On the Benefits of Abstraction in Concurrent Haskell

DISSERTATION

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
im Fachbereich 16 (Elektrotechnik/Informatik)
der Universität Kassel

vorgelegt von

Michael Christian Lesniak

Datum der Disputation: 1. Juni 2012

Summary

Nowadays, even commodity hardware consists of several processing units. This allows for parallel computing, i.e. a problem is split into several tasks that can be run in parallel. However, parallel programming is still difficult, even for experts. One possibility to ease the implementation of parallel programs is abstraction. While traditional programming languages work on a rather low level of abstraction, functional programming languages such as Haskell provide techniques for sophisticated abstractions.

In this thesis, we aim at analyzing how well different kinds of abstraction support programming with Concurrent Haskell, a library for shared-memory parallel programming in Haskell. Our research is focused on two areas of interest. First we compare synchronization approaches which differ in their level of abstraction. Second we investigate how abstraction techniques can be used to hide details of parallelization.

For a comparison of synchronization approaches, we considered locks, compare-and-swap operations, and software transactional memory, chiefly in the context of two problems. First, we implemented a thread safe priority queue on top of the skiplist data structure. Second, to compare the approaches in a more high-level scenario, we implemented the taskpool design pattern for different taskpool variants (global taskpools, and private taskpools with and without task stealing). Additionally, we provided an abstraction layer that allowed for a convenient and idiomatic formulation of taskpool algorithms.

For analyzing if Haskell's abstraction techniques allow to hide complex details of the parallelization, we first examined stencil-based algorithms. For this purpose we developed a library that allows for a declarative description of stencil-based algorithms as well as their parallel execution. By means of the declarative interface, the thread-based parallel implementation is completely hidden from the user. Subsequently, we developed an embedded domain specific language (EDSL) for vertex-centric graph algorithms as well as an execution platform that allows for their automatic parallel execution. We provided several examples which demonstrated that the EDSL allows for a concise yet understandable formulation of graph algorithms.

Zusammenfassung

Heutzutage haben selbst durchschnittliche Computersysteme mehrere unabhängige Recheneinheiten (Kerne). Wird ein rechenintensives Problem in mehrere Teilberechnungen unterteilt, können diese parallel und damit schneller verarbeitet werden. Obwohl die Entwicklung paralleler Programme mittels Abstraktionen vereinfacht werden kann, ist es selbst für Experten anspruchsvoll, effiziente und korrekte Programme zu schreiben. Während traditionelle Programmiersprachen auf einem eher geringen Abstraktionsniveau arbeiten, bieten funktionale Programmiersprachen wie z.B. Haskell, Möglichkeiten zur fortgeschrittenen Abstrahierung.

Das Ziel der vorliegenden Dissertation war es, zu untersuchen, wie gut verschiedene Arten der Abstraktion das Programmieren mit Concurrent Haskell unterstützen. Concurrent Haskell ist eine Bibliothek für Haskell, die parallele Programmierung auf Systemen mit gemeinsamem Speicher ermöglicht. Im Mittelpunkt der Dissertation standen zwei Forschungsfragen. Erstens wurden verschiedene Synchronisierungsansätze verglichen, die sich in ihrem Abstraktionsgrad unterscheiden. Zweitens wurde untersucht, wie Abstraktionen verwendet werden können, um die Komplexität der Parallelisierung vor dem Entwickler zu verbergen.

Bei dem Vergleich der Synchronisierungsansätze wurden Locks, Compare-and-Swap Operationen und Software Transactional Memory berücksichtigt. Die Ansätze wurden zunächst bezüglich ihrer Eignung für die Synchronisation einer Prioritätenwarteschlange auf Basis von Skiplists untersucht. Anschließend wurden verschiedene Varianten des Taskpool Entwurfsmusters implementiert (globale Taskpools sowie private Taskpools mit und ohne Taskdiebstahl). Zusätzlich wurde für das Entwurfsmuster eine Abstraktionsschicht entwickelt, welche eine einfache Formulierung von Taskpool-basierten Algorithmen erlaubt.

Für die Untersuchung der Frage, ob Haskells Abstraktionsmethoden die Komplexität paralleler Programmierung verbergen können, wurden zunächst stencil-basierte Algorithmen betrachtet. Es wurde eine Bibliothek entwickelt, die eine deklarative Beschreibung von stencil-basierten Algorithmen sowie ihre parallele Ausführung erlaubt. Mit Hilfe dieses deklarativen Interfaces wurde die parallele Implementation vollständig vor dem Anwender verborgen. Anschließend wurde eine eingebettete domänenspezifische Sprache (EDSL) für Knoten-basierte Graphalgorithmen sowie eine entsprechende Ausführungsplattform entwickelt. Die Plattform erlaubt die automatische parallele Verarbeitung dieser Algorithmen. Verschiedene Beispiele zeigten, dass die EDSL eine knappe und dennoch verständliche Formulierung von Graphalgorithmen ermöglicht.

Contents

List of Figures

Chapter 1
Introduction

1.1 Motivation

The effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer.

Edsger W. Dijkstra [42]

Over the last sixty years, computing power increased enormously. In 1953, the mainframe IBM 650 could execute 0.06 *thousand* instructions per second [203]. It had a price of $150.000 [105], which is, taking inflation into account, about $1.287.000 today. Nowadays, a commodity computer which costs a few hundred dollars can easily execute 10.000 *million* instructions per second [203]. One of the most important reasons for this enormous increase in computing power as well as the decrease in price were the technological advancements in the development of transistors [173].

Transistors are the basic building blocks of any modern central processing unit (CPU). In the last decades, transistors did not only become cheaper in production but they also became smaller and faster. Hence, more transistors can be assembled to form a processing unit and the CPU's clock rate can be increased. In 1965, this development has been predicted by Gordon Moore: *Moore's Law* states that *"the density of transistors on chips doubles every 24 months"* [147]. However laws of nature make it increasingly difficult to meet this prediction [116, 210]: Due to the heat generated by millions of active transistors which run with a high clock rate and which are packed on a small area, it becomes increasingly difficult to keep the temperature of the CPU low. In addition, quantum effects begin to play a major role in the interaction of transistors. Active transistors may influence their neighbors accidentally. Therefore, in the last years a different approach has been taken to satisfy the continuous demand for faster computers. Instead of increasing the clock rate and the number of transistors of a single processor, the number of distinct computation units (cores) is increased [106]. This transition is widely called the *multicore revolution.*

Unfortunately, the availability of several cores does not automatically increase the speed in already existing programs. A program which has been developed for the execution on a single processing unit will still run on a single core, despite the availability of others. Some speed can be gained since the operating system can schedule different programs to run on different cores [194]. For example, a web browser remains usable on a system with two cores despite the execution of a spreadsheet application performing

CPU intensive operations at the same time. Nevertheless, in the long run modern hardware will consist of dozens and even hundreds of cores (*manycore era*) [106]. To allow for an efficient usage of this hardware, programs have to be *parallelized*.

As of yet, the parallelization of a program is difficult, even for experts. Over the years, different approaches to develop parallel programs have emerged. These approaches differ in their flexibility as well as their degree of abstraction. As a rule of thumb, the more flexible an approach is, the less abstract is its underlying programming model. In the following we describe some widely used libraries as well as standards for parallel programming. Several other, less common approaches, are for example presented in [117, 124, 162].

A well-known low-level library for parallel programming in C and C++ is *POSIX Threads* [20, 205]. It is based on the concept of threads, where a *thread* is an independently running part of a program [123, 124]. In thread-based parallel programming, a program consists of multiple concurrently running threads that communicate by accessing shared memory. Concurrent access to this memory must be synchronized. The POSIX Threads library implements functions for thread control and synchronization, which operate on a low level. Unfortunately, the low degree of abstraction combined with programming in rather low-level languages gives rise to numerous potential sources for errors [162, 184]. Although tools like Halgrind [199] are able to find some typical programming errors, parallel programming with POSIX Threads remains a tedious and error-prone task.

OpenMP (Open Multi-Processing) is a standardized application programming interface for parallel programming [28, 150, 151] available for different programming languages (C, C++ and Fortran) and platforms (Unix, Windows, etc.) [151]. Programming with OpenMP is less error-prone than programming with POSIX Threads since OpenMP works on a higher level of abstraction. In contrast to using POSIX Threads, a developer does not have to handle the mundane details of thread-based parallel programming with OpenMP. Instead, code that is meant to be run in parallel is annotated. The actual parallelization, i.e. thread creation, task distribution, and thread synchronization, is supported by the compiler. Although OpenMP allows for a more abstract formulation of parallel programs, the supported programming languages do still not provide sophisticated abstraction techniques (see below). Hence it is still complicated to write correct and efficient programs [193].

A third approach is based on the idea of explicit message passing. In this model, threads do not have direct access to data of other threads. Instead, they send messages and receive messages. Message passing programs tend to have fewer bugs than shared-memory programs, since synchronization of shared state is a common source of errors [52, 54, 132]. On the other hand, designing a parallel algorithm which uses message passing tends to be more complicated, since task distribution and data locality are more important for the parallel performance than in shared-memory parallel programming. The de-facto standard for parallel programming with message passing is *MPI* (Message Passing Interface) [44, 68, 183]. Similar to OpenMP, MPI defines an interface that has been implemented by different vendors. While OpenMP focuses on shared-memory parallel programming, MPI is often used for distributed memory machines. MPI offers a wide range of communication primitives. For example, it defines functions for blocking as well as non-blocking point-to-point communications and collective operations for broadcasting or gathering data [183]. Similar to POSIX Threads and OpenMP, MPI is a low level approach. For example, when sending messages, their size in bytes has to be stated.

A different approach to parallel programming is taken by novel programming languages like Chapel [26, 27, 37], X10 [29, 175, 209], or Fortress [3]. These kinds of languages especially support abstractions for data parallelism, task parallelism, and data distribution amongst others in their language design. They also support a variety of types with a high level of abstraction, e.g. for matrices. Operations on these types are (partially) parallelized automatically. The introduced novel programming languages are not yet commonly used and are still an active area of research.

To allow for the development of programs for the multicore area, parallel programming has to become easier. Hence, approaches with a higher degree of abstraction are necessary.

A programming paradigm which supports quite sophisticated abstraction techniques is based on the notion of functions as regular types. This paradigm is called *functional* programming [97]. In 1930, the era of functional programming began with Church's description of the λ-calculus [204]. The λ-calculus is a formal system to describe aspects of computability by means of functions, their application and their definition [93, 204]. A program in a functional language is stated as an expression. The compiler and runtime system decide how and when to evaluate its subexpressions [161]. This allows for more freedom and therefore more possibilities for optimization. In particular, functional programming languages are amenable for automatic parallelization: Since subexpressions can not influence each other, theoretically they can be evaluated in parallel.

Unfortunately, identifying tasks is not sufficient for an efficient parallelization. Problems like task selection or task distribution have to be solved, too [71]. Therefore, as far as we know, automatic parallelization of arbitrary code has not yet been implemented successfully for any general-purpose functional programming language. On the other hand, automatic parallelization for a *particular domain*, e.g. array or list computations, is quite successful: For example, *SAC* (Single Assignment C) [67, 174] is a functional programming language with a syntax similar to C. SAC treats multi-dimensional arrays as first-class types and the SAC compiler automatically generates parallelized code for array operations for different target architectures. Another example is DPH (Data Parallel Haskell) [23, 24, 111] (see also Section 2.3.4). DPH defines a special notation for finite lists (arrays). Operations on them are automatically compiled to parallelized code.

In general, research of (automatic) parallel programming approaches within the functional programming paradigm has a long history. We refer to [14, 71, 129, 172, 176] for detailed surveys. In the following we describe some of today's widely used functional programming languages as well as their approach(es) to parallel programming.

Erlang is a functional programming language specifically designed for the development of telecommunication applications [8, 9, 50, 51]. It supports concurrent programming, hot swapping of code in a running application [9], and especially fault tolerance. Parallel programming is based on the actor model [2] in which independently running processes communicate by means of asynchronous messages.

Several functional programming languages are based on *LISP* (LISt Processing) [65, 177]. LISP programs are written as abstract syntax trees which can easily be manipulated by macros [121]. In addition, LISP's syntactic flexibility is often used to define domain specific languages [56, 65]. Invented in 1950 [141], LISP is special among the mentioned functional languages because it still influences programming language development. While the original LISP is seldom used, different languages that are based on LISP, emerged. *Common LISP* [186] supports procedural, functional as well as object-oriented programming, and has a large standard library. Nevertheless, the standard does not define functions for parallel programming. As an addition to the standard, Bordeaux-Threads [35] define an interface to thread-based shared-memory parallel programming. This interface is part of several Common LISP implementations [35]. In contrast to Common LISP, *Scheme* [185] has been designed to be easy to understand and implement. Therefore its language and library definition is quite small. In particular, Scheme is often used in teaching [1]. The Scheme standard does not define any functions for parallel programming, although different implementations define their own primitives, e.g. [169]. A novel member of the LISP family is *Clojure* [33, 70]. Since Clojure is executed on the Java Virtual Machine [126], Clojure programs can access all libraries of the Java platform, including Java threads [63]. In addition, Clojure supports parallel programming with actor-based message passing as well as synchronization with software transactional memory [32].

OCaml is a functional programming language with additional support for imperative and object-oriented programming [21, 145, 182]. It implements many of the properties of a modern functional language, e.g. a static type system, type inference and pattern matching. The standard OCaml system supports only low-level parallel programming with threads and mutexes [182]. OCaml is special amongst functional languages since it serves as the basis for *F#* [181]. While functional programming languages are usually neglected regarding commercial support, F# is fully supported by Microsoft as part of the .NET platform [143]. Similar to Clojure, F# programs can access all constructs of .NET for parallel programming [22].

One of the most well-known and currently researched functional programming languages is *Haskell* [89, 100, 135]. Haskell supports many advanced concepts of functional programming, e.g. polymorphic types, type inference, or monads amongst others. Particularly it allows for the parallelization of algorithms with various programming techniques [112, 114, 134] that differ in their degree of abstraction, efficiency and drawbacks: On the one hand, semi-explicit parallel programming allows to annotate expressions which can be evaluated in parallel [134]. The runtime system decides when and how to evaluate these expressions (in parallel). On the other hand, Haskell supports more conventional parallel programming with threads [160] (see below). For synchronization of data, it offers low-level approaches like locks [160] and atomic compare-and-swap operations [90, 191] but also sophisticated techniques like software transactional memory [72]. Given the number of published research papers, research focused on semi-explicit programming in the last years. We think that this focus was motivated by Trinder et al.'s seminal paper *Algorithm + Strategy = Parallelism* [197] which introduces the idea of evaluation strategies. Evaluation strategies are lazy higher-order functions which implement a separation of algorithmic and parallel code. Over the years, the approach of evaluation strategies and semi-explicit annotations has been continuously refined [137, 138, 139]. The focus on this approach led to the neglect of Haskell's other approaches for parallelism, in particular, of Concurrent Haskell. *Concurrent Haskell* [160] is a concurrency extension (library) that allows for explicit parallel programming similar to programming with POSIX Threads. It defines functions for thread control as well as different low-level data structures for synchronization. Hence, by using Concurrent Haskell, parallel algorithms can be implemented in a rather traditional way but with the benefits of a high-level functional language with sophisticated abstraction techniques.

In this thesis, our goal is to analyze how well abstraction at different degrees supports programming with Concurrent Haskell. How can traditional imperative data structures, design patterns and programming approaches be adapted to a modern functional programming language? Do commonly used idioms of the functional paradigm as well as high-level constructs of Haskell such as monads allow to hide the complexity of parallel programming with Concurrent Haskell? Or, more generally speaking, what are the particular advantages and disadvantages of a modern functional language for parallel programming? Since answering these questions in general is difficult, we focus our research on two different areas: Synchronization is crucial for shared-memory parallel programs, hence we first compare the synchronization techniques available in Concurrent Haskell regarding their parallel performance as well as their development effort. Second we explore how well advanced techniques of Haskell can hide the complexity of parallel programming in Concurrent Haskell. For examination of these areas we chose four different topics of parallel programming that vary in their degree of abstraction and their approach to parallelization. These topics are described in detail in the next section.

1.2 Contribution

The main part of this thesis covers two areas of interest: 1) comparison of synchronization approaches and 2) exploration of abstraction techniques to hide details of parallelization. Each area of interest is the focus of two sections of this thesis (see Figure 1.1 on the following page). Particular topics are chosen to cover different aspects of parallel programming by means of Concurrent Haskell as well as to examine how different concepts of abstraction in Haskell (e.g. monads, higher-order functions, or typeclasses), can be used to hide the complexity of parallel programming.

First, by means of a concurrently accessible data structure we examine the advantages and disadvantages of low-level synchronization techniques in Concurrent Haskell versus high-level ones. Second, the implementation of a parallel design pattern allows us to compare synchronization techniques as well as different abstraction approaches of Haskell to provide an additional problem-specific layer to this design pattern. Third, by implementing a library we show how complex aspects of thread-based parallel programming can be completely hidden behind a declarative interface. Fourth, we show how a specialized language can be used to hide parallelization in cases where a more flexible problem description is necessary. In the following we describe each topic in detail.

Data structures, e.g. priority queues, stacks and trees, are fundamental building blocks in almost any parallel algorithm, and their efficient and correct implementation is of great importance [146]. We examine **priority queues** which are a well-known data structure to store and retrieve elements from some ordered set [36]. Priority queues sup-

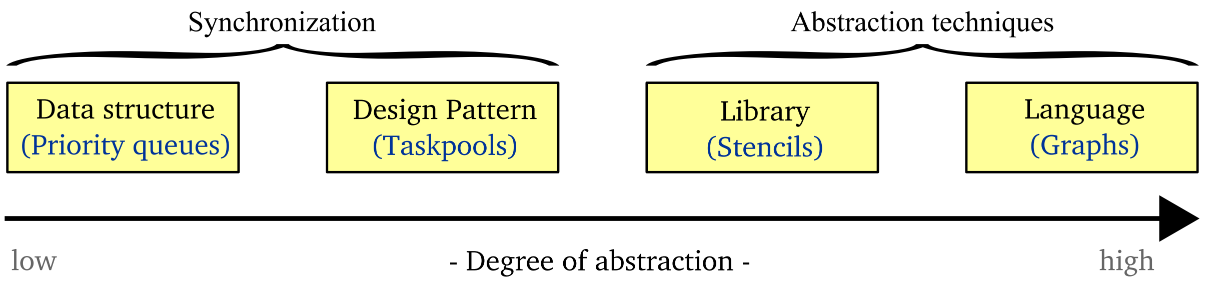


Figure 1.1: Topics of this thesis, ordered by degree of abstraction.

port at least two operations: *insert* adds an element to the queue, and *deleteMin* removes and returns the minimal element. The operations are non-trivial to implement and allow many optimizations. For the underlying implementation we use skiplists [164, 165], since they are more efficient [164] than binary heaps [36]. Skiplists provide a general dictionary interface but can also be used as priority queues. Internally they have a pointer-based, list-like structure which allows to experiment with Concurrent Haskell's different synchronization approaches (locks, compare-and-swap (CAS), and software transactional memory (STM), see Section 2.3). Hence, we can examine the advantages and disadvantages of high-level approaches to synchronization in a scenario with high contention. We compare the different synchronization approaches regarding their parallel performance on the one hand and their development and debugging complexity on the other. To briefly describe the results of our experiments, the low-level lock-based and CAS-based approaches scale comparably well. The high-level STM variant is always slower by some order of magnitude since it does not scale well if a synchronized operation takes a long time. A coarse-locked heap-based variant [36] is developed for comparison with a straightforward implementation of a priority queue. It scales as well as the low-level approaches due to its better cache locality. Regarding the complexity of development, STM has been much easier to apply and easier to debug. Unfortunately, STM does not yet support the scalability that is necessary for developing thread-safe data structures in general. For today's use cases and a relatively low number of cores, a rudimentary and straightforward implementation, i.e. coarse-locked heaps, can also be sufficient.

Data structures work on a fairly low level regarding the implementation of a whole software system. Design patterns are more abstract as they try to formalize software engineering principles within a framework of a common vocabulary [60]. Such patterns also exist for parallel and functional programming [140, 167]. One of the best-known and most relevant parallel patterns is the **taskpool** [140]. It is used to distribute tasks to different threads. The taskpool is a building block for many parallel algorithms. We analyze different variants of taskpools which differ in their functionality, complexity of development and synchronization approach. For the description of taskpool-based algorithms we use high-level functional approaches such as typeclasses and monads to hide the underlying parallelization by means of Concurrent Haskell. For synchronization, locks as well as software transactional memory are used. The criterion for comparison is their parallel performance as well as their usability. In contrast to the previous results for the skiplist, STM scales well since synchronized operations are rather short. At the same time, it allows for an easier development.

Libraries are collections of functions which are used to solve a particular problem class. Internally they often use abstract data types and (parallel) design patterns so their design becomes more comprehensible and their implementation easier to maintain. A library can support a declarative description, i.e. the user does not have to state *how* a problem should be solved. Instead, he simply states the *properties* of the problem. Then an underlying implementation is used to compute a solution based on this description. Hence, the library becomes easier to use and problem-specific optimizations are easier to apply. To examine the concept of declarative libraries, we design and implement a prototype of such a library which is used for multi-dimensional stencil-based algorithms. *Stencil-based algorithms* work on a grid and perform computations on each grid element iteratively. A **stencil** is a pattern that defines the dependencies within such computations. The library consists of types and functions for a declarative description of stencil-based algorithms as well as a prototype for a parallelized execution platform. We analyze the scalability of the developed library using different well-known stencils. Results demonstrate that Haskell allows for a concise declarative description of stencil problems and enables their scalable parallelization.

From a user's point of view, a problem should be stated in its natural context containing a problem-specific vocabulary. This approach to describe a problem is known as a domain specific language (DSL) in software engineering [56]. For its users, the limited vocabulary is more intuitive and results in less errors during development. While developing the underlying execution platform, DSLs, like declarative libraries, offer potential for efficient optimizations. Nevertheless, DSLs are seldom developed from scratch. The implementation of a compiler as well as the whole ecosystem that a modern language provides (debuggers, profilers and libraries) is rarely justified. Instead, modern DSLs are typically *embedded* into a host language which provides an underlying infrastructure. These kinds of DSLs are called Embedded Domain Specific Languages (EDSLs). **Graph** algorithms such as finding shortest paths, clustering or matching can be described by a limited vocabulary: Graphs consist of vertices and edges, hence the number of operations and types is rather small. Nevertheless, it can be quite challenging to write and optimize graph algorithms, in particular for multicore hardware [66]. We develop an EDSL that allows for the concise description of graph algorithms. We also develop an execution platform which facilitates an automatic parallel execution of these graph algorithms. Results indicate that advanced functional concepts (e.g. monads) are particularly useful for the implementation of a concise language. Additionally, they allow to hide all details of the implementation. Concurrent Haskell allows for the implementation of the execution platform which scales very well.

1.3 Structure

The rest of this thesis is structured as follows. In Chapter 2 we describe basic terms and concepts of parallel and functional programming. We start with an introduction to parallel programming in general. Then we depict Haskell and its different parallelization and synchronization concepts. In chapter 3 we explain skiplists and their synchronization approaches and describe the implementation of thread-safe priority queues. We analyze the performance as well as the development effort of the different synchronization approaches. In Chapter 4 we examine several variants of the taskpool pattern and analyze their development effort and their scalability. In Chapter 5 we describe our declarative stencil library. We show how to describe standard stencils and describe the implementation for their parallelized computation. In Chapter 6 we give an introduction to (E)DSLs, describe the design and implementation of our EDSL, give examples of concisely formulated graph algorithms and benchmark the parallel performance. In Chapter 7 we outline possible future work and reflect on our research.

All contributions have already been published in separate papers. This is reflected in the structure of the four main chapters: after reading Chapter 2, each of the following chapters can be read independently.

1.4 Acknowledgments

I owe my deepest gratitude to Prof. Dr. Claudia Fohry. She gave me the opportunity to do research about the exciting combination of parallel programming and functional languages. While giving me the room to work my own way she was always willing to discuss my questions and ideas.

My special gratitude also goes to Dr. Clemens Grelck. I feel fortunate that he has agreed to become my second supervisor. Albeit (too) few, I really enjoyed our telephone conferences discussing various topics about my thesis.

Without Susanne Jurkowski, this thesis would not have been possible. I would like to thank her for her encouragement, understanding and patience when I was lost in the dark realms of type errors and deadlocks. From the bottom of my heart I can say that we match the type signature `love :: ♀->♀->💕`.

I would also like to thank my parents for their continuous support during all those years. Although they might have never understood why I spent so much time in front of boring gray boxes instead of going outside, they always encouraged and supported me in any way they could. Thanks!

There are numerous other people who deserve a special thank you: Claudia Huerkamp for supporting all my, sometimes rather fancy, wishes for office supplies. Raffaele Biscosi for keeping all the machines running that I have used and providing hundreds of interesting facts. The people from Kassel's Ki-Aikido Dojo helped to keep me calm and relaxed, even in times of approaching deadlines. Finally, I would also like to thank my students. Their questions and their curiosity made it possible for me to combine theory and practice.

1.5 Disclaimer

Trademarks and brand names have been used without indicating them explicitly. The absence of trademark symbols does not imply that a name or a product is not protected. All trademarks are the property of their respective owners.

Chapter 2
Foundations

In this chapter we first present an overview of important concepts and approaches of parallel programming. We also describe the key concepts of the functional programming language Haskell as well as the different concurrency constructs of Haskell which are used in this thesis.

2.1 An Introduction to Parallel Programming

In this section first we introduce basic terms of parallel programming and briefly explain a systematic approach to the design of parallel algorithms. Afterwards, we show why (functional) parallel programming is, despite much research, still difficult compared to sequential programming. Since Concurrent Haskell works on shared-memory systems only, we focus our introduction on this paradigm.

2.1.1 Basic Terms

In the following we briefly introduce some of the most common terms of shared-memory parallel programming, which are used throughout the thesis. A more thorough definition of these and related terms can be found in introductory parallel programming books, for example [123, 124, 166, 207].

In *parallel computing*, a problem is split into several *tasks* that can be run in parallel. Key reasons for parallelization include the decrease of a program's running time, and the natural parallelism in the underlying algorithm [124]. A *thread* is a single sequential flow of control within a program [63]. In a *concurrent* application, multiple threads of control are executed. Depending on the underlying hardware and scheduling, they are executed simultaneously. This allows for the parallel computation of tasks by means of threads.

One is often interested in how much faster a parallel program runs compared to its sequential counterpart under the precondition that both programs solve the same problem instance. The ratio between sequential and parallel running time is called the *speedup* S_p

$$S_p = \frac{T_1}{T_p}$$

T_1 is the running time of the best sequential implementation and T_p is the running time of the parallel variant when using p threads. In most cases a linear speedup is not to be expected due to the overhead of the parallelization. In particular, [123, 124] state three reasons for a sub-linear speedup. First, many parallel programs require time-consuming communication between threads. Second, parallel programs often have sections of non-parallelizable code (see below, Amdahl's Law). Third, thread management and task distribution induce a certain overhead. Depending on the problem, threads may become idle when tasks are not evenly distributed. On the other hand, a superlinear speedup ($S_p > p$) might occur due to a better utilization of caches, as well as for other reasons [123].

Most parallel programs include sequential sections which can not be parallelized. To determine the theoretically maximum speedup of a parallel program, *Amdahl's Law* can be used [5]. Amdahl's Law states that the maximum speedup A_p is determined by

$$A_p = \frac{1}{P_s + \frac{P_p}{p}}$$

P_s is the proportion of the program that can not be parallelized and P_p is the proportion that can be parallelized. For example, a program with a sequential proportion of only 5% which runs using 16 threads can only gain a maximal speedup of $(0.05 + \frac{0.95}{16})^{-1} \approx 9.14$.

Amdahl's Law is rather pessimistic. Fortunately, in many parallel programs the proportion of the sequential sections decreases with larger problem instances. In this case, *Gustafson's Law* is applicable [47, 69]. According to Gustafson, users want to compute solutions to their problem instances within a practical time limit. If hardware with more cores is available, users will utilize these cores to compute larger problem instances. Hence, the proportion of parallelizable work increases. Thus, an arbitrary speedup $S_p < p$ can be achieved by choosing problem instances that are large enough.

Memory access time is an important factor for the performance of (parallel) programs [45, 104]. Many shared-memory systems are based on *symmetric multiprocessing* (SMP): the access time to any memory cell is identical for each core. Other architectures have non-uniform memory access due to independent memory banks being dedicated to different cores. To improve access time, many systems have a hierarchy of different memory types which differ in their respective access time and size. A particularly fast albeit small type of memory is called *cache*. When a core accesses a memory cell, the cell's value and the values of adjacent cells are loaded into the core's cache; the set of loaded cells is called *cache line*. Subsequent accesses to these memory cells from this core are served by the cache. If one of the memory cells is modified by another core, the cache line is reloaded. Keeping data of the cache synchronized with the shared memory is a complex topic and we refer to [104, 195] for more information. While caches allow for improved access times, they also increase the difficulty of performance-oriented programming. Two aspects which have to be kept in mind are false sharing and data locality. *False sharing* occurs when threads share a cache line unintentionally. Even in case threads never share data, their common cache line will be updated if one thread updates its local data. By paying attention to *data locality*, data necessary for a computation should be stored in memory continuously (spatial locality) and accesses to the same data should be concentrated in time (temporal locality). Hence data is loaded into the cache with few memory accesses.

2.1.2 Designing Parallel Algorithms

Understanding the design phase of a parallel algorithm eases understanding the design and implementation decisions we have made in subsequent chapters. Therefore, we describe a three-step approach for designing parallel algorithms for shared-memory systems (see Figure 2.1). It is loosely based on similar approaches, e.g. by Grama et al. [123] or by Foster [55]. In the following we explain the single steps in detail.

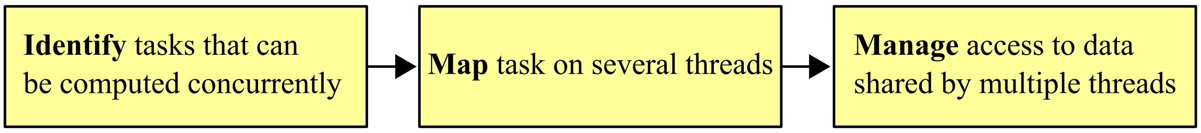

Identify tasks that can be computed concurrently
Map task on several threads
Manage access to data shared by multiple threads

Figure 2.1: Three-step approach for the parallelization of algorithms.

Identifying Parallelizable Work

In the parallelization of an algorithm the first step is the identification of tasks which can be processed in parallel. The process of identifying these tasks is called *decomposition*. Depending on the algorithm to be parallelized, different approaches are possible. With **recursive decomposition** tasks are generated using a divide-and-conquer strategy: a problem is recursively divided into a set of sub-problems until a subproblem's size reaches a certain limit. For algorithms which work on large data sets, **data decomposition** is a viable strategy: depending on the problem, either the structure of input data, output data, intermediate data, or a combination of them is used for partitioning and thus task derivation. **Exploratory decomposition** is used for search or optimization problems. Tasks are derived by partitioning the search space. For algorithms with computationally expensive branches, **speculative decomposition** can be used. By this approach, different tasks (pre-)compute different branches while the right one is selected. Note that efficient parallelization might require the combination of different decomposition strategies.

Mapping Techniques

After parallelizable tasks have been identified, they have to be mapped to available threads. An efficient mapping aims at assigning tasks such that all threads are busy (load balancing), caches are well utilized (data locality), and the overall computation is finished as quickly as possible. While perfect load balancing is desirable, its computation is $\mathcal{NP}$-complete, if possible at all [38]. Hence, different heuristic approaches have been developed. Depending on the task characteristics (e.g. time for task generation, task size, a priori knowledge of task size, or size of associated data) different approaches are appropriate. These approaches can be classified into two categories, static and dynamic.

By using a **static mapping**, tasks are assigned prior to the execution of the algorithm. A common static mapping is data partitioning, which is especially useful if data decomposition has been used for task identification. There are different approaches which differ in their compromises between an effective load balancing and good data locality [150]. By using a *block distribution*, contiguous data *blocks* are distributed equally among all threads. Hence, data locality is good, but this distribution can lead to load imbalances if the computation time varies too much between blocks. An approach which favors load balancing over data locality is a *cyclic distribution*, where the block size is chosen to be minimal, such that many more blocks than threads are available and are assigned in a round-robin manner. A good compromise between these two approaches is a *block-cyclic distribution*. For this distribution, data is divided into more blocks than threads, although not as many as in a cyclic distribution. Then these blocks are mapped in a round-robin manner.

If tasks are generated dynamically, or if the computation time varies much between tasks, a **dynamic mapping** is often more efficient. By using a *centralized dynamic distribution*, all available tasks are managed by means of a synchronized central data structure called taskpool. If a thread wants to acquire a task or store a new one, it accesses the taskpool. While this approach can often be implemented in a straightforward way, special care has to be taken to ensure scalability and to prevent load imbalances. To increase scalability for the price of a more complicated implementation, a *distributed dynamic distribution* is viable. Available tasks are distributed among the threads and each thread may send tasks to other threads and receive tasks from others.

There are many more and sometimes quite problem-specific mapping techniques. An elaborate overview of these techniques as well as a detailed description of the mentioned ones can be found in [55, 123, 166] for example.

Access Management

Since we focus our description on shared-memory parallel programming, all threads share a common address space and thus can access all data immediately. To prevent corruption, e.g. due to inconsistent write operations, synchronization of concurrent accesses is necessary. In general, the correct and efficient handling of concurrent access to shared data is quite platform- and language-specific. Therefore, problem-independent guidelines are difficult to give. Various techniques for data synchronization by means of Concurrent Haskell are described in detail in Section 2.3.2.

2.1.3 Why Parallel Programming is Hard

In this section we describe why parallel programming is more difficult compared to sequential programming [16, 62, 142, 189, 190]. Our discussion is organized in three parts. These parts reflect the lifetime of a parallel program: planning and implementation, testing and debugging, and optimization. In addition we briefly discuss the lack of information and education about parallel programming. Finally, we describe reasons why parallel programming might be even more difficult in the functional programming language Haskell than in imperative programming languages.

Planning and Implementation

In the phase of designing a new parallel program, difficult decisions have to be made (see Section 2.1.2). In contrast to sequential programming, well-tested and published knowledge about best practices for these decisions are still in their infancy. Therefore, personal experience with parallel programming is still one of the most important factors for an efficient parallel design (see below). This is even more so if an already existing algorithm or program should be parallelized, since many algorithms were designed with single-core performance in mind. For example, there might be artificial and unnecessary task dependencies, but which improve the performance on single cores. Another aspect which complicates porting of a sequential algorithm is that, depending on the parallel programming approach, intermediate steps can not be tested and examined independently. An algorithm to be parallelized often has to be refactored until its parallelization is completely done.

Testing and Debugging

The parallel programming model for shared-memory systems which is well-supported by many mainstream languages uses threads as well as manual synchronization of shared data. Often, difficult to control scheduling of threads and subsequently the non-determinism of the data access patterns cause various problems: First, reasoning about a program becomes more difficult, since the order of execution of instructions is not determined. Second, debugging and testing becomes more difficult. For testing, in sequential programming unit tests are usually used. While these tests can guarantee the correct functionality in the context of a single-threaded program, they are nearly useless to find errors like deadlocks or data races which occur because of the scheduling. Traditional debugging techniques, e.g. debuggers or even the simple printing of expressions might lead to a different scheduling and thus the disappearance of the observability of the bug.

Due to the rise of parallel programming in the last years, nowadays there is tool support for parallel debugging for mainstream languages [107]. In some parallel programming systems, e.g. OpenMP, the mentioned problems are not as severe since the developer has better control over some aspects of the parallel execution.

Optimization

Optimizing parallel programs is more difficult than optimizing their sequential counterparts. Despite a relatively high degree of abstraction in modern languages and high-level libraries, the underlying hardware is still an important factor for a program's execution time. For example, the cache behavior of a program is generally important for its efficient execution [45]. In general, optimizing for the purpose of a good, hardware-independent performance on today's multicore processors is complicated, since they might share caches, their cores are connected differently etc.

Lack of Information and Education

Despite the fact that multicore processors are common nowadays, parallel programming is still not mainstream regarding the education of computer science students. Students' curriculum consists mostly of programming courses for sequential programming. In many cases a course on parallel programming is only optional. Furthermore, it is still difficult to find books or online resources about parallel programming, especially for beginners. The current introductory books focus mainly on high-performance computing or numerical problems.

Additional Problems with Parallel Programming in Haskell

Parallel programming in Haskell causes additional problems and the mentioned issues are emphasized, respectively.

Lack of tools. The commercial interest in (parallel) programming in Haskell is negligible. Besides, in our opinion, the development of tools is disrespected in the academic community. For these reasons there are only few tools for the optimization of parallel programs. Except of Threadscope [109], we are not aware of any parallel profilers or other approaches to analyze parallel performance.

Lack of information. The amount of online information about parallel functional programming is smaller than the information for mainstream languages. While introductory tutorials and books exist (e.g. [112, 127, 134]) and of course research papers are published, intermediate textbooks simply do not exist.

Lazy Evaluation. Haskell is a lazy evaluated language. This evaluation strategy is quite beautiful for sequential programming but it complicates parallel programming in Haskell. We discuss this problem as well as possible solutions in detail in Section 2.3.3.

2.2 The Functional Programming Language Haskell

We provide an introduction to the functional programming language Haskell. First, we give an abstract of the history of Haskell. Then we present an overview of interesting and important concepts of the Haskell language (laziness, side-effect free programming and the type system). We close this section mentioning introductory and advanced resources for further reading.

2.2.1 History

In 1930, Alonzo Church introduced the λ-calculus, which is a formal system to describe different aspects of computability by means of functions [204]. This system forms the basis for many functional programming languages. The ideas of the λ-calculus were used for the development of LISP (LISt Processing) by McCarty in 1950 [141]. In the following decades, many functional languages and concepts were developed, although none of these were widely-used, except LISP. In 1985, the lazy functional language *Miranda* was introduced and became quite popular [198]. Miranda was a commercial product hence usage and further development by the academic community were nearly impossible. At the Functional Programming Languages and Computer Architecture Conference in 1987, researchers discussed the general situation of functional programming development in the scope of an informal meeting:

> ”[...] there had come into being more than a dozen non-strict, purely functional programming languages, all similar in expressive power and semantic underpinnings. There was a strong consensus at this meeting that more widespread use of this class of functional languages was being hampered by the lack of a common language.” [113]

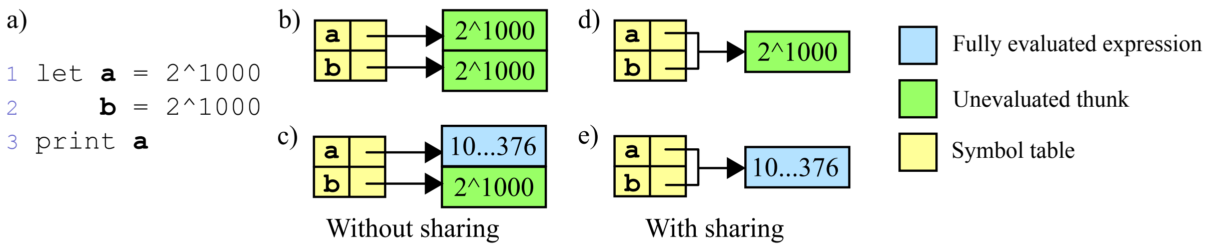
This discussion led to the definition of Haskell (named after the logician Haskell B. Curry). The first version of the language definition (Haskell 1.0) appeared in 1990. Haskell 98, the first complete language standard enclosing a definition of the standard library, was published in 1998 [113]. The latest standard is Haskell 2010 [135]. A detailed report of Haskell's historical development including the languages that effected its development can be found in [100].

2.2.2 Important Language Concepts

As explained in the following, Haskell is non-strict (lazy) as well as side-effect free. It is based on an expressive type system with automatic type inference. In the following sections, we briefly describe each of these aspects.

Lazy Evaluation

A programming language with *lazy evaluation* postpones the computation of an expression until the expression's value is actually needed. In the meantime only a *thunk* is generated. This thunk represents the computation (see Figure 2.2).


a)

```
1 let a = 2^1000
2     b = 2^1000
3 print a
```

Figure 2.2: Illustration of lazy evaluation without and with sharing. a) source code. b) symbol table after the execution of lines 1 and 2 (without sharing). c) symbol table after the execution of line 3 (without sharing). d) analogous (with sharing) e) analogous (with sharing)

A technique often mentioned in the context of lazy evaluation is *sharing*. If the evaluation strategy supports sharing, equal expressions share the same thunk. Thus unnecessary computations are prevented. In the framework of the compiler used in this thesis, Glasgow Haskell Compiler 7.0.3 [92], sharing of common subexpressions is supported in `let`-bindings [76]. More sophisticated sharing strategies must be implemented manually.

Advantages. In a lazily evaluated programming language, unnecessary computations, for example, the evaluation of parameters in a function call, are prevented. One exemplary use case is the definition of additional control structures without any language support for macros. Consider the function `when` [75]

```
when :: Monad m => Bool -> m () -> m ()
when p s =  if p then s else return ()
```

In the framework of a programming language with strict evaluation, the second parameter `s` is evaluated unconditionally.

Lazy evaluation can not only be used to define control structures but it also allows for an efficient composition of functions. The function `any` returns `True`, if any of the elements in the list meets the given predicate [11]:

```
any :: (a -> Bool) -> [a] -> Bool
any p = or . map p
```

With a lazily evaluated language, the evaluation of the expression `any (>100) [1..10^6]` terminates after the list element `100` has been examined. With a strict language all values of the list are examined.

Our last example examines infinite data structures, in particular infinite lists. In this case, lazy evaluation allows a concise and elegant formulation of self-referencing lists. For example, the Fibonacci series can be defined recursively by

```
fibonacci :: [Int]
fibonacci = 0 : 1 : zipWith (+) fibonacci (tail fibonacci)
```

Although the function definition has no explicit termination condition, lazy evaluation strategy ensures that only necessary elements of the list are computed: `head fibonacci`, for instance, immediately evaluates to `0` without any further computation.

A more thorough introduction to the advantages of lazy evaluation (and higher order functions) can be found in the seminal paper *Why Functional Programming Matters* by John Hughes [101].

