_edges; ++i) {
7      // Get id of target node
8      size_t target_id = graph[node].edges[i].target;
9      // Retrieve tentative distance value for node
10     size_t old_d = graph[target].distance.load(std::memory_order_relaxed);
11     // Calculate distance when going through this node
12     size_t new_d = d + graph[node].edges[i].weight;
13     // Update tentative distance if a better distance value was found
14     while(old_d > new_d) {
15       if(graph[target].distance.compare_exchange_strong(old_d, new_d,
16           std::memory_order_relaxed)) {
17         // Spawn task for updated distance value
18         Pheet::spawn_s<Self>(
19               Strategy(new_d, &graph[target].distance),
20               graph, target, new_d, pc);
21         break;
22       }
23     }
24   }
25 }
```

In our parallel implementation, which is presented in Listing 7.13, each node that has to be relaxed corresponds to a task in the scheduling system. For the sake of comparability to other

works on single-source shortest paths, we will use the terms *node* and *relax* instead of *task* and *execute* throughout this section. Whenever the tentative distance value of a node is updated, a new task is spawned and prioritized based on the tentative distance value. This is similar to Dijkstra's algorithm, which utilizes a priority queue storing nodes ordered by tentative distance value to determine the next node to relax. In our case, the part of the priority queue is taken over by the scheduling system.

We diverge from Dijkstra's algorithm whenever a better distance value is found for an active node in the priority queue. Instead of updating the priority using a *decrease key* operation, we reinsert the node into the priority queue. This is necessary due to the fact that our current schedulers do not allow changing the priority of a task once it has been submitted to the scheduling system. The previous instance of the same node will be recognized by the scheduling system as a *dead task* (see also Section 2.7.4), allowing the priority queue to get rid of the node at its earliest convenience. All this is achieved using a *scheduling strategy* (see also Section 2.7) for which the implementation is shown in Listing 7.14. For better readability, boilerplate functions and type definitions have been omitted in the listing.

Listing 7.14 Implementation of the SSSP strategy in Pheet. (simplified)

```
1  class Strategy2SsspStrategy : public BaseStrategy {
2  public:
3    // Inform scheduling system about the preferred priority queue implementation
4    typedef PriorityQueueImplementation TaskStorage;
5
6    Strategy2SsspStrategy(size_t stored_distance,
7      std::atomic<size_t>& current_distance)
8    : stored_distance(stored_distance), current_distance(&current_distance) {}
9
10   bool prioritize(Self& other) {
11     // Tasks with smaller distance execute first
12     return distance < other.distance;
13   }
14
15   bool dead_task() {
16     // A task is recognized as dead if the tentative distance has been updated
17     // since it was created
18     return current_distance->load(std::memory_order_relaxed) < stored_distance;
19   }
20 private:
21   size_t stored_distance;
22   std::atomic<size_t>* current_distance;
23 };
```

7.8.4 Theoretical analysis

In previous work [144, 148] we have analysed the SSSP algorithm to gain an understanding of the requirements on the ordering semantics for concurrent priority queues. The analysis is based on a simplified model of task-parallel computations: execution occurs in temporal phases and, in each phase, up to P nodes are selected from a priority queue and relaxed in parallel. Our analysis is restricted to Erdős-Rényi random graphs [54, 64].

The goal of this analysis was to show that, for quantitatively relaxed priority queues, an upper bound on the amount of *useless work* generated can be given, and that this amount is small compared to useful work. In addition, we hoped to gain an insight into the exact

semantics of quantitative relaxation required to achieve such bounds. One important result of this analysis was that, to achieve reasonable bounds, it is sufficient to provide structural ρ-relaxation guarantees (Section 5.1.3). This opened up the possibility to implement more scalable ρ-relaxed priority queue which are not required to provide temporal ρ-relaxation guarantees. One result coming out of this is the new k-LSM priority queue sketched in Section 5.8.18. While this priority queue is still a work in progress, we were already able to use it for our experiments in this section.

The full analysis of the SSSP algorithm can be found in Appendix 7.A.

7.8.5 Results

We give averages over executions on these 20 graphs. Since no visible difference exists between the results obtained with GCC and the Intel compiler, we only show results obtained with the Intel compiler.

The following variants of SSSP are shown:

Sequential A sequential implementation of Dijkstra's algorithm for SSSP that relies on a binary heap as priority queue.

Priority Work-Stealing Our parallel SSSP algorithm run on our priority work-stealing scheduler.

Centralized k Our parallel SSSP algorithm run on a scheduler that uses the centralized k-priority queue as task queue.

Hybrid k Our parallel SSSP algorithm run on a scheduler that uses the hybrid k-priority queue as task queue.

CLSM Our parallel SSSP algorithm run on a scheduler that uses the concurrent LSM priority queue as task queue.

k-LSM Our parallel SSSP algorithm run on a scheduler that uses the k-LSM priority queue, a ρ-relaxed version of the concurrent LSM priority queue as task queue.

For our first set of experiments we fixed the parameter k for the hybrid k-priority queue and the k-LSM priority queue to 1024. For the centralized k-priority queue we used $k = 512$, since it does not support larger values for k. Both the concurrent LSM and k-LSM priority queues are still work in progress, and we are aware of some bottlenecks in the current implementation that limit scalability. While the results shown for these priority queues in this section already look promising, we expect even better results to be achievable in the future.

To gain an insight into the useless work performed by the parallel algorithm, we have counted the number of node relaxations executed by the SSSP benchmark. In a sequential execution, where always an active node with minimal distance label is relaxed next, the number of node relaxations will always equal the number of nodes in the graph, since each node is relaxed exactly once. All additional node relaxations performed by the parallel algorithm are therefore *useless work*.

This can be seen in Figure 7.18. On a single processor, all variants perform exactly 10000 node relaxations, which is the number of nodes in the graph. With rising number of threads, the executions relying on priority queues with only local ordering semantics (Priority Work-Stealing and CLSM) perform a high amount of useless work. CLSM performs less node relaxations than priority work-stealing due to the use of spying, instead of stealing.

The ρ-relaxed priority queues perform significantly less useless work. For the k-LSM priority queue, which provides structural ρ-relaxation with $\rho = k(P-1)$ the amount of useless

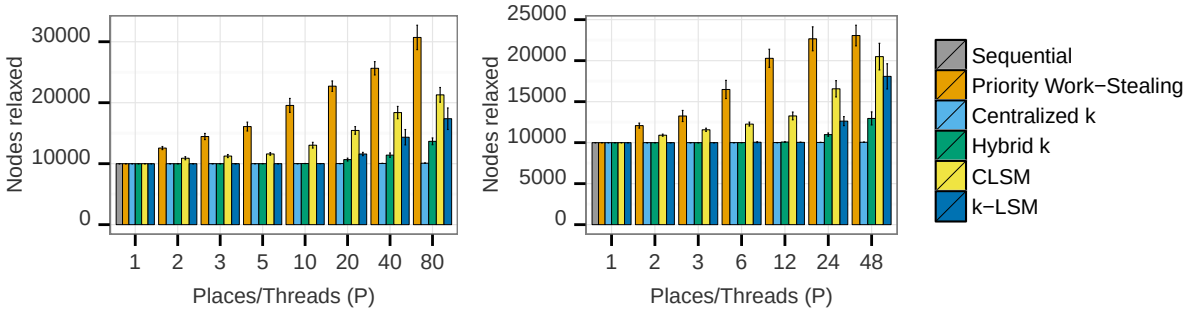


Figure 7.18: Node relaxations performed by SSSP on dense random graphs on *Mars* (left) and *Saturn* (right). ($n = 10^5$, $p = 50\%$, Intel compiler)

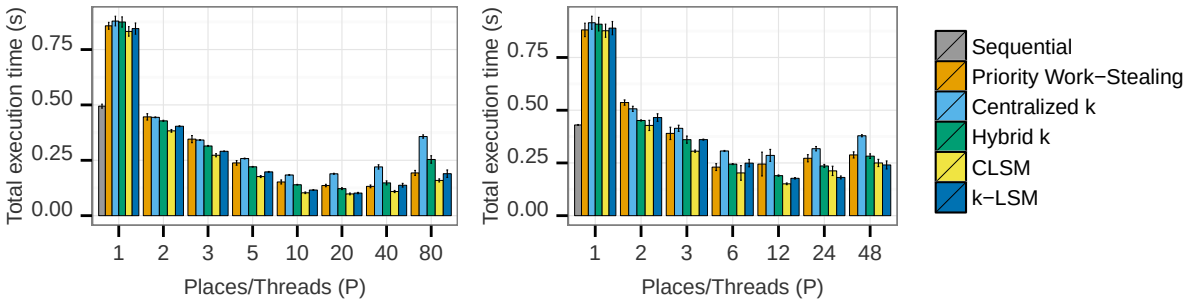


Figure 7.19: Average execution time of SSSP on dense random graphs on *Mars* (left) and *Saturn* (right). ($n = 10^5$, $p = 50\%$, Intel compiler)

work still increases visibly with rising numbers of threads. This can be expected since a larger number of threads allows more tasks to be skipped. Slightly better results for the same k can be achieved by the hybrid k-priority queue, since it provides temporal ρ-relaxation in addition, but the benefits are small in comparison to the additional synchronization overhead. The smallest amount of useless work is performed by the centralized k-priority queue. Even with 80 threads on *Mars* it only performs 80 useless node relaxations.

Although ρ-relaxed priority queues perform significantly less work, the actual performance advantage is much smaller. Execution time for all variants is shown in Figure 7.19. What can be seen is the high overhead of the parallel algorithm on all schedulers compared to a sequential execution. This can be explained by the small granularity of tasks, and the reinsertion of tasks for a specific node whenever a new distance label was found. Nonetheless, some speedup can be achieved with two or more threads on *Mars* and three threads on *Saturn*. No scalability can be achieved when using more than one NUMA node, and even some slowdown can be seen. The current implementation stores the whole graph on a single NUMA node. We tried an implementation that distributes the graph to all NUMA nodes, but it had even worse scalability, and thus the results shown here do not take this into account.

Even though the centralized k-priority queue performs almost no useless work, its limited scalability makes it impractical for such a fine-grained algorithm. The other ρ-relaxed algorithms show better scalability, but even there the performance advantage over priority work-stealing is limited, even though priority work-stealing performs significantly more work. The best performance is achieved by the concurrent LSM though, which even outperforms its ρ-relaxed counterpart.

To gain better insight into the influence of the parameter k on performance and guaran-

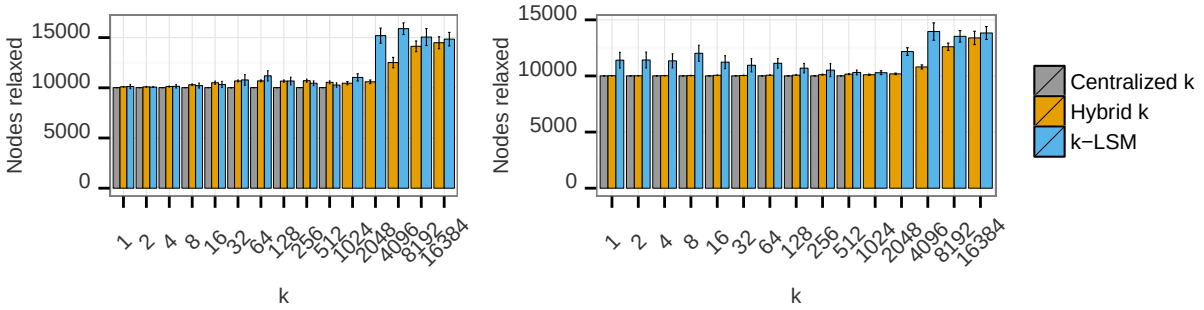


Figure 7.20: Node relaxations performed by SSSP on dense random graphs for varying k on *Mars* (left, $P = 20$) and *Saturn* (right, $P = 12$). ($n = 10^5$, $p = 50\%$, Intel compiler)

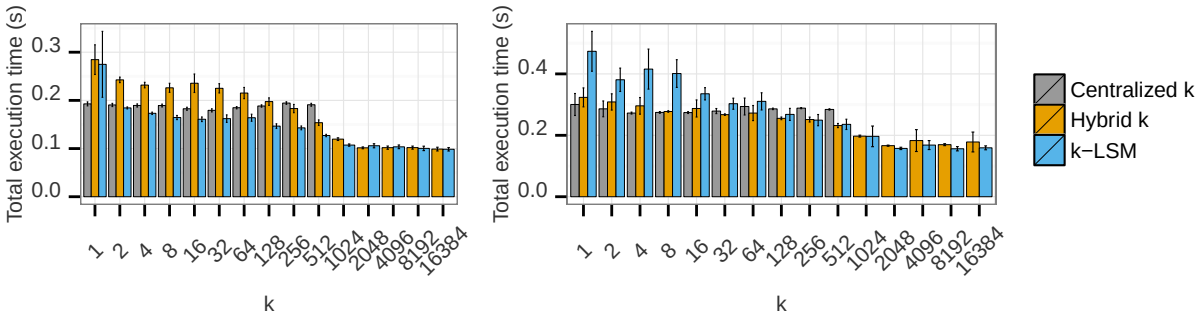


Figure 7.21: Average execution time of SSSP on dense random graphs for varying k on *Mars* (left, $P = 20$) and *Saturn* (right, $P = 12$). ($n = 10^5$, $p = 50\%$, Intel compiler)

tees provided by the ρ-relaxed priority queue, we also compare different values of k. Since the parallel SSSP algorithm does not scale to multiple NUMA nodes, we have restricted our experiments to a single NUMA node ($P = 20$ on *Mars* and $P = 12$ on *Saturn*).

The number of node relaxations for varying k are shown in Figure 7.20. As can be seen, the amount of additional useless work created stays small for all variants for $k \leq 1024$. For larger k, the amount of useless work increases up until a certain point where both the hybrid k-priority queue and the k-LSM priority queue keep all their work locally, and thus have behaviour similar to the concurrent LSM. The amount of useless work created by the k-LSM priority queue increases more abruptly than for the hybrid k-priority queue since it only fulfils structural ρ-relaxation requirements.

The execution times for varying k are presented in Figure 7.21. Both the hybrid k-priority queue and the k-LSM priority queue have worse performance than the centralized k-priority queue for small k, but allow for larger k to be used, which can help to improve performance and scalability. For the centralized k-priority queue performance peaks around $k = 32$. The small, but visible slowdown can be explained by the linear search through k elements which is required in the worst case for every insertion.

Notes on the experiments

The priority queues and the SSSP algorithm are topics of current research, and the experiments here present a snapshot of this research. Due to this, some results that are shown here differ from results presented previously [144, 148]. Since this previous work some performance regressions have occurred, especially with regard to scalability on NUMA systems, and it is

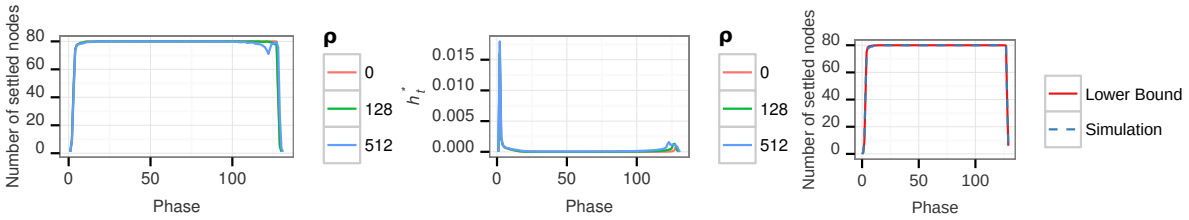


Figure 7.22: From left to right: nodes settled per phase; difference between biggest and smallest tentative distance of nodes relaxed per phase; comparison between the theoretical bound and the simulation. ($n = 10000$, $P = 80$, $p = 50\%$)

not yet clear what has caused these regressions.

The performance of the centralized k-priority queue for small k is a positive surprise, on the other hand. Recent improvements to this priority queue are most likely the cause of this change, and we are still investigating the implications of those changes.