Disadvantages. Lazy evaluation complicates reasoning about the time- and space-behavior of a program, especially compared to reasoning under strict evaluation. Consider this definition of the `foldl` function

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl _ y []      = y
foldl f x (y:ys) =
    let x' = f x y in
    foldl f x' ys
```

as well as the expression `foldl (+) 0 [1..10000]`. Due to lazy evaluation, instead of aggregating the intermediate result in successive calls, many thunks are generated :

```
   foldl (+) 0           [1..10000]
-> foldl (+) (0+1)       [2..10000]
-> foldl (+) ((0+1)+2)   [3..10000]
         ...                   ...
```

These thunks are only evaluated if the result of the expression is actually needed, e.g. by printing it. While this does not cause any memory problems for small lists, computations with larger lists can fail due to excessive memory consumption.

Lazy evaluation also complicates parallel programming. If a program computes expressions in parallel, only thunks for these expressions are generated unless explicit evaluation is enforced. Techniques to allow for the strict evaluation of expressions are described in Section 2.3.

Programming Without Side Effects

Haskell is a *side-effect free* (pure) language: each function returns the same result given the same input in each call. In consequence, expressions in Haskell are *referentially transparent*. That means that any expression can be replaced by its computed value. For example, in the function

```
f :: [Int] -> Int
f = (+2^20) . sum . map (\x -> x + 2^20)
```

the expression `2^20` can be bound to a variable `n` and the function can be rewritten as

```
f :: [Int] -> Int
f xs =
    let n = 2^20 in
    (+n) $ sum $ map (\x -> x + n) xs
```

Note that this transformation can also be done in reverse: each occurrence of a variable can be replaced by its definition. This allows for safe refactoring of functions. In addition, reasoning about properties of functions is easier because external factors, e.g. global variables, can not change the semantics of a function.

Unfortunately, side effects are an important part of many computations. A function that returns input from the keyboard would be useless if it had to be pure. On each call it would have to return the same input. The concept for handling side effects in Haskell is based on *monads*. Since monads are a well-discussed topic we refer to [85, 99, 127] as well as Section 2.2.3 for a list of resources that explain monads in detail. Haskell's most important monad for concurrent programming and mutable data structures is the `IO` monad [112]. There are several types of computations that are executed in the context of the `IO` monad, for example computations that have to execute input- or output operations, or computations that have side effects on mutable data. The type system ensures that pure functions can be called by non-pure functions but not vice versa.

Note that a programming language which supports referential transparency is a strong candidate for today's multicore systems: since evaluations of different expressions can not influence each other, expressions can be evaluated in parallel. While this approach works in theory, practice has shown that in many cases *too many* parallelizable evaluations are generated[71]. For most of these evaluations, parallelization is not worth the additional overhead.

Haskell's Type System

Haskell features a static type system, i.e. the types of expressions are checked at compile time. This type system supports many sophisticated techniques and concepts, e.g. automatic type derivation, type classes and polymorphic types. We briefly illustrate these techniques and concepts by means of examples.

Polymorphic Types. Polymorphic types allow the user to generalize the type of a function call by means of a type variable. A *type variable* indicates that any type can be used. Hence, a function's signature can contain not only types but also type variables. For a polymorphic function the standard example is the `map` function

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Its first parameter is a function `f` that receives a value of type `a` and returns a value of type `b`. By means of `map`, the function `f` is applied to all elements of a list of values, whereby all list elements have type `a`. Thereby, a list of values of type `b` is generated.

Type Classes. A type classes defines a set of functions to enforce their implementation. The implementation must be in a type that is an instance of the type class. A popular example is the type class `Eq`. It defines the functions `==` and `/=`:

```
class Eq a where
  (==) :: a -> a -> Bool      -- test on equality
  (/=) :: a -> a -> Bool      -- test on inequality
```

A developer can define an instance of `Eq` for his own data types. Then all functions with a type variable which must be an instance of `Eq` can be used. For the exemplary algebraic data type `D`

```
data D = A | B Int
```

an instance of `Eq` is defined by

```
instance Eq D where
    A   == A   = True
    A   == _   = False
    B n == B m = n == m
```

The function `elem :: Eq a => a -> [a] -> Bool` checks if a list contains a given element by comparing elements pairwise. After defining the `Eq` instance for `D`, the expression `elem (B 3) [A, B 6, A, B 3]` is valid, for example.

Automatic Type Inference. In many traditional programming languages with a static type system, e.g. Java and C++, types of variables have to be declared explicitly. In contrast, Haskell's type inference system uses the Hindley–Milner type inference mechanism [39, 94, 144] to derive the types of expressions at compile time automatically (unless they are stated explicitly). Hence the developer has the chance to omit type signatures while finding type-based programming errors is still possible. We illustrate the basic idea of this mechanism by an informal example. Consider the function

```
f x n =
 let cs = show x in
 replicate n cs
```

In this case the goal is to infer the type of the function `f`. For this purpose, the inference engine tries to find a consistent set of types that match the parameters of the called functions. Assuming that the function `replicate` has the type `Int -> a -> [a]`, which is defined by the standard library, the first parameter of `replicate` must be of type `Int`. Therefore `n` must have type `Int`. Since the second parameter of `replicate` is polymorphic, no information for `cs` can be derived by the call. But `cs` is bound to the expression `show x` and the function `show` has the type `Show a => a -> String`. Hence, two informations can be derived. First, `x` must be an instance of the `Show` type class and second, `cs` is of type `String`. Therefore the function `f` must have the type `f :: Show a => a -> Int -> [String]`. A more through discussion of Haskell's type checker (and an implementation in Haskell itself) can be found in [110, 154].

2.2.3 Further Reading

In this section we could only give a short insight into the programming language Haskell. Many different books, research articles and blog articles have been written, and we especially recommend the following resources to gain a deeper understanding of Haskell and functional programming in general.

An introduction is Graham Hutton's *Programming in Haskell* [103]. This book covers most of the important concepts of functional programming (in Haskell), e.g. recursive functions, lazy evaluation, higher-order functions, and monadic effects. An introductory book which focuses on the concept of monads and less on the theoretical details of functional programming in general is Lipovaca's *Learn You a Haskell for Great Good!* [127]. A book that demonstrates that Haskell is by no means only an academic language is *Real World Haskell* by O'Sullivan et al. [157]. It covers binary data parsing, barcode recognition, database access as well as multicore and network programming, to name just a few examples.

Haskell's website (and the corresponding wiki) presents an overview of recent developments of the language as well as different implementations. It also provides a general list of links for numerous functional programming topics [89]. The Haskell mailing lists *haskell-cafe* and *haskell-beginners* are both active and cover topics from recent developments in functional programming research to beginner questions [84]. Finally, the internet relay chat (IRC) channel `#haskell` allows beginners as well as advanced users to discuss various topics about Haskell in realtime [83].

2.3 Parallel Programming in Haskell

In this section we describe the different approaches for parallelization in Haskell. We focus on Concurrent Haskell as we introduce Haskell's thread model and the existing approaches to synchronize mutable shared data. As stated before, lazy evaluation complicates an efficient parallelization. We elaborate on the reasons and show approaches to handle lazy evaluation in the context of parallelization. We conclude with a brief overview of other parallelization concepts in Haskell which are outside of Concurrent Haskell. Further information about these topics can be found in [112, 114, 134].

2.3.1 Threads in Concurrent Haskell

Haskell's parallel runtime systems allows for the execution of several independently running control flows at the same time. We call each control flow a *thread.* Since threads run concurrently, they can be used to implement parallel algorithms. A thread is generated (forked) by calling the function

```
forkIO :: IO () -> IO ThreadId
```

This function requires an arbitrary function `f` as its parameter. The function `f` will be executed in a new thread [82]. `forkIO` returns a thread identification that can be used to control the thread's state, e.g. terminating it. The following example forks ten threads which run concurrently. Each thread prints a particular number ranging from 0 to 9 and then terminates:

```
let numberThreads = 10
    range         = [0..numberThreads-1]
forM_ range $ \number -> forkIO $
  print number
```

The thread that executes the `main :: IO ()` function of a program is called the *main thread.* A specific feature of Concurrent Haskell is that the main thread does not wait on the termination of forked threads. Instead, the program (and thus all threads) finishes if the main thread finishes. Other languages, e.g. Java, wait until all spawned threads are finished [152]. Although this complicates a fork-join-based parallelization, it is straightforward to implement a modified fork-function that spawns a number of threads and awaits their termination (see Section 2.3.2)

2.3.2 Data Synchronization

There are three ways to synchronize mutable data which is shared by multiple threads: with `IORef`s, with `MVar`s, and with software transactional memory. These ways differ in their level of abstraction and their performance, and are described in the following.

Synchronization with `IORefs`

An `IORef`, defined in the module `Data.IORef` [90], is a container for a mutable variable inside the `IO` monad. The following example illustrates how an `IORef` is generated, read and modified by storing a new value:

```
ref <- newIORef 0 :: IO (IORef Int) -- Generate a new IORef
                                    -- for Int storage
print =<< readIORef ref             -- Read and print
                                    -- (prints 0)
writeIORef ref 1                    -- write 1
print =<< readIORef ref             -- Read and print
                                    -- prints 1
```

The function `modifyIORef :: IORef a -> (a -> a) -> IO ()` simplifies the modification of a value stored in an `IORef`. It allows to change the value by specifying a function which consumes the old value and returns a new one. For example, incrementing the value stored in the `ref` variable can be written as

```
modifyIORef ref (\n -> n + 1)
```

The modification of `IORef`s by means of the aforementioned operations, in particular `modifyIORef`, is not thread-safe. Between reading the value, applying the update function, and writing the updated value, the value might be altered by another thread. To ensure a thread-safe modification, the function `atomicModifyIORef :: IORef a -> (a -> (a,b)) -> IO b` can be used. It ensures that the three mentioned operations of `modifyIORef` (read, apply function, and write) are atomic and thus prevents corruption.

The possibilities for thread-safe modifications of `IORef`s are limited, since one can only use pure functions to alter the value. While a pure function is sufficient to implement compare-and-swap operations [191, 206], this limitation complicates the implementation of sophisticated synchronization approaches, as we will see later. A more common and flexible technique for synchronization in Concurrent Haskell are `MVar`s which we describe in the next section.

Synchronization with `MVar`s

Similar to an `IORef`, an `MVar` is used to store a mutable value [160]. However, compared to `IORef`s, `MVar`s provide support for thread-safe modifications and can also be used to block threads. That is, an `MVar` is in one of two states. Either it is *empty* or *full*. If a thread tries to read from an empty `MVar`, this thread is blocked until a value is stored by another thread. If reading succeeds, then the `MVar` becomes empty and the read operation returns the previously stored value. If a thread tries to write to an already full `MVar`, this thread is blocked until the value is read by another thread. Otherwise, if a thread tries to write a value to an empty one, the value is stored and the `MVar` is full. Figure 2.3 illustrates these concepts.

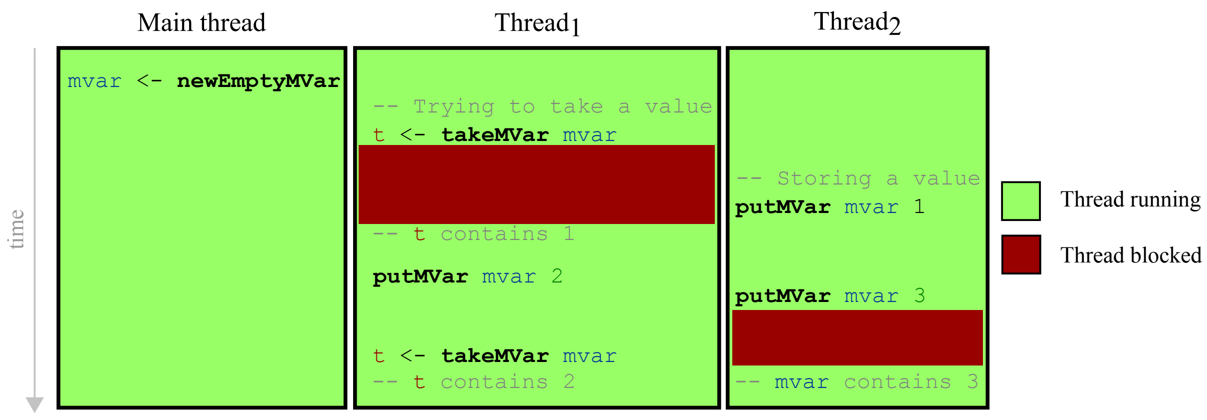


Figure 2.3: Illustration of thread blocking on an empty as well as a full `MVar`. Thread$_1$ blocks since the `MVar` is empty, Thread$_2$ blocks since the `MVar` is full.

As already mentioned in Section 2.3.1, the main thread does not wait until forked threads are terminated. As a use-case of `MVar`s we demonstrate how a function `forkForList` can be defined. The function forks a number of threads such that each thread processes one element of a given list. The calling thread waits until all list elements have been processed. The function `forkForList` is implemented as follows. First, a list of empty `MVar`s is generated:

```
forkForList :: [a] -> (a -> IO ()) -> IO ()
forkForList xs f = do
  let len = length xs
  mvars <- replicateM len newEmptyMVar
```

Then threads are forked. After a thread has finished its work, the respective `MVar` is filled:

```
  forM_ (zip xs mvars) $ \(x, mvar) -> forkIO $ do
    f x
    putMVar mvar ()
```

By trying to read the initially empty `MVar`s, the main thread waits until all threads are finished:

```
  mapM_ takeMVar mvars
```

`MVar`s are also the basis for more advanced concurrency primitives, e.g. unbound channels [160]. An (unbound) channel (`Chan`) can store multiple elements of the same type in a thread-safe manner. Similar to an `MVar`, a thread is blocked if it tries to read from an empty channel. However, storing a value is always possible. Figure 2.4 on page 30 presents an overview of common channel operations.

```
newChan      :: IO (Chan a)
readChan     :: Chan a -> IO a
writeChan    :: Chan a -> a -> IO a
-- (deprecated due to possible race conditions)
isEmptyChan  :: Chan a -> IO Bool
```

Figure 2.4: Overview of common `Chan` operations.

Despite supporting more sophisticated synchronization than `IORef`s, `MVar`s still work on a fairly low level. In contrast to that, a high level approach for synchronization is software transactional memory.

Synchronization with Software Transactional Memory

Software transactional memory (STM) is a modern approach for synchronizing concurrent access to shared data. STM adapted the idea of a *transaction* from database research to combine multiple modifications of shared data and handle them as one operation. In this section we describe Haskell's STM primitives [72, 154] as well as some examples of its usage.

General approach. In the framework of STM, shared mutable data is stored in a transactional variable. Such variables can only be modified within a transaction. If concurrent transactions modify values inconsistently, only one transaction succeeds and updates the shared data by its modifications. All other transactions are restarted with the updated values (see Figure 2.5).

STM primitives. In the Haskell implementation, STM's key type is the `STM` monad. In this monad only function calls to modify transactional variables and the evaluation of pure functions are possible. In particular, the monad does not permit the execution of functions of the `IO` monad. This is important since transactions might be restarted several times and the effects of `IO` functions might be irreversible.

Transactional variables have type `TVar`. For executing transactions, the function `atomically` is called. Figure 2.6 on page 32 presents an overview including the respective function signatures; the operations `retry` and `orElse` are described below. We can now formulate the example depicted in Figure 2.5:

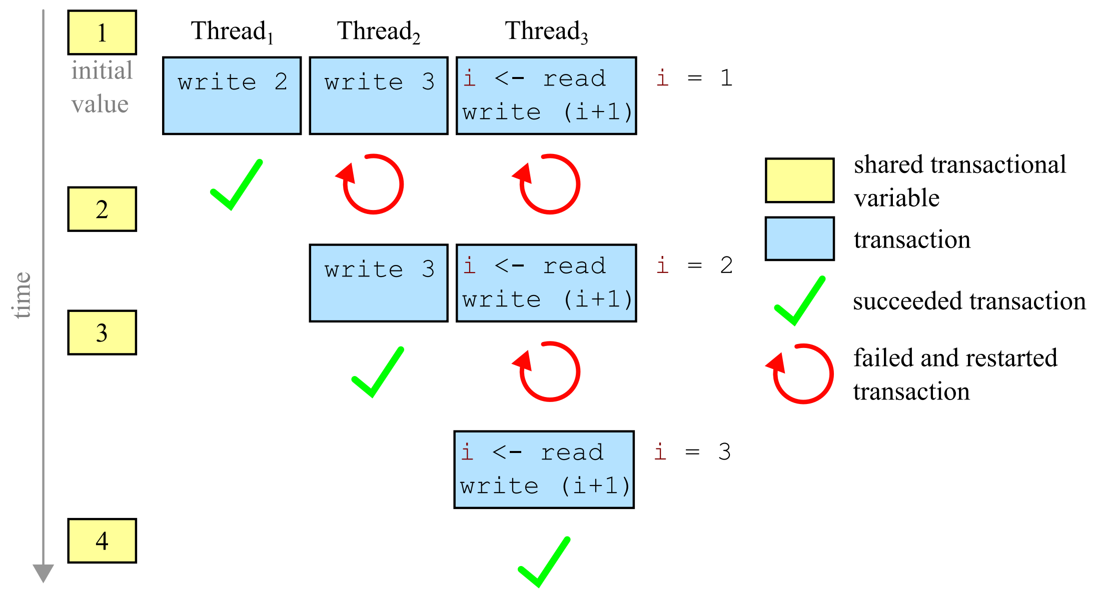


Figure 2.5: Illustration of software transactional memory. Note that in successive runs the order of succeeded and failed transactions might be different.

```
main :: IO ()
main = do
    tvar <- atomically (newTVar 1)
    forkIO $ atomically $          -- Fork Thread 1
        writeTVar 2
    forkIO $ atomically $          -- Fork Thread 2
        writeTVar 3
    forkIO $ atomically $ do       -- Fork Thread 3
        i <- readTVar tvar
        writeTVar (i+1)
```

STM operations. To allow more control over the restart of a transaction, or over the composition of two transactions, there are the functions `retry` and `orElse`, respectively.

The function `retry` blocks the current thread until one of the transactional variables, which have already been read in the current transaction, changes. When one of these variables was changed by another thread, the transaction is restarted. In contrast to automatically restarted transactions, a finer control and thus less contention can be achieved. Consider the following example which manages the synchronized access to an integer-valued resource [72]:

```
type Resource = TVar Int
putR :: Resource -> Int -> STM ()
```

```
-- Transactional variables
data TVar a
newTVar   :: a -> STM (TVar a)
readTVar  :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()

-- Executing transactions
atomically :: STM a -> IO a

-- Controlling transactions
retry  :: STM a
orElse :: STM a -> STM a -> STM a
```

Figure 2.6: Overview of common STM operations [72].

```
putR r i = do
    v <- readTVar r
    writeTVar (v+i)

getR :: Resource -> Int -> STM ()
getR r i = do
    v <- readTVar r
    if (v < i)
        then retry
        else writeTVar r (v-i)
```

If the function `getR` is called, the underlying transaction pauses for `v < i`. That means, the thread is blocked until the variable `r` is modified. Then the transaction is restarted.

The function `orElse` composes two transactions, i.e. the expression `a `orElse` b` means that the transaction `a` is run first. If `a` has to be restarted, the transaction `b` is started and the effects of `a` are abandoned. If `b` is retried, the composed transaction is retried as a whole. Furthermore, transactional variables read by either `a` or `b` are watched for modifications. Thus, a non-blocking `getR` can be implemented by [72]:

```
nonblockGetR :: Resource -> Int -> STM Bool
nonblockGetR r i =
    (get r i >> return True) `orElse` return False
```

2.3.3 Lazy Evaluation and Concurrency

Lazy evaluation complicates parallel programming by means of Concurrent Haskell. If the evaluation of an expression is not actually forced within a thread, only a thunk is generated. Hence, the actual computation could be started long after the thread is finished. The following example illustrates this problem. Although two threads are started to perform two computations, the results of the computations are only evaluated within the main thread when the result is printed:

```haskell
main :: IO ()
main = do
     var1 <- newEmptyMVar
     var2 <- newEmptyMVar

     forkIO $ do
          let a = <...expensive computation> :: [Int]
          putMVar var1 a
     forkIO $ do
          let b = <...expensive computation> :: [Int]
          putMVar var2 b

     a <- readMVar var1
     b <- readMVar var2
     print (sum var1 + sum var2)
```

To perform the expensive computations within the forked threads, their evaluation has to be enforced. The evaluation of an expression can be in different states (forms): An expression in *weak head normal form* is evaluated to its outermost data constructor and its subexpressions may or may not be evaluated. In contrast, an expression in *normal form* can not be evaluated any further. The common approach to enforce an evaluation to weak head normal form is to use the function `seq :: a -> b -> b`. It evaluates its first parameter to weak head normal form and returns its second element. Unfortunately, for parallel computations weak head normal form is not sufficient. For example, in the weak head normal form of a list only the list's first element is evaluated; for the `Maybe` type it is only determined if the value reduces to `Just` or `Nothing`, etc. For a deep evaluation, i.e. to normal form, the common technique is applying the function `deepSeq`. The expression `a ‘deepSeq‘ b` evaluates `a` to normal form and returns the (lazily evaluated) value of `b`. In the given example, the parallel computation of `var1` and `var2` can be enforced by rewriting the computation as follows:

```
    forkIO $ do
        let a = <...expensive computation> :: [Int]
        a `deepSeq` putMVar res1 a
    forkIO $ do
        let b = <...expensive computation> :: [Int]
        b `deepSeq` putMVar res2 b
```

Technically, the function `deepSeq` uses the typeclass `NFData` to enforce evaluation: Each type `a` that should be evaluated by `deepSeq` must be an instance of `NFData` and therefore has to define a function `rnf :: a -> ()`. The function call `rnf x` should reduce `x` to normal form by using a combination of `deepSeq` and `seq`.

2.3.4 Other Parallelization Approaches in Haskell

Outside Concurrent Haskell, Haskell supports other techniques for parallel programming. In this section we briefly describe three high-level approaches: semi-explicit parallelization by means of the `par` function(s) [114, 137], the recently developed `Par` monad [139], and Data Parallel Haskell [24].

Parallelization of Pure Functions with `par`

Pure functions seem to be ideal for (automatic) parallelization. Since no side effects can occur, all expressions can be evaluated in parallel, at least theoretically. Unfortunately this approach does not scale well, because usually too many small tasks are generated. Instead, the family of `par`-functions (see Figure 2.7) allows us to *annotate* expressions which are worthwhile to be parallelized. Since this approach has been subsumed recently

```
par  :: a -> b -> b
pseq :: a -> b -> b
```

Figure 2.7: Overview of functions for semi-explicit parallelization.

by the `Par` monad (described in the following section), we keep the following description short. The expression ``a `par` b`` notifies the parallel runtime system that the evaluation of the subexpression `a` in parallel with the subexpression `b` is worthwhile. The result of ``a `par` b`` is the result of `b`'s evaluation. Note that the parallel runtime system does not have to evaluate expressions in parallel. Instead, this is merely a suggestion. Note also that the previously mentioned problem about lazy evaluation still remains. But it can be resolved by the `pseq` function. The `pseq` function enforces the evaluation of its first subexpression. In ``a `pseq` b``, for example, `a` is evaluated to weak head normal form and the result of lazily evaluating `b` is returned. Although this approach works well, it is quite cumbersome in practice. That is especially true for a functional language which has many functions to work on lists, e.g. `map` and the functions in the `fold`-family. To allow for an easier formulation of parallelization for list-based algorithms, strategies are used [137]. A strategy defines how a particular expression should be evaluated. Combined with new higher-order functions, e.g. `parList` to evaluate list elements in parallel, parallel computations of complex data structures can be expressed more easily.

Parallelization with the `Par` Monad

Evaluating expressions by using semi-explicit parallelization combined with strategies as introduced in the previous section is still cumbersome. The parallel evaluation of data structures which are more complex than lists *"can be something of an art"* [139]. Another approach for parallelizing the evaluation of pure functions has been described recently [139]. It is based on the so called `Par` monad, in which computations are parallelized and results are collected (see Figure 2.8). A computation `c` in the `Par`

```
newtype Par a

-- Extract value from a parallel computation
runPar :: Par a -> a

-- "Fork" computations
fork :: Par () -> Par ()

-- Communicate using IVars
new :: PVar (IVar a)
get :: IVar a -> Par a
put :: NFData a => IVar a -> a -> Par ()
```

Figure 2.8: Overview of functions for parallel computations in the `Par` monad. [139]

monad is initiated by passing a function with result type `Par a` to the function `runPar`. To spawn a new computation, the function `fork` is called inside the `Par` monad. The spawned computation is executed in parallel to the computation `c`. Between different spawned functions communication is achieved by means of `IVars`. `IVars` are single assignment mutable references that block if they are read and no value has been stored.

Note that the value which is stored by the `put` function must be an instance of `NFData`. Hence it can be fully evaluated and so problems with lazy evaluation are prevented.

The following example, in which a parallel `map` function is implemented, illustrates the usage of the `Par` monad: The `spawn` function forks a new computation and returns an `IVar` which contains the result of the computation as soon as it is finished:

```
spawn :: NFData a => Par a -> Par (IVar a)
spawn p = do
  i <- new
  fork $ do
    x <- p
    put i x
  return i
```

The `parMapM` function uses `spawn` to fork a computation for each element of a list. Afterwards, results are collected:

```
parMapM :: NFData b => (a -> Par b) -> [a] -> Par [b]
parMapM f xs = do
  ivars <- mapM (spawn . f) xs
  mapM get ivars
```

Summarizing , the `Par` monad allows a deterministic and comfortable definition of the parallel evaluation of pure functions. By using `IVar`s and `NFData` combined with `put`, problems regarding late evaluation due to laziness cease to exist. Due to the recency of this approach it is still an open question if it can compete with Concurrent Haskell with respect to performance. The latter allows both the parallel computation of pure and impure functions and thus provides a greater flexibility for performance-oriented optimizations.

Parallelizing Array Computations with Data Parallel Haskell

Data Parallel Haskell (DPH) is a compiler extension to the Glasgow Haskell Compiler [92]. It supports (nested) data parallel computations [23, 24, 111] through the concept of *parallel arrays* and corresponding operations. In contrast to (lazily evaluated) lists, these parallel arrays are strict. That is, if the value of one of its elements is retrieved, the whole array is evaluated in parallel. However, similar to lists, DPH supports array comprehensions as well as many list operations that work on lists of finite length. For example, parallel computing of the dot product of two DPH arrays with the type `[:Double]` is implemented by

```
dotp_double :: [:Double:] -> [:Double:] -> Double
dotp_double xs ys = sumP [:x * y | x <- xs | y <- ys:]
```

DPH only supports a restricted number of array operations and is still work in progress. For instance it does not support type classes and defines only some of the `Prelude` functions for arrays. Particular attention must be paid when mixing code of DPH and standard Haskell code. Therefore, we decided against including DPH into our study.

Chapter 3
Thread-safe Priority Queues based on Skiplists

Although Haskell supports mutable data structures, first and foremost it is a side-effect free language. In consequence, there has not been much research about the implementation of mutable data structures, and in particular not on the different synchronization approaches that make mutable data structures thread-safe (locks, software transactional memory, compare-and-swap). We compare these approaches to each other with respect to parallel performance and development effort, referring to the example of a priority queue implemented with skiplists. For an additional comparison with a straightforward to implement priority queue, we present results for a coarse-locked heap-based priority queue.

3.1 Introduction

A priority queue is a data structure that allows the user to store and retrieve elements from a sorted set S [36]. It is often used as a basic building block for (parallel) algorithms. In this chapter we solely examine *min-priority* queues, i.e. priority queues that allow fast access to the smallest element of S. There are several and slightly different definitions of a min-priority queue, but consensus exists that it must support at least two operations such as `insert`(S, x) and `deleteMin`(S). The former inserts x into the set S and the latter removes and returns the smallest element of S.

A priority queue is called *thread-safe* if concurrently executed priority queue operations do not lead to errors such as returning any other element than the minimal one, failing insertion of an element, or leaving the priority queue in an undefined state. Being thread-safe is obviously of great importance and besides scalable performance the crucial aspect of any concurrently used implementation of a data structure.

Many different ways to implement a (thread-safe) priority queue exist. Traditionally priority queues are implemented with self-balancing binary search trees [180] or heaps [36, 102]. Although the sequential implementation of these data data structures is straightforward, their thread-safe and scalable implementation is difficult and other, less known data structures can have particular advantages. Among these alternative data structures is the *skiplist* [164]. A skiplist is a multi-level linked list data structure with shortcuts to randomly chosen elements. It has originally been invented for dictionaries, but its operations can be used for the implementation of priority queues as well. For programming languages like C and C++, skiplists outperform traditionally implemented priority queues with different synchronization approaches [53, 131, 165, 192].

For Haskell, prior to our work, there were no scalable thread-safe priority queue implementations. Moreover, to the best of our knowledge only one research paper about the advantages and disadvantages of different synchronization approaches for data structures has been published at all, and it referred to linked list [191]. Therefore we examined how different synchronization techniques scale and compare to each other for a thread-safe priority queue based on skiplists: traditional locks [160], software transactional memory [72], and compare-and-swap operations [191] (see Figure 3.1). For an additional comparison with a straightforward to develop and yet thread-safe priority queue, we also implemented a coarse-locked heap-based priority queue [36]. Unlike many other implementations, our priority queues fully support duplicates, i.e. multiple elements may have the same key. Duplicates are frequently needed in practice.

This chapter is structured as follows. In Section 3.2 we give an introduction to the

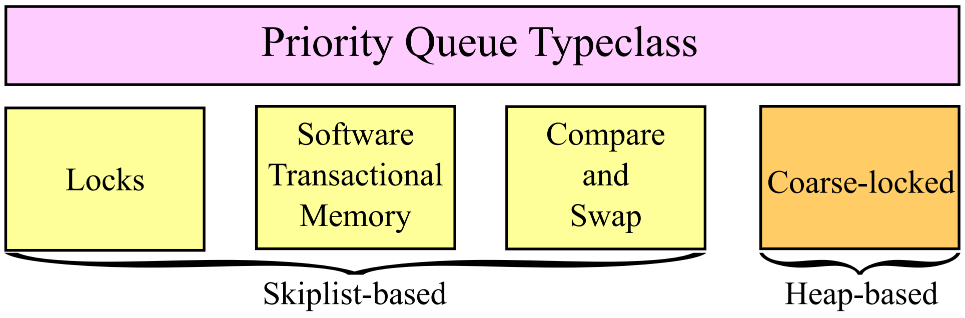


Figure 3.1: Overview of the different synchronization variants and priority queue implementations, which are examined.

(sequential) skiplist data structure and describe some of its algorithmic properties. In Section 3.3 we describe how a priority queue can be implemented with skiplist operations and we also define a typeclass for priority queue operations. In Section 3.4, we describe our different thread-safe implementations in detail and explain our test environment. In Section 3.5 we give reasons for the different benchmark scenarios that were used to compare performance, present benchmark results and analyze the outcomes. In Section 3.6 we discuss related work. Finally, in Section 3.7 we summarize this chapter and draw conclusions.

3.2 An Introduction to Skiplists

In this section we explain the design of the non-optimized and concurrency-unaware skiplist data structure as it has been invented for dictionaries [164]. We also describe support for duplicate keys and sketch algorithmic complexity. In this (sequential) version there is no need for handling synchronization. Hence we postpone the discussion of different synchronization approaches to Section 3.4.

3.2.1 The Data Structure

Basically a skiplist is a sorted linked list with additional levels for each node. By means of these additional levels partial linked lists are defined. A partial linked list typically connects only some of the nodes. Therefore it provides shortcuts from one element to distant elements by allowing to skip over elements of lower levels. In consequence, the time for typical linked-list operations is decreased. Analogous to linked lists, a skiplist has a `Begin` node which points to the first node of the list and additionally an `End` node with a key that is larger than any insertable key. Figure 3.2 on page 42 shows an example of a skiplist with three nodes of varying height.

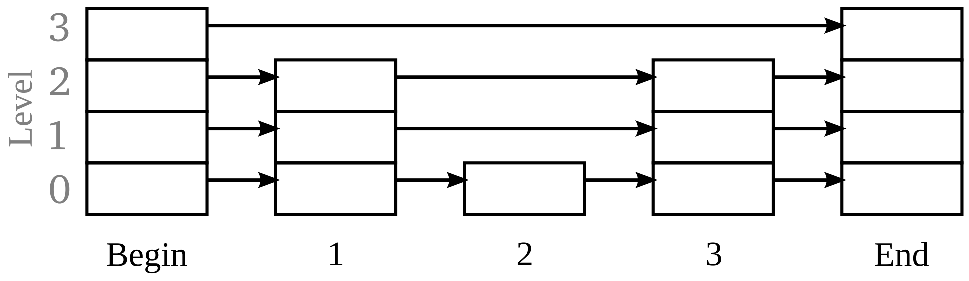


Figure 3.2: A skiplist of height 4 storing keys 1, 2 and 3. The nodes with keys 1 and 3 have height 3, the node with key 2 has height 1. There is no node with height 4.

A skiplist provides a dictionary-like interface that allows to search, insert and delete comparable key-indexed elements. The elements can be sorted either in increasing or decreasing order. Since we are interested in the implementation of min-priority queues, elements are sorted in increasing order such that the smallest element is stored after the `Begin`-node. As a dictionary, a skiplist S has to support at least the following three operations: 1) `search`(S, k) returns an element with the key k from S, 2) `insert`(S, k, v) inserts a new element v with key k into S and, (3) `delete`(S, k) removes an element with key k from S. If the key does not exist or if several nodes with the same key are stored, the behavior of the operations is implementation-dependent (see Section 3.2.2).

Searching for a given key k starts on the `Begin` node at the highest level of the skiplist. At each level the linked list is traversed by using look-ahead. If the key exists, it is found at the highest level of the first node that contains the key. If the key ahead is larger than k, traversal continues at the next lower level. If a node with a key larger than k is found at the lowest level, the given key does not exist. Figure 3.3 shows an exemplary search in a skiplist with six elements and a maximum of four levels.

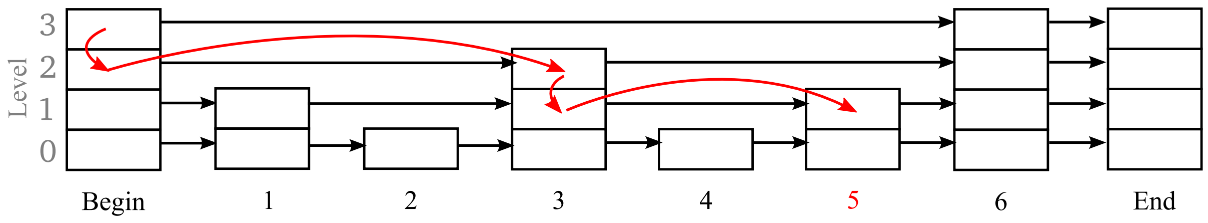


Figure 3.3: Search for the element with key 5. By using the shortcut on level 2, keys 1 and 2 are skipped. By using the shortcut on level 1, key 4 is skipped. The red arrows show the search order. Note that the search looks ahead to the next node.

Both insert- and delete-operations have to change pointers on the different levels. Therefore, a list of predecessors (one per level) is collected by a procedure that is similar to the search-operation but prior to advancing to a lower level, the last node on each

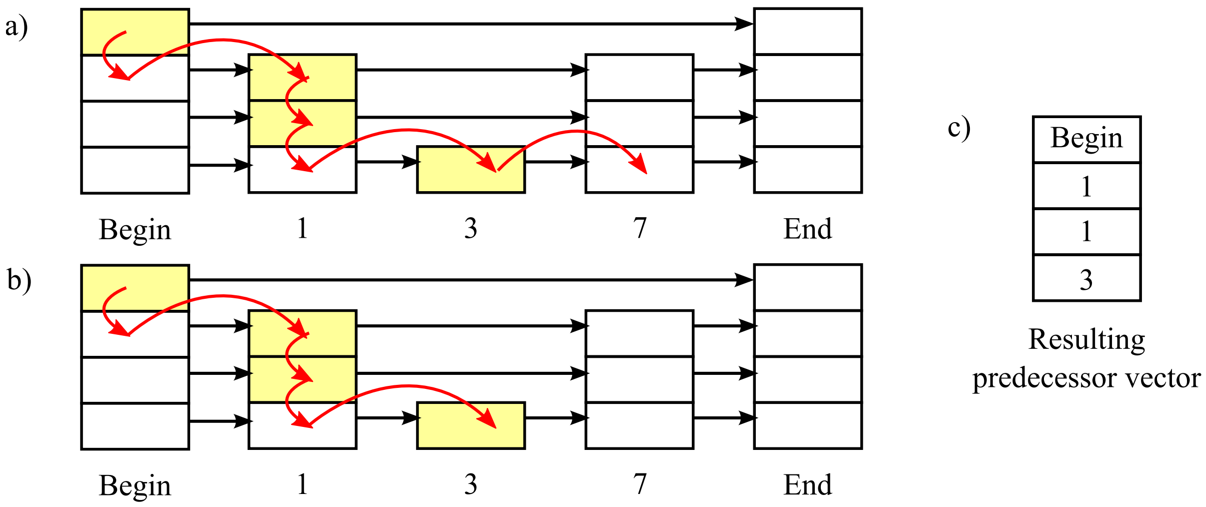


Figure 3.4: Predecessor collection for a) deleting the node with key 7. b) inserting the node with key 5. c) the resulting predecessor array. For deletion, the node to be deleted has only height 3 so the 4th level of the predecessor array will be ignored. Predecessors are marked yellow.

level is stored as the level-specific predecessor. If the key is found, traversal continues nonetheless up to the lowest level. For deletion, this approach allows to collect all direct predecessors of the node to be deleted (see Figure 3.4a)). For insertion a node with the given key and a random height is created first and then the predecessors are collected (see Figure 3.4b). After all predecessors have been found, the insert or delete operations for linked lists are applied level-wise. Note that in the sequential case the order of these level-wise modifications is of no importance.

3.2.2 Duplicate Key Handling

Many algorithms, e.g. finding shortest paths, rely on priority queues that support duplicate keys. In this case, the insertion of a key that already exists in the skiplist must not overwrite the existing one but create an additional element. In (sequential) skiplist implementations duplicate key handling is supported without an increase of the implementation complexity:

The look-ahead predecessor search traverses to the lower level if the search leads to a larger or equal key on the current level. Therefore, if an element with an already existing key is inserted, it is added in front of the others (see Figure 3.5 on page 44). The other operations work comparably, i.e. both search- and delete-operations return or remove the element with a given key that was added latest. As described in Section 3.4, duplicate support becomes more complex if threads apply operations concurrently.

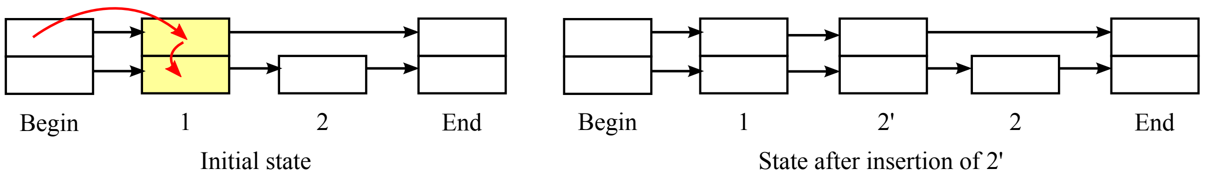


Figure 3.5: Insertion of the node 2', which has the same key as the node 2.

3.2.3 Skiplist complexity

In this section we describe briefly the computational complexity of skiplists. A more detailed formal analysis with elaborate proofs can be found in [164].

In contrast to many classical data structures the complexity of *probabilistic data structures* such as skiplists does not only depend on their deterministic operations but also on additionally required randomness. For skiplists such randomness is introduced by a probability p which determines the height of a newly inserted node: with probability p a node that is accessible over a linked list on level k should also be accessible on the respective list on level $k+1$.

The choice of p influences the search efficiency. If p is small many nodes have a low height and many pointers have to be traversed horizontally (see Figure 3.6a)) If p is large, some nodes have a large height. In consequence more nodes can be skipped over but vertical traversal is more time consuming (see Figure 3.6b)).

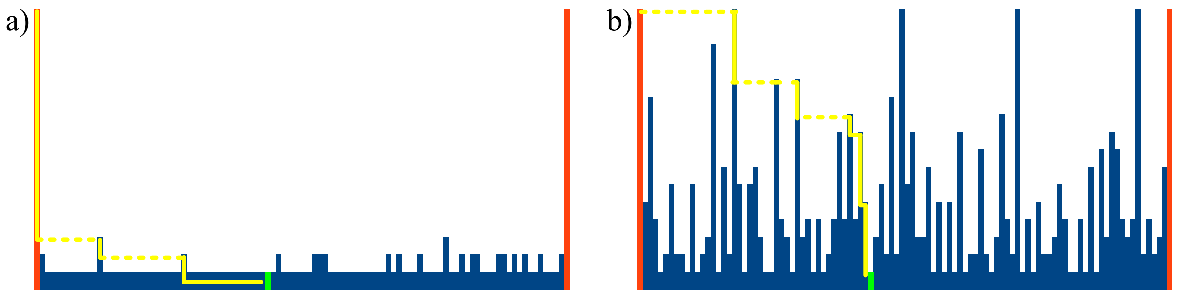


Figure 3.6: Visualization of pointer traversal for a skiplist with a) $p = 0.2$ and b) $p = 0.8$. The orange bars are the `Begin` and `End` node. The green bar shows the key to be searched. The yellow line shows the pointer traversal.

Based on this observation and the formal analysis in [164], for our implementation we defined $p = \frac{1}{2}$ and limit the maximum height. This value of p balances the advantages and disadvantages and allows for a complexity that is comparable to classical data structures. Employing $p = \frac{1}{2}$, skiplists have a probabilistic time complexity of $\mathcal{O}(\log n)$ and a worstcase complexity of $\mathcal{O}(n)$ for all dictionary operations. Note that skiplists have the same average complexity as e.g. balanced trees [120], but are typically easier to implement especially in concurrent scenarios [165].

3.3 A Priority Queue Typeclass for Skiplists

As stated in Section 3.1, (min-)priority queues have to support at least two operations, namely `insert` and `deleteMin`. Skiplists sort their elements in increasing order. The smallest element can therefore be accessed in $\mathcal{O}(1)$ and, since the maximum height of a skiplist is fixed (see Section 3.2), the first element can be removed in $\mathcal{O}(1)$. Since the smallest element is always the first element after the `Begin`-node an explicit predecessors search is not necessary. Furthermore modifying pointers to point to the node after the smallest element takes constant time. Therefore skiplists can be used as efficient data structures for implementing priority queues.

To ease experimentation with the different synchronization approaches, we defined a typeclass `PriorityQueue` which is instantiated by all skiplist variants (see Figure 3.7). The `deleteMin` function returns the key and value of the minimal element wrapped in `Just` or `Nothing` if the queue is currently empty. The `insert` function adds a key and its value to the queue.