7.8.6 Simulation

We have used a simulator to bridge between the findings of our theoretical model and the experimental results. While we developed our theoretical model, the simulator helped understand why ρ-relaxation gives such strong guarantees. The simulator uses the phase-wise execution model used in the theoretical analysis and allows us to vary the parameters P and ρ. The simulator stores all active nodes in a single array sorted by distance value. Execution proceeds in phases, where in each phase the first P nodes from the array are relaxed. At the end of each phase the array is updated with all new active nodes.

If $\rho > 0$, newly created active nodes are marked with a sequence id. To ensure randomness, nodes created in a single phase are shuffled first before assigning sequence id's. The nodes with the ρ highest sequence id's are stored separately from the sorted array of nodes. These nodes represent the nodes that might be ignored due to the ρ-relaxation. An exception is made if a node has the lowest distance value of all nodes. This node is guaranteed to be relaxed in the next phase, and is therefore added to the array of active nodes. A deterministic tie-breaking scheme is used to ensure that only one node has the lowest distance value of all at any time. In case that less than P nodes are available in the array, a random selection of all other active nodes is relaxed by the other places.

Simulation results

We ran our simulator in a setting that closely resembles the setup used in our experiments. We use exactly the same 20 random graphs used in the experiments and report the mean. The number of places, P, is set to 80, which corresponds to the 80 cores of the machine used in our experiments. We use three values for ρ: 0, which represents an ideal priority data structure, 128 and 512.

The first graph in Figure 7.22 depicts the number of nodes settled in each phase throughout the simulation. It can be seen that for most of the execution almost all nodes that are relaxed are already settled. Non-settled nodes are only encountered in the first phases. For higher ρ some variation can also be observed towards the end when a significant amount of nodes is not visible to all places. Throughout most of the execution almost all of the nodes that are relaxed are already settled.

The middle graph in Figure 7.22 shows h_t^*, the difference between the largest and smallest distance value of nodes relaxed in each phase. After only a few iterations, all of the nodes

that are relaxed have distance values close to each other, and the distance values only grow a bit at the end of the execution, a bit more with higher ρ. It is easy to see the close relationship between distance values and nodes settled per phase.

Finally, the last graph in Figure 7.22 gives a comparison between the theoretical lower bound and the number of settled nodes in the simulation. It can be seen that the calculated theoretical lower bound on the number of settled nodes, and the number of nodes settled in the simulation are very close.

7.9 Future Work

The set of Pheet benchmarks is in constant flux, with new benchmarks being added if needed, and redundant or unmaintained benchmarks being removed. In the following we explain the areas in which we plan to expand the Pheet benchmarks in the future.

One important area is the comparability to other task scheduling frameworks. For this we plan to port and maintain a subset of the Pheet benchmarks to the commonly used task scheduling frameworks Intel CilkPlus, Intel Threading Building Blocks (TBB) and OpenMP. This will improve comparability of results obtained with Pheet, and will help to expose any performance and scalability bottlenecks of Pheet.

Another topic that we plan to improve upon is the scalability of our memory-bound benchmarks in NUMA environments. For this, we intend to utilize priority scheduling to ensure that each worker-thread preferable executes tasks operating on data stored in the same NUMA node.

Finally, we plan to add benchmarks that help us to further expand and evaluate the Pheet framework. We are currently working on a benchmark for the *multi-criteria shortest path problem* [119], which we see as an interesting application of priority scheduling. We expect that better scalability for this algorithm can be achieved by using a concurrent pareto priority queue instead of a normal priority queue in the scheduler. For better evaluation of the container data structures used inside the Pheet framework (presented in Chapter 5) we also plan to implement benchmarks that allow to test their throughput directly, outside of a task scheduling context. This will allow to gain more insight into the scalability bottlenecks of these containers.

7.A Appendix: Theoretical Analysis of the SSSP Algorithm

The goal of this analysis is to show that, using ρ-relaxed priority queues, the amount of useless work generated is small compared to the actual work, and that bounds can be given on the amount of useless work generated. This analysis has been published in previous work [144].

For the theoretical analysis we use a simplified model of task-parallel computations: the system operates on a global pool of nodes (tasks), which are ordered by their tentative distance value. Execution occurs in temporal phases and, in each phase, up to P nodes with the lowest tentative distance values are relaxed. We assume an *ideal priority queue*, in which all nodes are visible to all places at the beginning of each phase. We are interested in upper bounding the amount of useless work that is performed during each phase. Similar bounds have previously been obtained for Δ-stepping and other SSSP algorithms [103, 104].

7.A.1 Formal model

We are given an undirected graph $G = (V, E)$ (with $n = |V|$ and $m = |E|$), a source node $s \in V$ and a positive weight function $\lambda : E \to \mathbb{R}^+$. For each temporal step t we maintain a partition of V into two subsets: $V = A_t \cup B_t$, of sizes α_t and β_t ($\forall t\ \alpha_t + \beta_t = n$). The set

$A_t = \{a_t(1), a_t(2), \ldots, a_t(\alpha_t)\}$ contains the active nodes, $B_t = \{b_t(1), \ldots, b_t(\beta_t)\}$ the inactive nodes. For each node $v \in V$ we also keep a tentative distance $\delta_t(v) \in \mathbb{R} \cup \{\infty\}$. Let $d_t(i) = \delta_t(a_t(i))$, we assume the nodes in A_t to be ordered by d_t, with ties broken arbitrarily, i.e., $\forall i \in \{1, \alpha - 1\}\ d_t(i) \leq d_t(i+1)$. Initially ($t = 0$) we have $A_0 = \{s\}$, $B_t = V \setminus \{s\}$, $\delta_0(s) = 0$ and $\delta_0(v \neq s) = \infty$. In each phase (up to) P active nodes $\Phi_t = \{a_t(1), \ldots, a_t(P)\}$ with lowest d_t are selected and relaxed, so that at the end of the phase the tentative distance of a generic node $w \in V$ is

$$\delta_{t+1}(w) = \min\left\{\delta_t(w), \min_{v \in \Phi_t}\{\delta_t(v) + \lambda(v, w)\}\right\} \ .$$

Any node (whether active or inactive) which had its tentative distance updated is moved into A_{t+1}, relaxed nodes which were not updated are moved into B_{t+1}, all the other nodes remain in their former sets for the next time phase. The algorithm terminates, at some time $\tau < n$, when there are no more active nodes, i.e., $A_\tau = \emptyset$ and $B_\tau = V$, with the nodes reachable from s having a finite distance.

We restrict our analysis to Erdős-Rényi random graphs [54, 64] of parameters n and p, i.e., graphs with n nodes, for which each of the $\binom{n}{2}$ possible edges has independent probability p to occur. Furthermore, we assign, independently for each edge, a weight uniformly distributed between 0 and 1: $\forall e \in E,\ \lambda(e) \in \mathcal{U}]0, 1]$. We assume the source node s to be chosen uniformly at random in V. In order to ensure, w.h.p., the connectedness of the graph, we also assume $p > \frac{(1+\epsilon)\ln n}{n}$ for some $\epsilon > 0$.

7.A.2 Useless work

We say that a node is *settled* at time t when its tentative distance is equal to its final distance. Every time that a node which is not settled is relaxed, useless work is performed, since the node will need to be relaxed again when its tentative distance is going to be updated (Dijkstra's algorithm only relaxes nodes which are settled, thus performing only useful work, but, on the other hand, it is hard to parallelize because of its dependencies). The following theorem (proof in Section 7.A.3) bounds the useless work W_t performed by our algorithm as a function of d_t.

Theorem 7.A.1. *Let W_t be the useless work performed at time t by our algorithm, using an ideal priority queue, and let $h_t(i, j) = d_t(j) - d_t(i)$. We can bound W_t from above as:*

$$W_t \leq \sum_{j=1}^{P}\left[1 - \prod_{i=1}^{j-1}\prod_{L=1}^{n-1}\left(1 - \frac{(p\, h_t(i,j))^L}{L!}\right)^{\frac{(n-2)!}{(n-1-L)!}}\right] \ .$$

Remark 7.A.2. *A simpler (but weaker) form of this bound can be obtained by substituting $h_t(i, j)$ with $h_t^* = \max_{i,j} h_t(i, j) = h_t(1, P)$.*

7.A.3 Proofs and lemmata

In order to simplify the analysis, we assume the following property to hold when the number of nodes n is large. The property has been experimentally validated using the simulator presented in Section 7.8.6.