```
{-# LANGUAGE MultiParamTypeClasses,
             FunctionalDependencies #-}

class Ord k => PriorityQueue pq k v | pq -> k, pq -> v where
    deleteMin :: pq -> IO (Maybe (k,v))
    insert    :: k  -> v -> pq -> IO ()
```

Figure 3.7: The `PriorityQueue` typeclass.

The `deleteMin`-function uses the type parameters `k` and `v` of the multi-parameter typeclass solely in its return type. Therefore with standard Haskell it is not possible to correctly type-check this definition. To allow type-checking we use a language extension called *functional dependencies* [115]. By stating `pq -> k, pq -> v`, we state that the return type of the function (`k` and `v`) can be derived from another type parameter (`pq`).

3.4 Thread-safe Priority Queue Variants

In this section we describe three variants of thread-safe skiplists, which are based on different synchronization approaches. The first uses locks, the second software transactional memory and the third compare-and-swap operations. Although we are solely interested in the implementation of priority queues, we implemented a complete dictionary interface and built the priority queue operations on top of them. We also briefly describe a heap-based priority queue which is synchronized with a coarse lock. This variant should not be compared to the previously mentioned ones regarding the efficiency of its synchronization mechanism, since the underlying priority queue implementation is fundamentally different. Instead, it serves as an example of the achievable parallel performance of a naive solution.

3.4.1 Lock-based Skiplists

Our lock-based implementation is based on the original lock-based concurrent variant of Pugh [165]. Pugh's approach has two properties that reduce lock contention and make the implementation scalable: First, locking is fine-grained, i.e. both insert and delete operations lock only a small part of the skiplist and merely for a short time. Second, an efficient lock-less traversal through the list is possible by using multiple-reader single-writer locks. This permits fast search operations and an efficient collection of predecessors.

The base types `Skiplist` and `Node` as well as necessary configuration variables are shown in Figure 3.8. Each skiplist is globally configured with two parameters, `maxLevel` and `probability`. The former defines the maximum node height and the latter the probability p for node level generation (see Section 3.2.3). The type `Skiplist` defines the `Begin` and `End` node only. The actual information is stored in the type `Node`. Each `Node` holds a key-value pair (if it is not the begin or end node), a unique id for handling duplicates, a deletion marker for signaling deletion to other threads, its height, as well as references to its successors at each level.

Pugh's approach requires multiple-reader/single-writer locks. This type of locks supports both concurrent reading of a shared value and its exclusive writing. While a thread holds the lock, other threads may still read the stored value. However, threads are blocked if they try to get the lock. Since Haskell's standard concurrency libraries do not have a built-in type for this type of locks, we developed a basic type by using `IORef`s and `MVar`s. While `IORef`s allow storing a mutable value, they can not be used to block threads to attain exclusive access. On the other hand, `MVar`s can be used to block threads

```
-- Skiplist-wide configuration
maxLevel :: Level
maxLevel = 32

probability :: Double
probability = 0.5

-- Data structure
data (Ord k) => Skiplist k v = Skiplist {
    skipBegin :: Node k v
  , skipEnd   :: Node k v
}

data (Ord k) => Node k v = Node {
    nodeValue   :: NodeValue k v
  , nodeId      :: Unique
  , nodeGarbage :: IORef Bool
  , nodePointer :: Pointers k v
  , nodeLevel   :: RWLock Level
}

-- Type definitions
data NodeValue k v = Value (k,v) | Begin | End deriving Eq
type Level         = Int
type Pointers k v  = IOArray Level (RWLock (Node k v))
```

Figure 3.8: Base types for the lock-based skiplist implementation.

but do not permit an efficient non-blocked reading of their contents. To implement a multiple-reader/single-writer lock, we combined both approaches and thereby defined a new type `RWLock`. `RWLock` encapsulates a modifiable value in an `IORef` and uses an `MVar` to attain the blocked locking:

```
data RWLock a = RWLock {
    rwLockValue :: IORef (Maybe a),
    rwLockLock  :: MVar ()
}
```

Note that the value can be empty, i.e. `Nothing`. This simplifies the implementation, since the initial value does not need to be supplied when a new lock is generated. To prevent unauthorized access, the internal details of the `RWLock` are hidden by Haskell's module system. Concurrent reading is implemented by `readIORef`. Obtaining the exclusive lock is attained by taking the value of the `MVar` and releasing it by storing `()`.

Initially, each skiplist is empty. All `maxLevel` pointers of the begin node point to the end node.

The insert and delete operations work level by level. Thereby locking is solely used to control shared access to the successor pointers of each node. Both operations initially need to generate an array of predecessor nodes (as marked in Figure 3.4a) and Figure 3.4b) on page 43).

By the use of the given key and value the `insert` operation at first generates an unlinked new node of random height. This node is inserted level by level, beginning with level 0. After insertion on the first level, the node can be found in a search operation. The additional levels provide shortcuts to reduce search time. On each level, insertion works as follows: The successor of the new node is set to the former successor. Afterwards the successor pointer of the predecessor is locked and this pointer is set to the newly generated node (see Figure 3.9).

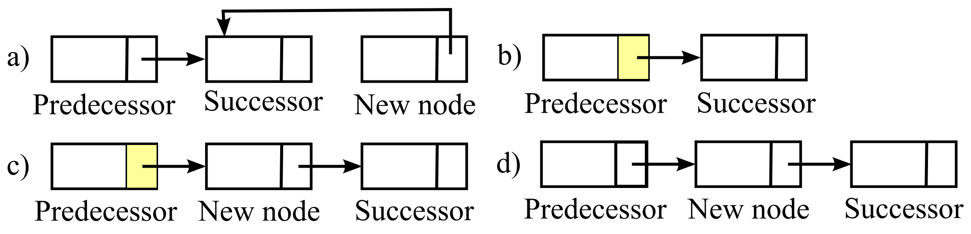


Figure 3.9: Locking to insert a node into the list on a particular level. a) initial state b) locking of successor pointer of the predecessor c) insertion of new node d) unlocking of the locked pointer.

To implement `deleteMin` we use a helper function `delete` which implements the delete function belonging to a dictionary data structure. Note that a general delete function is necessary for good scalability: If deleteMin was implemented as a function that can remove the first element of the skiplist only, multiple deleteMin operations would hinder each other. After predecessor search, `delete` checks the current state of a node's possible deletion: if `nodeGarbage` is false, no other thread deletes the node and it is set. Otherwise deletion is stopped and the operation may be repeated by searching for another node with the same key. Note that marking nodes to be deleted is solely an optimization, not strictly necessary: If a node was not marked, threads would concurrently try to delete it throughout its different levels. Since operations are still synchronized this would not corrupt the data structure, but it would decrease scalability. If the thread is able to delete the found node, it is removed level by level starting with the highest one. This is a safe operation since pointers on levels larger than 0 are shortcuts and so removing them does not lead to a (partially) corrupted skiplist. Thus a node is deleted if it is removed from level 0. Since other threads should be able to traverse through the node currently being deleted, a backreference to the node's predecessor is set. This allows continuous traversal (see Figure 3.10).

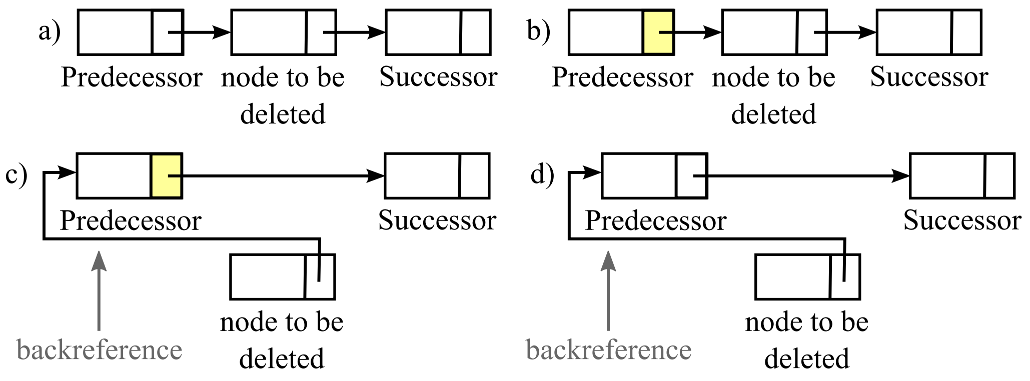


Figure 3.10: Deleting and locking of a node at a certain level. a) initial state b) locking of successor pointer of the predecessor c) setting new successor pointer of the predecessor and generating the backreference d) unlocking of the locked pointer.

The `deleteMin` operation reads the key of the successor belonging to the begin node and tries to delete it. If deletion is successful, the element is returned. Otherwise another thread deleted the element simultaneously and consequently the operation is restarted.

The lock on the node level (`nodeLevel`) is necessary to prevent corruption when a node is concurrently inserted and deleted. When a node is inserted, the lock is taken and released when insertion is finished. Deletion takes this lock prior to removing the node. This prevents the following case: A node with key k and height n is not fully inserted yet, e.g. insertion is currently at level $m < n$ and the key k should be deleted. Deletion will thus start at level m and work downwards, insertion will continue until all predecessor pointers on level $m + 1 \ldots n$ have been set. As a consequence, pointers without a corresponding node exist on these levels.

The order of concurrently *inserted* elements with the same key is not predetermined at different levels. Instead it depends on thread scheduling. One thread can insert a key k at level n, but another thread is first inserting another element with the same key k at level $n + 1$ (see Figure 3.11a) on page 50). Without additional information, the predecessor search would find the wrong pointers (see Figure 3.11b) on page 50). If key k should be deleted, a predecessor search finds m as the predecessor at level $n + 1$. At level n, the correct predecessor is k (and not m). To distinguish between different nodes with the same key, an unique identifier is added to each node. Thus a predecessor search finds the correct nodes at each level by comparing both the key and the unique identifier.

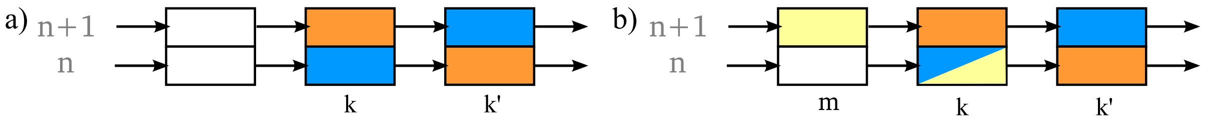


Figure 3.11: Illustration of a skiplist with two keys k (marked with different colors). a) Due to scheduling, the order of the keys is different between levels n and $n+1$. b) Predecessor search for deletion of key k. Predecessor are marked yellow.

3.4.2 Software Transactional Memory based Skiplists

For the second implementation we used software transactional memory for synchronization. As STM is a high-level approach, it is especially interesting to analyze if the promise of easier programming can compensate for the additional overhead of the transactional model.

We developed two variants of the STM skiplist. First a naive one in which the whole functionality for each operation is enclosed in one atomic block. Second a sophisticated one in which the different steps for each operation are divided into separate atomic blocks. The data types are similar to the lock-based ones except that `TVar`s are used instead of `IORef`s and `MVar`s (see Figure 3.12).

```
data (Ord k) => Node k v = Node {
      nodeValue    :: NodeValue k v
    , nodeId       :: Unique
    , nodePointer  :: Pointers k v
    , nodeLevel    :: TVar Level
}

type Pointers k v  = TArray Level (Node k v)
```

Figure 3.12: Base types for the STM-based skiplist implementation.

In the naive variant there is no need for a shared variable containing the deletion state. The deletion state is managed by the underlying transaction model implicitly. The STM based approach can be transcribed directly from the sequential variant by modifying mutable variables to `TVar`s and enclosing each operation in one atomic block (see Figure 3.13a)). As we illustrate in Section 3.5, the performance of this variant is rather insufficient due to the possibly long time each transaction takes and the high probability of conflicts and restarts.

A more reasonable approach is to dissect the transactions into independent level-wise sub-transactions. Now both the insert and the delete operations have two phases. In the first phase the predecessors for the correct position are searched. In case of deletion, the corresponding node is marked as to be deleted. In the second phase, the new node is deleted or inserted at the correct position, respectively. For each level a new atomic transaction is started to execute the previously mentioned linked-list operations (see Figure 3.13b)).

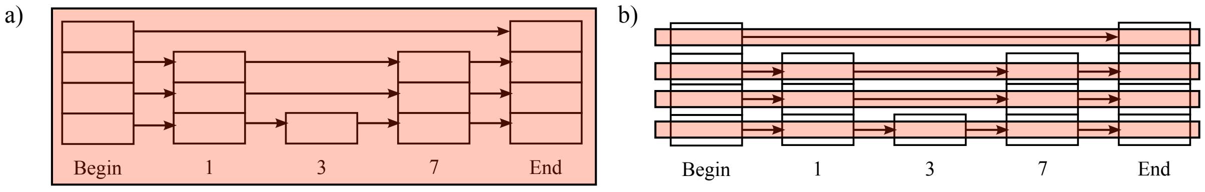


Figure 3.13: Illustration of the second phase of pointer modifications. Predecessor search is not shown, transactions are marked red a) naive transactional variant. One transaction is used for a whole operation. b) sophisticated transactional variant. For each operation independent transactions are used at each level.

It is possible to get on with the lock-based solution by dissecting the level-based transactions, i.e. a new transaction is started for each pointer modification. This approach would be identical to the lock-based approach. Thus all advantages of programming in the transactional model would be lost and the overhead would be higher.

3.4.3 Skiplists based on Atomic Compare and Swap Operations

Lock-free synchronization with atomic compare-and-swap operations promises to deliver better performance than both transaction- and lock-based synchronization. The reasons for this are the more fine-grained control of the operations as well as the absence of waiting for synchronization. Basically, skiplists are advanced linked lists. So we adopted the ideas about a lock-free implementation of linked lists from Harris and Fomitchev [53, 73], extending them to work at multiple levels. As stated in Section 3.4.1, deletion works downwards and insertion upwards level by level.

Atomic compare and swap operations guarantee consistency of shared variables by means of two atomic functions: test-and-set (TAS) and compare-and-swap (CAS). A TAS applies a function to the current value of the shared variable. Then it swaps the stored value with a new value if the function returns `True`. A CAS checks if the shared variable contains a specific value (usually the value of the variable read previously). Only if this is the case, CAS stores the new value. For both operations a returned flag indicates the outcome. Haskell code for modifying shared `IORef`s by TAS and CAS is shown in Figure 3.14 on page 52. Both operations use `atomicModifyIORef` internally. Therefore they are thread-safe and atomic.

```
-- If the test function called with the currently stored
-- value is true, store the new value.
atomTAS :: Eq a => IORef a -> (a -> Bool) -> a -> IO Bool
atomTAS ptr test new = do
    atomicModifyIORef ptr $ \cur ->
        if test cur then (new, True) else (cur, False)

-- If the currently stored value is also the passed value,
-- store the new value.
atomCAS :: Eq a => IORef a -> a -> a -> IO Bool
atomCAS ptr old new = atomTAS ptr (==old) new
```

Figure 3.14: Atomic compare-and-swap and test-and-set functions.

Again, data types are similar to the lock-based ones, and are shown in Figure 3.15. Instead of protecting pointers of successor nodes with a lock, they are stored in an `IORef` and modified by CAS and TAS operations. Similar to lock-based synchronization, pointers have to be marked in some way to signal their (future) modification. But with a lock-free synchronization model obviously it is not possible to mark them by locking. A common approach for compare-and-swap synchronization is to add a flag for each pointer. The flag can have the values `Unmarked` or `Marked`. A single CAS can be used to set the flag from `Unmarked` to `Marked`. Thereby a node is marked for the purpose of signaling its removal state. Therefore a `Pointer` contains a tuple which stores its marked state as well as the successor of the node at a particular level. The tuple is changed as described below. Instead of using a flag many low-level implementations modify the least significant bits of a memory address [53, 73, 202]. However, this architecture- and compiler-dependent approach is not directly applicable to Haskell with `IORef`s.

```
data Ord k => Node k v = Node {
    nodeValue   :: NodeValue k v
  , nodeId      :: Unique
  , nodeGarbage :: IORef Bool
  , nodePointer :: Pointers k v
  , nodeLevel   :: Level
}

data Mark           = Marked | Unmarked
type Pointer k v  = (Mark, Node k v)
type Pointers k v = IOArray Level (IORef (Pointer k v))
```

Figure 3.15: Base types for the compare-and-swap-based implementation.

Using CAS operations, insertion is similar to the lock-based approach, yet deletion needs to be more complicated to handle the case of interfering insertions and deletions (see Figure 3.16). To prevent corruption, the deletion is subdivided in two phases. In the first phase, the predecessor pointer of the node to be deleted (called N_d in the following) is searched and N_d's successor pointer is logically deleted using CAS. The search is repeated if CAS fails. This can be the case if another thread has already marked the pointer. This phase prevents concurrently running insertions from modifying this pointer. In the second phase, the node is actually deleted physically from this level (using a second CAS) by setting the pointer of the predecessor node to the successor of N_d. If another concurrent operation changed a pointer and the CAS failed, the node will be removed by an initiated search operation for the key as described in the following.

The key function used by both operations is the search for the predecessor. While searching, all nodes that are traversed and have previously been marked for removal by other threads are also deleted (see Figure 3.17 on page 54).

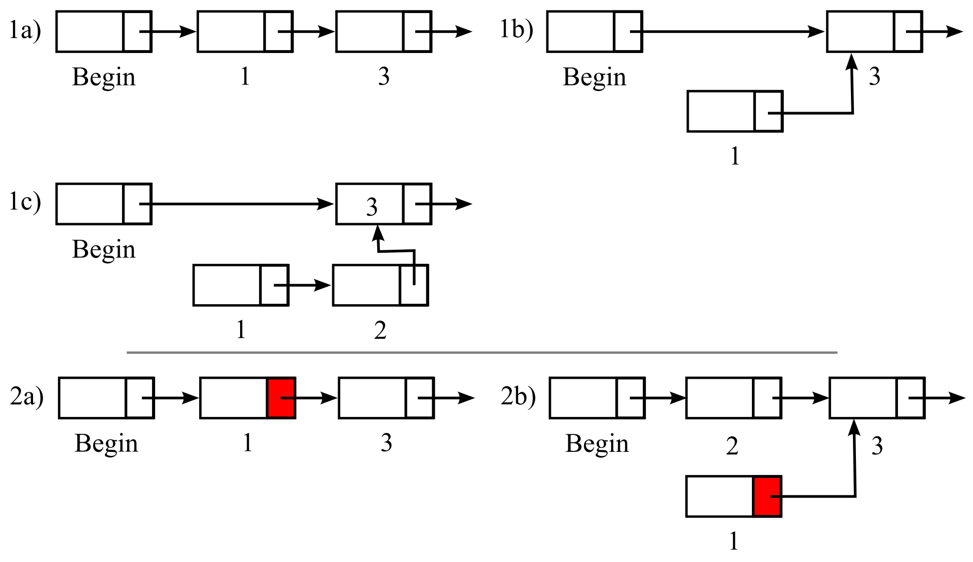


Figure 3.16: Consistency problems with concurrent deletion of node 1 and insertion of node 2: 1a) initial state 1b) use of a single CAS (modifying the successor of the Begin-node) 1c) concurrent insertion can lead to new nodes that are not reachable anymore. Solution using a two-phase deletion: 2a) deletion phase 1: logically mark the node as to be deleted 2b) after physical deletion and concurrent insertion.

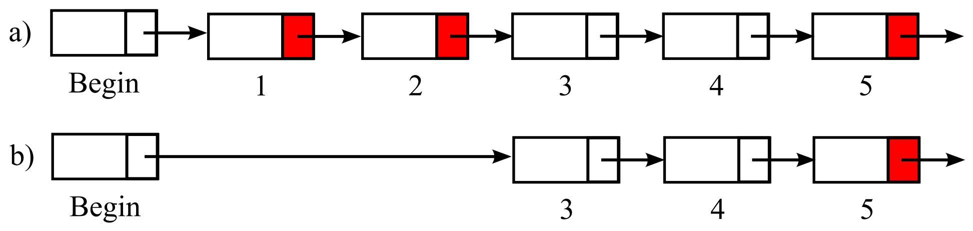


Figure 3.17: Illustration of deleting traversed nodes on a particular level by means of searching for a node with key 3. a) initial state: The marked nodes were marked by other threads b) state after traversal: The search function returns the beginning node as the predecessor and node 4 as the successor. Non-traversed nodes (e.g. node 5) are not influenced.

3.4.4 Coarse-Locked Heap-based Priority Queue

To compare the different variants of skiplists we implemented a coarse-locked heap-based priority queue which is easy to implement and synchronize.

Although a heap is tree-based, it is usually implemented by means of an array. To allow for traversal that is similar to a binary tree (traversing to right, left and parent nodes) nodes are mapped to specific indexes: For a node i the parent is accessible at index $\lfloor \frac{i}{2} \rfloor$, its left child at $2i$ and its right child at $2i+1$. For the priority queue operations, we applied the approach for heaps, described in e.g. [36].

The base types are shown in Figure 3.18. Note that all elements are stored in a mutable array whose elements are stored in memory as a continuous block. A single `MVar` prevents the heap from being concurrently accessed and modified. Note that in our benchmarks (see Section 3.5) the initially allocated array is large enough to store all values and thus there is no need to resize dynamically.

```haskell
-- Coarse-locking through a single MVar
data Skiplist k v = Skiplist {
    ref :: MVar (Heap k v)
}

data Ord k => Heap k v = Heap {
    heapData :: IOArray Int (k,v)
    -- Position of first free element
  , heapLast :: IORef Int
    -- Maximum capacity (for dynamic resizing)
  , heapMax  :: IORef Int
}
```

Figure 3.18: Base types for the coarse-locked heap-based implementation.

3.4.5 Testing the Implementations

It is difficult to develop concurrent programs. Hard to find errors can be overlooked easily, e.g. slight corruptions of the data structure. Proving the correctness of an implementation is still a topic of ongoing research [187]. To at least test the requested functionality, we implemented Johnson's algorithm [123]. It calculates the single-source shortest path (SSSP) [36] in sparse graphs in parallel using a thread-safe priority queue as its basic data structure. This algorithm uses both insert and deleteMin operations in a realistic scenario, and emphasizes the duplicate key support (since many duplicate keys occur). Moreover, it allows an easy comparison with a desired outcome: For reference, we developed a simple sequential implementation.

A test consists of a randomly generated graph and the execution of Johnson's algorithm. We ran each test a few hundred times for all four skiplist implementations and the heap-based variant. A test was evaluated to be successful when the result was equal to the sequential reference implementation.

We did not use the algorithm for performance measurement, since the effort per extracted element is exceptionally low. Johnson's algorithm can be improved in this respect (and it is very reasonable to do so in a real-world application), but this further development is not trivial and a research topic on its own right [158, 188]. Therefore it is out of scope regarding our research topic.

For testing in Haskell a common tool is Quickcheck [31]. We did not use this tool, however, since it is primarily targeted at side-effect free code and not appropriate for testing concurrent code which is executed in the IO-monad. In particular, its strength of automatic test case generation does not necessarily apply to non-deterministic bugs that are generally found in concurrent programs.

3.5 Benchmarks

We ran experiments on a 2.3 GHz 16-core AMD Opteron 6134 with 32 GB RAM running a Linux-kernel 2.6.38-8 with GHC 7.0.3. Garbage collection times ranged from 1% up to 10% for all benchmarks.

3.5.1 Scenarios

We examined two scenarios: First we tested the scalability of the implementations by means of a benchmark adapted from the literature [192]. Second we examined the speedup using a synthetic benchmark. This benchmark simulates a divide-and-conquer algorithm. As stated in Section 3.4.5 we did not implement a real-world algorithm, since even for simple algorithms the design space is large and currently mostly unexplored in Haskell.

Scalability

In the scalability benchmark each concurrent thread performs 1000 initial insertions, followed by 10000 operations, randomly chosen to be 50% deleteMin or insert. Keys were randomly chosen from the integer interval $(0, 10^6)$. These values and probabilities were adopted from other papers, e.g. [192]. We repeated each benchmark three times and report the average execution times for the three runs. Since the thread-specific random number generators are (re-)initialized with the same initial (but thread-specific) seed, the same sequential operations are performed in all runs as well as all implementations. This procedure guarantees reproducibility. Since *each* thread performs 11000 operations, the total number of operations increases with the number of threads. We conducted three experiments with varying workload after each deleteMin operation:

In the first experiment the thread that executes deleteMin receives the key-value k and computes π on $\frac{k}{35000}$ digits (see Figure 3.19). We chose the divisor 35000 such that

```haskell
-- Calculate pi to a specific number of digits and
-- discard results.
import Data.Number.CReal

calcPiPure :: Int -> ()
calcPiPure digits =
    showCReal (fromEnum digits) Prelude.pi `seq` ()
```

Figure 3.19: Code for π calculation.

for $k = 10^6$ this calculation takes 0.002s, which is a reasonable amount of computational work. For this amount of computational work results are shown in Figure 3.20 on page 58. Keep in mind that in all figures the time-axes are logarithmic. We also do not show times over 1000s so the lower graphs are more distinguishable. Both transactional variants perform badly, since the amount of computational work can not compensate for the overhead of the synchronization model. Nevertheless the advantage of dissecting the transactions is clearly visible. The CAS variant which does not lock explicitly and thus should outperform explicit locking, performs magnitudes of orders better than the transactional variants, but it is about three times slower than the lock-based and heap variants. This unexpected result can be explained by the high level of abstraction of the atomic CAS operations. Related work with C as the underlying language also describes a better performance of CAS-based skiplist implementations as compared to lock-based variants. Since C has a low level of abstraction and uses real pointers, CAS operations are much faster, especially as there is a supported assembly instruction since 1970 [206]. Although GHC internally uses these CAS instructions they were not available for types of a higher level or for mutable references at the time of development.

The good scalability of the lock-based variant was expected, since the implementation does not hold any global lock and to a large extent `MVars` are used, which are optimized. We were surprised tough, by the good scalability of the straightforward and easy to implement heap-based variant. Its coarse locking impairs performance, however this drawback is compensated by a cache-friendly memory layout: all operations are executed on a continuously allocated memory block. This provides a much better spatial locality and a better cache usage than the potentially distributed and small data chunks of skiplists. Note that this result does not imply that coarse-locking scales generally better as compared to the other synchronization variants, since the underlying priority queue implementations were fundamentally different. Instead, the result demonstrates that for some scenarios a less complex implementation might be a viable alternative.

In a second experiment we omitted the workload to analyze the impact of high contention. The results are shown in Figure 3.21 on page 58. As anticipated, the transactional variants perform quite bad since transactions are restarted (overall or on a level basis) very often. The compare-and-swap variant still scales well. Although both the heap and the lock-based variant scale better than the others, concurrent accesses to the heap occur so often that its cache advantage is dominated by frequent synchronizations.

It is interesting to analyze scalability in case of a higher amount of computational load since this occurs in algorithms (e.g. scheduling) as well. In our third experiment π was computed on $\frac{k}{2000}$ digits (about 0.035s for $n = 10^6$). We additionally reduced the number of initial insertions and operations. Results are shown in Figure 3.22 on page 58. While the different variants converge, the overall ranking stays the same.

Speedup

Our second scenario, a synthetic speedup benchmark, initially fills a priority queue with 10000 random elements. Computation takes up to 0.002s per element using the π calculation mentioned above. If an element is extracted from the queue for the first time, it is additionally reinserted two times with half its original value (approximately halving its next computation time). In addition, π is computed on $\frac{k}{35000}$ digits, with k being the extracted number. Thus it represents a computational intensive divide-and-conquer

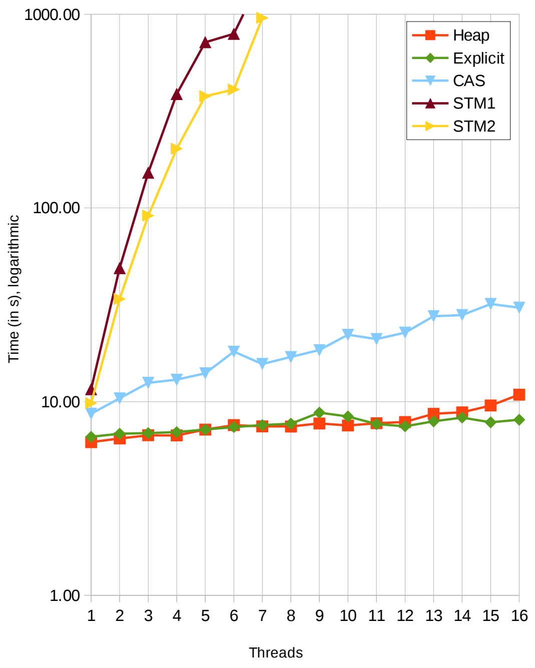


Figure 3.20: Scalability of priority queue implementations with computational load.

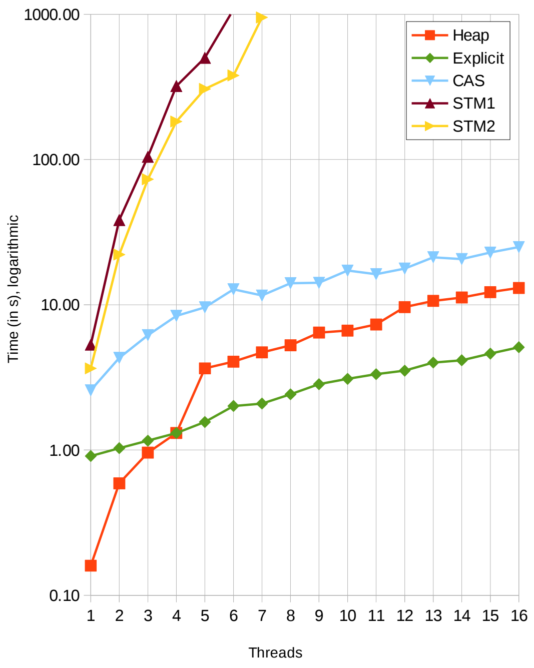


Figure 3.21: Scalability of priority queue implementations without computational load.

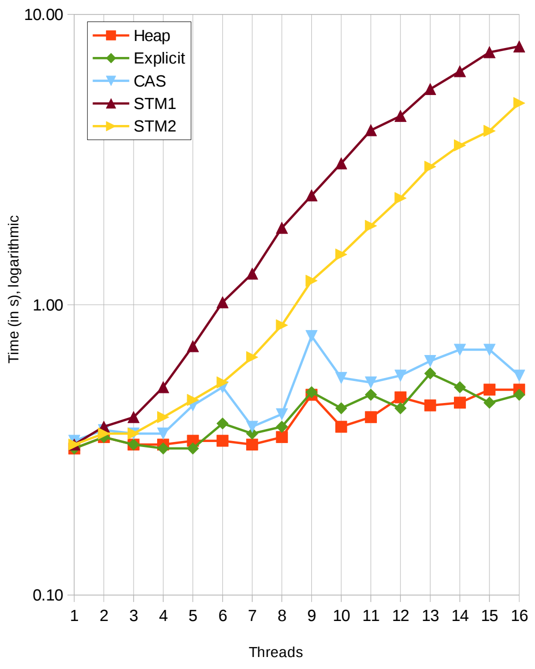


Figure 3.22: Scalability of priority queue implementations with larger computational load.

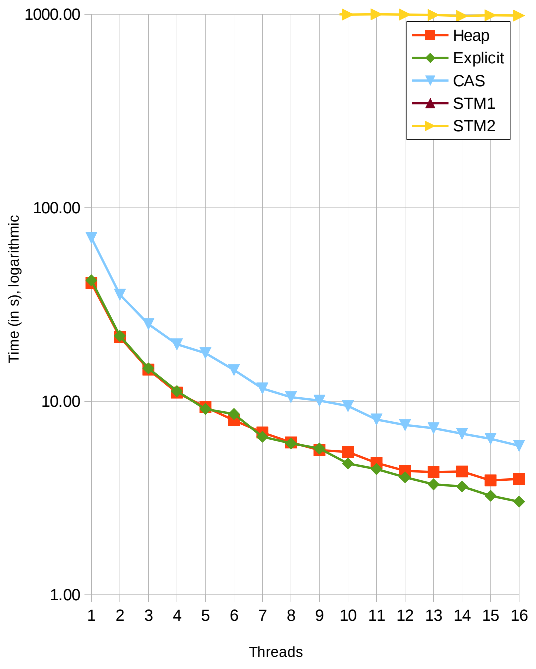


Figure 3.23: Speedup of different priority queue implementations.

algorithm. The results are shown in Figure 3.23. The transactional variants performed quite bad ranging from 1862s and 1132s for one thread up to 1925s and 983s for 16 threads for STM1 and STM2, respectively. Thus they are only sketched in Figure 3.23. In accordance with the scalability results the lock-based variant and the heap show the best speedup although the compare-and-swap variant performs quite similar. The two transactional variants show a small speedup but are three orders of magnitudes slower than the other variants.

Summary

In summary, both the lock-based variant and the heap-based variant perform best. Taking into account the better scalability of the lock-based variant compared to the heap-based variant, the former should be preferred especially if future large multicore and manycore systems will be used. The compare-and-swap variant also shows good scalability but the constant factors need to be improved to be competitive. The transactional variants are not practical under circumstances of high contention. Nonetheless, their development is much easier in scenarios with lower contention.

3.5.2 Comparison of Difficulty and Code Size

Finally we reconsider two non-performance oriented criteria: effort of programming and code size.

The heap-based variant was easy to develop: coarse-locked synchronization is conceptually simple to implement and array-based binary trees are well-documented (although we were not aware of any Haskell implementation).

The naive transactional variant was by far the easiest skiplist implementation. As stated in Section 3.4.2 a thread-safe variant could be developed by just using `TVars` for mutable variables. The dissected transactional variant could also be developed easily. Although we had to implement dissected level-based traversal, we did not have to consider common synchronization problems such as deadlocks.

The lock-based variant was more difficult to develop, despite the fact that it is the most researched one and the author was experienced in this approach. It had the well-known problems of rarely occurring deadlocks and thus difficult debugging. An advantage of the lock-based approach is its explicit semantics: if a potential cause for a bug has been identified, it is easy to reason about the behavior of the program and resolve the problem. The compare-and-swap variant was the most difficult to develop: programming errors did not entail any observable deadlocks. More often a slightly corrupted data structure resulted. This could result in errors that occurred much later in the execution and so debugging and reasoning about potential errors was very difficult.

While the code size varies from one developer to the other depending on their programming style, a comparison with one programmer approaches the complexity of the different implementations from a different angle. In each variant we counted source code lines, excluding comments and empty lines. The heap-based variant was the smallest with 125 lines of code. The transactional variants STM1 and STM2 were comparable with 237 and 250 lines of code, respectively. The two skiplist variants were significantly larger: each one contained 401 lines each.

3.6 Related work

A huge amount of research has been conducted on efficient and thread-safe implementations of priority queues. Generally they are implemented using binary heaps [36]. A common parallel variant that uses locks is described in Hunt [102]. A review of thread-safe priority-queue implementations as well as a discussion about their properties for coarse locks and transactional versions is given by Bauer et al. [13]. The review discusses Hunt heaps, parallel Fibonacci heaps, lock-free skiplists, and quantizing queues.

A probabilistic data structure that allows for the implementation of a priority queue is the skiplist. A sequential as well as a lock-based skiplist were first developed by Pugh [164, 165]. Both implementations yielded better performance than binary heaps. In Lotan et al. [131] one can find a detailed discussion about the performance characteristics of lock-based priority queues using skiplists. An algorithmic performance analysis of lock-free linked lists as well as future prospects for skiplists is described in Fomitchev [53]. Experimental results and a comparison between the performance of lock-free implementations and the performance of lock-based variants (Hunt, Lotan-Shavit) are illustrated in Sundell [192]. The contributions of Sundell and Fomitchev are both based on the implementation of lock-free linked lists as described in Harris [73], which itself was an improvement of the algorithms for lock-free linked lists from Valois [202]. A thread-safe skiplist implementation in Java can be found in the concurrency-aware collections of the standard library [153].

Sulzmann et al. analyzed the performance of concurrent linked list implementations in Haskell [191]. Like our work, they examined synchronization using `MVars`, `IORefs` and STM. But their benchmark depended on GHC's internal thread scheduling to a greater extent than our implementation: 8 threads were started and the number of utilized processors was varied. In contrast to the results demonstrated in Section 3.5, in [191] the lock-free CAS implementation outperformed the other variants. A possible reason for this difference is that their benchmarks a ratio of 2 searches to 1 delete to 4 inserts: searches need no synchronization at all and insertions and deletions are distributed all over the linked list. Our benchmarks have no searches and deletions cause more conflicts since `deleteMin` always tries to delete the first accessible element.

Although priority queues are a basis for many parallel algorithms, (as far as we know) there is no literature about thread-safe priority queue implementations in Haskell. There is also only few literature about mutable thread-safe data structures for explicit concurrency in general. The common way to implement a thread-safe data structure is to protect a pure data structure [149] by means of a coarse thread-safe container (e.g. a `MVar` or `IORef`). For example, this approach has been applied in the new GHC 7 IO manager [156]. One possible reason for the lack of research is that thread-safe data structures still work on a quite low level of abstraction. Haskell provides alternatives which allow for a more abstract definition of parallel algorithms (as we will also show in the following chapters).

3.7 Summary and Conclusion

We implemented thread-safe priority queues based on skiplists in Haskell. Synchronization of shared mutable variables was implemented using lock-based or lock-free techniques. Our lock-based variant uses `MVar`s, whereas the lock-free variants use software transactional memory (STM) or atomic compare-and-swap (CAS) operations, respectively. For STM, we developed two variants: a naive transformation of the sequential implementation and a variant that dissects the transactions into several parts. For an additional comparison with a straightforward to implement priority queue, we also implemented a coarse-locked heap-based variant. We tested the functionality of all variants by means of Johnson's algorithm for the single source shortest path problem on sparse graphs. Results were compared to a reference implementation.

For evaluation of the different synchronization approaches four different criteria were applied: scalability, speedup, programming effort and code size. The lock-based variant scaled best. We suppose that it will scale well even with more cores since it does not use any global lock. Typical problems of lock-based synchronization, e.g. deadlocks, made the development difficult. Surprisingly, the coarse-locked heap performed comparably well for many scenarios with low contention, but we suppose it will do worse the more concurrent threads are running. The coarse-locked heap could be developed easily. The compare-and-swap variant scaled well but its absolute performance was restricted by a slow atomic compare-and-swap operation. Since absolute performance is not on par with a well-tuned lock-based variant it can not yet be recommended before a more efficient compare-and-swap operation for mutable variables is available in Haskell. According to [136], a low-level CAS operation for `IORef`s will be available in one of the next versions of GHC. Development of the CAS variant was cumbersome since programming errors were difficult to find. The transaction-based implementations were easy to develop, but the initial overhead of the transactional approach as well as the costs for synchronization under the condition of high contention are much too high. So this approach does not seem to be a good choice for the synchronization of thread-safe data structures. Nevertheless, if the particular problem limits the number of concurrent accesses, they are a good alternative to traditional approaches.

To draw a conclusion we suppose that a *well-designed* lock-based synchronization is currently the best approach to synchronize shared data structures in Haskell, especially in case of high contention. This approach is well researched and will scale well on future multicore machines. Given the results from related work, compare-and-swap synchronization can outperform lock-based synchronization if the internal operations are optimized. For prototyping, few cores or low contention, synchronization with coarse locks or software transactional memory are viable alternatives. In contrast to coarse locking, transactional synchronization has the advantage that transactions can be combined easier which allows for more flexibility and more modularity in the program's design.

Summarizing on the usefulness of abstraction, the outcome of this chapter was rather negative. While high-level synchronization by STM allows for a convenient implementation of a thread-safe data structure, this advantage comes at a significant performance penalty. Possibly there are other data structures where STM-based synchronization scales better than in this case. Unfortunately, the synchronization approach must be developed from scratch and can not easily be based on existing implementations. This might limit the advantages of using a high-level synchronization approach in the first place.

Chapter 4
A Comparison of Lock-based and STM-based Taskpools

Design Patterns formalize principles of software engineering within a framework of a common vocabulary. Such patterns also exist for parallel and functional programming. One of the best-known and frequently used parallel design patterns is the taskpool. A taskpool allows to distribute tasks to threads. Therefore it is used as a building block for more sophisticated parallel algorithms. In this chapter we demonstrate how to implement different variants of the taskpool pattern. We implemented global and private taskpools with and without task stealing. We used locks as well as software-transactional memory for synchronization. Locks are a low-level approach to synchronization while software transactional memory provides an additional abstraction layer and therefore eases synchronization. We depict our benchmark scenarios and report on the scalability of the different variants of taskpools using two synthetic algorithms as well as LU decomposition.

4.1 Introduction

Design patterns are fundamental for the development of large software systems. They formalize reliable solutions and bring them together within a framework of a common vocabulary. Such a vocabulary eases communication between developers, comparison of different approaches, and the implementation of programs. While traditional design patterns often enclose object-oriented knowledge [60], parallel design patterns describe building blocks for parallel programs [168].

Task-based parallel algorithms partition a computation-intensive problem into *tasks* that can be processed independently. The parallel computation of these tasks is the key to an increase in computation speed. In general, tasks compute data that is needed by other tasks. Hence, dependencies are induced, which may or may not be known at compile time. Depending on the particular algorithm, tasks may start other tasks when being processed. Task dependencies can be modeled by means of a directed acyclic graph called *task-dependency graph*. In this graph, each node represents a task and each edge denotes a dependency. A task can be computed if all input tasks have finished. Some task-based algorithms such as LU decomposition (see [123] and Section 4.5.4) work in *iterations*. In each iteration, tasks are processed in parallel. Before the next iteration starts, all tasks of the current iteration must have been processed. In the dependency graph, therefore all tasks of an iteration are connected to all tasks of the successive iteration. See, for example, [55, 123, 140], for an overview of different types of dependency graphs.

A *taskpool* is a parallel design pattern that enables the load-balanced processing of independent and irregular tasks [123, 168]. All tasks are stored in a shared data structure which can be modified by at least two operations. For a taskpool T and a task t these operations are $\texttt{put}(T, t)$ and $\texttt{get}(T)$. The former operation adds t to the taskpool T and the latter retrieves a task from T. The semantic of these two operations depends on the particular implementation of the taskpool as well as on special requirements of the application. For example, `get` can either return or wait if the taskpool is empty. We present our definition in more detail in Section 4.3.

Task-based algorithms using a taskpool generally work as follows: First, the taskpool is generated and filled with initial tasks. Then threads are either created explicitly or already existing threads are deployed. The threads process one task at a time in parallel. Task processing finishes, when a particular condition is fulfilled or when all tasks have been processed.

The taskpool pattern allows us to examine different synchronization techniques as well as the abstraction mechanisms of Haskell: First, the number of tasks and the number of concurrent accesses to a taskpool are often large. Therefore the underlying parallel runtime system performing the particular synchronization mechanism is constantly under load. Second, taskpools are widely used in real-world applications. Mainly we were interested in analyzing if high-level synchronization constructs, e.g. software-transactional memory, could compete with traditional lock-based synchronization regarding scalability. As a second issue we examined if language constructs like monads and higher-order functions are able to facilitate the formulation of irregular task-based algorithms. We did not include compare-and-swap operations for their low-level nature. We based our investigation on three taskpool variants: global taskpools, as well as private taskpools with and without task stealing. For conciseness, we call a private taskpool with task stealing a shared taskpool. For each variant, synchronization was implemented by means of locks as well as software transactional memory. Furthermore, each variant was examined with three example problems: two synthetic algorithms with varying task structures and the LU decomposition of a matrix.

All six taskpool implementations are based on a custom *typeclass* and a *monad*. In addition, they refer to an identical set of *core functions* that allow the convenient formulation of task-based algorithms. An overview of the relation between the different synchronization techniques, taskpool variants, and language constructs is shown in Figure 4.1. Each taskpool variant implements the functions defined in the typeclass. The monad defines a taskpool-variant independent environment. Core functions provide a high-level interface to taskpool operations. These functions are executed in the environment defined by the monad and use the taskpool operations of the respective variant.

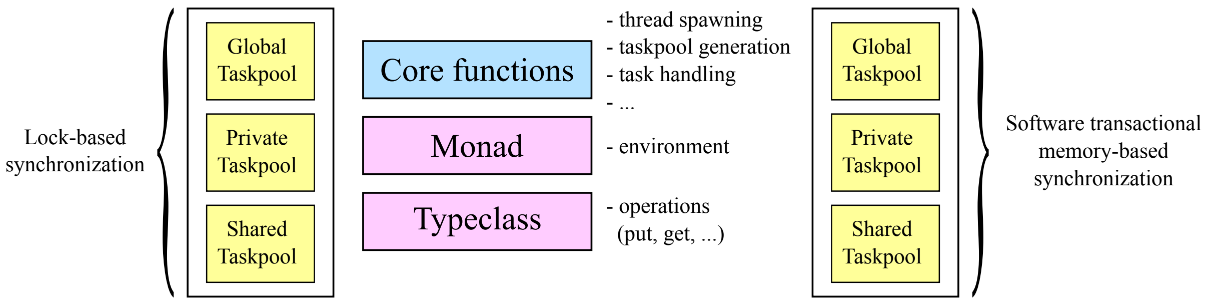


Figure 4.1: Relation between different taskpool variants, their synchronization primitives, and language constructs.

This chapter is structured as follows: In Section 4.2, we introduce the different taskpool variants. In Section 4.3 we use examples to describe a taskpool-independent typeclass as well as a monad for taskpool operations. In Section 4.4 we explain the various implementations in detail. In Section 4.5 we explain our example problems and experimental settings, and discuss the results. In Section 4.6, we compare our results with related work. Finally, in Section 4.7 we summarize this chapter and conclude.

4.2 Taskpool Variants

In this section we describe criteria for the usefulness of taskpools. Subsequently we explain the variants used in our experiments, and sketch some of their general advantages and disadvantages.

4.2.1 Taskpool Criteria

While designing the interface to the taskpools and implementing the different taskpool variants we were guided by three requirements:

1. **Performance.** The taskpools' implementation should allow for good performance and scalability, such that modern multicore systems can be utilized effectively. In addition, the overhead of the taskpools should be low so that there is a great deal of processing power left for computing the actual tasks.
2. **Usability.** The taskpool's interface should be simple to understand and use. Additionally, the interface should support the idioms of the underlying programming language, e.g. higher-order functions, typeclasses, and monads. Different variants of taskpools should be exchangeable so experimentation is easy.
3. **Task independence.** The taskpool's performance and usability should be independent from the underlying task structure. In addition to simple task structures the taskpool should also support complex task structures that involve dynamically generated tasks and arbitrary dependencies.

We chose the following variants since they cover a range of issues such as synchronization and access to thread-local storage.

4.2.2 Global Taskpools

In this variant all threads access a common memory area (see Figure 4.2). Global taskpools are easy to implement, since any thread-safe container is sufficient. On the other side, they scale badly when they are accessed frequently because synchronization is necessary for each operation. To prevent contention and thus enhance scalability, one approach is to add additional thread-local storage. This approach is implemented by a private taskpool and shown in the next section.

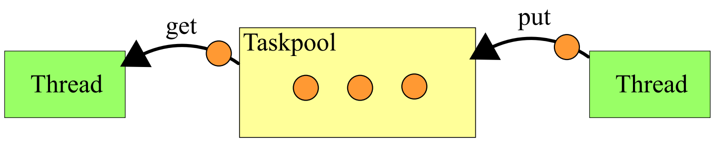


Figure 4.2: A global taskpool and two threads. The thread on the right side adds a task, the thread on the left side receives one.

4.2.3 Private Taskpools

A private taskpool uses a global taskpool and additionally for each thread a thread-local (private) storage (see Figure 4.3). Instead of transferring one task at a time, each access to the global taskpool transfers a set of tasks and stores it in the private storage. Successive `get`-operations of a thread are served from private storage until it is empty. It is rather difficult to determine how many tasks should be taken from the global taskpool at once. In our own implementation a `get`-operation transfers as many tasks as possible. The `put`-operation always adds new tasks to the global taskpool. Of course, other heuristics are possible [122, 193]. We chose the described approach since it is easy to implement and our focus is on synchronization. An advantage of using private storage is that only a few costly accesses to the global taskpool are necessary. Hence the scalability of this variant is usually better than that of a global taskpool. On the other side some task distributions may inhibit an effective load-balancing. For example, if a thread retrieves a set of computationally expensive tasks out of the global taskpool, other threads are idle, since they can not access those tasks anymore. In the next section an approach to allow idle threads to steal tasks and to continue task processing is shown.

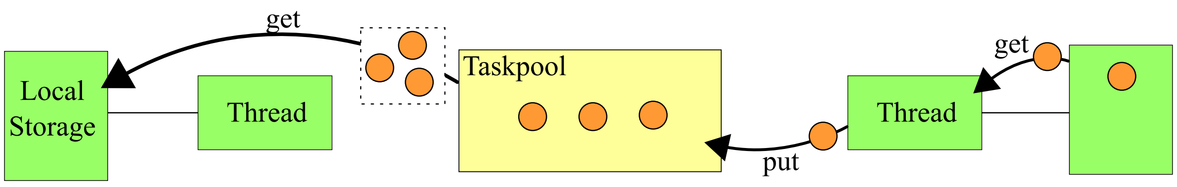


Figure 4.3: A private taskpool and two threads. When retrieving tasks, each thread tries to access its local storage first. If the local storage is empty, the global taskpool is accessed.

4.2.4 Shared Taskpools

A shared taskpool resembles a private taskpool but has an extended operation to transfer new tasks (see Figure 4.4 on page 68). If the thread's local storage as well as the global taskpool are empty, the thread-local storage of another thread is accessed and tasks are stolen from there. So threads do not have exclusive access to their thread-local storage. While this approach prevents threads without tasks, it requires more synchronization. Therefore, a shared taskpool is more complicated to implement than the previously mentioned variants.

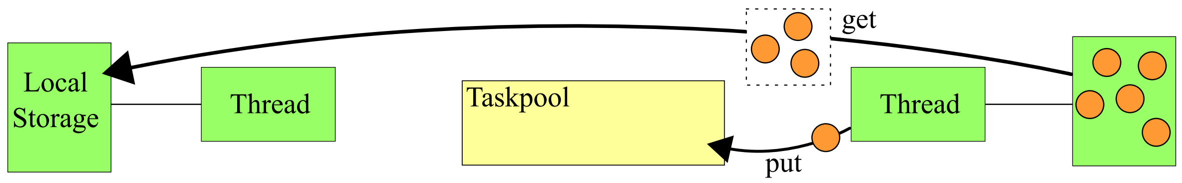


Figure 4.4: A shared taskpool and two threads. When retrieving tasks each thread tries first to access its local storage. If a `get`-operation can not find tasks in either the local storage nor the global taskpool, it tries to steal tasks from working threads next.

4.3 A Monadic Taskpool Interface for Taskpools

In this section we describe our general taskpool interface for Haskell. We also discuss the relevance as well as the implementation of core functions. These functions ease the description of task-based algorithms as well as the sharing of data between tasks. For a better understanding of the internal operations of the taskpool monad we deliberately depict a lot of source code.

4.3.1 Typeclass and Monad

The common interface to all taskpools is defined by a typeclass that defines a set of common operations for all taskpools. Building upon them, core functions are provided. These core functions provide a high-level interface for the implementation of task-based algorithms. The core functions as well as the taskpool operations use the environment that is defined by the custom monad. Thus, their definition is independent of a particular taskpool-variant.

The typeclass is shown in Figure 4.5. Its parameters `pool` and `task` refer to the underlying taskpool variant and the type of the tasks, respectively. The functional dependency [115] `pool -> task` covers the type dependency between the taskpool and the task's type. The `put`-function adds a task to the taskpool. It is called both to add initial tasks and dynamically generated tasks. *Initial tasks* are added sequentially prior to the computation. *Dynamically generated tasks* are tasks which are added by

```
class Taskpool pool task | pool -> task where
    put  :: task -> TPMonad pool ()
    get  :: TPMonad pool (Maybe task)
    wait :: TPMonad pool ()
```

Figure 4.5: The typeclass `Taskpool` which defines taskpool operations.

other tasks during their computation. The `get` function retrieves a task. If a task *t* is available, it returns `Just` *t*. The `get` function blocks if no tasks are available while other threads are still working. If all threads are finished, it returns `Nothing`. The `wait` function blocks until all threads are finished. Therefore this function can serve as a barrier between iterations (see Section 4.1 and Section 4.3.5).

As shown in Figure 4.5, the taskpool functions work in a monad named `TPMonad`. Its definition is shown in Figure 4.6. The monad's state stores the taskpool which is used by all threads as well as additional thread-specific data. In our implementations of the taskpool variants, an integer was the only additional data. Its purpose is explained in Section 4.4. The monad is implemented with a `StateT` transformer [88]. By using

```
type TPMonad pool a = StateT (Index,pool) IO a

-- Thread-specific state
type Index = Int
```

Figure 4.6: The monad `TPMonad` which encapsulates taskpool operations.

this transformer instead of a read-only `ReaderT`, this design is future proof: Since the state can be modified, threads can store performance statistics, for instance. We had to wrap the state around the `IO` monad, since explicit concurrency and synchronization only work in the `IO` monad. Beneficially using the `IO` monad, tasks can consist of both pure and impure computations.

4.3.2 Core Taskpool Functions

Core functions allow for a concise and easy formulation of task-based algorithms. For example, there are core functions for thread control, or for task processing. These functions use the `get`, `put` and `wait` functions defined in the typeclass, as well as the state defined in `TPMonad`. To illustrate the interaction between the different concepts, we describe some of the common (core) functions in the following.

The function `newTaskpool` generates a new taskpool. This function is defined in each of the different taskpool variants. Each taskpool variant is defined in a separate module.

Hence, by importing a particular module the specific `newTaskpool` function is defined. For example, for the lock-based global taskpool, which is implemented in the module `GPool`, `newTaskpool` is defined by

```haskell
module GPool

newTaskpool :: TPMonad (GPool task) () -> IO (GPool task)
newTaskpool initFunction = do
    pool <- newGPool
    let noState = undefined
    State.runStateT initFunction (noState,pool)
    return pool
```

By defining `initFunction` accordingly, initial tasks can be added. For example, the tasks represented by integers `1 :: Int`, `2` and `3` are added with

```haskell
pool <- newTaskpool $ do
    put 1
    put 2
    put 3
```

The function `eachTask` processes all tasks from the taskpool including dynamically generated tasks. This higher-order function receives the generated taskpool as well as a function to process a single task. It generates thread and waits until all tasks have been processed. `eachTask` is defined independently of the taskpool variant as

```haskell
type TaskProcessing pool task = task -> TPMonad pool ()

eachTask :: Taskpool pool task => pool -> TaskProcessing pool
  task -> IO ()
eachTask pool f = do
    State.runStateT work (undefined, pool)
    return ()
  where work = do
        forkN numCapabilities (thread f)
        -- Wait until all tasks have been processed.
        wait
```

The function `eachTask` uses two helper functions, one for thread handling (`forkN`) and one for task handling (`thread`). The `forkN` function creates the given number of threads.

Additionally, it inserts thread-specific information into the monad's state. Each thread is executed inside the `StateT` monad. The `StateT`'s state includes the accessible parts of the taskpool and the thread's own thread-specific data:

```haskell
type NumberThreads = Int

-- Forks threads that execute the function f. Returns the
-- ThreadIds of the created threads.
forkN :: Taskpool pool task => NumberThreads -> TPMonad pool
  () -> TPMonad pool [ThreadId]
forkN num f = do
    pool <- getPool
    io $ forM [0..num-1] $ \threadState ->
        forkIO $ do
            State.runStateT f (threadState, pool)
            return ()
  where getPool = snd `fmap` StateT.get
        io      = liftIO
```

We made `forkN` available in the `TPMonad` to allow for greater flexibility. As shown in Section 4.3.5, this approach allows to handle complex task dependencies. The function `thread` is called for each forked thread and controls the handling of tasks:

```haskell
-- Thread using the different taskpool implementations.
thread :: Taskpool pool task => TaskProcessing pool task
  -> TPMonad pool ()
thread f = do
    task <- get
    case task of
        Nothing    -> return ()
        Just task' -> do
            f task'
            thread f
```

Thus, the function tries to get a task out of the pool by using the taskpool-specific `get`-operation. Then it calls the function `f` with the task as the parameter. Task retrieval is repeated until no more tasks are available. For more complex scenarios we defined additional functions that allow more control of the different steps. We will describe them at the appropriate location in the next sections.

4.3.3 Exchanging Return Values

Most of the task-based algorithms do not only process tasks but also require a mechanism to store or combine the results of their calculations. Intermediate and final results can be stored in an algorithm-specific data structure if the problem at hand allows it. In particular, the tasks must access different parts, or the data structure must be synchronized. The LU decomposition, for example, uses a mutable array, and different threads modify different parts (see Section 4.5.4). If the algorithm does not use mutable structures, a mutable variable type named `TPVar` can be deployed for data exchange. Basically, a `TPVar` is an `MVar` with an additional operation `waitFor` operation to wait for the results of submitted tasks (see Figure 4.7). Internally, a new thread is forked to compute the forthcoming result. So the currently active thread can process a new task. The next section shows an example for using `TPVar`s.

```haskell
data TPVar a = TPVar {tpVar :: MVar a}

waitFor :: Taskpool pool task =>
  TPVar a -> (a -> TPMonad pool ()) -> TPMonad pool ()
waitFor (TPVar var) f = do
    state <- State.get
    io $ forkIO $ do
        value <- readMVar var
        State.runStateT (f value) state >> return ()
    return ()

-- A TPVar supports other familiar operations. Here, only
-- their type signatures are shown and the typeclass
-- annotation Taskpool pool task => is omitted.
newTPVarIO   :: IO (TPVar a)
readTPVarIO  :: TPVar a -> IO a
newTPVar     :: TPMonad pool (TPVar a)
writeTPVar   :: TPVar a -> a -> TPMonad pool ()
readTPVar    :: TPVar a -> TPMonad pool a
```

Figure 4.7: Type and functions for a blocking mutable variable `TPVar`.

4.3.4 Example: Fibonacci sequence

In this section we show how to calculate the Fibonacci sequence for any `n` in parallel using a taskpool. In particular we demonstrate the use of `TPVar`s. Note that without memoization and switching to a sequential implementation for small `n` the given parallelization approach is rather ineffective, but the sole purpose of this example is to show the use of the functions and types mentioned in the last sections.

In Haskell, the function for calculating the n-th element of the Fibonacci sequence is typically defined recursively by

```
fib 1 = 1
fib 2 = 1
fib n = fib(n-1) + fib(n-2)
```

We begin by generating a taskpool `pool` as well as a variable `result` for the final result. The taskpool is initialized with a tuple consisting of the starting value and the result variable

```
fibonacci :: Int -> IO Int
fibonacci n = do
    result <- newTPVarIO :: IO (TPVar Int)
    pool   <- newTaskpool $
        put (n, result)
```

If the numerical value of the task is less than 3, the base case of the recursion has been reached and the result is stored:

```
    eachTask pool $ \(task, var) -> do
        if task <= 2
            then writeTPVar var 1
```

If the value is greater than 2, two new tasks are spawned and two new `TPVars` are generated:

```
            else do
                v1 <- newTPVar
                v2 <- newTPVar
                put (task-1, v1)
                put (task-2, v2)
```

The `waitFor` function waits until a value is stored in the given variable and then executes the provided function. In the Fibonacci example, we have to wait until the results of both subtasks have been computed. Then the result is stored in the result variable of the task (see Figure 4.8 on page 74):

```
                    waitFor v1 $ \r1 ->
                        waitFor v2 $ \r2 ->
                            writeTPVar var (r1+r2)
```

When all tasks have been processed, the final result has been stored in `result`, which is read and returned:

```
       readTPVarIO result
```

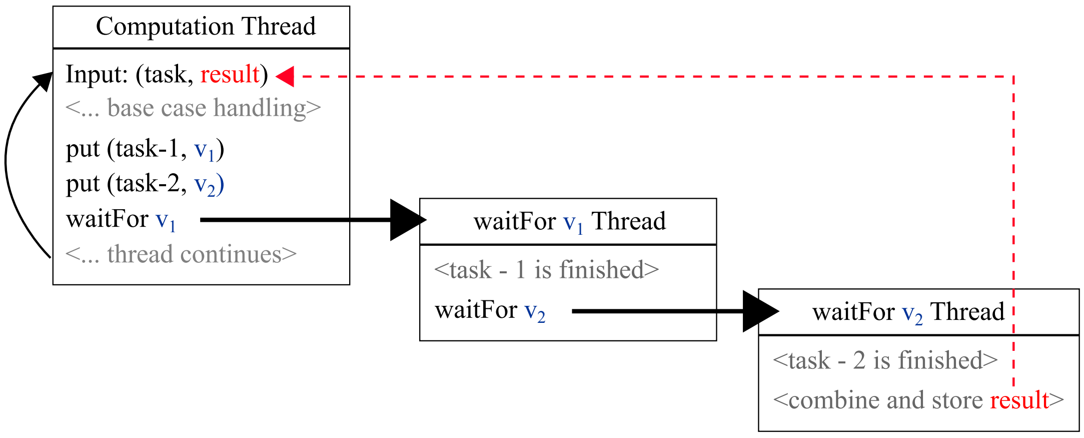


Figure 4.8: Illustration of the `waitFor` function for the Fibonacci example.

4.3.5 Example: Nested Task Dependencies

Some task-based algorithms have more complex dependencies. In this section we demonstrate that the `wait`-operation permits a concise and clear formulation of such algorithms, using a simplified version of a matrix algorithm as an example.

Consider a two-dimensional matrix with dependencies between elements (see Figure 4.9). While all elements in a column can be calculated in parallel, their particular value depends on the prior calculation of the value in the previous column. Although a

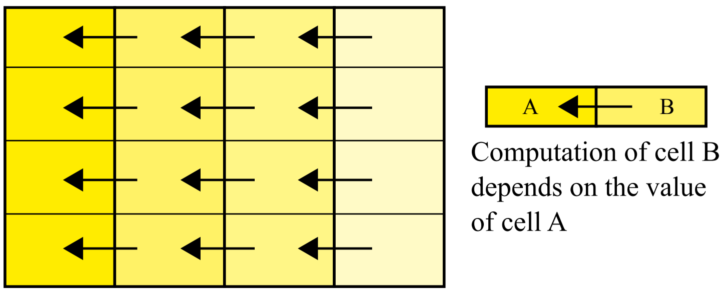


Figure 4.9: Example matrix of task dependencies. Sets of tasks that can be processed in parallel are colored differently.

manual spawning of subtasks might work for this simplified example, it is more complicated in general. A better approach is to divide the calculation into separate iterations. In our example an iteration consists of all tasks in a particular column. These dependencies are formulated by using the `wait` operation as follows. We begin by generating a taskpool without initial tasks and defining the width and height of the matrix:

```haskell
type Coordinates = (Int, Int)

nestedExample :: IO ()
nestedExample = do
    -- Create new global taskpool.
    pool <- newGPool :: IO (GPool Coordinates)

    let width  = 5
        height = 5
```

Instead of using `eachTask` we use a function `taskpool` that works similar to `eachTask`, but neither forks threads nor waits for their processing. Instead, it allows to execute any function in the `TPMonad`:

```haskell
    taskpool pool $ do
        -- Inside TPMonad. No threads forked
        -- and no waiting for their completion.
```

Inside the monad we fork threads that process each task. The function `threadForever` works similar to the introduced function `thread` (see page 71), except that when the taskpool is empty, it continues waiting for tasks instead of returning `Nothing`. Note that we store the thread identifiers (`tids`) of the forked threads:

```haskell
        tids <- forkN numCapabilities (threadForever $
            -- Arbitrary task processing, e.g.:
            \(x,y) -> io (print (x,y))
```

In the next step the set of tasks for a particular column is added. Then the threads started with `forkN` process these tasks. This procedure is repeated until all columns have been processed:

```haskell
        forM_ [x | x <- [1..width]] $ \x -> do
            forM_ [y | y <- [1..height]] $ \y -> do
                put (x,y)
            wait
```

Since a successive iteration can add new tasks, threads do not finish when the taskpool is empty. Therefore they need to be killed explicitly by `exitThreads`

```
        exitThreads tids
```

This function iterates over all thread identifiers and calls `killThread` [82] for each. Note that our own implementation can be easily modified to encapsulate thread identifier handling and thread termination in the taskpool state by either modifying `TPMonad` or adding a state transformer on top of `TPMonad`.

Nested task dependencies can also be modelled by multiple `newTaskpool` and `eachTask` calls. The described approach does not divide iteration handling and has lower overhead, since threads are not spawned for each iteration.

4.4 Taskpool Implementations

In this section we describe our lock-based and software transactional memory-based implementations for global, private and shared taskpools. We draw a comparison between each taskpool variant by demonstrating how to implement them using both lock-based and STM-based synchronization. We explain the global taskpool implementations in detail because they are the basis for the other two variants. For each taskpool variant we start with a general description of the used data structures and types. Then we describe taskpool initialization as well as the `get`, `put` and `wait` operations. We emphasize the `get` operation since it is the most interesting operation with respect to synchronization.

4.4.1 Global taskpools

Given the criteria of Section 4.2.1, no thread should terminate until all tasks have been processed. That means, threads should wait for new tasks even if the global taskpool is empty, because currently processed tasks could still add new subtasks. Therefore threads should only terminate if they are all idle *and* the global taskpool is empty. Also correctness should not depend on the order of (initial) taskpool operations. For example, threads should not terminate if they have been forked while the taskpool is still empty. This allows for a more natural formulation of some task-based algorithms (see Section 4.3.5).

Lock-based Implementation

The complete data structure of the lock-based global taskpool is shown in Figure 4.10. While `gPool` stores the tasks to be processed, the other elements exist solely for termination detection and efficient handling of waiting for new tasks.

```
data GPool a = GPool {
    -- Task storage.
      gPool      :: Chan a

    -- Synchronization and termination detection.
    , gState     :: IORef GState
    , gFinished  :: Chan (MVar ())
    , gWorking   :: MVar (Set ThreadId)
    , gWaiting   :: IORef [MVar ()]
}

data GState = Init | Wait

instance Taskpool (GPool a) a where
    put  = putGPool
    get  = getGPool
    wait = waitGPool
```

Figure 4.10: Data structure and typeclass instance definition for the lock-based global taskpool.

Taskpool Generation. The function `newGPool` generates all elements of the data structure. It sets the `gState` to `Init`, and sets all other elements to their respective empty states, i.e. empty sets and empty lists.

Getting tasks. Getting a task out of the pool is the most complex operation (see Figure 4.11 on page 78). It includes both accessing the taskpool and termination detection. In simple terms, the `get` operation works as follows: The executing thread acquires the lock that is shared by the `put` and `get` operations (①). Then it checks, if there are any tasks in the global taskpool. If so, one of them is retrieved, the lock is released, and the task is returned (③). In case there are no tasks, the thread checks if it is the last working thread (②). If other threads are working, the thread removes itself from the set of working threads. Then it begins to wait and releases the lock (④). If the thread has been the last working thread, termination (of this iteration) begins (⑤).

In the following we give a more detailed explanation of this operation and we show how different parts of the data structure are accessed. The `get` operation starts as it retrieves the single elements of the monad's and thus taskpool's state:

```
1  getGPool :: TPMonad (GPool a) (Maybe a)
2  getGPool = do
3      state@(_, (GPool pool state finish work waiting)) <-
```

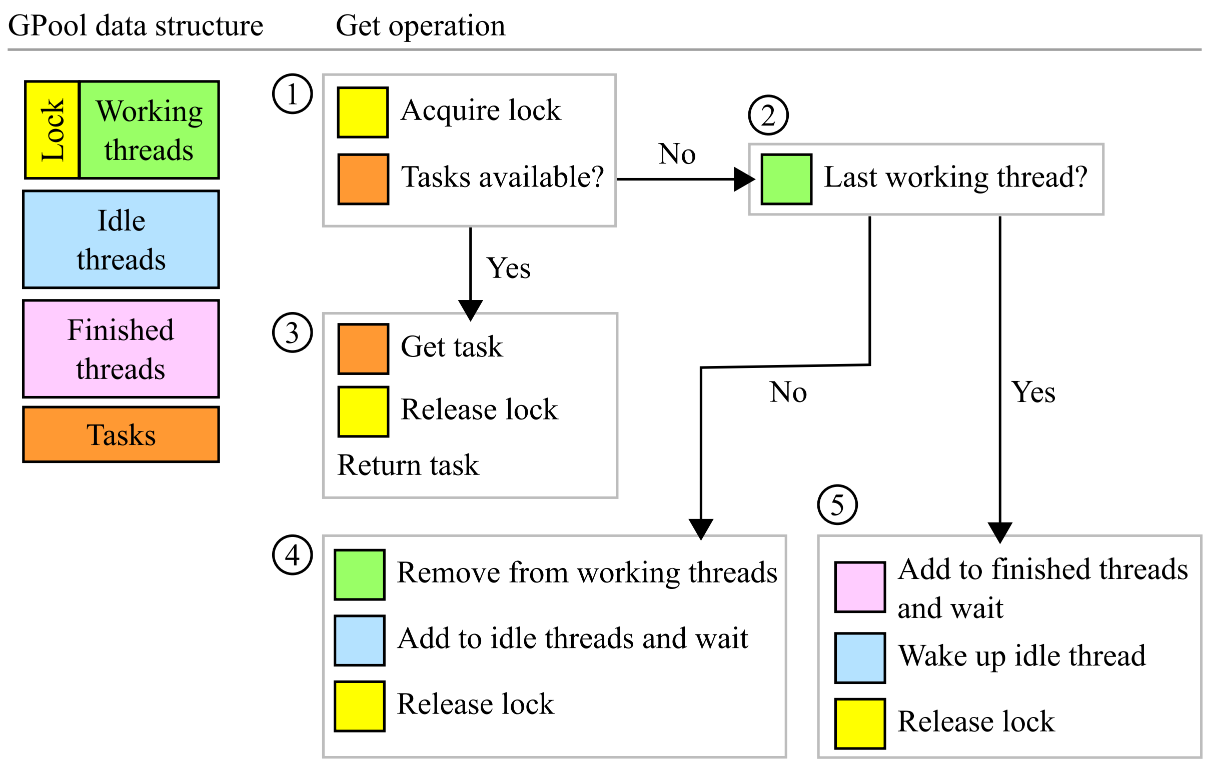


Figure 4.11: Flowchart of the `get` operation in the context of the global lock-based taskpool implementation.

```
        State.get
      io $ do
```

When a task is retrieved from the taskpool, a global lock (`gWork`) is taken to block other concurrent operations (①):

```
      set <- takeMVar work
```

An `MVar` can not only be used as a lock but also stores a value. In the context of the global lock-based taskpool the set of concurrently working threads is stored. These threads are defined by their `ThreadId`s. Instead of simply counting the number of active threads we chose to store the set of `ThreadIds` . This approach allows a more thorough overview of the taskpool's functionality. Since the number of active threads is usually small and set-operations work in $\mathcal{O}(\log n)$ [87], this decision did not cause a significant performance penalty. The next step is deciding if the taskpool is empty:

```
      empty <- isEmptyChan pool
```

Obviously, without blocking, race conditions could occur. If the taskpool is not empty, a task can simply be read by means of the `Chan`'s interface [82]. The lock is released and the retrieved task is returned:

```
    if (not empty)
      then do
        task <- readChan pool
        putMVar work set
        return (Just task)
```

We used a `Chan` to store tasks. Hence tasks are stored in FIFO order. Other task orderings, e.g. LIFO, can be implemented by using the types of `Data.Sequence` [86]. Since these types are not thread-safe by design, additional care has to be taken to ensure thread-safety in the other operations. If the taskpool is empty, the next steps depend on the taskpool's current state: If the taskpool is in the initial state `Init`, threads could have been forked before adding tasks; these threads have to wait. If the taskpool is in the `Wait` state (which indicates that task processing has started), a thread has to check if other threads are still working. If so, the thread has to wait (④). If no other threads are working, the current thread starts the termination of the current iteration (⑤). In the following we describe the handling for the `Init` and `Wait` states in detail.

Init state. In this state, threads may have been forked or tasks may have been added. The `wait` function has not been called yet, i.e. threads have to wait for new tasks and the start of task processing. This is implemented for each thread by generating an empty `MVar` by using the function `makeBlockVar` (see below). This empty `MVar` is added to the list of waiting threads:

```
      -- taskpool is empty
      else do
        op  <- readIORef state
        tid <- myThreadId
        let set' = Set.delete tid set
        case op of
            Init -> do
                tmp <- makeBlockVar waiting
                putMVar work set'
                takeMVar tmp
                <... restart get-operation >
```

To describe our implementation of waiting for new tasks in detail, at this point it is appropriate to explain two general ways to implement waiting: First, a thread can generate an empty `MVar` and block until it is filled (*blocked waiting*). Second, it can acquire a lock, check if a certain event occurred, release the lock, wait some time and retry (*busy waiting*). With busy waiting, the event source does not have to be aware of all listeners, but many locking operations are required and contention is increased. For scalability we used the first approach. It is implemented as follows: A thread that wants to be informed of newly arrived tasks generates an empty `MVar` by means of the function `makeBlockVar`:

```
makeBlockVar :: IORef [MVar a] -> IO (MVar a)
makeBlockVar mvars = do
    var <- newEmptyMVar
    modifyIORef mvars (var:)
    return var
```

This function adds the generated empty variable to the list of other blocked variables. Therefore the `gWaiting` field of `GPool` contains a list of all threads that are currently idle and wait for new tasks. Albeit using only `modifyIORef`, this operation is thread-safe since the thread modifying `gWaiting` still possesses the global lock. After storing the empty variable, the lock is released by writing the set of working threads in which the current one is removed. Afterwards the thread tries to access its blocking variable. When this variable is unblocked, a new task was added or the current iteration should finish (see below). In both cases, the `get` operation is restarted.

Wait state. The `Wait` phase is started when the main thread calls `wait`. This state implies that initial tasks have been added and thus task processing can begin. If the taskpool is empty and other threads are still working (and can thus add tasks), the thread waits by generating a blocking variable in the same way as described above:

```
            Wait -> do
                if (not (Set.null set'))
                    -- Other threads still working.
                    then do
                        <... analogous to 20-23>
```

If no other threads are working, the set of working threads is empty. Therefore the currently running thread has been the last one and all tasks of the current iteration have been finished. The current thread generates a new empty `MVar` `blockVar` and writes it to the `finished` channel. There it will be processed later by the `wait` operation. Afterwards a waiting thread is woken up (see below) and the current thread waits for the release of `blockVar`. This process continues until all threads have written their blockVars to the finished channel. After the MVars in waiting have been unblocked, the current iteration is finished. Consequently the get operation returns Nothing for all threads:

```
                        else do
                            blockVar <- newEmptyMVar
                            writeChan finish blockVar
                            unblock waiting
                            <...>
                            takeMVar blockVar
                            return Nothing
```

To wake up a thread, the first element of the list of blocking MVars is taken and the value () is stored. This action releases the block on takeMVar and the respective thread continues:

```
unblock :: IORef [MVar ()] -> IO ()
unblock w = do
    list <- readIORef w
    unless (null list) $ do
        let (m:ms) = list
        putMVar m ()
        writeIORef w ms
```

Note that the implementations of makeBlockVar and unblock imply a LIFO order. By means of the list operations, waiting threads are added to the front or removed from the front, respectively. So the last added thread will be the first to be unblocked. In practice this is not a problem, since it is of no importance which waiting thread processes a task.

Adding Tasks. To add a new task to the taskpool, the task is written to the global channel and a waiting thread is woken up. For this purpose, the set of working threads is locked (to prevent interferences of concurrent get operations) and an idle thread is woken up as described above.

Waiting for termination. The wait function is called by the main thread. This function controls the handling of dependencies between iterations (see Section 4.3.5). It waits until all initially and subsequently added tasks have been processed. When wait is called, the taskpool's state is set to Wait, and idle threads from the Init state are unblocked.

After task processing has started, the main thread waits until the `get` functions of all threads have added a blocking variable to the `finished` channel. In case there have been as many blocking variables collected as threads have been forked, all threads are blocked and wait for the current iteration to finish. Then a new iteration might be started. In any case, the taskpool's state is reset to `Init` and all blocking variables are released.

In the next section we will show how software transactional memory simplifies the implementation. The lock-based and the STM-based implementation are compared referring to the `get` operation.

STM-based Implementation

The STM-based implementation is simpler than the lock-based one since it does not require locks to protect the taskpool's data structure from concurrent access nor blocked variables for waiting on new tasks. The STM-based data structure is shown in Figure 4.12. Note that there is still an STM-based channel `sgFinished` of type `TChan` [74] to handle waiting between iterations.

```
data STMGPool a = STMGPool {
    -- Task storage.
      sgPool       :: TChan a

    -- Termination detection.
     , sgState     :: TVar GState
     , sgFinished  :: TChan (TMVar ())
     , sgWorking   :: TVar (Set ThreadId)
}

 <...>
```

Figure 4.12: Data structure and typeclass instance definition for the STM-based global taskpool.

Taskpool creation. A taskpool is created the same way as in the lock-based variant. All properties are empty and the initial state is `Init`.

Getting tasks. Again, `get` is the most complex operation. While its implementation is similar to the lock-based variant (see Figure 4.11 on page 78), the `retry` operation of the STM monad reduces the overall complexity to a great extent.

First the `get` operation starts a transaction in which the current thread is removed from the list of working threads:

```
 1 getSTMGPool :: TPMonad (STMGPool a) (Maybe a)
 2 getSTMGPool  = do
 3     STMGPool tasks state finished working <-
 4         fst <$> StateT.get
 5     io $ do
 6         tid <- myThreadId
 7         atomically $ do
 8             work  <- Set.delete tid <$> readTVar working
 9             writeTVar working work
```

Second, in a new transaction it is checked if there are still tasks available in the taskpool. If possible, a task is read, the thread is stored in the list of working threads again, and the task is returned by wrapping it into the `Right` type (see also line number 32 for handling of this result type):

```
10         result <- atomically $ do
11             empty <- isEmptyTChan chan
12             work  <- readTVar working
13             op    <- readTVar state
14
15             if (not empty)
16                 then do
17                     task <- readTChan chan
18                     writeTVar working (Set.insert tid work)
19                     return (Right task)
```

If the global taskpool is empty, further processing depends on the taskpool's current state. If it is in the `Init` state, the function waits by using `retry` until the state of the global taskpool changes. Note that in this case we do not have to generate blocking variables. If it is in the `Wait` state, the number of working threads is checked. If none of the threads are working, an empty `TMVar` is created and written to the `finished`-channel. In this case, we use the same approach as described in the last section:

```
20                 else do
21                     case op of
22                         Init -> retry
23                         Wait -> do
24                             if Set.null work
25                                 -- No other threads working.
```

```
                                          then do
                                              f <- newEmptyTMVar
                                              writeTChan finished f
                                              return (Left f)
                                          else retry
```

Note that the use of `retry` simplifies synchronization. By means of this function threads will be woken up automatically if tasks are inserted. Finally, `get` processes the `result` of the transaction. If the result matches to `Right a`, a task has been returned. If the result matches to `Left b`, all threads are finished. Then the thread waits until the next iteration is started or task processing is finished:

```
           case result of
               Left finishVar -> do
                   atomically $ takeTMVar finishVar
                   return Nothing
               Right task     -> return (Just task)
```

Adding tasks. Adding new tasks is simple. Due to the `retry` operation we do not have to take care of waking up blocked threads. Instead, a task is simply written to the global channel.

Waiting for termination

The STM-based `wait` operation works similar to the lock-based one. The function waits until the number of blocking variables matches the number of threads. Then it reset the state to `Init`, unblocks all variables and the threads continue.

In the next sections we describe two additions to the global taskpool, private thread-local storage and task stealing. Both additions are based on the global taskpool, hence the following descriptions focus on the particular differences.

4.4.2 Private Taskpools

The largest difference between global and private taskpools is that private taskpools have an additional thread-local storage. This storage is used to store a set of processable tasks. For both the lock-based and the STM-based variants this storage is implemented as follows. A private taskpool contains an additional field for an array [80, 81]. Each element of the array handles the local storage for one thread (see Figure 4.13). When the `get` operation is called (see Figure 4.14), it first checks if the local storage contains any tasks (①). If so, they are taken without accessing the global taskpool (②). Hence there is no contention. If no tasks are available locally, the thread tries to retrieve as many tasks as possible from the global taskpool (③). Then these tasks are stored in the thread's local storage (④). In the following, we describe the way thread-local storage was implemented in detail. We also explain how the operations of the lock-based and the STM-based global taskpools were modified.

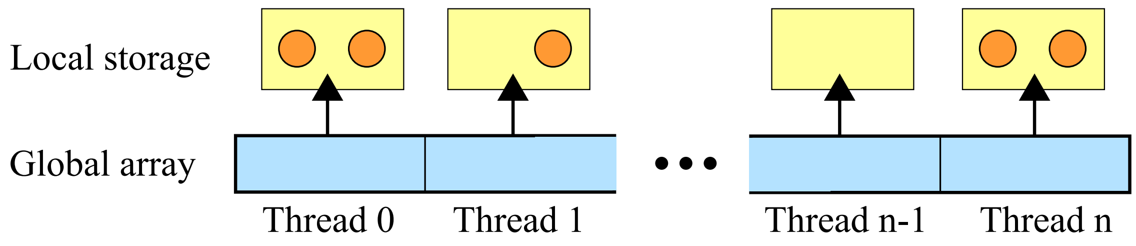


Figure 4.13: Implementation of thread-local storage. All threads have access to a common array, which is indexed by the thread's local data. The array is used to store thread-local tasks.

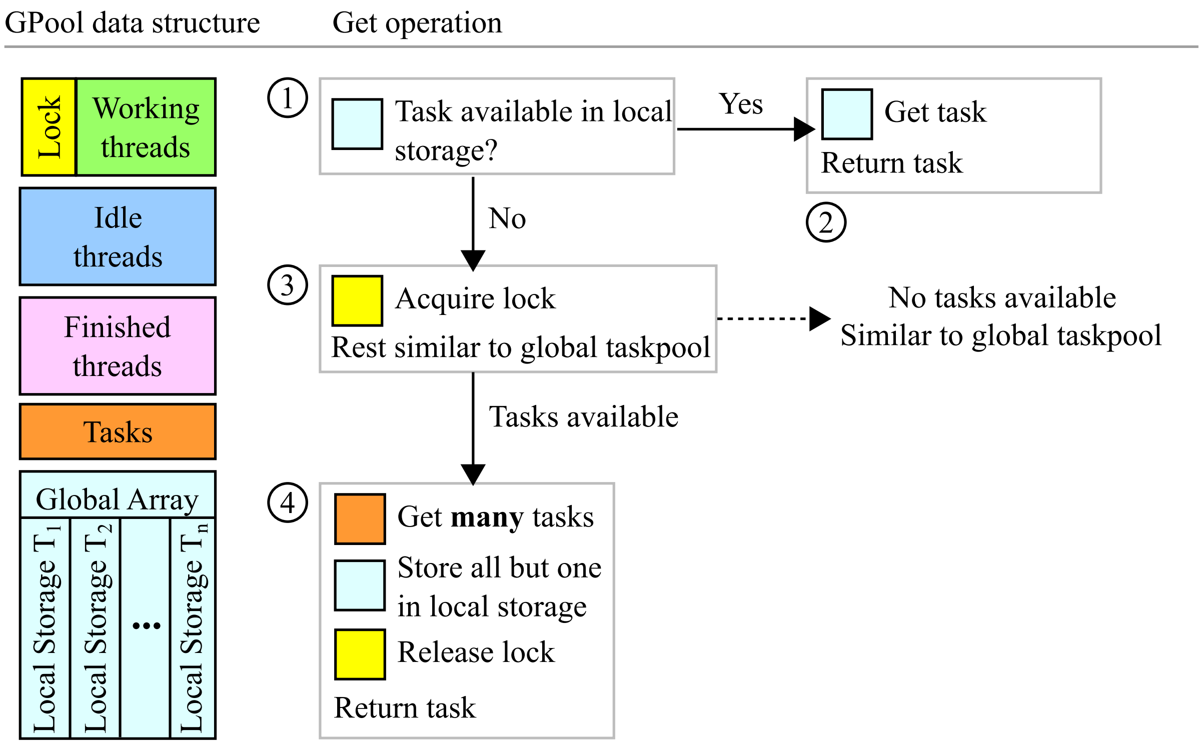


Figure 4.14: Flowchart illustrating the handling of local storage in the context of the `get` operation of a lock-based private taskpool.

Lock-based Private Taskpools

For the purpose of thread-local storage a global taskpool is enhanced to a private taskpool (`PPool`) as follows:

```
data PPool a = PPool {
    -- Task storage.
      pPool      :: CChan a

    -- Local storage.
    , pPrivate   :: IOArray ThreadIndex (Local a)
    , pSize      :: Size -- size of the local storage.
    <...>
}

type Size        = Int
type ThreadIndex = Int
```

To simplify matters, the new types `CChan` and `Local` are explained later on page 87. In the following, the semantics of the function calls should be obvious from the function names, e.g. `newX` to generate a new instance, `readX` to read a value type of `X`.

Taskpool Generation. In addition to the initialization of the already known fields, local storage has to be set up. When a new private taskpool is generated, the size of the local storage must be specified. For each thread the local storage is generated as follows:

```
newPPool :: Size -> IO (PPool a)
newPPool size = do
    chan    <- newCChan
    queues  <- replicateM numCapabilities (newLocal size)
    private <- newArray_ (0, numCapabilities-1)
    forM_ (zip [0..numCapabilities-1] queues) $ \(i,l) ->
        writeArray private i l
    <...>
```

Getting tasks. When the `get` operation is called, each thread first accesses its own local storage addressed through the global array and checks if it can serve the request from it:

```
getPPool :: TPMonad (PPool a) (Maybe a)
getPPool = do
    (index, (PPool pool private size state finished working
      waiting)) <- getState

    io $ do
        local <- readLocal private index
        empty <- isEmptyLocal local
        if (not empty)
            then do
                <... Read an element from local storage>
```

If the local storage is empty, a similar approach as for the global taskpool is applied. That is, if the global taskpool is empty, blocking variables are generated. Hence, waiting is initiated either for a change of the taskpool's state, or for the addition of new tasks. If the global taskpool is not empty, as many tasks as can be stored in the local storage are read at once. Then these tasks are stored in the local storage:

```
            else do
                <...>
                -- global taskpool is not empty:
                (task:rest) <- readCChan pool (size+1)
                writeLocal local rest
                <...>
                return (Just task)
```

Adding Tasks and Waiting for Termination. As the `wait` and `put` operations do not access the taskpools, they are implemented in the same way as for the global taskpool.

Global Task Storage with `CChan`. The global taskpool uses a `Chan` to store and retrieve tasks as well as to check if the taskpool is empty. Unfortunately, a `Chan` blocks on reading if it is empty. That is not sufficient for private taskpools, since their `get` operation retrieves many tasks at once. Therefore we extended a channel by a counter of the number of stored elements, and a non-blocking retrieve operation. The type `CChan` is defined (internally) by

```
data CChan a = CChan {
        cqLock :: MVar ()
      , cqList :: Chan a
      , cqSize :: IORef Int
}
```

This allows for locking and counting. When a task is written, it is stored in the channel. At the same time the internal counter is increased. When tasks should be retrieved, special effort is put into preventing the operation to block: If k elements are requested and n elements are stored, $min(k, n)$ elements are returned and the internal counter is modified accordingly.

Thread-local Storage with `Local`. All threads require a mechanism to store and retrieve their local tasks. Although any container-like structure would be sufficient, we used an array, because it allows a straightforward implementation and fast access, and it also eases the later implementation of task stealing (see Section 4.4.3). The type `Local` is defined by

```
data Local a = Local {
        aList :: IOArray ThreadIndex a
      , aCur  :: IORef Int
      , aMax  :: IORef Int
}
```

This type stores the array, the index of the next returnable element and the array's maximal size. At this point we omit the implementation details for `newLocal`, `isEmptyLocal`, `readLocal` and `writeLocal` since they are self-explanatory.

STM-based Private Taskpools

The STM-based variant of a private taskpool uses the same approach as the lock-based one. That means the STM-based variant uses an additional array to store thread-local data. Therefore, the STM-based private taskpools has the type `STMPPool`:

```
data STMPPool a = STMPPool {
      -- For task storage.
         stmChan      :: STMCChan a

      -- For local storage.
```

```
        , stmPrivate :: TArray ThreadIndex (LocalSTM a)
        , stmSize    :: Int

        <...>
    }
```

We also use a counting channel `STMCChan` as well as a global array that stores `LocalSTM`s. The taskpool operations are similar to the lock-based operations described in the previous section. Instead of generating and using arrays as well as channels in the `IO` monad we apply the respective functions in the `STM` monad. For comparison between the STM-based variant and the lock-based variant regarding the `get` operation, we describe the way tasks are retrieved by means of this operation as well as the implementation of `LocalSTM`.

Getting tasks. If the `get` operation is called, each thread first accesses its own local storage by retrieving it from the global array `private`. Then it checks if it can serve the request from it:

```
getSTMPPool :: TPMonad (STMPPool a) (Maybe a)
getSTMPPool = do
    (index, (STMPPool pool private size state finished
      working waiting)) <- getState

    io $ do
        -- Access local pool and check state.
        (local, empty) <- atomically $ do
            local <- readArray private index
            empty <- isEmptyLocal local
            return (local, empty)

        if (not empty)
            then do
                <... Read an element from local storage>
```

If the local taskpool is empty and there are tasks available in the global taskpool, the tasks are retrieved in the same way as mentioned above. They are then stored in the local storage:

```
              else do
                result <- atomically $ do
                  -- global taskpool is not empty:
                  (task:rest) <- readSTMCChan pool (size+1)
                  writeLocalSTM local rest
                   <...>
            <...>
```

Global Task Storage with `STMCChan`. Likewise the lock-based case, the STM-based `get` operation has to transfer a set of tasks at once. Therefore, we implemented a counting channel for the `STM` monad. This channel is defined by

```
data STMCChan a = STMCChan {
        scqList :: TChan a
      , scqSize :: TVar Int
}
```

Its definition is similar to the definition of `CChan`, although no lock is necessary. The implementation of the respective functions is straightforward and congruent to its `CChan` counterpart.

Local STM-based Storage with a `LocalSTM`. In accordance with the lock-based type `Local` in the `IO` monad we implemented an STM-based version for STM which uses the STM counterpart of an `IOArray`, `TArray` [74]. Its type definition is

```
data LocalSTM a = LocalSTM {
        asList :: TArray Int a
      , asCur  :: TVar Int
      , asMax  :: TVar Int
}
```

4.4.3 Shared Taskpools

In a shared taskpool idle threads try to steal tasks from working threads (see Figure 4.15). First a thread checks if tasks in its local storage are available (①). If so, one is returned (②). If no tasks are available in the local storage and the global taskpool is empty too (③), the thread checks if it is the last working one (④). If so, the approach described for the previous variants is applied. If other threads are working, the thread accesses the local storage of a randomly chosen working thread and retrieves some of its tasks (⑤). These tasks are then stored in the thread's own local storage.

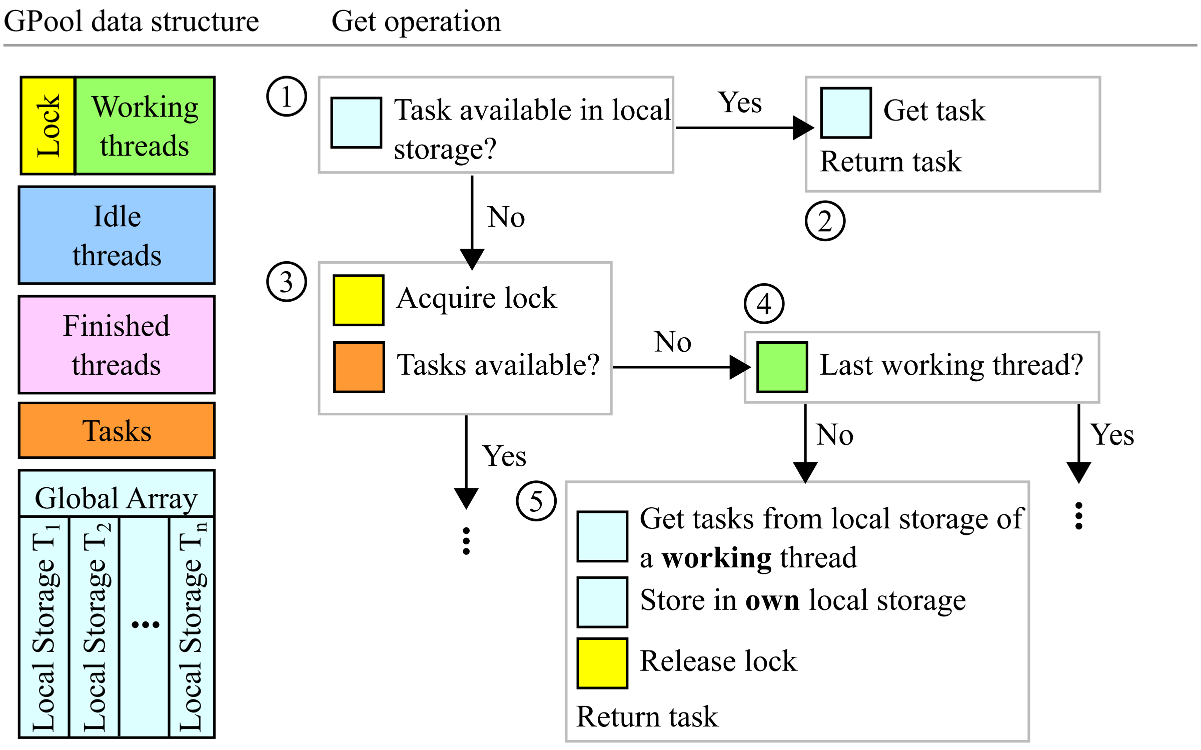


Figure 4.15: Flowchart of the `get` operation in a lock-based shared taskpool.

Lock-based Shared Taskpools

For task stealing a private taskpool's original data structure has to be modified in two ways: First, to synchronize concurrent accesses, all operations using local storage have to be protected by a lock. Second, the type of the set of working threads is enhanced. In addition to the thread identification the set also stores the array index to the thread's local storage. Hence, the type `SPool` for a shared lock-based taskpool is defined by

```
data SPool a = SPool {
        sPrivate :: IOArray ThreadIndex (LocalLock a)
      , sWork    :: MVar (Set (ThreadIndex, ThreadId))
      <...>
}
```

Since task stealing is only implemented in the `get`-operation the `wait` and `put` operations do not require any modifications.

In the `get` operation, if the global taskpool is empty, one of the working threads is chosen randomly and some of its locally stored tasks are stolen, i.e.

```
    <... code similar to a private taskpool>
        else do
            -- Other threads are working.
            tasks <- steal working private
            if (null tasks)
                then <... try again>
                else <... store locally>
```

The `steal` function chooses a working thread randomly. By means of the global array this function accesses the local storage of that thread and removes some of its tasks. In all implementations half of the available tasks are stolen.

STM-based Shared Taskpools

The STM-based variant supporting task stealing is similar to the lock-based one introduced above. Its type `STMSPool` is defined by

```
data STMSPool a = STMSPool {
      sstmWorking  :: TVar (Set (ThreadIndex, ThreadId))
    , sstmPrivate  :: TArray ThreadIndex (ArrayListSTM a)
    <...>
}
```

Compared to the lock-based variant the only difference worth mentioning is the lack of explicit synchronization for the `LocalSTM`s.

4.5 Benchmarks

We executed experiments with the different taskpool implementations on a 2.3 GHz 16-core AMD Opteron 6134 with 32 GB RAM running a Linux-kernel 2.6.38-8 with GHC 7.0.3. Each benchmark consisted of a benchmark problem and a taskpool variant. We ran each benchmark three times from one to sixteen cores. For speedup calculation the mean value of these runs was used. The baseline for the speedup calculations were sequential implementations for each problem. For all taskpool variants the local storage had space for 256 tasks. We chose instance sizes such that absolute runtime with sixteen cores was about thirty seconds.

4.5.1 Requirements for Benchmark Problems

For examining the taskpool performance and scalability we selected benchmark problems based on the following three considerations:

1. The problems should be processable independently of problem-dependent data structures. For example a problem in which results of all tasks were stored in synchronized linked lists, was inappropriate. Its additional synchronization operations would be time-consuming and allow many taskpool-independent optimizations [191].

2. The tasks should have different and unpredictable computation time. As mentioned in the introduction, this aspect is one main motivation for using a taskpool instead of a static distribution of tasks.

3. The tasks generated by the problem should not require much (dynamic) memory: since we test the implementations by means of a large number of tasks, that would stress the garbage collector and thus complicate the interpretation of the results.

4.5.2 Calculating Digits of Pi

Our basic and somewhat artificial first problem refers to the calculation of π for any number of digits. The main section of the benchmark function is shown in Figure 4.16 on page 93. The tasks consist of a list of numbers. Each number specifies the calculation

```
benchmarkPi :: IO ()
benchmarkPi = do
    <...>
    tasks <- <...create list of random numbers> :: IO [Int]
    pool  <- newTaskpool $
        mapM_ put tasks
    eachTask pool calcPi

calcPi :: Int -> ()
calcPi digits =
    showCReal (fromEnum digits) Prelude.pi `pseq` return ()
```

Figure 4.16: Benchmark function for π calculation.

of π to the given number of digits. This problem has the following properties: First, tasks do not spawn subtasks. Hence if the taskpool is empty, all tasks have been processed.

Second, since only arithmetic operations are used, the calculation is memory efficient. Third, since the overall number of tasks is known, we can assign tasks to threads manually beforehand, leaving out the taskpool. We can use this particular property to measure the maximal speedup and to derive the overhead of the taskpools for this scenario.

We benchmarked the π-calculation using two scenarios. The first scenario contains 131272 tasks with a random task size of 100 ± 10 (*pi-small*, ca. 0.0028s/task), the second scenario contains 8192 tasks with a random size of 1000 ± 100 (*pi-large*, ca. 0.09s/task). Since the tasks in pi-small are so short-lived, threads access the taskpool frequently to obtain new tasks. The benchmark in which tasks were assigned manually to threads beforehand is labeled *Manual*. The speedup graphs for pi-small and pi-large are shown in Figure 4.17 and Figure 4.18 on page 96, respectively. In the following we describe and discuss our observations:

a. For short-lived tasks the speedup is better if a lock-based variant is used: Since accesses to the taskpool occur extremely often, the overhead of the STM implementation is measurable.

b. For short-lived tasks, the lock-based shared taskpool performs better than manual distribution: In a manual distribution each thread receives a chunk of all tasks. Since tasks are generated randomly, threads become idle when they have processed their individual chunks. This is prevented if threads are allowed to steal tasks.

c. For sixteen cores, the global STM-based taskpool performs slightly better than the lock-based one. We suppose that STM's optimistic transactional approach [72] as well as its earlier restart of waiting threads (namely, when a transactional variable is changed) are the reasons for this small advantage. We also assume that these are the reasons for the surprisingly small difference between the lock-based and STM-based variants.

d. For large tasks the STM-based variants perform comparably well to the lock-based variants. In some benchmark runs they even slightly outperform the respective lock-based variants. Since the computation time per task is larger the taskpool is not accessed as often. Hence the impact of the transactional overhead on the runtime is not as significant.

e. The extraordinary jumps for both the private lock-based and private STM-based taskpools for large tasks can be explained by an unfortunate task distribution: larger tasks were stored in the thread-local storage and so idle threads could not process them.

Summarizing the results of this benchmark, STM-based and lock-based taskpools have a comparable speedup. This result refers to a scenario with an irregular task distribution and without any addition of new tasks. In the next section the speedup is analyzed in a scenario in which tasks are dynamically generated.

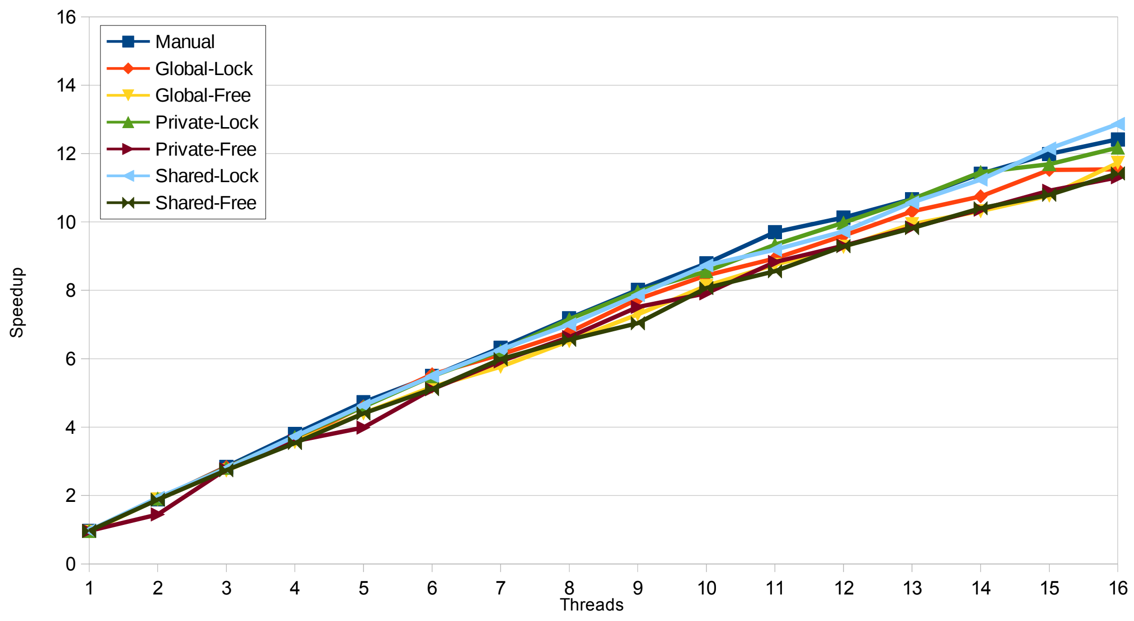


Figure 4.17: Speedup graph for pi-small: 131272 tasks with a random task size of 100 ± 10, average time ca. 0.0028s/task.

4.5.3 A Synthetic algorithm

The next benchmark problem is a synthetic algorithm [122] that involves spawning of subtasks: each task $A(i)$ typically generates two new subtasks, $A(i-1)$ and $A(i-2)$.

$$A(i) = \begin{cases} \{10f\} & \text{for } i \le 0 \\ \\ \{1f\}A(i-2)\{5f\}A(i-1)\{10f\} & \text{for } i > 0 \end{cases}$$

The values in curly braces describe artificial tasks that are computational intensive. As in the last section we chose the calculation of π. By varying f, the amount of computation per task can be modified. By varying the initial i, the number of tasks as well as the degree of irregularity can be chosen. In the base case ($i \le 0$) a single calculation is done and no dynamically generated tasks are spawned. In the normal case ($i > 0$) increasingly

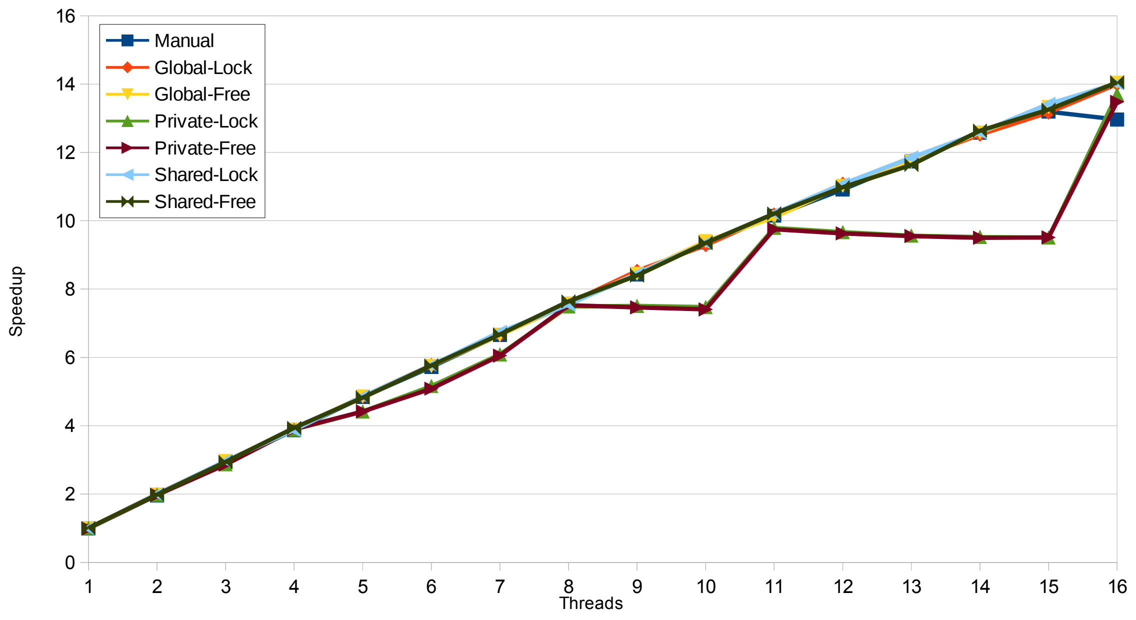


Figure 4.18: Speedup graph for pi-large: 8192 tasks with a random task size of 1000 ± 100, average time ca. 0.09s/task.

larger calculations are interleaved with spawning of smaller dynamically generated tasks. An implementation of the synthetic algorithm is shown in Figure 4.19.

We benchmarked the synthetic algorithm using two problem instances: 1) very small tasks ($f = 1$) and a large depth ($i = 24$), leading to 196418 tasks (*syn-small*), and 2) larger tasks ($f = 100$) and a smaller depth ($i = 15$), leading to 2584 tasks (*syn-large*). The speedup graphs are shown in Figure 4.20 and Figure 4.21 on page 98, respectively.

The results are quite similar to those for calculating π in the previous section. Additionally we made the following observations:

f. For short-lived tasks, the performance of the private and the shared STM-based variants is very low. We suppose that the overhead of the software transactional memory for (synchronized) thread-local storage is high: since the tasks are smaller than in pi-small, the storage is accessed more often. Note that without thread-local storage, the global STM-based variant performs comparably well.

g. For larger tasks, the speedup of the different variants is again comparably high. The private variants are slower, since large tasks remain unreachable for idle threads.

```
benchmarkSynthetic :: IO ()
benchmarkSynthetic = do
    <...>
    let cost  = <... f>
        depth = <... d>
    pool  <- newTaskpool $
        put depth
    eachTask pool $ \i ->
        if i > 0
            then do force $ calcPi (1*f + i - i)
                    put (i-2)
                    force $ calcPi (5*f + i - i)
                    put (i-1)
                    force $ calcPi (10*f + i - i)
            else force $ calcPi (10*f + i - i)


force :: v -> IO ()
force = v `seq` return ()
```

Figure 4.19: Implementation of the synthetic algorithm. Note that the subexpression `+i-i` is necessary to prevent compiler optimizations. The function `force` is used to enforce evaluation of the calculated value inside the processing thread.

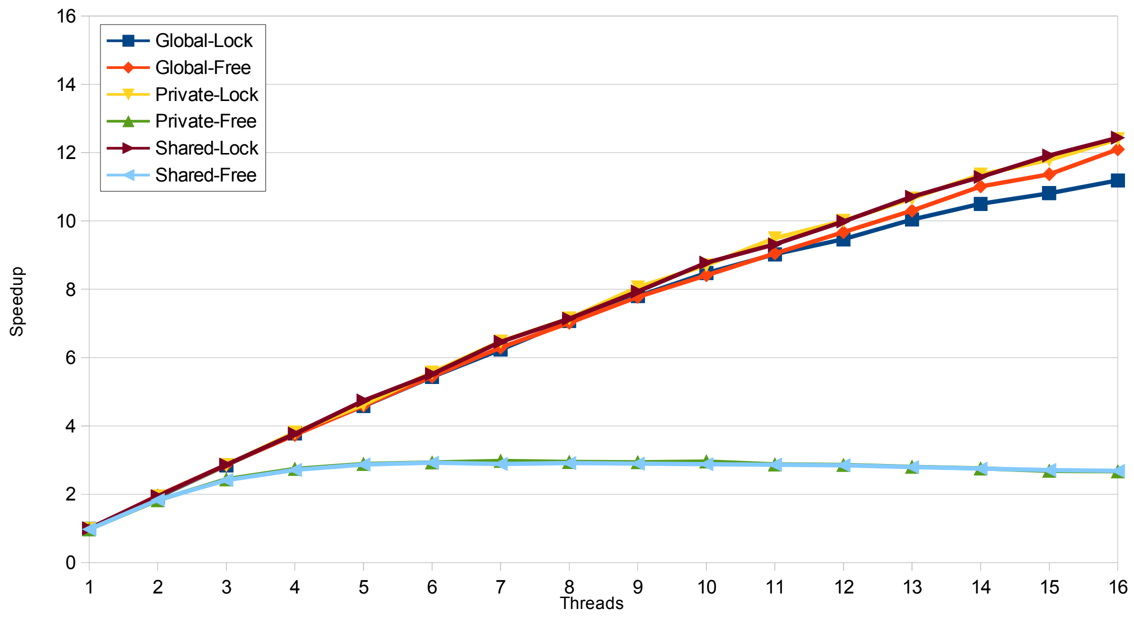


Figure 4.20: Speedup graph for syn-small: 196418 tasks with a task size of 1.

.

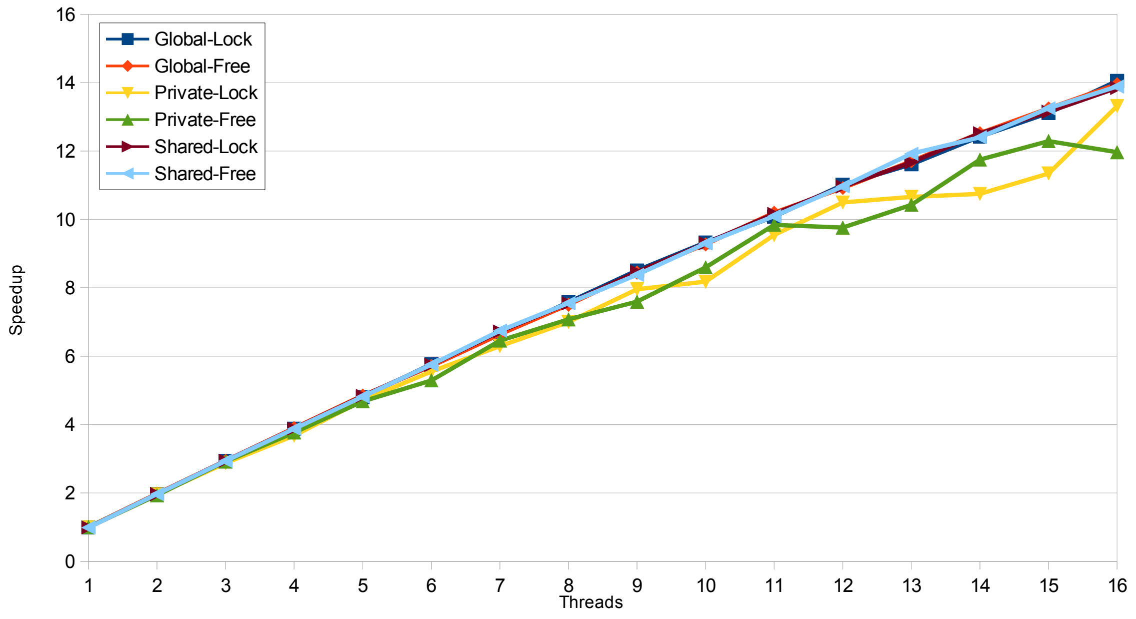


Figure 4.21: Speedup graph for syn-large: 2584 tasks with a task size of 100.

4.5.4 LU Decomposition

Solving a system of linear equations described by $Ax = b$ (with $A \in \mathbb{R}^{n\times n}$ and $x, b \in \mathbb{R}^n$) is common in many scientific applications. A well-known approach is the Gaussian Elimination Algorithm. If the system has to be solved for multiple b, the algorithm can be repeated, but a more efficient approach uses the *LU-decomposition* of A. This decomposition computes a lower triangular matrix L as well as an upper triangular matrix U. If the equation $Ax = b$ has to be solved and L and U are given, $Ly = b$ and $Ux = y$ can be calculated using forward and backward substitution, respectively.

Our approach to parallelizing LU is task-based and uses a block-based technique described in [123]. Let n be the width of the matrix. Then we consider submatrices from $n \times n$ to 1×1. Initial computations have to be done before each submatrix can be processed. We did not parallelize these initial computations since at most n values need to be computed, even for large matrices. Afterwards for each submatrix the rows can be computed independently (see Figure 4.22). These computations constitute the tasks which are added to the taskpool. After the last submatrix has been computed, the initial array contains both the L and the U matrices. The implementation is shown in Figure 4.23 on page 100.

There are two differences compared to the previous benchmark scenarios: First, both the number of tasks and the computation time per task decrease as decomposition

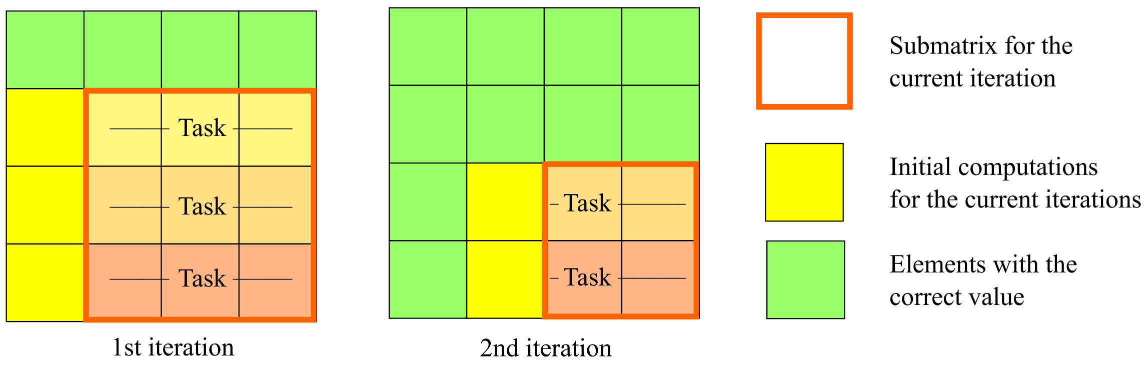

Task
Task
Task
1st iteration
Task
Task
2nd iteration
Submatrix for the current iteration
Initial computations for the current iterations
Elements with the correct value

Figure 4.22: Submatrices, initial computations, and tasks for the first and the second iteration of the LU decomposition of a 4×4 matrix. Simplified for presentation purposes.

progresses. Second, the calculation works in data dependent iterations. With progressing decomposition, the number and size of tasks decreases. For better performance one would apply a sequential algorithm if tasks become too small. We did not apply this optimization to study our taskpools with high contention, too.

In our experiments we measured the decomposition time for a randomly generated 1000×1000 matrix. For the different synchronization variants the speedup graphs are shown in Figure 4.24 on page 101. For the global STM-based implementation computation took about 14 seconds with 12 cores. Hence this was the best implementation. The sequential runtime was about 84 seconds.

We made the following observations:

h) All variants except the shared STM-based and global STM-based ones have a low speedup. The reasons can be seen in the simplicity of the sequential variant, which is implemented by means of a straightforward nested loop. The loop kernel solely consists of math and array operations, which can be efficiently optimized by the compiler. We think that the overhead of the taskpools in combination with the complex task structure led to the bad speedup. Without further optimizations, such as the previously mentioned switch to a sequential variant, it is difficult to achieve a better speedup.

i) Quite unexpectedly, both the global and the shared STM-based implementation perform significantly better than all other variants. We suppose that the optimistic synchronization approach of STM (see Section 2.3.2 and c)) is a reason. In addition, the initially high computation costs of each task led to less contention.

```
benchmarkLU :: IO ()
benchmarkLU =
     <... arr contains the matrix values>
    -- Shortens later code:
    bounds <- getBounds arr
    let rm = readMatrix bounds arr
        wm = writeMatrix bounds arr

    taskpool pool $ do
        forkN numCapabilities
          (threadForever (workRow rm wm n))
        forM_ [0..n-2] $ \k -> do
            forM_ [k+1..n-1] $ \i -> do
                put (i,k)
            wait

  -- calculate the values for one row
  where workRow rm wm n (i,k) = io $ do
        akk <- rm (k,k)
        aik <- rm (i,k)
        let lik = aik/akk
        wm (i,k) lik
        -- calculate rest of row
        forM_ [k+1..n-1] $ \j -> do
            aij <- rm (i,j)
            akj <- rm (k,j)
            wm (i,j) (aij - lik*akj)
        threadforever f = do
            task <- get
            case task of
                nothing -> do
                    io yield
                    return ()
                just t  -> f t
            threadforever f
```

Figure 4.23: Implementation of a task-based LU decomposition.

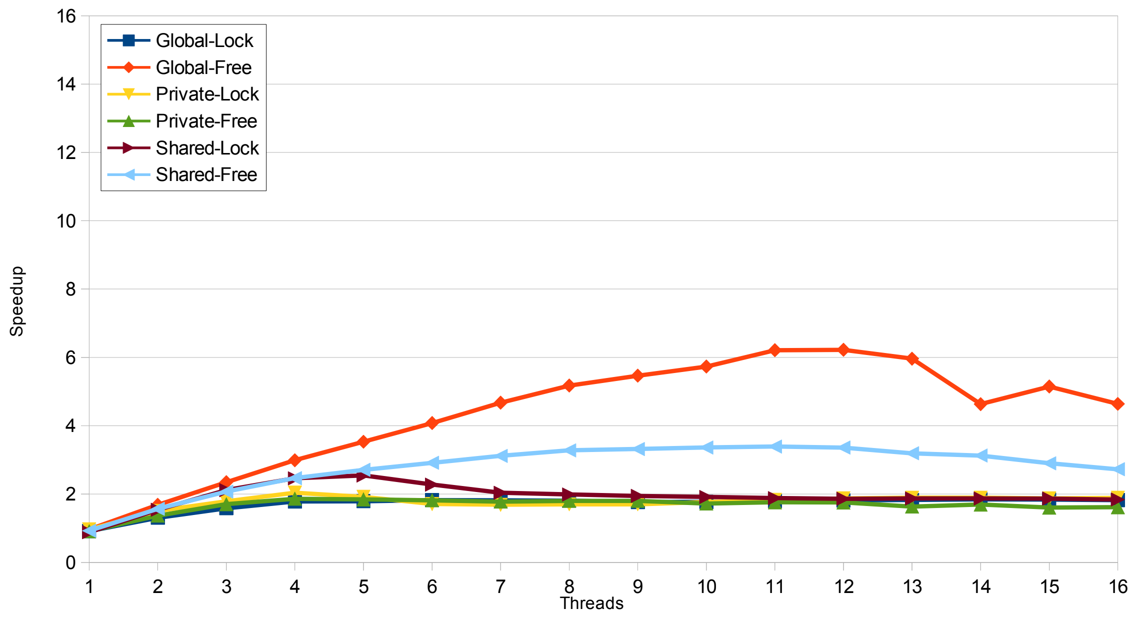


Figure 4.24: Speedup graphs for LU decomposition of a 1000 × 1000 matrix.

4.5.5 Summary

For typical taskpools, synchronization with software transactional memory has proven to be a useful alternative to a traditional lock-based approach. However, for taskpools with thread-local storage, benchmarks reveal a better result for lock-based implementations compared to STM-based implementations. Perhaps the outcome for the STM-based variants could be enhanced by the implementation of a compiler-supported thread-local storage (which would in turn better support STM), or a fundamentally different approach for the implementation of STM-based arrays. Nevertheless, with a transactional approach the developer gives up control. Hence, it can be difficult to anticipate the performance characteristics of an application.

4.6 Related work

There is a huge amount of parallel algorithms that use the taskpool pattern (see Rauber et al. [170] or Grama et al. [123] for an overview). An extensive discussion of the large taskpool design space has for example been given by Mattson [140].

For particular kinds of dependencies between tasks, the other parallelization approaches of Haskell (see Section 2.3.4) can be applied. Data Parallel Haskell can be used if tasks do not dynamically generate tasks and tasks only use pure computations. Then processable tasks are stored in a list and are automatically computed in parallel. The

`Par` monad is an option if tasks do dynamically generate tasks, but their computation is side-effect free. Due to these limitations, our approach is more general and due to the provided abstractions also easier to use. The nofib-benchmark suite [159] defines several (task-based) benchmark problems for semi-explicit parallelization. Example benchmarks are the calculation of mandelbrot sets, matrix multiplication, or raytracing. The respective tasks have an irregular structure, similar to ours, but use only pure computations. An analysis of occurring problems, e.g. space leaks and scalability issues, as well as a discussion of possible solutions is described by Marlow et al. [137]. An analysis similar to ours, but for the implementation of concurrent linked-lists instead of taskpools, has been done by Sulzmann et al. [191]. The authors used different synchronization approaches (STM, `MVars` and `IORefs`). Their study revealed similar results regarding the scalability and performance of STM. In contrast to our work, the other synchronization variants performed quite differently. We think that the main reason for this are the different techniques used for linked-list operations.

The parallel programming language Eden (Loogen et al. [130]) is an extension of Haskell. Eden adds constructs to control the parallel evaluation of expressions. Especially it eases the definition of skeletons [168] and allows a general and efficient definition of e.g. taskpool-based algorithms, as is shown by Berthold et al. [15]. The programming language Java had always supported parallel programming by means of threads [148]. In the last years, its support for more abstract approaches to parallel programming has been improved: For example, the `Executor` class of the concurrency framework [63] allows the definition of tasks and of their dependencies. Then these tasks are executed in parallel, whereby the developer does not have to control thread behavior or task dependency handling. For C++, the Intel Threading Building Blocks provide a template library for high-level parallel programming [171]. The developer only has to define the dependency graph of the tasks by means of a skeleton (see above).

Comparison of different taskpool variants as well as irregular tasks with synthetic problems has been made (for Java) by Korch et al. [122] and (for OpenMP) by Wirz et al. [208]. These studies revealed comparable results regarding the performance differences between global, private and shared taskpools.

4.7 Summary and Conclusion

We have described the implementation of different variants of the taskpool pattern (global taskpools, as well as private taskpools with and without task stealing) in Haskell. Synchronization was implemented by using lock-based as well as STM-based approaches. Each taskpool was benchmarked using two synthetic problems as well as the LU decomposition of a matrix. The following lessons about the difficulty of implementing the taskpool pattern, the respective performance of the variants, and the complexity of optimizations, were learned:

- The implementation of parallel programs using STM is easier and less error-prone than the implementation of their lock-based counterparts. With an STM-based approach, the developer does not have to keep synchronization issues in mind.
- The performance of lock-based and STM-based taskpools is comparable. Due to the optimistic nature of transactional synchronization, STM-based taskpools can even outperform lock-based ones. The sometimes worse performance of the STM-based variants is caused by two reasons: first, in scenarios with high contention the overhead of the transactional model is large. Second, the implementation of particular data structures, e.g. arrays, is at the moment not efficient in the transactional model.
- Finding performance bottlenecks is difficult for lock-based and even more difficult for STM-based programs. For both implementations there are neither advanced profilers nor other development tools. Reasoning about the performance is more difficult for STM than for locks since important details are hidden.

Drawing a conclusion, STM is useful for the implementation of parallel design patterns: it is easier to comprehend, more expressive, and performs comparably well to a lock-based approach. However, this only holds on the assumptions that first the contention is not too high and second the problem at hand does not require thread-local storage. In contrast to the results of the previous chapter, STM performs much better. We think that the main reason is that transactions are shorter, take less time and are therefore less often restarted for taskpools.

Summarizing on the usefulness of abstraction, the experiments in this chapter have shown that abstraction allows for a simpler formulation of task-based algorithms. In particular, Haskell's support for monads and higher-order functions allows to hide details of the underlying parallelization and allows the user to focus on task handling. It is rather difficult to come to a general conclusion regarding low-level synchronization approaches versus high-level ones, since both approaches have their particular advantages and disadvantages (see above). In general, by using the abstraction techniques of Haskell shown in this chapter, it is possible to first use STM for synchronization since it is easier to use. If performance problems occur, this synchronization approach can easily be replaced with a lock-based one.

Chapter 5
Declarative Description and Parallelization of Stencil Computations

In computer simulations the problem space is often partitioned in a grid. To the grid's elements, computations that use the values of adjacent elements, are applied. Stencils are patterns to describe the dependencies within such computations. In this chapter we demonstrate how stencil-based algorithms can be described *declaratively* in an idiomatic and convenient way. Furthermore, we explain the implementation of an execution platform that performs the computations in parallel. Finally we present benchmarks that show the scalability of this approach.

5.1 Introduction

Besides applied and pure sciences, computer simulation (*computational science* [179]) is an important research method. Some computer simulations of real-world phenomena approximate solutions by superimposing the real-world problem space with a discrete grid. Then to each grid element a computation function is applied iteratively. Thereby, the element's value is updated. The values of the function parameters are defined by a spacial pattern called stencil. A *stencil* describes the dependencies within these computations by stating the relative positions of accessed elements. The values at these positions are used in the computation to update the value at the current position. Algorithms that use stencils are called *stencil-based algorithms* (or *stencil algorithms*).

Despite progress in both hardware and software development the demand for increasing processing power for stencil computations is still prevailing. Increased processing power allows for a finer grid resolution and thus more detailed simulations, or reduces the time necessary for finishing a computation. One approach to speed up stencil computations is parallelization. Unfortunately, parallelization of stencil algorithms causes more complexity regarding algorithm design as well as regarding their implementation.

To manage this complexity, a library can be used. A library serves as an abstraction layer, such that rather complex details of the parallelization are hidden from the user. Furthermore, a library can provide (semi)automatic techniques to select problem-specific optimizations. This is even more so, if the library supports a declarative description of the problem. By means of a *declarative* description, the user does not have to state how the stencil-based algorithm has to be executed. Instead, he simply states the problem. A useful library should fulfill some properties:

- The provided functions and types should be flexible enough to be applied to different problems within the problem domain. At the same time, simple problems should be easy to describe.
- The library may offer different underlying implementations if there is not a single most efficient one, and assist or even automatize the choice for the most efficient one out of these. This approach is especially useful if the problem description is declarative.
- The library should efficiently use the available resources of modern shared-memory multicore hardware. Its design should allow for good scalability.

Although many different libraries are listed on the Haskell repository for libraries [79], just a fraction deals with concurrency and parallelization. In particular, there is no library for a concise description of stencils. Thus our research objective is to support stencils in Haskell and therefore to examine, how well parallelization can be hidden for a restricted problem context. For this purpose we implemented the prototype of a library that allows for a *declarative* description and parallelization of stencil-based algorithms (see Figure 5.1). We implemented just a prototype because a full-featured library has

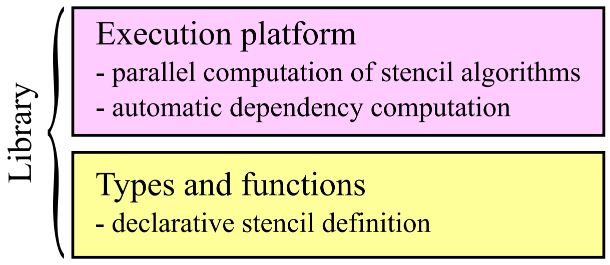

Library
Execution platform
- parallel computation of stencil algorithms
- automatic dependency computation
Types and functions
- declarative stencil definition

Figure 5.1: Overview of the library

many properties that are unnecessary for exploring the feasibility of the declarative approach, e.g. proper error handling, documentation or competitive performance. A particular feature of the declarative description is that the user is able to define any n-dimensional stencil. The purpose for this design decision was to examine how well stencils of any type can be parallelized with a single algorithm. In this chapter we describe

- The definition of types and functions to declaratively describe stencil-based algorithms with arbitrary n-dimensional stencils in Haskell.
- An implementation of an execution platform that allows the parallel computation of stencil-based algorithms on shared-memory multicore systems. In particular, the user does not have to keep parallelization in mind at all.
- An analysis of the scalability of the execution platform. For this purpose we benchmark two stencils which are commonly used in scientific applications, the Jacobi stencil and the Gauss-Seidel stencil.

The execution platform used in the present work is a further development of a previously described prototype [125]. In the previous version we supported two-dimensional grids with additional and limited support for three-dimensional grids. The user had to define additional dependency functions manually for each stencil. In contrast, in the presented work there are no limitations on the number of dimensions. Furthermore, preparations for the parallelization of the stencil computation are no longer needed.

Note that most examples in the present work use problems in two dimensions. The reason lies only in a less complex presentation and not in a general limitation of the declarative description scheme.

The following chapter is structured as follows. In Section 5.2 we give an introduction to stencil-based algorithms. By using voltage diffusion as a running example we demonstrate how our library allows a concise yet understandable formulation of stencil-based algorithms. In Section 5.3 we explain the declarative description scheme as well as the parallel execution platform in detail. Section 5.4 deals with the scalability of the execution platform. In Section 5.5 we compare our work to related research and Section 5.6 contains a summary and a conclusion.

5.2 An Introduction to the Library

In this section we introduce the basic concepts of our library. We explain the voltage diffusion problem which will be our running example in this chapter. We also show how the problem can be described concisely by functions and types of our library. To simplify the visualization of the voltage distribution we focus on two dimensions. Moreover, we sketch the extension to three dimensions for completeness.

5.2.1 The Voltage Diffusion Problem

A well-known problem for stencil-based algorithms is the *voltage diffusion problem* [162]. Here the task is to determine the voltage diffusion of a conductive metal sheet with predefined voltage at particular positions. Figure 5.2a) shows a simplified illustration of this problem for a two-dimensional metal sheet. A solution describes the voltage diffusion for the whole metal sheet. Although the problem can be modeled *algebraically* by a linear partial differential Laplace Equation

$$\frac{\partial v^2}{\partial x^2} + \frac{\partial v^2}{\partial y^2} = 0$$

a solution can only be approximated *numerically*, for example by applying the Jacobi Relaxation Method. The *Jacobi Relaxation Method* works as follows: Initially the problem space is superimposed with a discrete grid (see Figure 5.2b)). For each element of the grid the average value over all directly adjacent neighbors is computed iteratively. For example, for the two-dimensional metal sheet with $V_{i,j}^k$ denoting the voltage at point (i, j)

of the grid in iteration k, the voltage diffusion can be computed by iteratively evaluating

$$V_{i,j}^{(k)} = \frac{V_{i-1,j}^{(k-1)} + V_{i+1,j}^{(k-1)} + V_{i,j}^{(k-1)} + V_{i,j+1}^{(k-1)}}{4}.$$

Typically, this computation terminates if $\forall(i,j) : |V_{i,j}^{k} - V_{i,j}^{k-1}| < \epsilon$, i.e. for all points the difference of their respective voltage is less than ϵ in successive iterations. We call this termination condition *ϵ-condition*. In the next section we show how this formula and the underlying grid can be formulated declaratively in Haskell using our library.

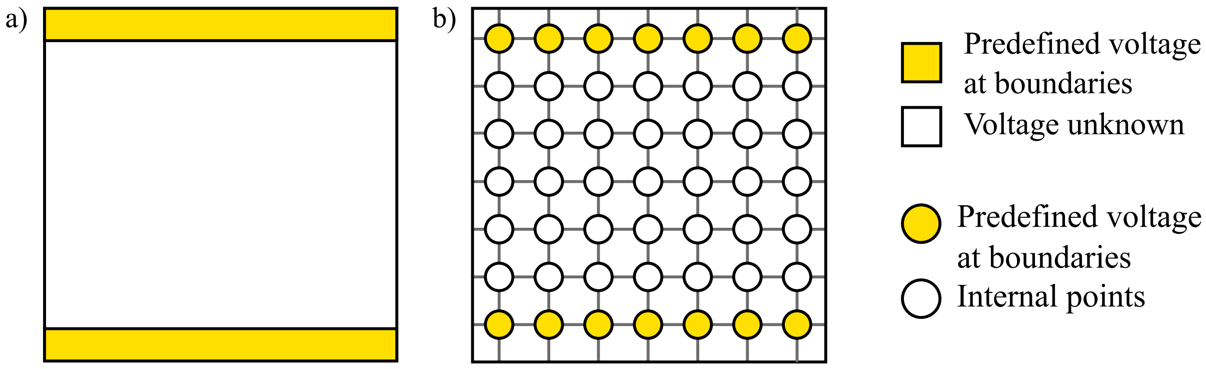

a)
b)
Predefined voltage at boundaries
Voltage unknown
Predefined voltage at boundaries
Internal points

Figure 5.2: a) Voltage diffusion problem for a conductive two-dimensional metal sheet. Voltage is applied at the top and at the bottom. b) Discretized voltage diffusion problem. Voltage is only determined for some points.

5.2.2 Modeling, Solving and Visualizing Voltage Diffusion

In this section we describe the voltage diffusion problem in Haskell by using types and functions of our library. We also show how its solution can be computed and visualized. In addition to the two-dimensional variant we briefly sketch the extension to three dimensions.

Modeling Voltage Diffusion

Each (declarative) problem formulation consists of three parts:

- The stencil that defines the dependencies to update the current element.
- The initial grid's dimension as well as the grid's values.
- The definition of the computation function which is used to compute updated values for each grid element.

In the following we describe each part in detail.

Stencil definition. To update an element, the Jacobi relaxation method uses values of adjacent elements from the previous iteration. In our library, a stencil (type synonym `Stencil`) consists of a list of dependencies (type synonym `Dependency`). Each dependency consists of a list of integers. The integer list's first element states the iteration of the referred element: it is 0 for the current iteration, 1 for the previous one and so on. The rest of the list elements define the relative position for each dimension separately. The two-dimensional Jacobi stencil can thus be defined by