Conjecture 7.A.3. *Throughout the execution of the ideal priority queue SSSP algorithm, for all values of $t \in \mathbb{N}$, $1 \leq i < j \leq P$ and $h \in]0, 1]$, the probability that there is a path of weight less than h between $a_t(i)$ and $a_t(j)$ is bounded from above by the probability that such a path exists in a random graph, between two (uniformly) random nodes.*

Lemma 7.A.4. *Let $h \in]0,1]$ and let $\pi^L = (\pi_0, \pi_1, \ldots, \pi_{L-1}, \pi_L)$ be a path in G chosen uniformly at random among the paths of length L, such that the subpaths $\pi' = (\pi_0, \ldots, \pi_{L-1})$ and $\pi'' = (\pi_{L-1}, \pi_L)$ both have weights smaller than h. Let $f^L(\lambda)$ be the probability density function associated with the total weight $\lambda(\pi^L) = \sum_{i=1}^{L} \lambda(\pi_{i-1}, \pi_i)$. We can write f^L as*

$$f^L(\lambda) = \begin{cases} \frac{\lambda^{L-1}}{h^L} & \lambda \in]0,h] \\ \frac{1}{h} - \frac{(\lambda-h)^{L-1}}{h^L} & \lambda \in]h,2h] \\ 0 & \textit{otherwise} \end{cases} .$$

Proof. The proof is by induction on L.

Base case. For $L = 1$, since the edge weight is uniformly distributed between 0 and 1, we clearly have

$$f^1(\lambda) = \begin{cases} \frac{1}{h} & \lambda \in]0,h] \\ 0 & \text{otherwise} \end{cases} .$$

Induction. We assume now that the inductive hypothesis holds for all values $l \leq L$. Let f_h^l be the probability density function obtained by conditioning its weight $\lambda(\pi^l)$ to be smaller than h, i.e.,

$$f_h^l(\lambda) = \begin{cases} \frac{l\lambda^{l-1}}{h^l} & \lambda \in]0,h] \\ 0 & \text{otherwise} \end{cases} .$$

We have $\lambda(\pi^L) = \lambda(\pi') + \lambda(\pi'')$, where π' and π'' are subpaths of length L and 1. Since $\lambda(\pi')$ and $\lambda(\pi'')$ are independent, the density function f^{L+1} can be obtained by convolution of f_h^L and f_h^1:

$$f^{L+1} = f_h^L * f_h^1 \qquad \Rightarrow \qquad f^{L+1}(\lambda) = \begin{cases} \frac{\lambda^L}{h^{L+1}} & \lambda \in]0,h] \\ \frac{1}{h} - \frac{(\lambda-h)^L}{h^{L+1}} & \lambda \in]h,2h] \\ 0 & \text{otherwise} \end{cases} ,$$

which concludes the induction. □

Corollary 7.A.5. *The probability that a (uniformly random) path π^L has $\lambda\left(\pi^L\right) < h$, conditioned to $\lambda(\pi') < h$ and $\lambda(\pi'') < h$, is equal to $\frac{1}{L}$.*

Proof. Just integrate f^L between 0 and h. □

Proof (Theorem 7.A.1). Let $1 \leq i < j \leq \alpha_t$ and let $\pi_t^L(i,j) = (\pi_0 = a_t(i), \pi_1, \ldots, \pi_L = a_t(j))$ be a path between $a_t(i)$ and $a_t(j)$ of length L; we denote the weight of $\pi_t^L(i,j)$ as

$$\lambda\left(\pi_t^L(i,j)\right) = \sum_{k=0}^{L-1} \lambda(\pi_k, \pi_{k+1}) .$$

A node $a_t(j)$ is not settled if and only if there exists $i < j$ such that there exists a path $\pi_t^L(i,j)$ with $\lambda\left(\pi_t^L(i,j)\right) < d_t(j) - d_t(i)$. Note that the non-existence of a particular path with weight less than $h_t(i,j)$ does not decrease the probability for another different path not to exist. Therefore, being $h_t(i,j) = d_t(j) - d_t(i) \leq 1$, the probability $q_t(j)$ that $a_t(j)$ is settled can be bounded as

$$\begin{aligned} q_t(j) &\geq \prod_{i=1}^{j-1} \prod_{L=1}^{n-1} \Pr\left[\nexists \pi_t^L(i,j) : \lambda\left(\pi_t^L(i,j)\right) < h_t(i,j)\right] \\ &= \prod_{i=1}^{j-1} \prod_{L=1}^{n-1} \left(1 - r_t^L(i,j)\right) , \end{aligned}$$

where $r_t^L(i,j)$ is the probability that a path $\pi_t^L(i,j)$, with weight less than $h_t(i,j)$, exists. Assuming that we are relaxing the first P nodes of A_t, we can compute the expected value of the useless work performed at time t as $W_t = \sum_{j=1}^{P}(1 - q_t(j))$. Let $\tilde{r}_t^L(i,j)$ be the probability that a *particular* path $\pi_t^L(i,j)$ exists, with weight less than $h_t(i,j)$; we can bound $r_t^L(i,j)$ as

$$r_t^L(i,j) \leq 1 - \left(1 - \tilde{r}_t^L(i,j)\right)^{\frac{(n-2)!}{(n-1-L)!}} .$$

Note that there exists a path $\pi_t^L(i,j)$ with weight less than $h_t(i,j)$ if and only if the two subpaths $\pi' = (\pi_0, \ldots, \pi_{L-1})$ and $\pi'' = (\pi_{L-1}, \pi_L)$ exist, their weights are smaller than $h_t(i,j)$, and so is the sum of their weights. Because of Conjecture 7.A.3 and Corollary 7.A.5 we have $\tilde{r}_t^1(i,j) = p\, h_t(i,j)$, which finally implies

$$\tilde{r}_t^L(i,j) = \frac{\tilde{r}_t^{L-1}(i,j)\tilde{r}_t^1(i,j)}{L} = \frac{\left(\tilde{r}_t^1(i,j)\right)^L}{L!} ,$$

$$r_t^L(i,j) \leq 1 - \left(1 - \frac{\left(p\, h_t(i,j)\right)^L}{L!}\right)^{\frac{(n-2)!}{(n-1-L)!}} .$$

□

7.A.4 k-priority data structures

We can adapt our theoretical framework to support ρ-relaxed priority queues, which allow that up to ρ tasks may not be visible to all places, and may therefore not be executed even though they would have been with the ideal data structure. The bound of Theorem 7.A.1 can be adapted by changing the sum over all j's to only the j's corresponding to nodes $a_t(j)$ which have been actually relaxed ($\sum_{j=1}^{P} \to \sum_{j \in R_t}$, with $R_t = \{j : a_t(j) \text{ has been relaxed}\}$). Similarly to the previous case, a simpler form of this bound can be obtained by substituting $h_t(i,j)$ with h_t^*, defined as the difference between the largest and smallest tentative distance of nodes relaxed at time t, which implies $h_t^* \leq \max_{i,j} h_t(i,j) = h_t(1, P+\rho)$.

7.A.5 Weakening the requirements of ρ-relaxation

Our first implementations of ρ-relaxed priority queues, namely the *centralized* (Section 5.3) and the *hybrid k-priority queue* (Section 5.4) rely on *temporal* ρ-relaxation (see Section 5.1.3), allowing only the last ρ items added to the data structure to be ignored (ρ items per thread for the hybrid k-priority queue). As it turned out, our theoretical model does not require the temporal formulation of ρ-relaxation, the weaker *structural* formulation suffices.

This result opened up possibilities for relaxed priority queues that achieve similar bounds but do not need to maintain the temporal property, leading to the development of the *k-LSM priority queue* (Section 5.8.18).