```
computeVoltageDiffusion :: IO ()
computeVoltageDiffusion = do
    let stencil =        -- Jacobi stencil
            [ [1,-1,  0]     :: Dependency
             ,[1, 0, -1]
             ,[1, 1,  0]
             ,[1, 0,  1] ] :: Stencil
```

Another well-known stencil is the Gauss-Seidel stencil. Here, the referred values of the left and upper element are retrieved from the current iteration instead of the previous one. Apart from that, the Gauss-Seidel stencil is analogous to the Jacobi stencil. While the Gauss-Seidel stencil converges faster, it complicates parallelization as we will show in Section 5.3. The two-dimensional Gauss-Seidel stencil can be formulated as

```
    let stencil =        -- Gauss-Seidel stencil
            [ [0,-1,  0]     :: Dependency
             ,[0, 0, -1]
             ,[1, 1,  0]
             ,[1, 0,  1] ] :: Stencil
```

For comparison, both stencils are shown in Figure 5.3. Note that both stencils can be used for the voltage diffusion problem.

Grid initialization. Grid initialization starts by stating the grid's size. On the one hand, the larger the grid, the more grid elements exist and hence the more detailed the simulation is. On the other hand, the more grid elements exist, the more time is needed for their computation. For our example, the grid's dimension is 40×40:

```
        size = [40, 40] :: Dimension
```

The grid's size is annotated by a type synonym called `Dimension`. After the grid's size has been stated, the grid has to be filled with initial values. Our library defines functions

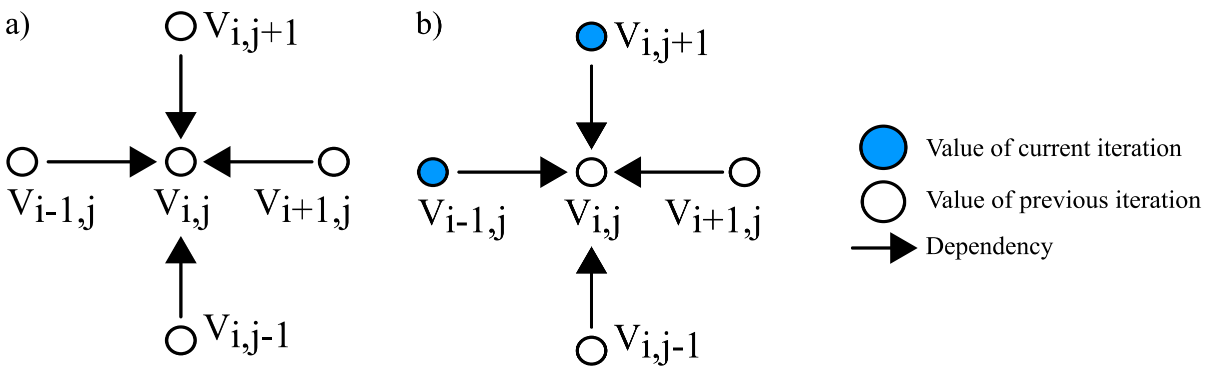


Figure 5.3: Visualization of a a) Jacobi stencil and b) a Gauss-Seidel stencil. In the following figures we use the coloring of values to refer to the current and previous iterations without explicitly stating so.

to comfortably fill values at the boundaries of two-dimensional grids; other fill patterns are easy to add. For our example, at the bottom the voltage is 50 (`B 50`) and at the top it is 100 (`T 100`). Each boundary value has the type `Border`:

```
          borders = [B 50, T 100] :: [Border]
      grid <- getGrid stencil size
      gridBorders grid borders
```

The function `getGrid` returns an empty grid whose boundaries are filled by `gridBorders`. From the user's point of view, a grid resembles an array and provides typical functions to access its values.

Computation function definition. To update the elements, the values of the neighbors referred to by the stencil have to be combined in a computation. For the Jacobi relaxation method, for instance their average is computed. Computation functions are annotated with the type synonym `Function`. A `Function` receives the position of the currently computed element as well as the values of the neighbors referred to by the stencil. Note that the element's position is not a typical part of a stencil computation. Nevertheless, it allows for more flexibility in the grid definition. For example, to simulate non-conductive areas, particular areas of the grid can have a constant value. Since for the Jacobi relaxation method the position of the element is not necessary, the computation of the average is defined by

```
          compute :: Function
          compute = (\_ values -> sum values / 4)
```

In this section we have demonstrated how to model a stencil-based algorithm for the simulation of voltage diffusion using just a few lines of Haskell. We did not have to specify informations outside of the problem domain. In particular, some stencils, e.g. the Gauss-Seidel stencil, create dependencies between different elements within the same iteration. Since our library allows to describe the problem declaratively, the user does not have to pay any attention to the details of the computation. In the next section we show how the stencil computation can be initiated.

Computing Voltage Diffusion

The focal idea of a declarative stencil library is to solve a stencil-based problem without the user having to state explicitly how elements should be updated. For that reason this section is quite short. To compute the solution to the voltage diffusion problem for an ϵ-condition with $\epsilon = 0.1$, we call the `run` function as follows:

```
    let condition = Epsilon 0.1 :: Condition
    run grid stencil compute condition
```

After the `run` function is finished, the voltage of each element is stored in the grid. For saving the grid's values the library defines various file-based output formats, e.g. a table for two-dimensional matrices or a list of points with their respective values. The saved values can be visualized, as we show in the next section.

Visualizing the Voltage Diffusion Solution

To graphically present two-dimensional and three-dimensional grids, we wrote shell scripts for a standard Linux system which use the well-known graph plotting program gnuplot [58].

Before the result of a computation can be visualized, it has to be stored in a file first. To save the result in a file named `output`, the following code can be used:

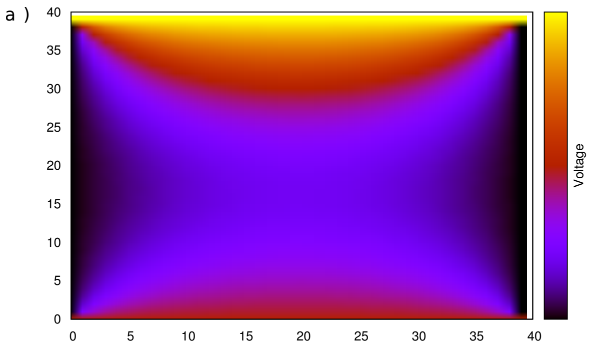
```
    let stringRepresentation = show grid
    writeFile "output" stringRepresentation
```

To present the result on screen, we call

```
    $ visualizeSolution output
```

in a command shell. Other output formats, e.g. storing the visualization as an image, can also be defined. Figure 5.4a) shows the voltage diffusion if voltage is applied to the top as well as the bottom of the metal sheet. Another example, with the voltage source in the center is shown in Figure 5.4b).

In the next section we briefly depict how the description has to be extended to compute a solution for the three-dimensional variant of the voltage diffusion problem.



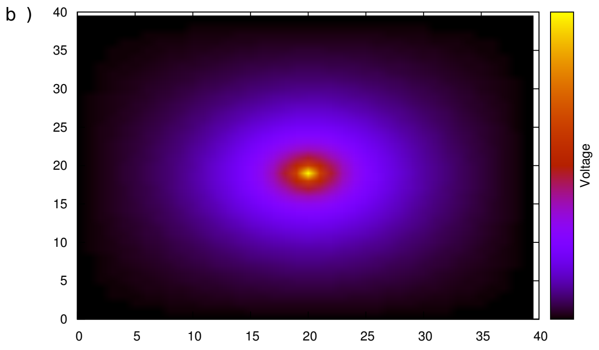


Figure 5.4: a) Voltage distribution on a conductive two-dimensional metal sheet with voltage applied at top and bottom. b) Voltage distribution on a conductive two-dimensional metal sheet with a voltage source in the center.

Defining, Computing and Visualizing a Three-Dimensional Voltage Diffusion

The three-dimensional voltage diffusion problem is similar to the two-dimensional one, but the additional third dimension has to be kept in mind when the problem is defined. Hence, the three-dimensional voltage diffusion problem can be described as follows:

```
computeVoltageDiffusion :: IO ()
computeVoltageDiffusion = do
    let size      = [20, 20, 20] :: Dimension
        stencil   = -- 3D Jacobi stencil
          [ [1,-1,  0,  0]   :: Dependency
           ,[1, 1,  0,  0]
           ,[1, 0, -1,  0]
           ,[1, 0,  1,  0]
           ,[1, 0,  0, -1]
           ,[1, 0,  0,  1]]  :: Stencil
        compute :: Function
        compute = (\_ values -> sum values / 6)
        <...>
```

For presentation purposes of the visualization we reduced the resolution of the grid. Voltage is applied at both sides of the cube (code not shown). The `run` function is called analogous to the two-dimensional case. Again the result is visualized, see Figure 5.5 on page 114. The illustration takes two different perspectives into account.

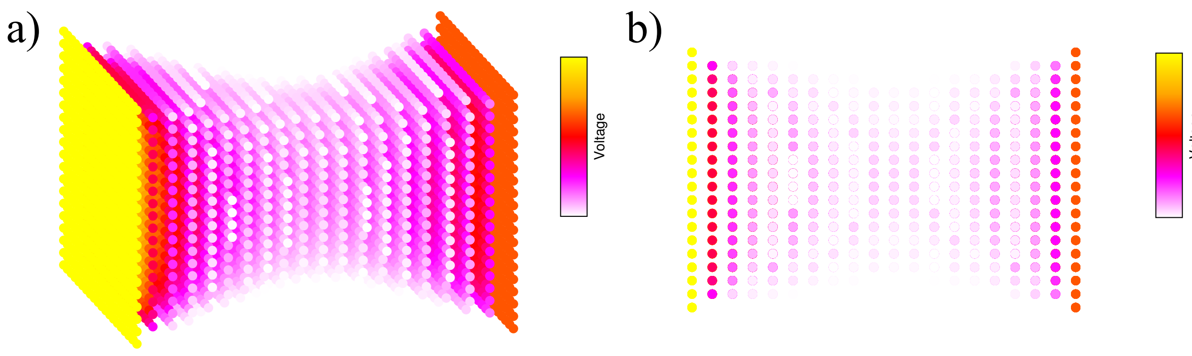


Figure 5.5: Voltage distribution in a conductive three-dimensional metal sheet with different voltages applied at the left and right side. a) isometric view b) side view.

5.3 Implementation

In this section we describe the design and implementation of the library. First we motivate the use of lists combined with type synonyms for the definition of stencils. Continuing the running example, the internal representation of the two-dimensional voltage diffusion problem is illustrated. Then we describe the implementation of the execution platform which allows for a parallel execution of a stencil-based algorithm. Finally we describe the multi-dimensional grid structure as well as the automatic dependency computation which is necessary for the parallel execution.

5.3.1 Lists versus Algebraic Data Types and Tuples

The core of a stencil-based algorithm is the definition of the stencil. Since the library should allow to model stencil problems of any dimension, we can neither use algebraic data types nor tuples. Both approaches are limited at compile time regarding their number of parameters. Thus they do both not allow for an easy formulation of a stencil whose length depends on the problem at hand. Instead for the definition of stencils we use lists since they can be of any length.

Type safety is a common problem with lists, especially if different lists contain elements of the same type. For example, the type system can not determine that `[100,100]` is to be used as the grid dimension and that its incidental usage in a stencil definition is to be dealt with as an error. To prevent these kinds of problems we use type synonyms to distinguish between different types of lists as well as to annotate functions. In our particular use case we think that type synonyms entail to an easier reading of the code compared to types defined by `newtype`. For example, if stencils were wrapped in types instead of being annotated by type synonyms, the definition of the Jacobi stencil (see page 110) would be written as

```
    let stencil = Stencil
            [Dependency [1,−1,  0]
            ,Dependency [1, 0, −1]
            ,Dependency [1, 1,  0]
            ,Dependency [1, 0,  1]]
```

In contrast to type constructors, type synonyms can be omitted conveniently. If the user decides to add them explicitly (as shown in the previous section), the compiler can still find type errors.

5.3.2 Internal Grid Representation

In this section we illustrate the internal grid representation by using the voltage diffusion problem and the Gauss-Seidel stencil for an example: The metal sheet is partitioned into a grid with dimensions 6 × 6 (see Figure 5.6a)). Since the Gauss-Seidel stencil refers to current elements as well as to elements from the previous iteration, a copy of the grid is used to refer to past values.

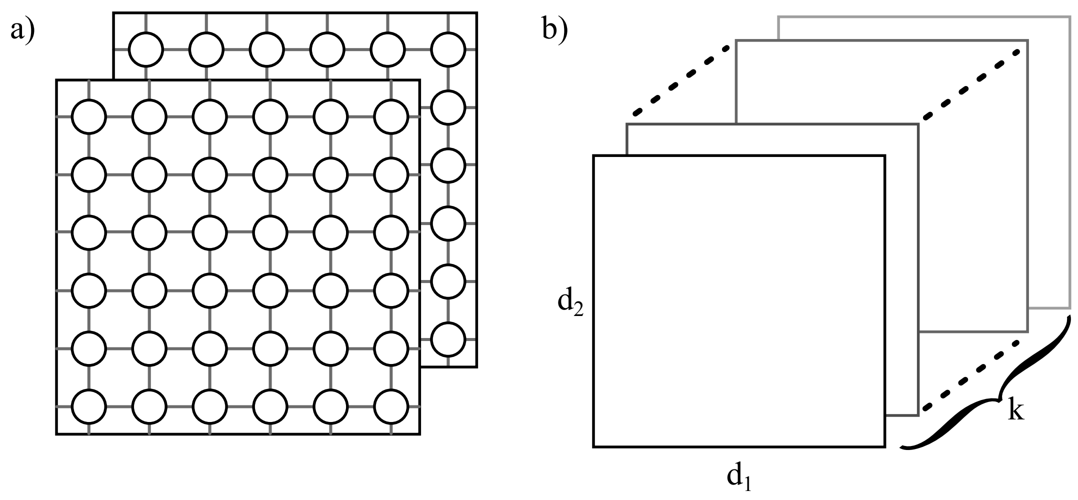


Figure 5.6: a) Grid partitioning of the metal sheet from the voltage diffusion problem. The second grid allows to refer to values from the previous iteration. b) Generalized illustration of the internal grid structure of a two-dimensional grid with dimensions $d_1 \times d_2$ and a stencil referring to k previous iterations at maximum.

Generally, elements of previous iterations are stored by using an additional dimension in the grid: For a grid with dimensions $d_1 \times d_2 \times \ldots \times d_n$ and a stencil which refers to k previous iterations at maximum, a grid with dimensions $k \times d_1 \times d_2 \times \ldots \times d_n$ is used internally (see Figure 5.6b)). By translating the reference to an iteration, it is guaranteed that the values of the previous k iterations are accessible: In iteration i, the element at grid position $x_1 \times x_2 \times \ldots \times x_n$ of iteration j can be accessed at position $((i+j) \mod (k+1)) \times x_1 \times x_2 \times \ldots \times x_n$. The details of the grid implementation are discussed in Section 5.3.4.

5.3.3 Parallel Execution of Stencil-based Algorithms

In this section we motivate the use of blocks as a fundamental design choice. Then we describe how element updates are computed in parallel taking their dependencies into account.

Using Blocks for Parallelization

In a stencil-based computation, grid elements can be computed in parallel. Nevertheless it would be inefficient to do so on a per-element basis. Since the computation of a single element is usually fast, the overhead of synchronization would be much larger than the gain achieved by parallelization. The common approach to reduce the parallel overhead is combining many small tasks into a group (see [123, 140] and Section 2.1). We follow this approach and call such a corresponding group of elements a block.

A *block* is an n-dimensional continuous part of the grid that is processed by a single thread. A block consists of a list of separate intervals for each dimension, i.e.

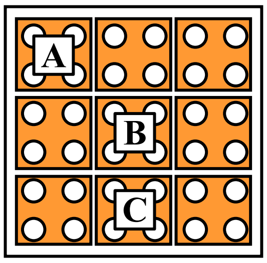
```
type Block    = [Interval]
type Interval = (Int, Int)
```

The *size* of a block (*block size*) is the cross product of the lengths of each dimension. The block size is an important factor in determining the amount of possible parallelization: If the block size is too small, threads process their respective blocks too fast. Therefore to get a new block they have to synchronize frequently and the overall performance is low. If the block size is too large, threads may become idle. This causes bad load balancing and again results in bad overall performance. For our implementation, we define a minimal length of 20 for each block dimension. In practice, grids are much larger, such that a sufficient number of blocks is available for parallel processing. We chose 20 by experimentation on our test machine, but the user can specify other values. This might result in better performance on his particular system, e.g. due to other cache sizes.

Similar to stencil dependencies, blocks have dependencies from other blocks: block B is said to *depend* on the block A (or, block B has a dependency on block A), if elements in B depend on elements in A. Analogously, an element *depends* on another element, if a dependency from iteration 0 refers to it.

Continuing our running example, Figure 5.7a) shows an exemplary block division with a block size of [2,2] for the voltage diffusion problem. Figure 5.7b) shows its formal representation by means of the introduced types. Note that the formal representation is hidden from the user and only used internally.

a)



b)

```
A = [(0,1), (0,1) :: Interval] :: Block
B = [(2,3), (2,3)]
C = [(2,3), (4,5)]
```

Figure 5.7: a) Block division of the voltage diffusion grid. Blocks are marked orange. b) Formal representation using the types `Interval` and `Block`.

Computing Element Updates (in Parallel)

The computation of grid elements is divided in two phases (see Figure 5.8) on page 118. First we present an overview of these phases. Then each phase is described in detail.

In **phase I**, repeatedly used components are precomputed (①). For that purpose, first blocks that do not depend on other blocks are determined. We call these blocks *initial blocks*. Second, the correct order of element updates for single blocks is computed in parallel. A correct order in which elements are updated is necessary for particular stencils (e.g. Gauss-Seidel). This prevents that values of elements from the current iteration are referenced, although they have not been updated yet. A more detailed explanation of the problem as well as our solution is given in Section 5.3.5. Afterwards, the threads which process blocks are started (②).

Phase II is the iterative part of the stencil-based computation. Blocks which have their dependencies fulfilled are stored in a shared queue; note that this approach is similar to a taskpool-based algorithm (see Chapter 4). At the beginning of each iteration this queue is filled with the positions of the precomputed initial blocks of phase I (③). Each thread accesses the queue to retrieve a block (④). Using the precomputed order, elements of the retrieved block are updated. Afterwards, the dependencies of yet unprocessed blocks are checked. If the processed block is the last one that is needed to fulfill all dependencies of an unprocessed block, this block is added to the queue. Threads retrieve blocks until every block of the present iteration has been computed (⑤). To determine termination, the termination condition is checked. If it is not fulfilled, the initial blocks are added again to the queue to start the next iteration (③). Otherwise computation stops (⑥).

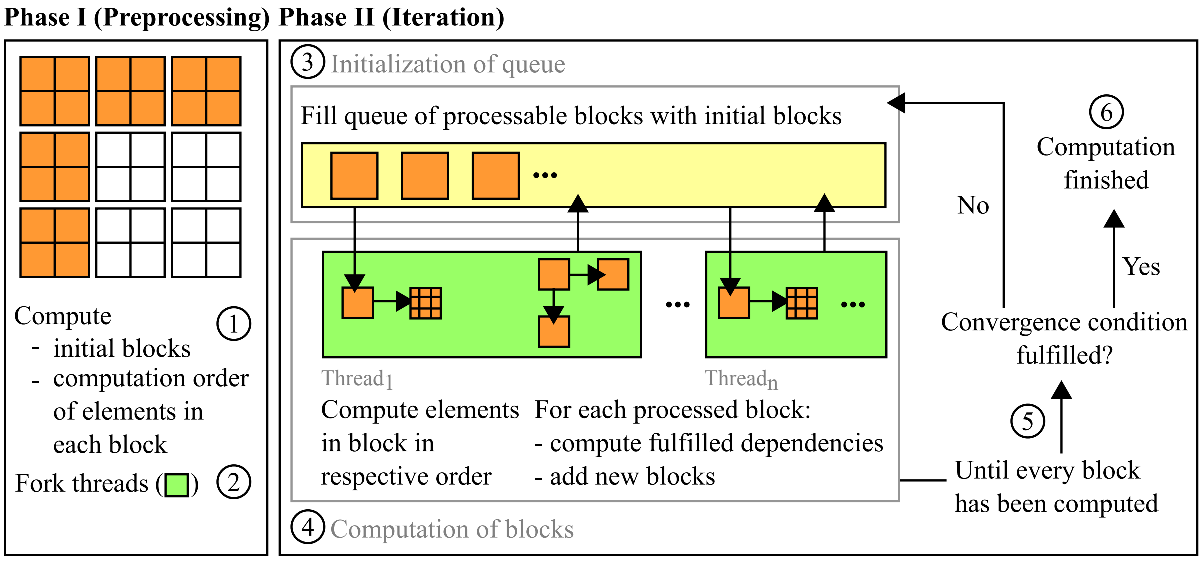

Phase I (Preprocessing)
Phase II (Iteration)
Compute
- initial blocks
- computation order of elements in each block
1
Fork threads (🟩)
2
3 Initialization of queue
Fill queue of processable blocks with initial blocks
...
Thread1
Threadn
Compute elements in block in respective order
For each processed block:
- compute fulfilled dependencies
- add new blocks
4 Computation of blocks
No
6 Computation finished
Yes
Convergence condition fulfilled?
5
Until every block has been computed

Figure 5.8: General overview of the (parallel) element computation.

Phase I (Preprocessing). In the preprocessing phase parallelization is straightforward. While the computation of the initial blocks is fast and thus parallelization is unnecessary, computing the order of element updates is computationally expensive and thus parallelized as follows: First, the position of every block is computed and stored in a channel. Then threads are forked. Until all blocks have been processed each thread retrieves one block at a time and computes the order for element updates for this particular block. To allow fast access to the order of elements of a block in phase II, the order is stored in a structure with type `ElementOrder`:

```
type ElementOrder = BlockArray (IOArray Int Position)
type BlockArray a = IOArray Int a
```

A `BlockArray` is a one-dimensional array. Each block is mapped to a unique position in this array (see Section 5.3.4 for the basic idea of this mapping). This type of mapping is called *block-indexed*. Each array element contains another array. This array stores an order of positions for element updates in the respective block.

Continuing our running example (although slightly modified for presentation purposes), Figure 5.9 shows a part of a `BlockArray` and a possible order of correct element updates for an upper left 4×4 block and a Gauss-Seidel stencil. Phase I is finished by forking threads which will process blocks. Subsequently, phase II is started.

Phase II (Iteration). In phase II, blocks of the grid are processed repeatedly until the predefined termination condition of the problem is fulfilled. Figure 5.10 shows a detailed illustration of this phase. Each of the threads forked at the end of phase I executes

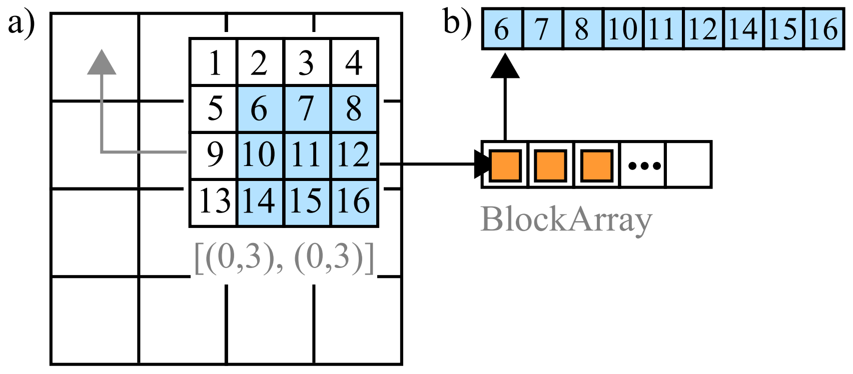


Figure 5.9: Storing element updates. a) A grid divided into 4×4 blocks. The upper left block is shown in detail. The numbers define the order in which elements are updated for a Gauss-Seidel stencil. b) The block array and a possible order for the correct update of elements for a Gauss-Seidel stencil.

the following five steps: First, a thread retrieves a block from the input channel (①). Second, the precomputed positions for the element updates are retrieved (②). Since the orders of element updates are never modified, no synchronization is necessary. By means of these positions, elements are then updated in the correct order (③). To determine which blocks have their dependencies fulfilled, another block-indexed array of type `BlockArray (MVar [Block])` is used. If a block has been processed, the (fulfilled) dependencies of dependent blocks are updated (④). Since several threads may access a block's dependencies at the same time, each array element is synchronized by an `MVar`. If all dependencies of a block are fulfilled, it is stored in the input channel for processing (⑤). Finally, the maximal ϵ-difference is sent to the response channel which has the type `Chan Double`. The ϵ-values of all blocks are used to compute the maximal ϵ value of the iteration.

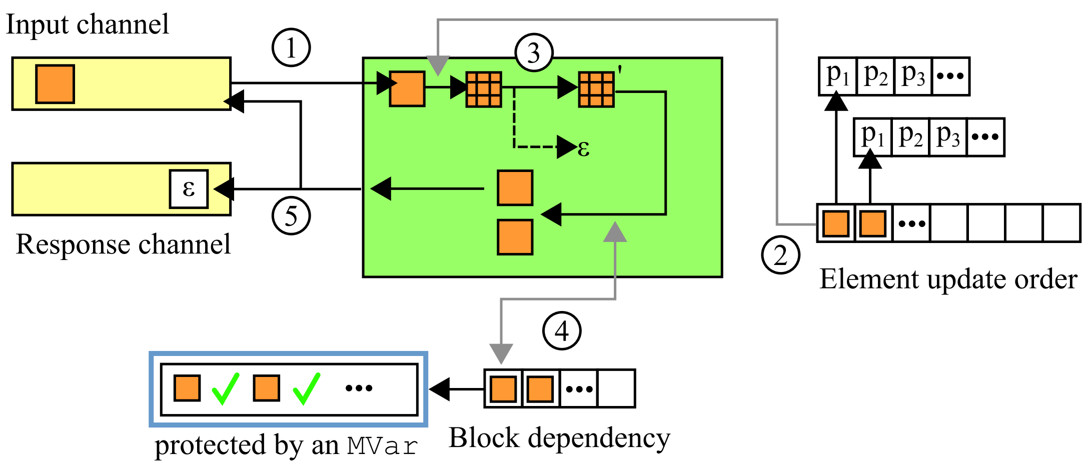


Figure 5.10: Flowchart for the phase II in the parallel element update.

5.3.4 Implementation of a Multidimensional Cache-aware Grid

In this section we describe the design and implementation of a multidimensional grid type which is used to store the grid elements. It is difficult to design a *general* cache-aware algorithm or data structure (e.g. [59]), but in our approach we use the particular access structure of stencil algorithms such that caches can be well utilized.

Our design focuses on minimizing the effects of false sharing while increasing data locality at the same time (see Section 2.1). In stencil-based computations, the update of an element requires only values of adjacent neighbors as well as values from previous iterations. If blocks are arranged such that all elements of a block *and* elements from previous iterations are aligned in memory continuously, data locality is improved (see Figure 5.11). With the aid of such an alignment, a thread which processes a block will predominantly affect memory cells that are only present in its own cache line. Thus unnecessary cache reloading is reduced. In the following we explain the core type and core function of our implementation.

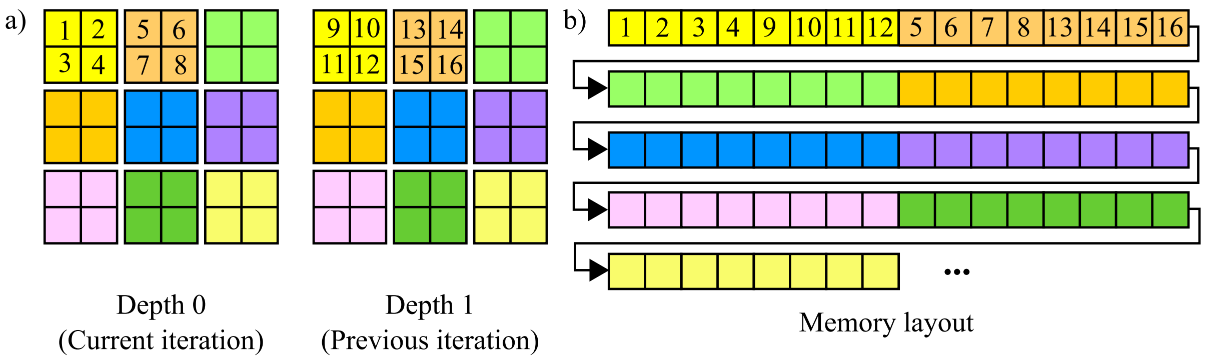


Figure 5.11: Simplified illustration of memory alignment for a two-dimensional array and the Gauss-Seidel stencil. a) Two-dimensional grid for storing values of the current and the previous iterations. b) Actual memory layout. Different colors denote memory areas that are used for processing one block.

The grid is implemented as an one-dimensional array of type `IOUArray Int Double`. The core function of our implementation is the translation operation from an n-dimensional list-based position to an one-dimensional array index whereat the desired memory layout is ensured. It is defined by

```
type Position  = [Int] -- Absolute position of an element
                       -- (without the iteration index)
type Dimension = [Int] -- Dimension of the grid

translate :: Dimension -> Position -> Int
translate dim pos = f' dim pos
  where f' [_]    [x]    = x
        f' (w:ws) (p:ps) = p + w * f' ws ps
```

All but the first list values of the position are used to map the n-dimensional block position to the array. As mentioned in Section 5.3.2, the first value of a position determines the referenced iteration. This value is handled separately such that blocks of successive iterations are aligned continuously, hence improving data locality. Since this additional alignment is based on the same principle as the previously described one, its implementation is not shown here.

Since the list-based operations for determining a particular array index are computationally expensive, we implemented specializations for grids which are often used in stencil computations: For example, many stencil-based algorithms work on a two-dimensional grid. In this case, a `translate`-functions which works solely by means of arithmetic functions and without list operations is used to compute the array index. This approach improves access time to a *large* extent. Although this optimization is also possible with a non-declarative description, a declarative one greatly eases its implementation.

Finally we cite reasons for our design decisions:

- It was not appropriate to use the predefined array type `IOUArray`. Although `IOUArray` supports any number of dimensions, the actual number has to be known at compile time. While it would be possible to extract this information from the stencil at compile time, e.g. by using Template Haskell[178], we would still have no control over the actual memory alignment.

- We chose to support only primitive types (e.g. `Doubles`) as array elements. Although all stencil-based computations we are aware of are numerical, the implementation can easily be further developed to allow arbitrary types. Note that such a more generalized approach might cause a performance degradation, since unboxed arrays (`IOUArray`) have to be replaced by the more general but less efficient boxed arrays (`IOArray`) [80].

- To further reduce the effects of false sharing a common approach is padding. By *padding*, dummy elements are added at the boundaries of blocks. This approach prevents reloading of the cache line if adjacent threads modify their data. In stencil-based computations, values of adjacent blocks are needed anyway. Therefore we think that padding will not improve cache efficiency significantly but it will complicate the implementation.

5.3.5 Dependency Computation

In a stencil-based algorithm computation order of elements has to respect dependencies (see Figure 5.12). In our previous implementation we used a database in which the correct

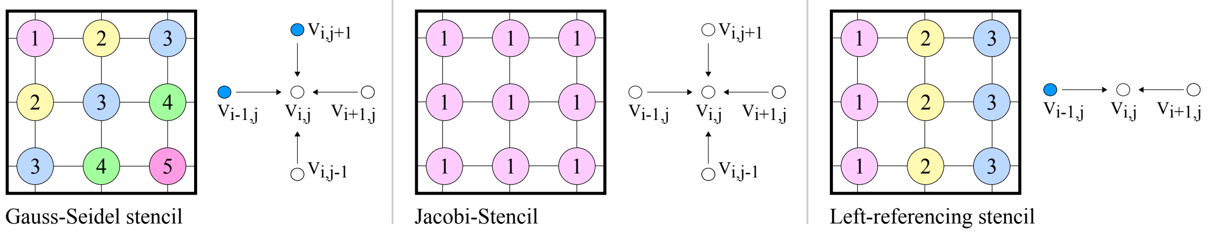


Figure 5.12: A possible correct order of element updates for different stencils. A computation order is indicated by numbers. Equal numbers denote elements which can be computed in parallel.

order of evaluation was defined for commonly used stencils. Although this approach allowed to hide stencil-dependent data from the user, it only supported stencils that were already in the database. It was quite easy to add further stencils, nevertheless it weakened the promise of a declarative library. For the current prototype, we implemented the automatic deduction of block dependencies as well as the deduction of a correct order of element updates from a given stencil. In the following we first describe the automatic dependency computation for the block order and then briefly explain the approach for their respective elements. As you will see, these two approaches are quite similar.

Dependency Computation for Blocks

The dependency computation for blocks is used in two cases. First, initial blocks have to be computed. These blocks are processed at the beginning of an iteration (see Figure 5.8 on page 118, ③). Second, after a block has been processed, dependent blocks have to be determined to check if these can be processed in turn (see Figure 5.10 on page 119, ④). In both cases, the same approach is used.

The block dependency computation has to take into account only dependencies of the current iteration, i.e. dependencies whose first coordinate is 0. Blocks, respectively their elements, from previous iterations can always be referred to and thus do not induce any dependency. The subset of a stencil S with this property also defines a stencil: we call this subset *current* S. For example, the *current* Gauss-Seidel stencil is

```
stencil =        -- current Gauss-Seidel stencil
  [ [-1,  0]  :: Dependency
   ,[ 0, -1]] :: Stencil
```

By comparison, the current Jacobi stencil is `[] :: Stencil`, since none of the dependencies refer to the current iteration.

Initial Block Computation. To determine initial blocks, the following procedure is applied to every block of the grid (see Figure 5.13): First, the coordinates of the boundary elements of each block are computed. For each of these coordinates, the stencil is applied. A block is an initial block, if the stencil's application results in a coordinate which is outside the grid's dimension. Element values at these coordinates have to be predefined by the problem definition.

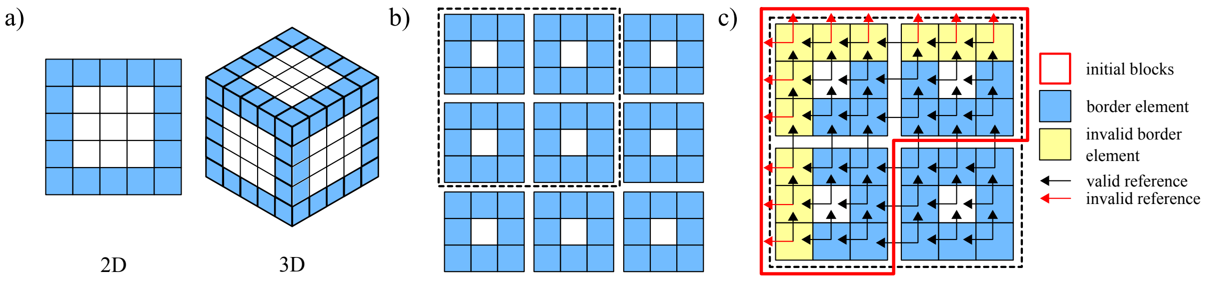


Figure 5.13: Illustration of initial block computation. a) Boundary elements for a two-dimensional block and isometric view of some boundary elements for a three-dimensional block. b) All boundary elements in a 9×9 grid with blocks of size 3×3 c) Magnification of the dashed area. For each boundary element, dependencies are shown for a current Gauss-Seidel stencil. The blocks surrounded by the red line are initial blocks.

Continuing our running example, we show that block A from Figure 5.7 on page 117 is an initial block with respect to a current Gauss-Seidel stencil. However, this is not the case for block B from the same figure. The coordinates of the boundary elements for each of the two blocks are

```
eleA = [(0,0), (0,1), (1,0), (1,1)]
eleB = [(2,2), (2,3), (3,2), (3,3)]
```

If the stencil is applied to each dependency separately, the referred coordinates of each boundary element are

```
appA = [(-1, 0), (-1,1), (0, 0), (0,1)    -- for [-1,0]
       ,( 0,-1), ( 0,0), (1,-1), (1,0)]   -- for [0,-1]
appB = [(1,2), (1,3), (2,2), (2,3)        -- for [-1,0]
       ,(2,1), (2,2), (3,1), (3,2)]       -- for [0,-1]
```

Some coordinates in `appA` are negative and thus outside the grid's dimension. Thus block `A` is an initial block. In contrast, the coordinates in `appB` are all inside the grid's dimension. Hence dependencies of `B`'s boundary elements can be fulfilled and `B` is not an initial block.

Dependent Block Computation. Dependent blocks of a block (called *source* block) are computed in two steps: stencil inversion and block computation. In the first step, the current stencil is inverted. For that purpose, all elements of each dependency are multiplied by -1 which inverses their respective direction. For example, the inverted current Gauss-Seidel stencil is