8 Summary and Outlook

The work in this thesis, which was originally started as an attempt to implement an extension to the task scheduling model, has grown to a work that spans three separate fields related to parallel computing: *Programmability*, *Scheduling* and (non-blocking) *Synchronization*.

Our work on *programmability* aspects was concerned with providing an easy to use programming model and programming patterns for task parallelism, as well as, extensions to the task parallel programming model that can both help to make the model more general, and, thus, suitable for more applications and to simplify the implementation of efficient programs in the task parallel model. Furthermore, not all of the presented extensions to the task parallel programming model can be supported by standard task schedulers, and thus it was necessary to develop new schedulers that can support these models.

Efficient implementations of the presented task parallel programming models require efficient *synchronization* primitives. The most important such primitives are task queues, which have a significant impact on the scalability of a task scheduler. Furthermore, the order in which tasks are returned by a task queue can have a huge impact on time-, space- and communication bounds of a scheduler.

The work in these three fields resulted in *Pheet*, a task-parallel library that supports the presented programming models. Its plug-in architecture makes Pheet interesting on its own, making it a useful tool for research and teaching that enables the rapid prototyping and evaluation of new schedulers and synchronization primitives.

8.1 Lessons Learned

While each of the fields touched by this thesis provides enough open problems to justify a thesis on its own, this work shows that new insights can be gained by adopting a broader view. As an example, a scalable task queue implementation is of limited use, if the resulting scheduler has bad time- and/or space bounds. At the same time, due to resource limitations, there is always a trade-off between the depth at which a problem is analysed, and how broad the adopted view is. So even for this work compromises were made, and fields like compilers and real-world applications based on the task-parallel programming model left out.

Touching more than one field, and, thus, working with more than one community also led to the sobering observation that the current system in academia often discourages interdisciplinary work. In the peer reviewing process, reviewers tend to demand more details and a deeper discussion of work in their field of expertise, even if it is not the main contribution of a work. As an example, it is easier to publish work about a concurrent priority queue, than to publish work that also puts this priority queue in context, hence giving a better understanding as to what the requirements of such a priority queue are. The reason for this is that it leaves

reviewers dissatisfied who are domain experts for the context and not priority queues.

8.2 Outtakes

The development of a large and complex library like Pheet pushes the limitations of the programming language it is implemented in. This naturally results in the development of new programming patterns to work around these limitations. It also sparks ideas for extensions to programming languages to work around such limitations. Similarly, work on non-blocking synchronization primitives helps to understand the limitations of current hardware synchronization primitives, and creates ideas as to how these limitations can be solved.

While all of these ideas would be interesting contributions on their own, they all have in common that they are out of the scope of this work. Due to limited resources these ideas had to be left unexplored.

8.3 Future Work

Current work focusses on strategies, and in particular the priority scheduling aspect. The ultimate goal is to support an arbitrary hierarchy of strategies, and to only pay for the overhead of priority scheduling if necessary. The two-level concurrent ordered container from Section 5.5 is a first step in this direction. We plan to build upon this design to support complex composed applications efficiently.

The efficient implementation of concurrent priority queues is a topic interesting on its own, and we are working on both priority task queues as well as standalone priority queues. The implementation of standalone priority queues, will also allow for a direct comparison with other priority queue implementations, thus putting our work into context.

Our current focus is on quantitatively relaxed priority queues, and we believe that there is still room for improvement both with regard to scalability, as well as ordering guarantees. Our new concurrent LSM priority queue already shows promising behaviour, but so far there is still room for improvement with regard to a ρ-relaxed implementation. Since all of our current ρ-relaxed priority queues only relax the insertion of items into the priority queue, there can still be a high congestion for the removal of the highest priority item.

Finally we are planning to perform a complete and fair benchmark study of Pheet and other task scheduling frameworks to gain more insight into performance and scalability of Pheet. For this we plan to port the Pheet benchmarks to other common task-parallel programming models and systems. We also intend to port the recent *problem based benchmark suite* [124] to Pheet, which will give us access to a wider range of task-parallel applications.

8.4 Final Remarks

Research is a learning experience, and writing a thesis covering four years of research is an important opportunity to reflect upon the lessons learned. It allows to revisit old work under a new light, and to apply knowledge gained over the years to improve upon this work. To use a common saying: *You are your own worst critic*, and therefore results of such reflection are not always flattering. At the same time, being able to criticise one's own work shows how oneself developed since the time the work was initially done.

In research, every question answered opens up a multitude of new questions. Each solution one finds is only a part of a bigger puzzle, and research has a tendency to never be finished. Due to this, starting to write a thesis is a very rewarding experience. It allows to draw a line, and creates the illusion of producing something final and polished. In the end,

this is an illusion. This thesis should be read as a snapshot taken after four years of research, which contains many interesting ideas, some already published, and many more that are not. This work certainly provides more than enough material to fill another four years of research. Therefore, such a thesis can never be complete, but it can be the start of something bigger.

Bibliography

[1] Umut A Acar, Guy E Blelloch, and Robert D Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3):321–347, 2002.

[2] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: a tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[3] Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-Linearizability: Relaxed consistency for improved concurrency. In *14th International Conference on Principles of Distributed Systems, OPODIS '10*, volume 6490 of *Lecture Notes in Computer Science*, pages 395–410, Berlin, Heidelberg, 2010. Springer.

[4] Shivali Agarwal, Rajkishore Barik, Dan Bonachea, Vivek Sarkar, Rudrapatna K Shyamasundar, and Katherine Yelick. Deadlock-free scheduling of X10 computations with bounded resources. In *19th ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 229–240, New York, NY, USA, 2007. ACM.

[5] Alok Aggarwal and S Vitter, Jeffrey. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[6] Kunal Agrawal, Charles E Leiserson, and Jim Sukha. Executing task graphs using work-stealing. In *24th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '10, pages 1–12. IEEE Computer Society, 2010.

[7] Ravindra K Ahuja, Thomas L Magnanti, and James B Orlin. *Network Flows*. Prentice-Hall, 1993.

[8] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[9] Andrei Alexandrescu, Hans Boehm, Kevlin Henney, Doug Lea, and Bill Pugh. Memory model for multithreaded C++. Technical Report N1680, The C++ Standards Committee, 2004.

[10] Dan Alistarh, Justin Kopinsky, Jerry Li, and Nir Shavit. The SprayList: A scalable relaxed priority queue. Technical Report MSR-TR-2014-16, Microsoft Research, February 2014.

[11] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001.

[12] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf,

Samuel Webb Williams, and Katherine A Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley, 18 December 2006.

[13] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, 1994.

[14] Hagit Attiya and Jennifer L Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

[15] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[16] Eduard Ayguade, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[17] Henri E Bal and Matthew Haines. Approaches for integrating task and data parallelism. *IEEE Concurrency*, 6(3):74–84, 1998.

[18] Dmitry Basin, Rui Fan, Idit Keidar, Ofer Kiselov, and Dmitri Perelman. CAFÉ: Scalable task pools with adjustable fairness and contention. In *25th International Conference on Distributed Computing, DISC '11*, Lecture Notes in Computer Science, pages 475–488, Berlin, Heidelberg, 1 January 2011. Springer.

[19] Siegfried Benkner, Sabri Pllana, Jesper Larsson Träff, Philippas Tsigas, Uwe Dolinsky, Cedric Augonnet, Beverly Bachmayer, Christoph Kessler, David Moloney, and Vitaly Osipov. PEPPHER: Efficient and productive usage of hybrid computing systems. *IEEE Micro*, 31(5):28–41, 2011.

[20] Petra Berenbrink, Tom Friedetzky, and Leslie Ann Goldberg. The natural Work-Stealing algorithm is stable. *SIAM Journal on Computing*, 32(5):1260–1279, 2003.

[21] Guy E Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, November 1989.

[22] Guy E Blelloch, Phillip B Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, 1 March 1999.

[23] Guy E Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *1st ACM SIGPLAN International Conference on Functional Programming*, ICFP '96, pages 213–225, New York, NY, USA, 1996. ACM.