```
-- inverted current
-- Gauss-Seidel stencil
stencil =
   [[ 1,  0]  :: Dependency  -- (1)
   ,[ 0,  1]] :: Stencil     -- (2)
```

In the second step, the inverted stencil is applied to the source block to compute the coordinates of dependent blocks. For each dependency of the inverted stencil, the source block coordinates are translated into the respective direction (see Figure 5.14). This approach results in the dependent blocks. Invalid blocks, i.e. those with coordinates outside the grid's dimension, are filtered.

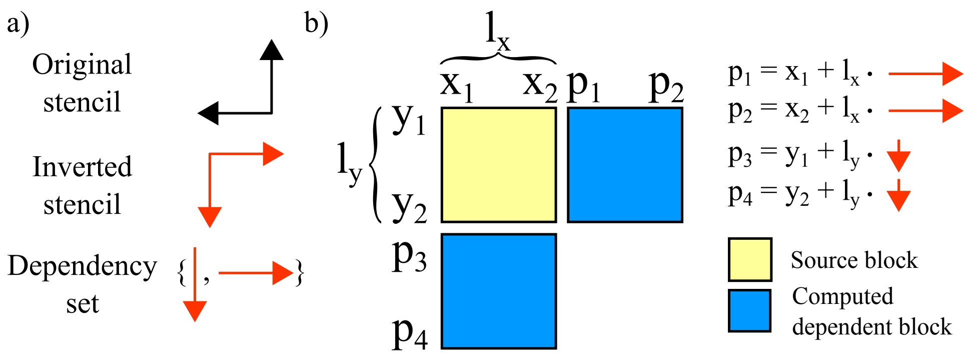


Figure 5.14: Exemplary computation of block dependencies. a) A current Gauss-Seidel stencil, its inversion and the subset of the stencil dependencies. b) Illustration of the computation to determine the respective dependent blocks.

For example, for a source block `A = [(0,1), (0,2)]` and a block size of 2×3 (for the purpose of a clearer illustration we deviate from the usual 2×2 blocksize), dependent blocks are

```
-- Using (1), i.e. [->, V] = [1, 0]
--   x-coordinates:
--       l_x  = 2
--       ->   = 1
--       p_x1 = 0 + l_x * -> = 2
--       p_x2 = 1 + l_x * -> = 3
--    y-coordinates:
--        l_y   = 3
--        V     = 0
--        p_y1  = 0 + l_y  * V = 0
--        p_y2  = 2 + l_y  * V = 2
B1 = [(2,3), (0,2)]

-- Using (2), analogously
B2 = [(0,1), (3,5)]
```

Element Order Computation

Figure 5.12 on page 122 showed examples for a correct evaluation order of of block elements. Since this order depends on the stencil it has to be computed at runtime. For each block the following two steps are performed, which are similar to the dependent block computation (see Figure 5.10 on page 119).

First, the *valid boundary* of a block is computed. This is a set of positions (called *valid positions*), which are closest to the block's boundary but do not have dependencies outside the grid. By using the valid positions, the initial positions are determined. An *initial position* is a valid position which does not have a dependency to other valid positions (see Figure 5.15). In the second step, initial positions are stored in a queue and processed successively: For each position, its dependent positions are determined by applying the inverted stencil. For each such position it is checked if all its dependencies are fulfilled. If that is the case, the particular position is added to the queue. The computation is finished, if the queue is empty, hence all reachable positions have been computed and are stored. The correct order of the positions is stored in a `BlockArray` and is accessed when a block is updated in phase II.

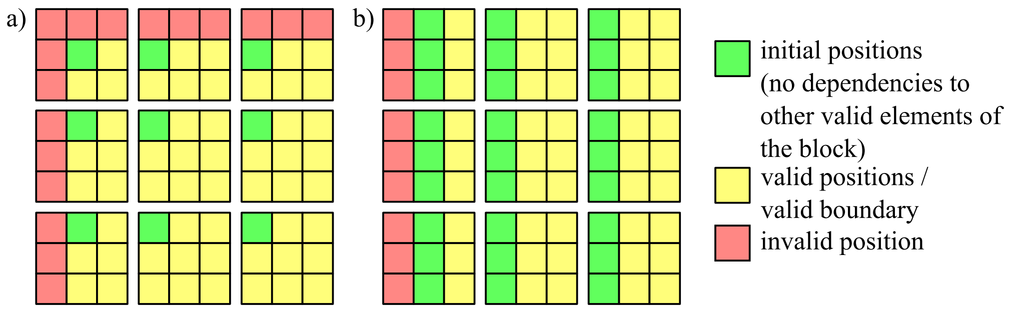

a)
b)
initial positions (no dependencies to other valid elements of the block)
valid positions / valid boundary
invalid position

Figure 5.15: Illustration of valid, invalid and initial positions for a 9×9 grid and a block size of 3×3. a) For the current Gauss-Seidel stencil b) For the current left-referencing stencil (see Figure 5.12 on page 122).

5.4 Benchmarks

We performed our experiments on a 2.3 GHz 16-core AMD Opteron 6134 with 32 GB RAM running a Linux-kernel 2.6.38-8 with GHC 7.0.3. We ran each benchmark three times from one to sixteen cores. The mean value was used for speedup calculation. In contrast to other chapters we did not compare the parallel version to a sequential version but to our implementation running on a single core: Any sequential version that implements only one stencil-based algorithm will be more efficient than our implementation since it is less general. Therefore such a comparison would favor the sequential version. We chose the grid sizes so that the absolute runtime on sixteen cores was about 30 seconds.

We tested our implementation with the Jacobi stencil as well as the Gauss-Seidel stencil. Since the results were similar, we do not present results of benchmark runs of similarly structured stencils, e.g. for image processing [118].

5.4.1 Jacobi Stencil

For benchmarking a Jacobi-Stencil we chose a two-dimensional grid with dimensions 400×400 and an ϵ-based convergence criterion of $\epsilon = 0.1$. Until the convergence criterion was reached, 243 iterations had to be calculated. The Jacobi stencil allows to examine the scalability of a stencil which has no dependencies on elements of the current iteration. Hence, all blocks can be computed in parallel. The benchmark result is shown in Figure 5.16.

Compared to our published paper, speedup results were similar and we could also measure speedup beyond eight cores. Similar to other benchmark results in this thesis, we achieved a typical speedup curve and reached a maximum speedup of nearly 10 when using 16 cores.

5.4.2 Gauss-Seidel Stencil

Compared to the Jacobi stencil, the Gauss-Seidel stencil limits the available maximal parallelism. This stencil induces dependencies for its left and upper elements of the current iteration. This causes dependencies between blocks which limits the possible scalability. Besides using the Gauss-Seidel stencil, all other parameters of the previous benchmark were taken. The speedup graph is presented in Figure 5.17 on page 128.

For up to 8 cores the speedup is similar to the previous benchmark. However, as expected the block dependencies reduce the maximal available parallelism and thus the speedup.

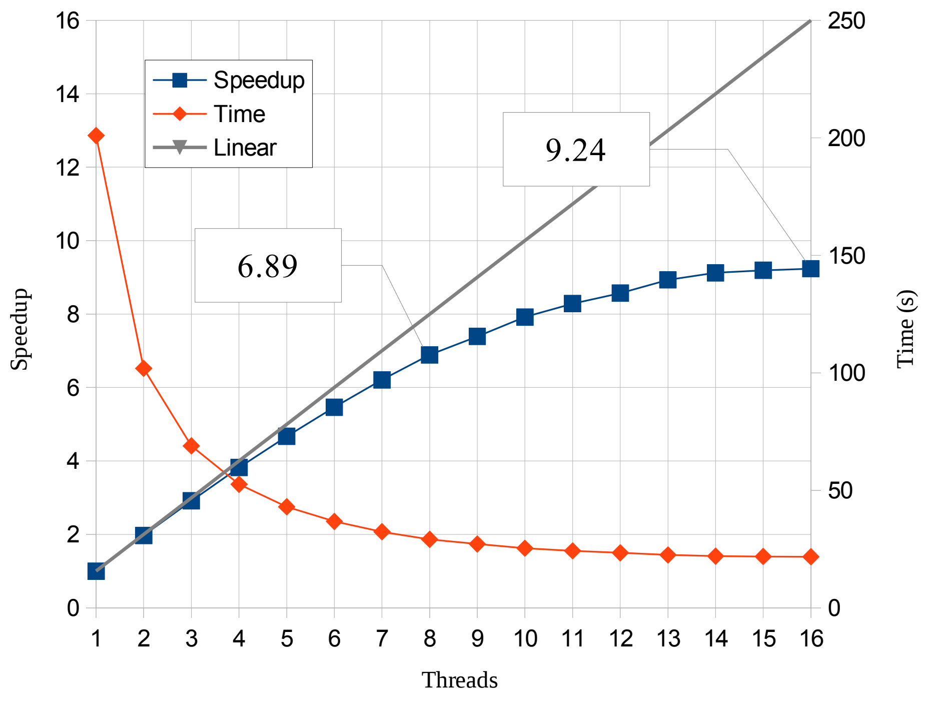


Figure 5.16: Speedup graph and absolute computation times for voltage diffusion simulation with a Jacobi-stencil on a 400 × 400 grid and $\epsilon = 0.1$
.

5.4.3 Summary

In this section we illustrated the results of speedup benchmarks for two commonly used stencils. The results show a reasonable scalability of this particular execution platform.

Note that the absolute performance of the execution platform is not optimal yet. It is well known that list-based computations in Haskell have a huge overhead compared to fixed-size data types. Compile time optimizations, as for example applied in [118], might enable a better absolute performance.

5.5 Related Work

Stencil-based calculations are one of the key patterns in scientific computing [10]. For this reason, much research has been done to describe stencils and stencil-based algorithms as well as to improve their performance by using parallelization. Traditionally, stencil algorithms are written in C/C++[207] and parallelized using OpenMP for shared- and MPI for distributed-memory architectures. For example, the C++-framework Janus by Gerlach et al. [61] provides a template-based system to express (among other mesh-like structures) stencil-based calculations and to calculate them in parallel using both MPI and OpenMP.

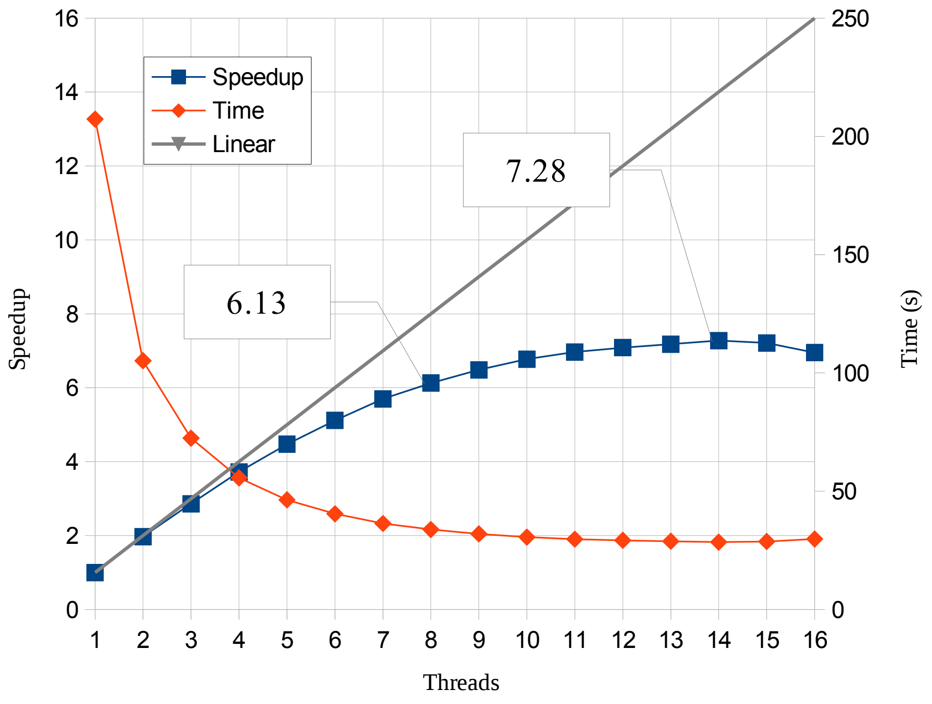


Figure 5.17: Speedup graph and absolute computation times for voltage diffusion simulation with a Gauss-Seidel stencil on a 400 × 400 grid and $\epsilon = 0.1$
.

Outside Haskell, the general idea of providing a high-level framework for parallel stencil calculations is an active research topic: Dursun et al. describe a parallel framework that uses different levels of parallelization which depend on the given stencil to provide an efficient parallel calculation [46]. Kaushik et al. demonstrate how an optimization strategy for a stencil can be determined automatically [40]. Christen et al. go even further by describing a domain specific language and a compiler which generate optimized code to compute stencils using OpenMP and CUDA [30]. These ideas are quite promising and interesting, especially with a purely declarative stencil description which can guide the optimizations as well as the compilation.

Another approach to parallel stencil calculation are by using novel high-level programming languages, like Chapel[26], Fortress[3], or SAC [67], which provide automatic parallelization of array-based operations. For stencils without dependencies to the current iteration this is a viable approach. For example, Barret et al. describe how to implement stencil algorithms in Chapel using seemingly sequential code [12]. In such high-level languages stencil calculations are easy to express. Thus they are an interesting alternative to our description, although they limit the possibilities of stencil-dependent optimizations.

Since our first paper has been published in 2010, the interest for stencil-based computations in the Haskell community has risen. Orchard describes a graphical approach to define one- and two-dimensional stencils called Ypnos [155]. It allows to describe stencils and their dependencies visually in source code. It does not provide a parallel execution platform. Although our library allows a convenient description of multi-dimensional stencils it is still in a prototypical state. For high-performance computations with arrays, there is the parallel array library REPA [118]. In particular, Lippmeier et al. describe a similar declarative approach for stencil algorithms by means of REPA [128]. While the approach is not as flexible as our library regarding the handling of convergence conditions, benchmarks using stencils for image processing show a performance comparable to the industry standard image library OpenCV.

5.6 Summary and Conclusion

In this chapter we have shown how to define general stencil-based algorithms in Haskell and how to execute them in parallel by means of Concurrent Haskell. Our contributions are twofold. First, we introduced types and a list-based notation which allow a flexible and yet concise and understandable formulation of n-dimensional stencil problems. Our notation should feel natural to most Haskell developers, since lists are one of Haskell's main data structures. In particular, our approach allows for the fully declarative formulation of stencil problems: besides stating the problem, i.e. dependencies, initial grid values, and grid dimensions, no further information is necessary. Second, we described the implementation of a prototypical platform that allows the parallel computation of the declaratively described stencil problem. To briefly state our results, for stencils that do not refer to elements of the current iteration (e.g. Jacobi stencil), speedup is about 9.2 for 16 threads. For stencils with such dependencies (e.g. Gauss-Seidel), speedup is about 7.2 for 16 threads. To summarize, we have shown that stencil computations can be fully described declaratively in Haskell in an *idiomatic* way. Although our execution platform is just a prototype it shows that a scalable implementation for parallel stencil computations by means of Concurrent Haskell is possible.

As far as abstraction is concerned, we have shown that complex details of parallel programming can be hidden from the user by providing a declarative interface. In general, a declarative approach for a well-defined problem domain does not only free the user from handling unnecessary details but also allows for more freedom regarding problem-specific optimizations.

Chapter 6
Describing and Executing Graph Algorithms

Graph algorithms have fundamental applications in the real world but can be both cumbersome to implement in traditional languages and difficult to execute efficiently on modern multicore hardware. The Bulk Synchronous Parallel model of computation allows for the definition of vertex-centric computations on graphs. In this chapter we describe an embedded domain specific language for specifying such algorithms, and show an implementation of an execution platform that allows to execute them on multicore systems in parallel. We provide several examples which demonstrate that the EDSL allows for a concise yet understandable formulation of graph algorithms.

6.1 Introduction

Graph algorithms such as finding shortest paths, clustering or matching have fundamental applications in the real world. It can be quite challenging for non-experts to write and optimize them, in particular on multicore hardware [66].

A new approach to implement graph algorithms is based on Valiant's Bulk Synchronous Parallel (BSP) model of computation [200, 201]. BSP models a concurrent computation as a set of independent processing nodes with local state that communicate solely by explicit message passing. This model has recently been adapted to support a new *vertex-centric* approach for graph algorithms called *Pregel* [133]. In Pregel, an algorithm performs *local* (sub)computations for each vertex: these computations do not have direct access to the whole graph structure but only to the local vertex state, a list of the vertex' neighbors and received messages. Pregel uses C++ as the underlying description language and defines a class `Vertex` which methods are overwritten to implement the actual vertex behavior. In our opinion, the usage of C++ has drawbacks for both users and implementators of the system. For users, concepts like iterators and inheritance are rather unintuitive in the context of graph algorithms. For implementators, problem-specific optimizations are difficult, since the vertex-centric C++-code can involve arbitrary functions. A solution to both problems is to restrict the description of graph algorithms to a small sublanguage.

Domain specific languages (DSLs) are a well-known software-engineering concept to describe solutions or problems in a restricted domain [56]. Successful examples of DSLs are Verilog for hardware design [196] and SQL for databases [34]. For DSL users, the reduced vocabulary is more intuitive and makes development less error-prone. For implementators, DSLs offer potential for efficient implementation, since boundary conditions and particular features of the problems are known in advance and can be used for optimizations such as parallelization on shared-memory multicore machines. Although DSLs have both theoretical and practical advantages, e.g. are easier to prove for correctness and are self-documenting, they are seldom developed from scratch. The implementation of a compiler as well as the whole ecosystem that a modern language offers (debuggers, profilers and libraries) is rarely justified. Instead, modern DSLs are typically *embedded* into a host language, which provides the underlying infrastructure. These DSLs are called Embedded Domain Specific Language (EDSL). Haskell is well-suited as a host language due to its high degree of abstraction [98] such as support for higher-order functions and monads. Successful Haskell EDSLs have been developed for hardware design [4], programming GPUs [48], describing graphics [49] and financial contracts [115].

In this chapter we combine the advantages of an EDSL and the ideas of the bulk synchronous parallel model and Pregel (see Figure 6.1). We call our combination of the language and its execution platform *Palovca* for PArallel LOcal Vertex Calculations in hAskell. Our contributions are

- The definition of an embedded domain specific language to describe vertex-centric graph algorithms and an evaluation of Haskell's suitability as a host language for this purpose.
- The implementation of an execution platform that runs on shared-memory multicore systems. It is based on explicit concurrency, i.e. manual synchronization through locking and explicit thread control.
- An experimental investigation of the platform with various benchmarks.

To summarize our results, the expressiveness of the EDSL is high and allows a concise and comprehensible formulation of vertex-centric algorithms. Our experiments have shown that the implementation scales very well. Depending on the particular graph, its edge distribution and the computational load per vertex, we achieve speedups between 9 and 11 with 16 threads.

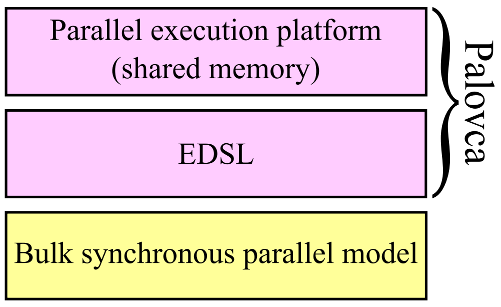

Parallel execution platform (shared memory)
EDSL
Bulk synchronous parallel model
Palovca

Figure 6.1: Overview of Palovca.

The rest of this chapter is structured as follows. Section 6.2 describes Palovca's underlying computational model. Section 6.3 shows its syntax and examples of vertex-centric algorithms and their implementation in Palovca. Section 6.4 describes the implementation of the language and execution platform in detail. Section 6.5 explains our benchmarks and discusses the experimental results. In Section 6.6 we review related work and Section 6.7 contains a summary and a conclusion.

6.2 Palovca's Computational Model

In this section we briefly introduce the BSP model and show how it can be applied to directed graphs. To keep the presentation simple we only give an informal description and refer to [133, 200, 201] for formal definitions.

6.2.1 The Bulk Synchronous Parallel Model of Computation

The BSP model that Palovca is based on independent processing nodes and message passing as its core concepts [200, 201]. A BSP computation is divided into a sequence of discrete steps called *supersteps* (see Figure 6.2). In each superstep nodes work independently and are allowed to

① perform computations on their local data

② receive messages sent in the previous superstep

③ send messages to other nodes

These three operations can be arbitrarily interleaved. Each superstep finishes with a barrier that guarantees that all operations of the previous superstep have finished. In particular, all messages sent in superstep n are delivered and may be retrieved in superstep $n + 1$.

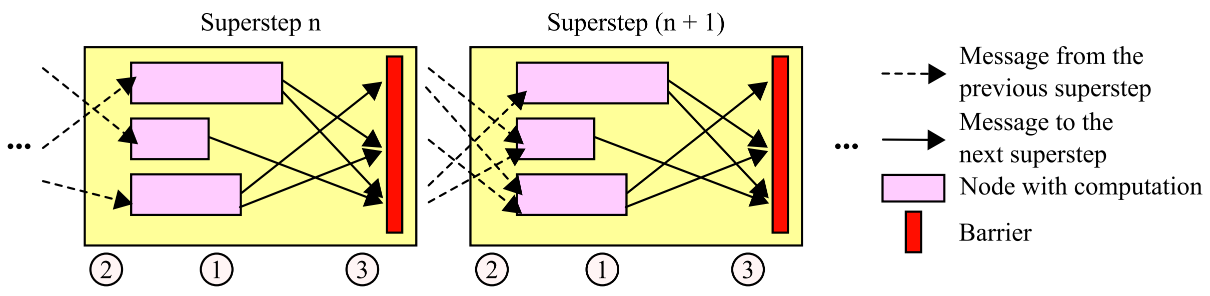


Figure 6.2: Visualization of superstep-based computation in the BSP model.

An advantage of BSP is its conceptual simplicity. By restricting communication to message passing, synchronization problems such as race conditions and deadlocks do not occur. By allowing solely computations on local data, modularity is enforced. In particular, developing algorithms as well as reasoning about their properties becomes easier.

6.2.2 Graphs and BSP

The central idea behind Pregel and Palovca is the application of BSP to graph problems by mapping vertices and edges to components in the BSP model. We call graph algorithms in this model *vertex-centric algorithms*. The mapping is as follows: Vertices correspond to nodes and (directed) graph edges define the source and destination for messages. A message sent from a vertex is delivered to *all* vertices that are connected to the source vertex by an edge (see Figure 6.3).

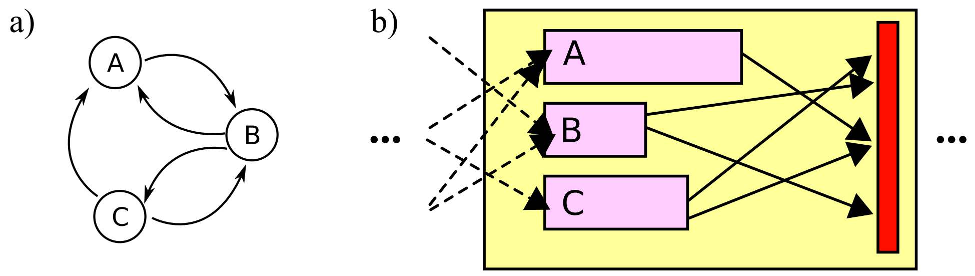


Figure 6.3: Example: a) source graph with 3 connected vertices b) BSP-based model of the source graph showing a single superstep with computations and communications.

To handle termination, each vertex is in one of two activity states, *active* or *inactive*. Initially all vertices are active and run the vertex computation. In each superstep, a vertex can switch to the inactive state and is only reactivated by receiving new messages in the following superstep. When all vertices are inactive, the computation is finished.

With this approach, graph algorithms can be expressed quite elegantly, in particular with the Palovca language, as will be shown in the following section. Moreover, we show in Section 6.5 that it also allows for an efficient parallelization.

6.3 An EDSL for Vertex-Centric Graph Algorithms

In the following we describe the syntax and core functions of the EDSL. Several examples of well-known graph algorithms formulated with the vertex-centric paradigm illustrate that the syntax is concise but easy to understand. We focus on directed graphs whose vertices as well as edges can be tagged with values.

6.3.1 The Palovca language

A major design requirement for the Palovca language was to allow a concise yet understandable formulation of vertex-centric algorithms. Therefore many functions typical for graph algorithms and BSP are part of the language, e.g. `size` to get the number of vertices in the graph, or `step` to get the number of the current superstep, with supersteps being numbered $0, 1, \ldots$

The language is internally based on a type that describes graphs (`Graph`), and a custom monad (`GraphM`) that handles the global state and the local vertices' states. Both have type parameters `v`, `e` and `m` for the local vertex value, the edge values and the value of the messages, respectively. All types need to be instances of `NFData` (see Section 6.4) but we omit that in nearly all type signatures for conciseness. An initial graph is typically created by calling one of the file-based input functions mentioned below. Palovca only supports directed graphs, although undirected graphs can easily be created by inserting symmetrical edges. A calculation with a graph is expressed with the function

```
run ::  Graph v e m -> GraphM v e m () -> IO ().
```

It receives a graph and a function working in the `GraphM` monad as parameters. The function is executed for each active vertex (in accordance with the computational model mentioned in the last section) until all vertices are inactive.

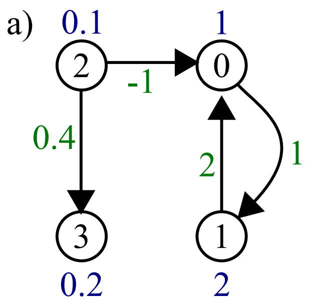
Figure 6.4 gives an overview of core functions. The local vertex state is accessible and modifiable by the `get` and `set` functions, respectively. A graph-wide unique identifier for each vertex is obtained by `identifier`. A vertex switches to inactive by calling `halt`. The number of connected vertices (neighbors) is obtained by `neighbors`. Messages sent in the previous superstep are obtained by calling `messages`, messages for the next

```
-- Graph query
size            :: GraphM v e m Int
step            :: GraphM v e m Int

-- Vertex modifications
modifyVertices  :: Graph v e m -> GraphM v e m () -> IO ()

-- Vertex local state
identifier      :: GraphM v e m Index     -- type Index = Int
get             :: GraphM v e m v
set             :: v -> GraphM v e m ()
halt            :: GraphM v e m ()

-- Message passing
messages        :: GraphM v e m [m]
send            :: m -> GraphM v e m
sendEdges       :: (e -> m) -> GraphM v e m ()
neighbors       :: GraphM v e m Int
```

Figure 6.4: Overview of core functions in Palovca.

superstep are sent by `send` and `sendEdges`. While `send` sends the same message to all neighbors, `sendEdges` allows to modify the message with respect to the edge value of each particular neighbor.

For file-based input we defined a data format that specifies the vertex value, its neighbors and their possible edge values (see Figure 6.5). Parsers for frequent types are predefined in Palovca. Moreover, adding own parsers to support arbitrarily complex graphs and vertex and edge values is easily possible.



b)

```
0 1.0 1,1
1 2.0 0,2
2 0.1 0,-1 3,0.4
3 0.2
```

Figure 6.5: Example: a) source graph. Blue numbers denote vertex values, and green numbers edge values. b) definition of this graph in a Palovca-supported format.

Some algorithms use a global convergence criterion to check for termination. In addition, collecting statistical data between supersteps might be useful. Therefore, Palovca supports *aggregators* (see Figure 6.6 and Figure 6.7 on page 138): In each superstep a vertex can send a value to an aggregator with `aggregate` (①). To reduce the number of type parameters, the aggregated value must *currently* have the same type as the vertex value. All aggregated values of a superstep are combined to a single value using a previously defined aggregator function (②). The new value is available in the next superstep and accessible by calling `aggregator` (③).

Other algorithms, e.g. clustering or matching, need to change the graph topology. Our current version of Palovca supports adding and removing edges, and adding new ver-

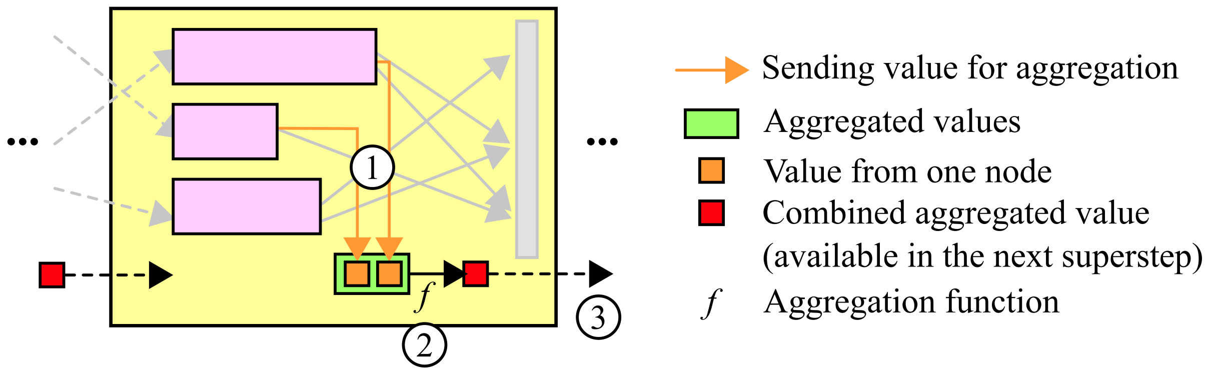

...
1
...
f
2
3
Sending value for aggregation
Aggregated values
Value from one node
Combined aggregated value (available in the next superstep)
f Aggregation function

Figure 6.6: Visualization of the aggregation function.

```
setAggregator :: Graph v e m -> ([v] -> v) -> IO ()
aggregate     :: v -> GraphM v e m ()
aggregator    :: GraphM v e m v
```

Figure 6.7: Aggregator functions for global communication between supersteps.

tices. Moreover, Palovca contains various functions that ease the formulation of vertex-centric graph algorithms such as selective sends and random neighbor selection.

In the following we describe several examples of vertex-centric algorithms and show their concise and understandable implementation in our EDSL. The examples serve as an introduction to the Palovca EDSL as well as a general introduction to vertex-centric algorithms.

6.3.2 Example: Pagerank

The pagerank algorithm [18] is widely known for being the foundation of the Google search engine: the pagerank of a webpage is a numerical value that is the higher the more pages point to it. In our modeling of this scheme vertices stand for pages, and edges denote links. An example is shown in Figure 6.8. The formulation of this algorithm in Palovca is shown in Figure 6.9 and the C++-based Pregel version is shown in Figure 6.10.

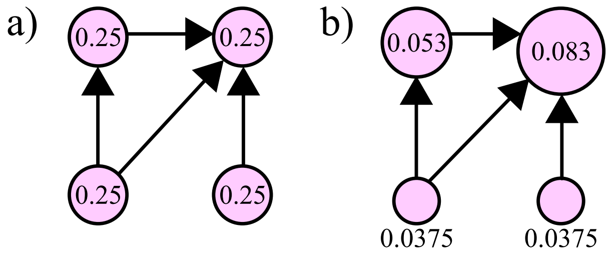


Figure 6.8: Visualization of the a) initial state and b) final state of the pagerank computation. Larger radiuses denote larger pagerank values for the example graph.

Both vertices and messages are of type `Double` and edges are of type `None` (a type synonym for `()`), i.e. do not store any value. Each vertex state is initialized with $\frac{1}{N}$ where N denotes the size of the graph. In each superstep the new pagerank for each vertex is computed and distributed: First, a vertex collects all weights from its incoming neighbors with the `messages` function. Second, it accesses the graph size with `size` and updates its new pagerank with the shown formula using `set`. Third, the vertex distributes its updated value to its outgoing neighbors. The calculation stops after a given number of calculation steps. A nice property of having a lazy host language is

```haskell
pagerank :: GraphM Double None Double ()
pagerank = do
  msgs <- messages
  s    <- size
  let value = 0.15 / (toEnum s) + 0.85 * (sum msgs)
  set value

  s <- step
  if s < 30
      then do
          n <- neighbors
          send (value / toEnum n) -- (*)
      else halt
```

Figure 6.9: The pagerank algorithm in the Palovca EDSL.

```cpp
class PageRankVertex : public Vertex<double, void, double> {
public:
  virtual void Compute(MessageIterator* msgs) {
    if (superstep() >= 1) {
      double sum = 0;
      for (; !msgs->Done(); msgs->Next())
        sum += msgs->Value();
      *MutableValue() = 0.15 / NumVertices() + 0.85 * sum;
    }

    if (superstep() < 30) {
      const int64 n = GetOutEdgeIterator().size();
      SendMessageToAllNeighbors(GetValue() / n);
    } else {
      VoteToHalt();
    }
  }
```

Figure 6.10: The pagerank algorithm in C++ using the Pregel API [133].

shown at `(*)`: we do not need to check that the number of neighbors (`n`) is non-zero. If it is zero, no messages are sent and the division is never evaluated.

6.3.3 Example: Single source shortest path

Finding the shortest path between vertices in a graph is one of the most important real-world applications of graph theory [36]. In the single-source shortest-path (SSSP) problem we want to find the shortest path between a single source and every other vertex. An example is shown in Figure 6.11.

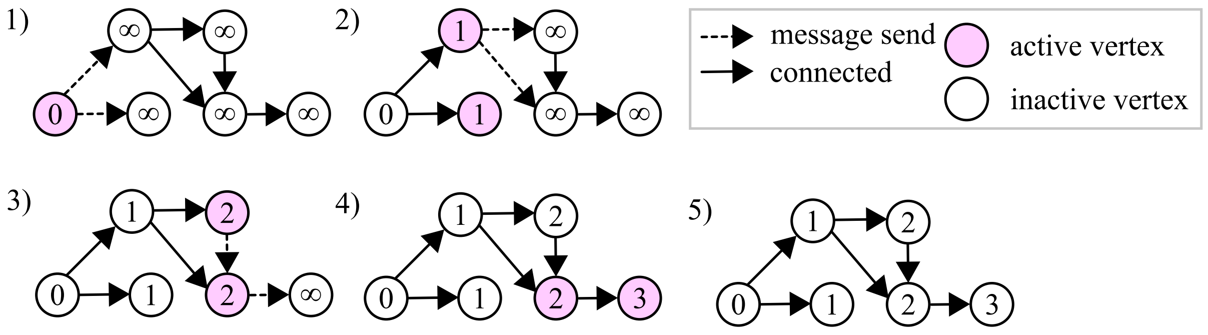


Figure 6.11: Visualization of the single source shortest path algorithm. All edges have the value 1. The computation stops after five steps since all vertices are inactive.

The vertex-centric formulation of this algorithm is shown in Figure 6.12 and works as follows. Vertex value, edges and messages are of type `Double`. The value defines the distance from the source and is initially set to $1e30$ (denoting infinity). In an initialization phase that precedes the execution of the `run`-function, all vertices except the source vertex (here the vertex with identifier 0) are set inactive using `modifyVertices`. In each superstep all active vertices receive messages from their incoming neighbors that denote the minimal distances of these neighbors to the source vertex. The minimal value of the distances is compared to the currently stored one: if it is smaller, the stored one is updated and sent to each outgoing neighbor, which reactivates them. When all vertices are inactive, the computation is finished.

6.3.4 Example: Semi-Clustering

Our third example is a semi-clustering algorithm [133]. It greedily divides an undirected graph with weighted edges into a set of C_{max} sub-graphs (clusters) with a maximum of V_{max} vertices in each. In contrast to traditional clustering problems, vertices can belong to more than one cluster. Semi-clusters in this example are chosen such that their score

$$S_c = \frac{I_c - f_B B_c}{\frac{V_c(Vc-1)}{2}}$$

```
sssp :: GraphM Double Double Double ()
sssp = do
  i <- identifier
  let dist = if i == 0 then 0 else 1e30 -- 1e30 as infinity

  minVal <- (minimum . (dist:)) `liftM` messages
  cur    <- get
  when (minVal < cur) $ do
    set minVal
    sendEdges (minVal+)
  halt

main :: IO ()
main = do
  -- Initialization phase
  g <- <graph reading>
  modifyVertices g $
      i <- identifier
      when (i > 0) halt
  run g sssp
```

Figure 6.12: The SSSP algorithm in the Palovca EDSL.

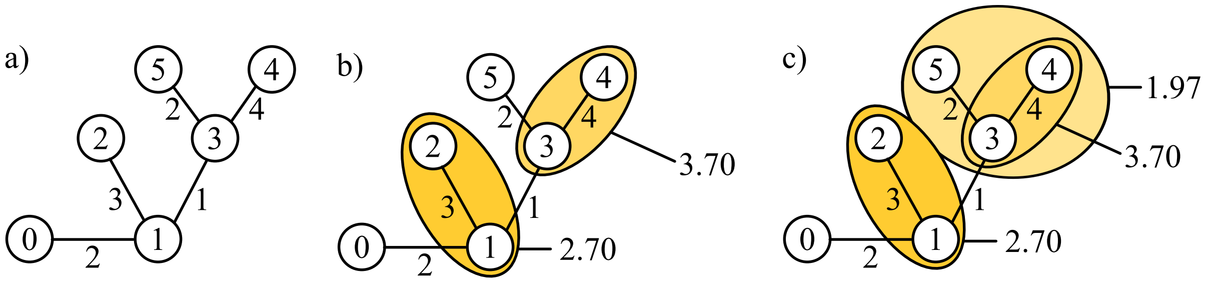


Figure 6.13: Examples of different semi-clusters for $V_{max} = 3$ and $f_B = 0.1$. The semi-clusters are colored and their respective value is labeled. a) initial graph b) semi-clusters for $C_{max} = 2$ c) semi-clusters for $C_{max} = 3$.

is maximized. Here, I_c denotes the sum of all internal edges, B_c the sum of boundary edges, V_c the number of vertices in the cluster and f_B a user-specified score factor. Figure 6.13 shows examples of different semi-clusters.

The general idea of the algorithm is that each vertex contains a list of local best semi-clusters which is distributed to all neighbors in each superstep. At the beginning of each superstep, each vertex receives a list of the best semi-clusters of its neighbors and adds itself to these if possible. It computes the best scoring C_{max} local semi-clusters and redistributes them to all its neighbors.

Each vertex' local value (type `Value`) stores a list of semi-clusters with their respective score whereby a semi-cluster is a list of the unique identifiers of the contained vertices:

```haskell
type Value       = [SemiCluster]
type SemiCluster = (Score, [Index])
type Score       = Double
type C           = GraphM Value Double Value
```

The type `Index` allows to refer to a particular vertex and the type `C` shortens type annotations. In an initialization phase (not shown here) the local list of each vertex V contains a single semi-cluster with V itself and a score of 1. This semi-cluster is sent to all neighbors of V. In the beginning of each superstep each vertex V receives its messages. Each message contains a list of semi-clusters. This list is partitioned into those clusters which already have V_{max} elements or have V already included (`rest`) and the others (`add`):

```haskell
calc :: Int -> Int -> Double -> C ()
calc cmax vmax fb = do
  sc <- messages
  i  <- identifier
  let notIncluded =
         \(_,c) -> not (i `elem` c) && (length c < vmax)
      (add, rest) = partition notIncluded (concat sc)
```

The score of those semi-clusters which could still include V (`add`) is updated by adding V; we do not show the implementation of `updateScore` since it contains mostly numerical computations:

```haskell
  edges <- neighbourList
  let add' = map (updateScore i edges) add
```

The function `neighbourList :: GraphM v e m [(Index, e)]` returns a list of each neighbor with its corresponding edge value. After adding the updated semi-clusters to the local state, it contains the C_{max} best scoring semi-clusters:

```haskell
  known <- get
  let containingMe = known ++ add'
      global = take cmax
             $ reverse
             $ sortBy (compare `on` fst)
             $ nub' (containingMe ++ rest ++ add)
```

```
    set containingMe
```

The function `nub'` is similar to the library function `Data.List.nub` [91], i.e. it removes duplicate elements from a list. `nub'` compares the list of semi-clusters to determine if two list elements are equal. Since rounding errors can occur, only the vertex indices and not the scores are compared:

```
nub' :: [SemiCluster] -> [SemiCluster]
nub' = nubBy (\a b -> snd a == snd b)
```

The computation is finished after a given number of supersteps and the semi-clusters of the first vertex can be collected by accessing the aggregator value. When the computation continues, this vertex' local best semi-clusters are sent to all neighbors to be processed in the next superstep:

```
    s <- superstep
    if (s < 10)
        then send global
        else do
            halt
            when (i == 0) $
              aggregate global
```

Other approaches for termination are also possible, e.g. when the list of the best semi-clusters stops changing. This can be checked by using the aggregator to compare the local best semi-clusters of all vertices. We omit this approach to simplify the presentation.

6.3.5 Example: Bipartite Matching

In this section we consider *bipartite* graphs, i.e. undirected graphs with two distinct sets L and R of vertices. Edges exist only between vertices of the sets. A *matching* is a subset of all edges such that no two edges share a common endpoint (see Figure 6.14 on page 144). In a *maximal matching* the subset of found edges is maximal, i.e. by adding another edge an endpoint would be shared [41]. A bipartite matching algorithm determines such a matching. In this section we present the Palovca version of a *randomized* maximal matching algorithm [7].

The algorithm is iterative and each iteration consists of phases 0,1,2 and 3 (see Figure 6.15 on page 144). In the first phase all non-matched vertices in L send a request to be matched to their connected vertices from set R and vote to halt. In the second phase, each vertex in R which is not yet matched randomly chooses one of the received

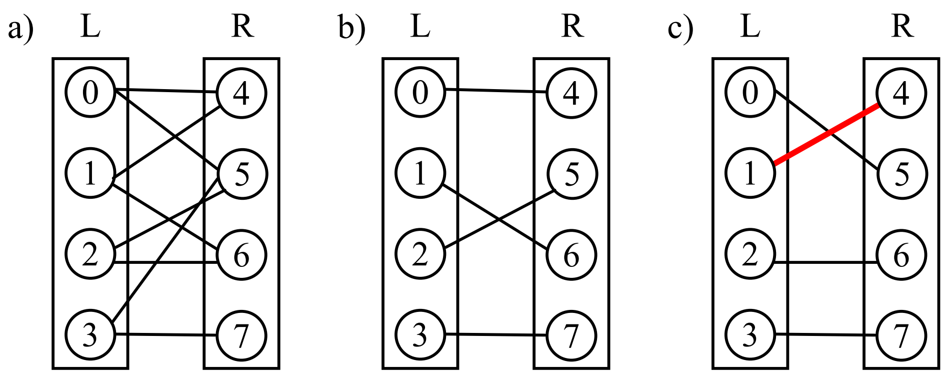


Figure 6.14: Examples of different bipartite matchings. a) initial graph b) a maximal matching c) another maximal matching. If the red edge would be omitted, the matching would be not maximal.

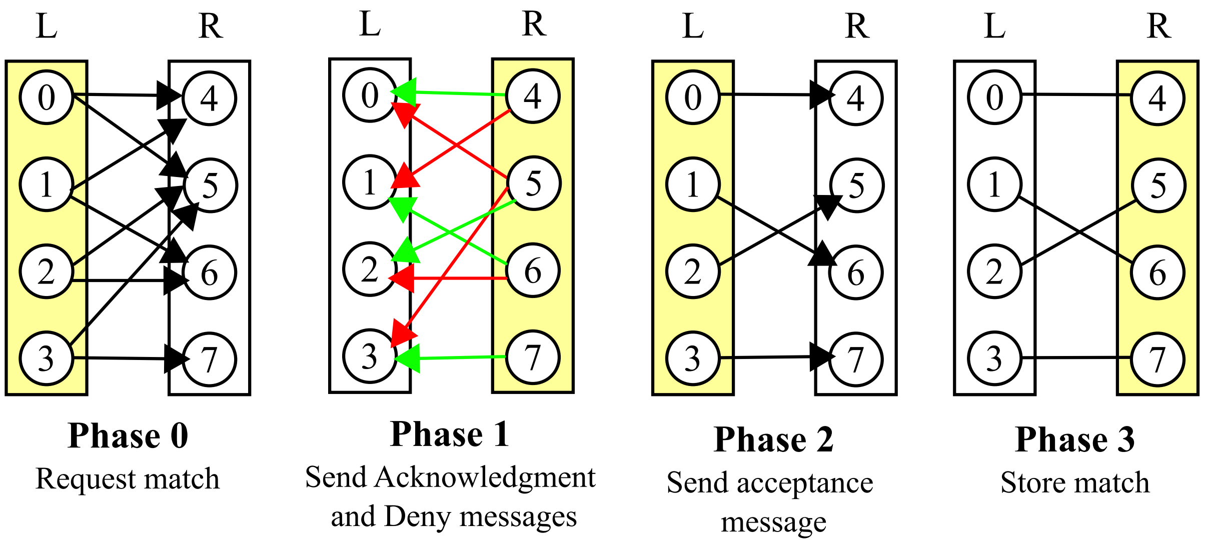


Figure 6.15: Visualization of the four phases in the bipartite matching algorithm. The active vertices of the graph are marked with a yellow box, the inactive vertices with a white box. Note that this example shows the ideal case in which no second iteration is needed.

requests. It sends an acknowledgment message to the vertex of the chosen request and sends denial messages to the sources of all other requests. In the third phase each unmatched vertex in L randomly chooses one of the received acknowledgment messages, sends one acceptance message, and votes to halt. In the fourth phase each unmatched vertex in R may receive an acceptance message. In this case, it stores its match and votes to halt. These four phases are repeated until all vertices of both sets have been matched or no further match is possible. In the following we show the Palovca implementation of this algorithm.

The local data of a vertex consists of the set it is in and a possible match:

```
data Value = Value {
    inSet      :: Set
  , matchState :: Status
}

data Set    = L | R
data Status =
    Matched Index
  | Unmatched
```

According to the description of the algorithm, four different types of messages are possible:

```
data Message =
    RequestMatch Index      -- send from L
  | Accept Index            -- send from L
  | Ok Index                -- send from R
  | Deny                    -- send from R
```

To distinguish between the different phases, we compute the superstep modulo 4, i.e. the computation is in one of the phases 0, 1, 2 or 3. With the computed phase a helper function `inPhase` is called:

```
calc :: GraphM Value None Message ()
calc = do
  phase <- (`mod` 4) `liftM` superstep
  inPhase phase
 where inPhase :: Int -> GraphM Value None Message ()
```

Phase 0. Vertices in L send a message `RequestMatch` with their identifier to all connected vertices (which are in R):

```
       inPhase 0 = do
         val <- get
         -- Only work on unmatched vertices from L.
         when (val == Value L Unmatched) $ do
           i <- identifier
           send (RequestMatch i)
           halt
```

Note that pattern matching in Haskell allows a concise and understandable formulation to chose only unmatched vertices in L.

Phase 1. Unmatched vertices in R randomly choose a vertex and send an acceptance message.