[24] Robert D Blumofe, Matteo Frigo, Christopher F Joerg, Charles E Leiserson, and Keith H Randall. DAG-consistent distributed shared memory. In *10th International Parallel Processing Symposium*, IPPS '96, pages 132–141, Washington, DC, USA, 1996. IEEE Computer Society.

[25] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[26] Robert D Blumofe and Charles E Leiserson. Space-Efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, 1998.

[27] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[28] Hans-J Boehm and Sarita V Adve. Foundations of the C++ concurrency memory model. In *2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 43 of *PLDI '08*, pages 68–78, New York, NY, USA, June 2008. ACM.

[29] Anastasia Braginsky, Alex Kogan, and Erez Petrank. Drop the anchor: Lightweight memory management for non-blocking data structures. In *25th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 33–42, New York, NY, USA, 2013. ACM.

[30] Richard P Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, April 1974.

[31] 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, February 1998.

[32] Francois Broquedis, Jerome Clet-Ortega, Stephanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In *18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP '10, pages 180–186. IEEE Computer Society, 2010.

[33] F Warren Burton. Storage management in virtual tree machines. *IEEE Transactions on Computers*, 37(3):321–328, 1988.

[34] F Warren Burton and M Ronan Sleep. Executing functional programs on a virtual tree of processors. In *1981 Conference on Functional Programming Languages and Computer Architecture*, FPCA '81, pages 187–194, New York, NY, USA, 1981. ACM.

[35] Daniel Cederman and Philippas Tsigas. Supporting lock-free composition of concurrent data objects. In *7th ACM International Conference on Computing Frontiers*, CF '10, pages 53–62, New York, NY, USA, 2010. ACM.

[36] Soumen Chakrabarti, James Demmel, and Katherine Yelick. Models and scheduling algorithms for mixed data and task parallel programs. *Journal of Parallel and Distributed Computing*, 47(2):168–184, December 1997.

[37] Rohit Chandra, Anoop Gupta, and John L Hennessy. COOL: An object-based language for parallel programming. *IEEE Computer*, 27(8):13–26, 1994.

[38] 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. In *20th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[39] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *17th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM.

[40] Jens Clausen and Jesper Larsson Träff. Implementation of parallel branch-and-bound algorithms – experiences with the graph partitioning problem. *Annals of Operations Research*, 33:331–349, 1991.

[41] S Cohen, R Rosner, and A Zidon. *Paralisp Simulator (reference manual)*, 1983.

[42] Richard Cole and Vijaya Ramachandran. Resource oblivious sorting on multicores. In *37th International Colloquium Conference on Automata, Languages and Programming, ICALP '10*, volume 6198 of *Lecture Notes in Computer Science*, pages 226–237, Berlin, Heidelberg, 2010. Springer-Verlag.

[43] Teodor Gabriel Crainic, Bertrand Le Cun, and Catherine Roucairol. Parallel branch-and-bound algorithms. In El-Ghazali Talbi, editor, *Parallel Combinatorial Optimization*, pages 1–28. Wiley, 2006.

[44] Andreas Crauser, Kurt Mehlhorn, Uli Meyer, and Peter Sanders. A parallelization of dijkstra's shortest path algorithm. In *Mathematical Foundations of Computer Science 1998*, pages 722–731. Springer Berlin Heidelberg, 1 January 1998.

[45] Mark Crovella, Prakash Das, Cezary Dubnicki, Thomas LeBlanc, and Evangelos Markatos. Multiprogramming on multiprocessors. In *3rd IEEE Symposium on Parallel and Distributed Processing*, IPDPS '91, pages 590–597. IEEE Computer Society, 1991.

[46] Lawrence A Crowl, Mark Crovella, Thomas J Leblanc, and Michael L Scott. The advantages of multiple parallelizations in combinatorial search. *J. Parallel Distrib. Comput.*, 21(1):110–123, April 1994.

[47] Frederic Desprez and Frederic Suter. Impact of mixed-parallelism on parallel implementations of the strassen and winograd matrix multiplication algorithms. *Concurrency and Computation: Practice and Experience*, 16(8):771–797, 2004.

[48] David L Detlefs, Paul A Martin, Mark Moir, and Guy L Steele, Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–271, 1 December 2002.

[49] Manuel Diaz, Bartolome Rubio, Enrique Soler, and Jose M Troya. Integrating task and data parallelism by means of coordination patterns. In *15th International Parallel and Distributed Processing Symposium*, IPDPS '01, pages 1077–1077. IEEE Computer Society, April 2001.

[50] M Drozdowski. *Scheduling for Parallel Processing*. Computer Communications and Networks. Springer, 2010.

[51] Jörg Dümmler, Thomas Rauber, and Gudula Rünger. Communicating Multiprocessor-Tasks. In *Languages and Compilers for Parallel Computing*, pages 292–307. Springer Berlin Heidelberg, 1 January 2008.

[52] Jörg Dümmler, Thomas Rauber, and Gudula Rünger. Programming support and scheduling for communicating parallel tasks. *Journal of Parallel and Distributed Computing*, 73(2):220–234, February 2013.

[53] A Duran, J Corbalan, and E Ayguade. An adaptive cut-off for task parallelism. In *2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 1–11, Piscataway, NJ, USA, November 2008. IEEE Press.

[54] Paul Erdös and Alfréd Rényi. On random graphs. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

[55] Francine Evans, Steven Skiena, and Amitabh Varshney. Optimizing triangle strips for fast rendering. In *7th Conference on Visualization*, VIS '96, pages 319–326, Los Alamitos, CA, USA, October 1996. IEEE Computer Society Press.

[56] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[57] Faith Ellen Fich, Victor Luchangco, Mark Moir, and Nir Shavit. Obstruction-Free algorithms can be practically Wait-Free. In *Distributed Computing*, Lecture Notes in Computer Science, pages 78–92. Springer Berlin Heidelberg, 1 January 2005.

[58] Ian Foster, David R Kohr, Jr., Rakesh Krishnaiyer, and Alok Choudhary. A Library-Based approach to task parallelism in a Data-Parallel language. *Journal of Parallel and Distributed Computing*, 45(2):148–158, September 1997.

[59] Keir Fraser. *Practical lock-freedom*. PhD thesis, King's College, University of Cambridge, 2004.

[60] Daniel P Friedman and David S Wise. Aspects of applicative programming for parallel processing. *IEEE Transactions on Computers*, C-27(4):289–296, 1978.

[61] Matteo Frigo, Pablo Halpern, Charles E Leiserson, and Stephen Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *21st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM.

[62] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Symposium on Foundations of Computer Science*, FOCS '99, pages 285–297, New York, NY, USA, 1999. IEEE Computer Society.

[63] Michael R Garey and David S Johnson. *Computers and Intractability: a guide to NP-completeness*. W. H. Freeman and Company, New York, 1979.

[64] Edgar N Gilbert. Random graphs. *Annals of Mathematical Statistics*, 30(4):1141–1144, 1959.

[65] Ronald Lewis Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 1966.

[66] Ronald Lewis Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[67] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic support for generic programming in c++. In *21st ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 291–310, New York, NY, USA, 2006. ACM.

[68] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *23rd IEEE International Parallel and Distributed Processing Symposium*, IPDPS '09, pages 1–12. IEEE Computer Society, 2009.

[69] Yi Guo, Jisheng Zhao, Vincent Cavé, and Vivek Sarkar. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *24th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '10, pages 1–12. IEEE Computer Society, 2010.

[70] Phuong Hoai Ha, Philippas Tsigas, and Otto J. Anshus. NB-FEB: A universal scalable Easy-to-Use synchronization primitive for manycore architectures. In *Principles of Distributed Systems*, Lecture Notes in Computer Science, pages 189–203. Springer Berlin Heidelberg, 1 January 2009.

[71] Robert H Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. In *1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 9–17, New York, NY, USA, 1984. ACM.

[72] Robert H Halstead, Jr. MULTILISP: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[73] Yijie Han, V Y Pan, and John H Reif. Efficient parallel algorithms for computing all pair shortest paths in directed graphs. In *4th ACM Symposium on Parallel Algorithms and Architectures*, SPAA '92, pages 353–362, New York, NY, USA, 1992. ACM.

[74] Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 67(12):1270–1285, December 2007.

[75] D Hendler and N Shavit. Non-blocking steal-half work queues. In *21st Symposium on Principles of Distributed Computing*, PODC '02, pages 280–289, New York, NY, USA, 2002. ACM.

[76] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *16th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 206–215, New York, NY, USA, 2004. ACM.

[77] Thomas A Henzinger, Christoph M Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. In *40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 317–328, New York, NY, USA, 2013. ACM.

[78] Kieran T Herley, Andrea Pietracaprina, and Geppino Pucci. Fast deterministic parallel branch-and-bound. *Parallel Processing Letters*, 9(3):325–333, 1999.

[79] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[80] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

[81] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008.

[82] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[83] Christopher F Joerg. *The Cilk System for Parallel Multithreaded Computing*. PhD thesis, Massachusetts Institute of Technology, 1996.

[84] Nicolai M Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, 2012.

[85] Richard M Karp and Yanjun Zhang. Randomized parallel algorithms for backtrack search and Branch-and-Bound computation. *Journal of the ACM*, 40(3):765–789, 1993.

[86] Robert M Keller, Frank C H Lin, and Jiro Tanaka. Rediflow multiprocessing. In *28th IEEE Computer Society International Conference*, COMPCON '84. IEEE Computer Society, 1984.

[87] Christoph W Kessler and E Hansson. Flexible scheduling and thread allocation for synchronous parallel tasks. In *ARCS Workshops, 2012*, pages 1–7, February 2012.

[88] CW Kessler, N Melot, P Eitschberger, and J Keller. Crown scheduling: Energy-efficient resource allocation, mapping and discrete frequency scaling for collections of malleable streaming tasks. In *23rd International Workshop on Power and Timing Modeling, Optimization and Simulation*, PATMOS '13, pages 215–222, 2013.

[89] CM Kirsch, M Lippautz, and H Payer. Fast and scalable k-FIFO queues. Technical Report 2102-04, Department of Computer Sciences - University of Salzburg, June 2012.

[90] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 47 of *PPoPP '12*, pages 141–150, New York, NY, USA, February 2012. ACM.

[91] David A Kranz, Robert H Halstead, Jr., and Eric Mohr. Mul-T: A high-performance parallel lisp. In *1989 ACM SIGPLAN Conference on Programming Language Design and Implementation*, volume 24 of *PLDI '89*, pages 81–90, New York, NY, USA, June 1989. ACM.

[92] D Krishnaswamy and P Banerjee. Exploiting task and data parallelism in parallel hough and radon transforms. In *1997 International Conference on Parallel Processing*, pages 441–444, August 1997.

[93] Alexey Kukanov and Michael J. Voss. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal*, 11(4), 2007.

[94] Vipin Kumar and V Nageshwara Rao. Parallel depth first search, part II: Analysis. *International Journal of Parallel Programming*, 16(6):501–519, 1 December 1987.

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

[96] I-Ting Angelina Lee, Aamir Shafi, and Charles E Leiserson. Memory-mapping support for reducer hyperobjects. In *24th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 287–297, New York, NY, USA, 2012. ACM.

[97] Charles E Leiserson. The Cilk++ concurrency platform. *Journal of Supercomputing*, 51(3):244–257, 2010.

[98] Charles E Leiserson and Tao B Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *22nd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 303–314, New York, NY, USA, 2010. ACM.

[99] A Lenharth, D Nguyen, and K Pingali. Priority queues are not good concurrent priority schedulers. Technical Report TR-11-39, Department of Computer Science, The University of Texas at Austin, 2011.

[100] Vasileios Liaskovitis, Shimin Chen, Phillip B Gibbons, Anastassia Ailamaki, Guy E Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Michael Kozuch, Todd C Mowry, and Chris Wilkerson. Parallel depth first vs. work stealing schedulers on CMP architectures. In *18th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '06, pages 330–330, New York, NY, USA, 30 July 2006. ACM.

[101] Jeremy Manson, William Pugh, and Sarita Adve, V. The Java memory model. In *32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[102] Ernesto Queirós Vieira Martins. On a multicriteria shortest path problem. *European Journal of Operational Research*, 16(2):236–245, May 1984.

[103] Ulrich Meyer. Average-case complexity of single-source shortest-paths algorithms: lower and upper bounds. *Journal of Algorithms - Special issue: Twelfth annual ACM-SIAM symposium on discrete algorithms*, 48(1):91–134, 2003.

[104] Ulrich Meyer and Peter Sanders. Δ-Stepping: A parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[105] Maged M Michael. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[106] Maged M Michael and Michael L Scott. Correction of a memory management method for Lock-Free data structures. Technical report, University of Rochester, 1995.

[107] Maged M Michael, Martin T Vechev, and Vijay A Saraswat. Idempotent work stealing. In *14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 45–54, New York, NY, USA, 2009. ACM.

[108] Girija J Narlikar and Guy E Blelloch. Space-efficient scheduling of nested parallelism. *ACM Transactions on Programming Languages and Systems*, 21(1):138–173, January 1999.

[109] S Olivier, J Huan, J Liu, J Prins, J Dinan, P Sadayappan, and CW Tseng. UTS: An unbalanced tree search benchmark. *Languages and Compilers for Parallel Computing*, pages 235–250, 2007.

[110] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1 June 1996.

[111] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, 1982.

[112] Ruben Perez. Speculative parallelism in Intel Cilk Plus. Master's thesis, Massachusetts Institute of Technology, 2012.

[113] Judit Planas, Rosa M Badia, Eduard Ayguadé, and Jesus Labarta. Hierarchical Task-Based programming with StarSs. *International Journal of High Performance Computing Applications*, 23(3):284–299, August 2009.

[114] Andrei Radulescu and Arjan J C Van Gemund. A low-cost approach towards mixed task and data parallel scheduling. In *International Conference on Parallel Processing*, pages 69–76, 2001.

[115] S Ramaswamy, S Sapatnekar, and P Banerjee. A framework for exploiting task and data parallelism on distributed memory multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1098–1116, November 1997.

[116] Thomas Rauber and Gudula Rünger. Mixed task and data parallel executions in general linear methods. *Scientific Programming*, 15(3):137–155, January 2007.

[117] Peter Sanders. Fast priority queues for parallel branch-and-bound. In *Parallel Algorithms for Irregularly Structured Problems (IRREGULAR)*, volume 980 of *Lecture Notes in Computer Science*, pages 379–393, 1995.

[118] Peter Sanders. *Lastverteilungsalgorithmen für parallele Tiefensuche*, volume 10/463 of *Fortschrittsberichte VDI*. VDI-Verlag, 1997. PhD Thesis.

[119] Peter Sanders and Lawrence Mandow. Parallel label-setting multi-objective shortest path search. In *27th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 215–224, 2013.

[120] Nir Shavit and Itay Lotan. Skiplist-based concurrent priority queues. In *14th International Parallel and Distributed Processing Symposium*, IPDPS '00, pages 263–268. IEEE Computer Society, 2000.

[121] Nir Shavit and Asaph Zemach. Diffracting trees. *ACM Transactions on Computer Systems*, 14(4):385–428, November 1996.

[122] Jun Shirako, Vincent Cavé, Jisheng Zhao, and Vivek Sarkar. Finish accumulators: An efficient reduction construct for dynamic task parallelism. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 264–265. Springer Berlin Heidelberg, 1 January 2013.

[123] Jun Shirako, David M Peixotto, Vivek Sarkar, and William N Scherer. Phasers: A unified deadlock-free construct for collective and point-to-point synchronization. In *22nd International Conference on Supercomputing*, ICS '08, pages 277–288, New York, NY, USA, 2008. ACM.

[124] 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 *24th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 68–70, New York, NY, USA, 2012. ACM.

[125] Marc Snir. Depth-size trade-offs for parallel prefix computation. *Journal of Algorithms*, 7(2):185–201, June 1986.

[126] Fengguang Song, A YarKhan, and J Dongarra. Dynamic task scheduling for linear algebra algorithms on Distributed-Memory multicore systems. In *International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '09, pages 1–11, November 2009.

[127] Daniel Spoonhower, Guy E Blelloch, Phillip B Gibbons, and Robert Harper. Beyond nested parallelism: Tight bounds on work-stealing overheads for parallel futures. In *21st Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 91–100, New York, NY, USA, 2009. ACM.

[128] Mark S Squillante and Edward D Lazowska. Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, 1993.

[129] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 19 May 2013.

[130] Jaspal Subhlok and Gary Vondran. Optimal use of mixed task and data parallelism for pipelined computations. *Journal of Parallel and Distributed Computing*, 60(3):297–319, March 2000.

[131] Hakan Sundell. Wait-Free reference counting and memory management. In *19th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '05, page 24b. IEEE Computer Society, 2005.

[132] Gerald Jay Sussman, Harold Abelson, and Julie Sussman. Structure and interpretation of computer programs. *New England Journal of Public Policy*, 1985.

[133] Oussama Tahan, Mats Brorsson, and Mohamed Shawky. Introducing task cancellation to OpenMP. In *OpenMP in a Heterogeneous World*, pages 73–87. Springer Berlin Heidelberg, 1 January 2012.

[134] Marc Tchiboukdjian, Nicolas Gast, Denis Trystram, Jean-Louis Roch, and Julien Bernard. A tighter analysis of work stealing. In *Algorithms and Computation*, pages 291–302. Springer Berlin Heidelberg, 1 January 2010.

[135] P Tsigas and Yi Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000. In *11th Euromicro Workshop on Parallel, Distributed and Network-Based Processing*, PDP '03, pages 372–381. IEEE Computer Society, February 2003.

[136] John D Valois. Lock-free linked lists using compare-and-swap. In *14th ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 214–222, New York, NY, USA, 1995. ACM.

[137] John D Valois. *Lock-free data structures*. PhD thesis, Rensselaer Polytechnic Institute Troy, 1996.

[138] Hans Vandierendonck, Kallia Chronaki, and Dimitrios S Nikolopoulos. Deterministic scale-free pipeline parallelism with hyperqueues. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 32:1–32:12, New York, NY, USA, 2013. ACM.

[139] Georgios Varisteas and Mats Brorsson. Palirria: Accurate on-line parallelism estimation for adaptive Work-Stealing. In *Proceedings of Programming Models and Applications on Multicores and Manycores*, PMAM'14, pages 120:120–120:131, New York, NY, USA, 2007. ACM.

[140] Boris Weissman. Performance counters and state sharing annotations: a unified approach to thread locality. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '98, pages 127–138, 1998.

[141] Martin Wimmer. Wait-free hyperobjects for task-parallel programming systems. In *27th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '13, pages 803–812. IEEE Computer Society, 2013.

[142] Martin Wimmer, Daniel Cederman, Jesper Larsson Träff, and Philippas Tsigas. Configurable strategies for work-stealing. *CoRR*, abs/1305.6474, 05 2013.

[143] Martin Wimmer, Daniel Cederman, Jesper Larsson Träff, and Philippas Tsigas. Work-stealing with configurable scheduling strategies. In *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 315–316, New York, NY, USA, 2013. ACM.

[144] Martin Wimmer, Daniel Cederman, Francesco Versaci, Jesper Larsson Träff, and Philippas Tsigas. Data structures for task-based priority scheduling. *CoRR*, abs/1312.2501, 09 December 2013.

[145] Martin Wimmer, Manuel Pöter, and Jesper Larsson Träff. The Pheet task-scheduling framework on the Intel Xeon Phi coprocessor and other multicore architectures. In *Workshop on Multi-threaded Architectures and Applications*, MTAAP '13. IEEE Computer Society, 2013.

[146] Martin Wimmer and Jesper Larsson Träff. Work-stealing for mixed-mode parallelism by deterministic team-building. In *23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 105–115, New York, NY, USA, 2011. ACM.

[147] Martin Wimmer and Jesper Larsson Träff. A work-stealing framework for mixed-mode parallel applications. In *Workshop on Multi-threaded Architectures and Applications*, MTAAP '11. IEEE Computer Society, 2011.

[148] Martin Wimmer, Francesco Versaci, Jesper Larsson Träff, Daniel Cederman, and Philippas Tsigas. Data structures for task-based priority scheduling. In *19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 379–380, New York, NY, USA, 6 February 2014. ACM.

[149] Wm A Wulf and Sally A McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, March 1995.

[150] Bwolen Yang and David R O'Hallaron. Parallel breadth-first BDD construction. In *6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, volume 32 of *PPoPP '97*, pages 145–156, New York, NY, USA, June 1997. ACM.

Curriculum Vitae

Name Martin Wimmer

Place of Birth Vienna, Austria

Research Interests

Parallel Algorithms, Graph Algorithms, Task Scheduling, Data Structures, Non-blocking Synchronization, Programming Languages and Models, Compilers, Parsing

Education

2010 – 2014 PhD in Computer Science at Vienna University of Technology

2007 – 2010 Master in Scientific Computing at University of Vienna

2000 – 2006 Bachelor in Business Informatics at University of Vienna

Teaching Experience

2012 – 2014 : Developed and taught part of a graduate course on *Advanced Multiprocessor Programming* at Vienna University of Technology.

2011 Taught an undergraduate course on *Theoretical Computer Science* at University of Vienna

2011 Developed and taught part of a graduate course on *The Art of Multiprocessor Programming* at University of Vienna

2010 Teaching assistant in a graduate course on *High Performance Computing* at University of Vienna

2009 Teaching assistant in a course on *Computational Drug Design* at University of Vienna

Professional Experience

2014 Lecturer at Vienna University of Technology

2013 Research assistant at Vienna University of Technology

2010 – 2012 Research assistant in the PEPPHER project

2008 – 2010 Research and teaching assistant at University of Vienna

Publications

1. Martin Wimmer, Francesco Versaci, Jesper Larsson Träff, Daniel Cederman, and Philippas Tsigas. Data structures for task-based priority scheduling. In *19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 379–380. ACM, 2014.

2. Martin Wimmer, Daniel Cederman, Francesco Versaci, Jesper Larsson Träff, and Philippas Tsigas. Data structures for task-based priority scheduling. *CoRR*, abs/1312.2501, 2013.

3. Martin Wimmer. Wait-free hyperobjects for task-parallel programming systems. In *27th IEEE International Parallel and Distributed Processing Symposium*, IPDPS '13, pages 803–812. IEEE Computer Society, 2013.

4. Martin Wimmer, Manuel Pöter, and Jesper Larsson Träff. The Pheet task-scheduling framework on the Intel Xeon Phi coprocessor and other multicore architectures. In *Workshop on Multi-threaded Architectures and Applications*, MTAAP '13. IEEE Computer Society, 2013.

5. Martin Wimmer, Daniel Cederman, Jesper Larsson Träff, and Philippas Tsigas. Work-stealing with configurable scheduling strategies. In *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 315–316. ACM, 2013.

6. Martin Wimmer, Daniel Cederman, Jesper Larsson Träff, and Philippas Tsigas. Configurable strategies for work-stealing. *CoRR*, abs/1305.6474, 2013.

7. Christoph W. Keßler, Usman Dastgeer, Mudassar Majeed, Nathalie Furmento, Samuel Thibault, Raymond Namyst, Siegfried Benkner, Sabri Pllana, Jesper Larsson Träff, Martin Wimmer. Leveraging PEPPHER Technology for Performance Portable Supercomputing. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '2012, pages 1395–1396. IEEE Computer Society, 2012.

8. Martin Wimmer and Jesper Larsson Träff. Work-stealing for mixed-mode parallelism by deterministic team-building. In *23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 105–115. ACM, 2011.

9. Martin Wimmer and Jesper Larsson Träff. A work-stealing framework for mixed-mode parallel applications. In *Workshop on Multi-threaded Architectures and Applications*, MTAAP '11. IEEE Computer Society, 2011.

10. Martin Wimmer. Programming Models for Parallel Computing. Master's Thesis. 2010.