```
      inPhase 1 = do
        val <- get
        when (val == Value R Unmatched) $ do
          msgs <- messages
          ok   <- random (0, length msgs - 1)

          me <- identifier
          forM_ (zip [0..] msgs) $ \(i,m) -> do
            let RequestMatch d = m
            if i == ok
              then sendTo d (Ok me)
              else sendTo d Deny
          halt
```

The function `sendTo` allows to selectively send a message to a particular vertex and `random` returns a random number in the given interval.

Phase 2. Unmatched vertices in L receive messages sent by vertices in R in the previous phase. In case of an acceptance message, an acknowledgment message is sent and the local state is changed:

```
      inPhase 2 = do
        val <- get
        when (val == Value L Unmatched) $ do
          let accept m = case m of
                                Ok _ -> True
                                _    -> False
          msgs <- filter accept `liftM` messages
          unless (null msgs) $ do
            ok <- random (0, length msgs - 1)
            let Ok a = msgs !! ok
            me <- identifier
```

```
                sendTo a (Accept me)
                set (Value L (Matched a))
                halt
```

Phase 3. Unmatched vertices in R wait on acceptance messages and set their local state accordingly:

```
          inPhase 3 = do
            val <- get
            when (val == Value R Unmatched) $ do
              msgs <- messages
              unless (null msgs) $ do
                let [Accept a] = msgs
                set (Value R (Matched a))
            when (inSet val == R) halt
```

6.4 Implementation

In this section we describe the implementation of the EDSL and its parallelization by means of Concurrent Haskell. The presentation is occasionally simplified for conciseness.

6.4.1 Implementing Palovca in Haskell

The two most important types of Palovca are `Graph` for graphs, and `Vertex` for single vertices (see Figure 6.16 on page 148). As already stated in Section 6.3.1, all type variables need to be instances of `NFData`. This avoids that the evaluation of expressions is delayed due to lazy evaluation. Instead evaluation is forced inside concurrently running threads.

In the following we explain the elements of the two types by going through the computation of a single superstep for a vertex-centric algorithm `f`. The computation is divided into three phases: vertex computation (which includes message passing), aggregator computation, and termination checking. Another phase, topology modification, handles vertex and edge modifications, but it is not shown for being similar to the aggregator phase.

Phase 1: Vertex Computation

All vertices are stored in a `GArray (1)` which resembles an `IOArray` [80, 81] but supports additional parallel map- and fold-like operations; its internal details are explained in the

```
data (NFData v, NFData e, NFData m) => Graph v e m = Graph {
        gVertices    :: GArray (Vertex v e m)       -- (1)
      , gSuperstep   :: Int                         -- (2)
      , gAggregator  :: IORef (Maybe ([v] -> v))    -- (3)
      , gAggValue    :: Maybe v                     -- (4)
      , gAggChannel  :: SChan v                     -- (5)
}

data (NFData v, NFData e, NFData m) => Vertex v e m = Vertex {
        vIndex       :: Index                       -- (6)
      , vValue       :: IORef v                     -- (7)
      , vHalt        :: IORef Bool
      , vEdges       :: IORef [Edge e]              -- (8)
      , vMessages    :: (SChan m, SChan m)          -- (9)
}

type Index    = Int
type Edge a   = (Index, a)
type SChan m = IORef [m]
```

Figure 6.16: The `Graph` and `Vertex` data types contain informations for representing arbitrary graphs and BSP-based computation.

next section. The first phase, which is computationally most expensive, evaluates `f` on all vertices of the graph: for inactive vertices nothing is done, for active vertices `f` is executed in a monadic `GraphM` context. `GraphM` is a `StateT`-based wrapper encapsulating the graph as well as the currently computed vertex inside the `IO` monad. This allows to use channel-based concurrent communication and mutable variables:

```
type GraphM v e m = StateT (ComputeState v e m) IO

data ComputeState v e m = ComputeState {
        cVertex :: Vertex v e m
      , cGraph  :: Graph v e m
}
```

Functions in the domain specific language therefore access and modify the current vertex or the graph. For example, modifying the local vertex state `(7)` with the Palovca function `set` is internally defined by

```
set :: v -> GraphM v e m ()
set v = modify (const v)
```

```
modify :: (v -> v) -> GraphM v e m ()
modify f = do
    v <- vertex
    liftIO $ modifyIORef (vValue v) f

-- Internal functions
vertex :: GraphM v e m (Vertex v e m)
vertex = access cVertex

access :: (ComputeState v e m -> a)  -> GraphM v e m a
access f = f `liftM` StateT.get
```

Note that the use of higher-order functions also improves conciseness of the Palovca-internal code. In addition to computing `f`, message passing is also performed in the first phase. Each vertex has a graph-wide unique identifier `(6)` which serves as its index in the `GArray`. The directed edges to its neighbors are stored as (identifier, edge-value) tuples `(8)`. When a message is sent, all tuples are iterated over to contact neighbors. The basis for sending messages to all neighbors is the `sendTo` function, which sends a message to a particular vertex. It accesses a particular message channel `(9)` over the `GArray` by

```
sendTo :: Index -> m -> GraphM v e m ()
sendTo dst msg = do
    vs <- access (gVertices . cGraph)
    v  <- liftIO (vs `GArray.at` dst)
```

In the BSP model messages are buffered between successive supersteps (see Section 6.2). We use a pair of channels for each vertex to implement this buffering (see Figure 6.17). In even supersteps messages are read from the first channel and written to the second one, in odd ones this is reversed:

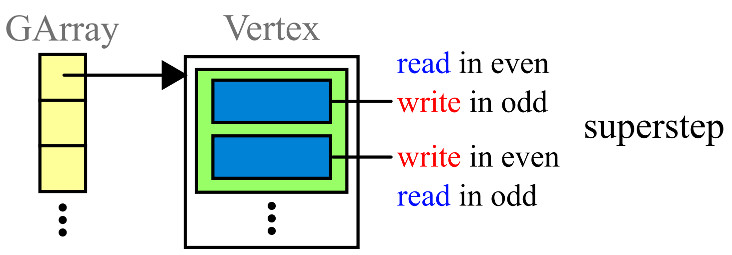


Figure 6.17: Visualization of BSP-based message passing with two buffers. The green box resembles (9) in Figure 6.16, the blue boxes resemble an `SChan`.

```
    s <- superstep
    let choose = if s `mod` 2 == 0 then fst else snd
        chan   = choose (vMessages v)
    liftIO $ atomicModifyIORef chan (\msgs -> msg:msgs)
```

In a previous version we separated message channel pairs to prevent contention which could occur when too many threads access the channel of one vertex. Each vertex had an *array* of pairs of message channels, such that each thread wrote messages to its own non-shared channel. Instead of improving performance it actually slowed down the computation. We think one reason was the increased number of lookups to access the correct channel for writing. Another reason was the additional overhead of combining the input of all channels for later reading.

Note that although we use the term channel (`SChan`) to describe the message buffers, we internally implement a message buffer as a list of elements wrapped in an `IORef` and modified by `atomicModifyIORef`. For our particular use case, our measurements showed that it was slightly faster than traditional channels.

Phase 2: Aggregator Collection and Distribution

In the first phase, where the vertex state was updated and messages were sent, vertices were also able to submit values to the global aggregator by calling `aggregate`. The `aggregate` function accesses the graph's aggregator channel `(5)` and writes a value to it. In the second phase a new global aggregation value is calculated by using the combination function initially defined in `(3)`: all values from the channel are read, combined and stored in `(4)`. The combined value is therefore available in the following superstep. This phase is currently implemented sequentially: All aggregated values are stored in a channel. For parallelization, the stored values would have to be read from the channel and stored in a `GArray`. While an implementation of this approach is straightforward, it induces a certain overhead. In preliminary tests the sequential computation was sufficiently fast, i.e. the time for the computation of the aggregated value is negligible compared to the other phases.

Phase 3: Termination Detection

In the third phase all vertices are scanned: if an inactive vertex has new messages waiting in its channel, its state is changed back to active, and it will participate again in phase one of the following superstep. It is then checked if any active vertices exist. If not, the overall computation is stopped. Otherwise, phase one is restarted.

Since all operations on vertices are performed independently, the order of execution is not important. In fact, the operations can be performed in parallel (using map in all phases and an additional fold in phase three). Since vertices are stored in a `GArray`, parallel operations are implemented as `GArray` functions and are discussed in the following section.

6.4.2 Dynamic Arrays and Parallel Vertex Evaluation

The `GArray` data structure, which is used to store the vertices of the graph implements a dynamic (growable) array with additional support for parallel map- and fold-like operations.

A growable array is needed for vertex addition (vertex removal is not supported yet). Our implementation uses the traditional way to implement such arrays: if the array is full, a new one with twice its original size is created and the old contents are copied. Initially we implemented a chunk-based growable array. It basically works like a linked list where the list nodes are arrays. If the last array of the list is full, a new empty one is appended. Elements are accessed over their index by traversing through the list. While this approach prevents copying of elements to the larger array, the time to access elements with large indices increases due to the needed traversal. Since vertex additions are less frequent than the access to elements over their indices, we chose the copy-based approach.

The four phases mentioned in the last section need two parallel operations: executing a function for every element, and evaluating a binary function for every successive pair of elements. They resemble the well-known `mapM_`-function and a fold-like `foldM`-function, respectively. In the following we describe parallel implementations for both.

`mapM_` Implementation

For parallelization of `mapM_`, the used part of the array is divided into chunks of a definable size (see Figure 6.18 on page 152). Forked threads work on these chunks in parallel. For each element of the chunk the given function is called. By default the chunksize is chosen such that each thread works on one chunk (see Section 2.1.2).

The `mapM_`-implementation of `GArray` is similar to `Prelude.mapM_` but restricted to the `IO`-monad and needs an additional parameter that defines a chunksize:

```
mapM_ :: ChunkSize -> (a -> IO b) -> GArray a -> IO ()
type ChunkSize = Int
```

Note that for more irregular vertex-centric algorithms with large differences in the local vertex computation time, load balancing can be achieved by manually choosing a smaller chunk size. We intentionally did not implement a `mapM`-function that also returns the results of the called function since it is not required in our context.

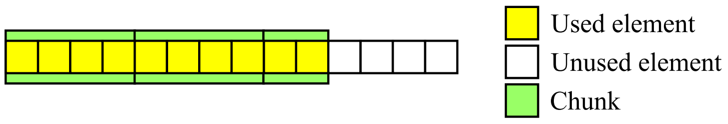


Figure 6.18: Visualization of the `GArray.mapM_` function. The array has a length of 14 elements, of which 10 elements are used. The chunksize is 4 and the three chunks may be processed in parallel.

`foldM` Implementation

To collect and combine information about vertices, for example to obtain a summary of the current activation states, we need a parallelized fold-like function:

```
foldM :: ChunkSize -> (b -> a -> IO b) -> (b -> b -> b) -> b
  -> GArray a -> IO b
```

`foldM` calls the fold function `b -> a -> IO b` for each chunk in parallel (using the above scheme) and returns the combined result. The combination of the results of the chunks is computed sequentially since the number of chunks is small and the combinator function is fast to evaluate (see Figure 6.19).

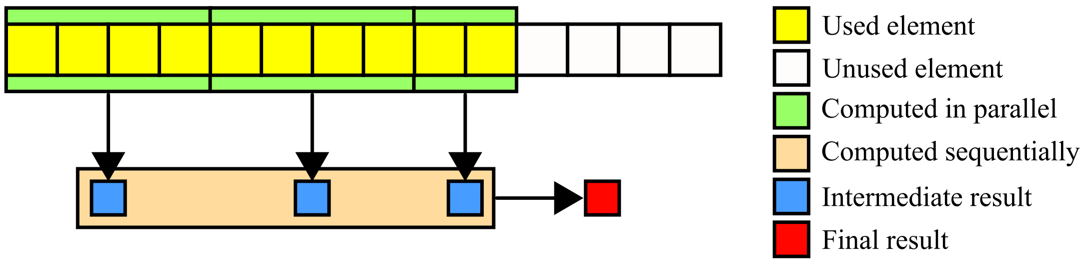


Figure 6.19: Visualization of the `GArray.foldM` function.

6.5 Benchmarks

We executed experiments on a 2.3 GHz 16-core AMD Opteron 6134 with 32 GB RAM running a Linux-kernel 2.6.38-8 with GHC 7.0.3. Similar to Pregel and to ease comparison, we measured only the pure computation time, i.e. excluded graph reading. The speedup is calculated relative to the single thread version, and all stated calculation times are the average of three runs.

The number of generated messages and thus allocated short-living chunks of memory is quite high and puts a lot of stress on the parallel garbage collector. By increasing the amount of memory that is allocated at once, we decrease the calls to the garbage collector and, since the collectors runs over the allocated data in parallel, increase the overall performance. For all benchmarks we chose `-A1G -H16G` for sequential and `-A1500M` for parallel runs as additional GHC runtime parameters (found by experimentation). The option `-H` sets the initial heap size (with `G` for gigabyte and `M` for megabyte as potential suffixes) and `-A` sets the size of the allocation area if the garbage collector needs to allocate more memory [77]. Although these parameters worked well for all our benchmarks, it should be stated that parameter tuning for a particular algorithm and graph type might result in even better speedups.

We chose four different benchmarks with varying number of active vertices and message sizes. In the pagerank benchmark the computational time per vertex is nearly equal and all vertices work until the calculation is finished. In the single-source shortest path benchmark the number of active vertices depends on the graph structure and is more irregular. In the semi-clustering benchmark computational times are also more irregular and the message size increases over the course of the computation. In the bipartite matching benchmark the computation time per superstep is small. In addition, matched vertices remove edges to unmatched vertices, hence the number of sent and received messages decreases over time. To keep the presentation of the matching algorithm simple, edge removal was not shown in the example of Section 6.3.5.

We used randomly generated graphs with v vertices. The probability that two vertices are connected, i.e. an edge is generated for them, was p. The values for v and p were as follows: For the first two benchmarks we chose $v = 10^5$ and for the third $v = 10^4$. To see the effect of additional computational load through more communication, we chose two different probabilities for edge generation, $p_1 = 0.0001$ and $p_2 = 0.0002$ such that in the second case of each benchmark the number of edges is doubled on average. Note that the number of vertices is much smaller than in the Pregel benchmarks since Pregel runs on a distributed system with hundreds of machines and therefore can handle much larger data sets. Since Pregel is not publicly available we were not able to directly compare running times with our testing machine.

6.5.1 Pagerank

Figure 6.20 on page 154 shows the benchmark results for the pagerank algorithm. Since all vertices are active over the whole computation and by default each thread works on a chunk of the same size, both speedup graphs correspond to a typical speedup curve. Linear speedup is not reached as it is typical for parallel computations. For p_1, the small computation time of the pagerank algorithm in combination with few connections between vertices lead to a stagnation of speedup if more than 12 threads are used. For p_2, the influence of more communication is visible. When the number of edges is doubled, more communication and thus more computations per vertex have to be performed, which results in better scaling.

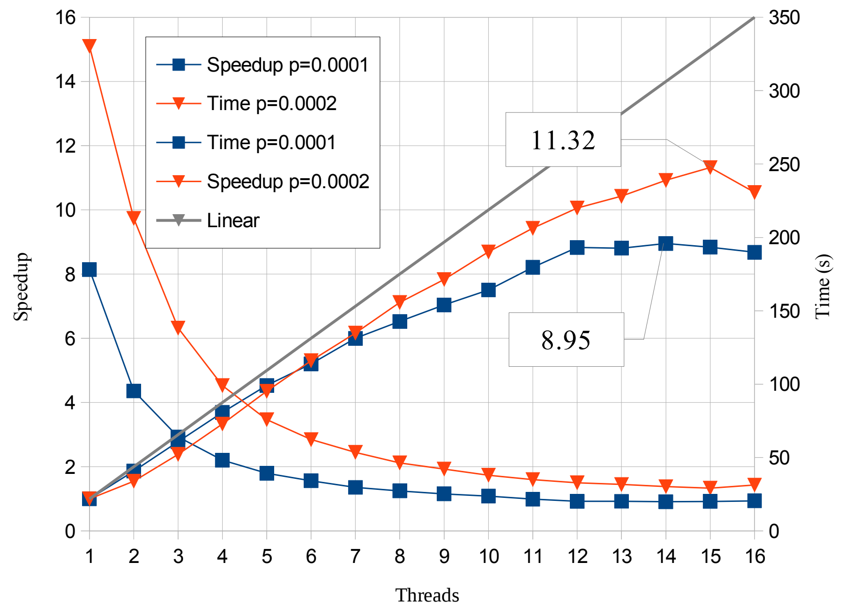


Figure 6.20: Pagerank speedup and computation times for a random graph with $v = 10^5$ and $p_1 = 0.0001$ and $p_2 = 0.0002$, respectively. In this figure as well as the following ones, increasing graphs illustrate the speedup, decreasing graphs illustrate computation time.

6.5.2 Single Source Shortest Path

Figure 6.21 shows the results for the SSSP algorithm. Due to the algorithm, all but the first vertex are initially inactive and only become active over time. An efficient parallelization is thus more difficult. Nevertheless, both speedup graphs resemble a typical speedup curve. The jump of the speedup graph for p_2 at eight cores might be due to cache effects.

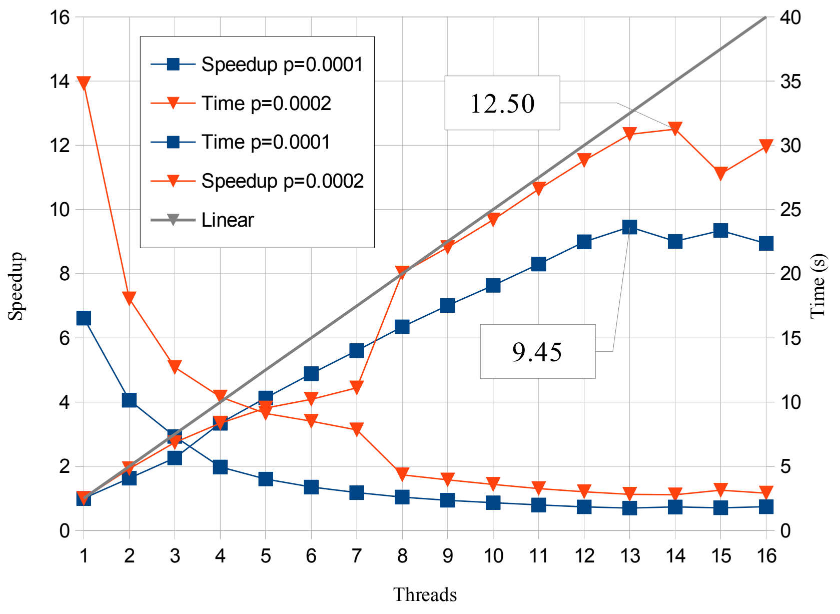


Figure 6.21: SSSP speedup and computation times for a random graph with $v = 10^5$ and $p_1 = 0.0001$ and $p_2 = 0.0002$, respectively.

6.5.3 Semi-Clustering

In the semi-clustering algorithm, the local computation time is higher and more irregular than in the previous two benchmarks. Figure 6.22 on page 156 shows the speedup graphs, which scale to about 10 for 16 threads. Both speedup graphs are similar and all threads are utilized, since the computational load is already high for p_1.

6.5.4 Bipartite Matching

In contrast to the previous benchmarks, the bipartite matching algorithm does not perform intensive computations nor does it send many messages. Since edges are removed once vertices are matched, fewer messages are sent over time. These properties lead to less parallelism, which is mirrored in the speedup graphs for p_1 and p_2 (see Figure 6.23 on page 156). While both graphs show a speedup, it is not as steep as in the previous benchmarks.

6.5.5 Summary

In this section we have presented different benchmarks with various properties regarding the amount of computational work per vertex, or the size and number of sent messages. Although the actual speedup is obviously dependent on the algorithm, all but one benchmark show a typical speedup curve with a speedup of about 10 with 16 threads.

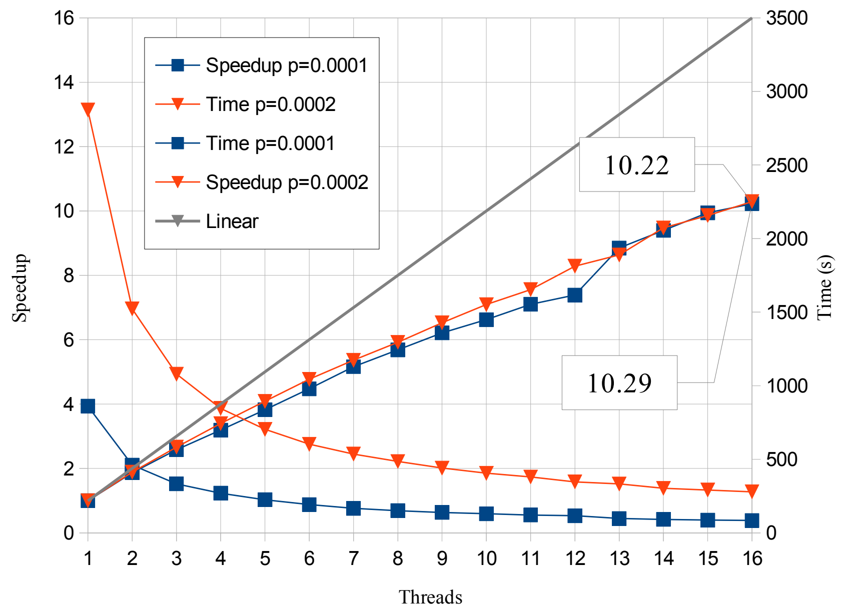


Figure 6.22: Semi-clustering speedup and computation times for random graphs with $v = 10^4$, $C_{Max} = 20$, $V_{Max} = 20$. Probabilities are $p_1 = 0.0001$ and $p_2 = 0.0002$, respectively.

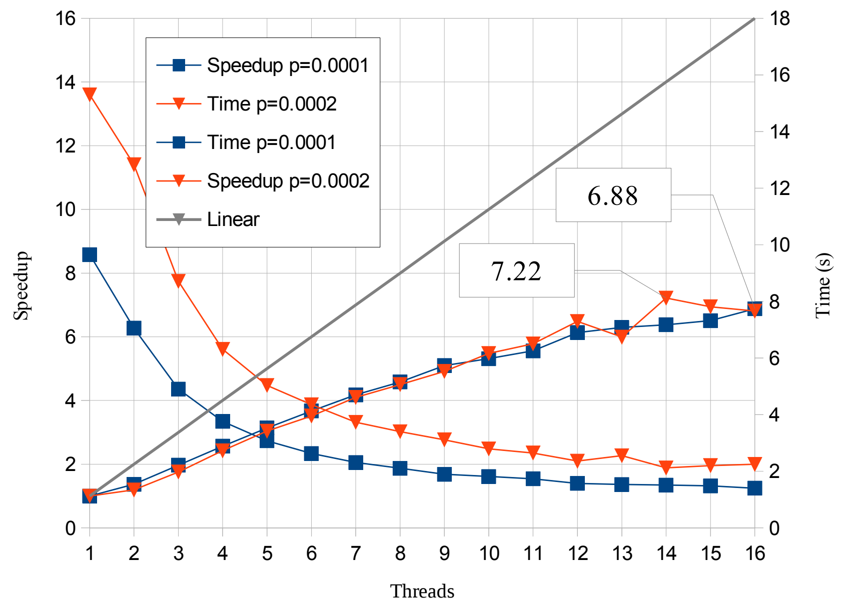


Figure 6.23: Bipartite matching speedup and computation times for a random graph with $v = 10^5$ and $p_1 = 0.0001$ and $p_2 = 0.0002$, respectively.

6.6 Related Work

The foundation of Palovca's computational model is the Bulk Synchronous Parallel model of Valiant [200, 201]. While some BSP libraries exist for classical programming languages, e.g. the Paderborn University BSP library by Bonorden et al. [17] or the Green BSP library by Goudreau et al. [64], we are not aware of any Haskell-based implementations. We can think of three Haskell techniques that could be used to implement BSP-like computations, or a BSP library, respectively: Glasgow Distributed Haskell (GdH) [163], the CHP library [19] and data parallel Haskell [111]. GdH implements parallel and concurrent computations on distributed-memory systems. It allows to work with the same concurrent primitives that are traditionally used for parallel programming with Haskell in a distributed environment. CHP implements combinators to allow communication of concurrent processes. Its computational model, which is based on Hoare's communicating sequential processes [96], is similar to BSP: Processes communicate over synchronous channels, interact via barriers and do not share any local data. Despite its similarity with BSP we did not choose CHP for the core of Palovca since computations in the `IO`-monad are rather difficult to express but were necessary for efficiency. Since vertex-centric computations are compellingly data-parallel, an efficient execution should be possible with parallelized array operations as they are provided in Data Parallel Haskell. Unfortunately it is yet unclear how to handle communication between vertices in a side-effect free (pure) environment efficiently, although this might be an interesting topic for future work.

Palovca was strongly influenced by Malewicz et al.'s paper on Pregel [133]: they implemented a distributed platform in C++ for large-scale vertex-centric graph algorithms with billions of vertices. Albeit difficult to compare due to their different architectural basis, their implementation achieves similar speedups as ours: when utilizing 16 times as many worker threads (absolute numbers were not stated in [133]) they reach speedups of about 10 for the SSSP benchmark.

As we mentioned in Section 6.1, Haskell EDSLs are quite successful for a variety of domains. To the best of our knowledge there are no other DSLs for any problems that parallelize the execution nor libraries that allow to describe general (vertex-centric) graph algorithms. The closest approach to Palovca is the containers package, which defines the module `Data.Graph`. It allows to describe graphs and perform standard algorithms such as depth-first search [119].

6.7 Summary and Conclusion

In this chapter we defined and implemented Palovca (PArallel Local Vertex Calculations in hAskell) which consists of a Haskell-based embedded domain specific language (EDSL) for vertex-centric graph algorithms and an execution platform for shared-memory multicore machines. The underlying computational model, which influenced the definition of the language's vocabulary, is an adaption of Valiant's Bulk Synchronous Parallel model of computation. The EDSL defines a vocabulary for vertex-centric graph algorithms and is implemented on top of a custom monad and higher-order functions. The underlying execution platform uses Concurrent Haskell for the parallel execution of the graph algorithms. Hence, a user can focus on writing algorithms by means of the provided vocabulary. All details of the parallelization as well as helper code for the EDSL are hidden through the monadic interface. To examine the scalability of the execution platform as well as show the conciseness and simplicity of the EDSL, we presented various examples which differed in their development complexity as well as their performance characteristics. We have shown implementations of a pagerank algorithm, a single source shortest path computation, a semi-clustering algorithm, and a bipartite matching algorithm. In our benchmarks we executed these algorithms in parallel and achieved speedups between 7.5 and 11 with 16 threads, depending on the particular algorithm and source graph. Unlike other data-parallel applications we did not yet achieve nearly linear speedups, since the irregularity of the vertex computations can be high. To summarize, we found that the general approach of implementing a BSP-based computation wrapped in an EDSL with Haskell as the host language and applying it to vertex-centric computations is feasible, allows for great expressiveness, and enables good parallel performance.

As far as abstraction is concerned, our experiments have shown that parallelization and embedded domain specific languages work very well together: Monads allow to implement a domain specific language with a problem-specific vocabulary. The type system enforces that no other operations can be executed. Higher-order functions allow the user to write concise yet understandable code. Hence, the combination of Concurrent Haskell for the underlying execution platform on the one hand, and a monadic approach for the definition of a problem-specific language on the other works and supports the convenient and efficient parallelization of a particular problem domain. This flexibility is yet unmatched in traditional programming languages.

Chapter 7
Perspectives and Closing Remarks

7.1 Future Work

In this section we briefly describe potential avenues for future work. First we show research directions for each of the examined topics separately. Then we reflect on general ideas for future research.

Thread-Safe Priority Queues based on Skiplists. First, despite their sole use as priority queues in this thesis, skiplists are general dictionary structures. A comparison of their concurrent performance for arbitrary insertions and deletions seems interesting. Second, the Glasgow Haskell Compiler did not support native (and thus fast) atomic compare-and-swap operations at the time of our experiments. This situation will change with one of the next releases [136]. We think that native compare-and-swap operations will improve the performance of the CAS skiplist implementation to a large extent. Additionally, an improved CAS operation allows to implement other lock-free data structures [43, 57] efficiently as building blocks for parallel algorithms. Third, implementing thread-safe variants of pure data structures (e.g. finger trees [95]) with Concurrent Haskell looks interesting and has not yet been well researched. It might be interesting to find out whether purity can ease the implementation or improve the scalability of thread-safe variants.

A Comparison of Lock-based and STM-based Taskpools. First, by implementing more efficient alternatives for thread-local storage for STM, the private taskpools with and without task stealing may deliver better parallel performance. Second, since taskpools are so widely used and our performance results are equal to those for imperative languages [208], developing advanced taskpool variants in Haskell is a viable alternative to implementations in imperative languages. In particular, taskpool-based algorithms could be *annotated* with information about the computed problem. Then these information is used to, for example guide the implementation in choosing an efficient taskpools variant. Haskell's type system and our monadic implementation would allow for a concise implementation of this approach. Third, exploring the advantages and disadvantages of the different taskpool variants by implementing additional real world examples, e.g. from [170], might be worthwhile. Fourth, both DPH and the `Par` monad allow for the implementation of particular kinds of taskpool algorithms. Hence a further investigation of the respective advantages and disadvantages of these approaches is interesting.

Declarative Description and Parallelization of Stencil Computations. First, as a preliminary stage, a stencil database for widely-used stencils can be created. For each defined stencil, an optimized implementation for its computation would be used. If the stencil is not known, the existing generalized implementation is chosen. Second, all information about the stencil is known at compile-time. Hence, the absolute performance of the computations can be improved by using Template Haskell [178] or other means of compilation to generate stencil-specific code. Third, the advantages of the current dynamic computation can be used for automatic tuning. The execution platform can measure different performance characteristics and, for example, switch to another grid implementation which is more efficient in the particular scenario. In fact, there is already much research for stencil-based autotuning [30, 40, 46], although not for Haskell.

Describing and Executing Graph Algorithms. First, we have not yet used the advantages of having a restricted problem domain or the knowledge of the particular graph structure. This information can be used to optimize computations, e.g. by modifying parameters of the execution platform at runtime depending on the irregularity of the vertex computations. Second, further experiments with larger graphs could explore scalability and the efficient usage of the garbage collector with high amounts of short-living data. Third, it would be interesting to use the EDSL to generate source code in a low-level language and link it to an already existing BSP library, as for example done in [6]. Fourth, the vertex-centric computations are compellingly data parallel. Implementing an execution platform that uses GPUs might significantly improve the performance. An intermediate step might be the implementation of an execution platform which uses Data Parallel Haskell to generate GPU code [25].

General Future Work. The current rise of experimental shared-memory many-core architectures [106] poses the question of the scalability of our implementations. In particular, on this architectures memory access time is often non-uniform and depends on the position of a memory cell relative to a core [104]. The behavior of Haskell's parallel runtime system on this type of architecture is largely unexplored, since GHC's parallel runtime does not optimize for data locality [78]. While it is difficult to access experimental platforms in general, Intel provides access to some of them [108]. Since there is experimental Haskell support, it would be interesting to analyze the scalability and possible modifications of our programs on these platforms.

As we have demonstrated, certain parallel design patterns work well combined with the abstraction techniques of Haskell. Our approach could be adapted to other parallel design patterns to form a common library. For the purpose of a concise and idiomatic formulation of parallel algorithms, this library could use, for example, monads, higher-order functions, or typeclasses, as in our approach. In particular, this would allow for an easier adoption of concepts which were developed for traditional programming languages while still enabling the benefits of Haskell.

Besides Concurrent Haskell there are also other parallelization approaches in Haskell. In the functional programming community, semi-explicit parallelization is favored due to its rather declarative nature, and Data Parallel Haskell is favored due to the automatic parallelization of its computations. As of yet, the advantages and disadvantages of each approach have not yet been compared for different scenarios side-by-side. In particular, it might be worthwhile to have a catalog which enlists various scenarios and their respective performance characteristics, best practices, and pitfalls for each parallelization approach.

7.2 Closing Remarks

Our goal was to research how well abstraction at different degrees supports programming with Concurrent Haskell. We were interested in answering the question whether commonly used idioms of the functional paradigm as well as high-level constructs of the Haskell language allow to ease parallel programming with Concurrent Haskell. How can traditional imperative data structures, design patterns and programming approaches be adapted to a modern functional programming language? To answer these questions, we examined two areas of interest: 1) comparison of synchronization approaches and 2) exploration of abstraction techniques to hide details of parallelization.

For a comparison of synchronization approaches, we first examined the advantages and disadvantages of low-level as well as high-level synchronization techniques. We implemented a thread-safe priority queue on top of the skiplist data structure. For synchronization, locks, software transactional memory (STM) as well as compare-and-swap (CAS) operations were compared. Our experiments showed that the high-level approach of STM scaled worse compared to the other approaches, although STM made the implementation easier to develop. Second, to compare locks with STM in a scenario where transactions are short-lived, we implemented the taskpool design pattern for different taskpool variants (global taskpools, and private taskpools with and without task stealing). Additionally, we provided an abstraction layer that allowed for a convenient and idiomatic formulation of taskpool algorithms. Our experiments showed that both synchronization approaches scaled comparably well.

Summarizing our analysis of synchronization approaches, a high-level approach to synchronization such as STM has advantages as well as disadvantages. While implementing synchronization is much easier and less error-prone than with locks or compare-

and-swap operations, special care has to be taken to ensure that transactions are short. Locks may in fact be easier to use if there is already a well-known lock-based solution which can be adapted. In general, we propose to use STM in the following way: By providing an abstraction layer, details of the underlying synchronization approach can conveniently be hidden from the user. If the necessity arises, the abstraction layer also allows for an easy replacement by more complex but also more efficient synchronization approaches such as locks. The sophisticated abstraction techniques of Haskell, e.g. monads, typeclasses, and higher-order functions amongst others, are especially useful for implementing such layers.

For our exploration if Haskell's abstraction techniques allow to hide complex details of the parallelization, we first examined stencil-based algorithms. We developed a library that allows for a declarative description of stencil-based algorithms as well as their parallel execution. Besides stating the problem, no further information is necessary. In particular, all aspects of the parallel computation are hidden. With this library we have shown that an idiomatic description of stencil-based algorithms is possible. In particular, the problem notation should feel natural to most Haskell developers. Subsequently, we developed an embedded domain specific language (EDSL) for vertex-centric graph algorithms as well as an execution platform that allows for their automatic parallel execution. By means of the EDSL a type-safe, concise and convenient formulation of vertex-centric graph algorithms is possible. By means of the execution platform, all aspects of the parallelization are hidden from the user. We provided several examples which demonstrated that the EDSL allows for a concise yet understandable formulation of graph algorithms. Our benchmarks showed that the execution platform scales well.

Summarizing our analysis of Haskell's abstraction techniques, in cases where solutions to problems of a particular domain are parallelized, either a declarative formulation or a domain specific language can hide aspects of Concurrent Haskell and thus hide complex details of parallel programming. The sophisticated language constructs of Haskell allow for a convenient and concise formulation of these approaches.

In Chapter 1 we wrote that *"[...] the parallelization of a program is difficult, even for experts."*. With our thesis we have demonstrated that a modern functional language like Haskell offers several sophisticated techniques to implement abstraction. These techniques can be used to ease parallel programming even for rather traditional parallel programming models such as Concurrent Haskell. While the flexibility of Haskell is yet unmatched in traditional programming languages, we believe that the introduced approaches would be worthwhile to adapt by these languages, such that parallel programming becomes easier, even for non-experts.

List of publications

Michael Lesniak. Palovca: Describing and Executing Graph Algorithms in Haskell. In *Fourteenth International Symposium on Practical Aspects of Declarative Languages (PADL 2012), number 7149 in Lecture Notes in Computer Science (LNCS)*, pages 153 – 167. Springer Verlag, Berlin, 2012.

Michael Lesniak. Thread-safe Priority Queues in Haskell Based on Skiplists. In *Twelth International Symposium on Trends in Functional Programming (TFP 2011).* To appear in Lecture Notes in Computer Science (LNCS).

Michael Lesniak. A Comparison of Lock-based and Lock-free Taskpool Implementations in Haskell. In *Proceedings of the International Conference on Computational Science (ICCS 2011)*, Procedia Computer Science Volume 4, pages 2317 – 2326, 2011.

Michael Lesniak. PASTHA – Parallelizing Stencils in Haskell. In *Proceedings of the POPL 2010 Workshop on Declarative Aspects of Multicore Programming (DAMP 2010)*, pages 5 – 14, ACM, 2010.

Bibliography

The existence and content of online resources, i.e. reference entries with an accompanying URL, have last been checked on 06. March 2012.

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996

[2] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986

[3] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, Inc., 2007

[4] Nélio Muniz Mendes Alves and Sérgio de Mello Schneider. Implementation of an Embedded Hardware Description Language Using Haskell. *Journal of Universal Computer Science*, 9(8):795–812, 2003

[5] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. AFIPS '67, pages 483–485, 1967

[6] Christopher Kumar Anand and Wolfram Kahl. A Domain-Specific Language for the Generation of Optimized SIMD-Parallel Assembly Code. SQRL Report 43, Software Quality Research Laboratory, McMaster University, May 2007

[7] Thomas E. Anderson, Susan S. Owicki, James B. Saxe, and Charles P. Thacker. High Speed Switch Scheduling for Local Area Networks. *ACM Transactions on Computer Systems*, 11:98–110, 1993

[8] Joe Armstrong. A history of Erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1–6–26, New York, NY, USA, 2007

[9] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007

[10] 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, EECS Department, University of California, Berkeley, 2006

[11] Lennart Augustsson. More Points For Lazy Evaluation.
`http://augustss.blogspot.com/2011/05/more-points-for-lazy-evaluation-in.html`

[12] F. Barrett, P. C. Roth, and S. W. Poole. Finite Difference Stencils Implemented Using Chapel. Technical Report TM-2007/119, ORNL Technical Report, 2007

[13] Daniel Bauer and Kristijan Dragicevic. A survey of concurrent priority queue algorithms. In *International Symposium on Parallel and Distributed Processing*, pages 1–6, 2008

[14] Andy Ben-Dyke. The History of Parallel Functional Programming.
`ftp://ftp.cs.bham.ac.uk/pub/dist/func-prog/drafts/history/History.tex.gz`

[15] Jost Berthold, Mischa Dieterle, Rita Loogen, and Steffen Priebe. Hierarchical master-worker skeletons. In *Proceedings of the 10th international conference on Practical aspects of declarative languages*, PADL'08, pages 248–264, Berlin, Heidelberg, 2008. Springer-Verlag

[16] Guy Blelloch. Is Parallel Programming Hard?
`http://software.intel.com/en-us/articles/is-parallel-programming-hard-1`

[17] Olaf Bonorden, Ben H.H. Juurlink, Ingo von Otte, and Ingo Rieping. The Paderborn University BSP library. *Parallel Computing*, 29(2):187–207, 2 2003

[18] Sergey Brin and Lary Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. In *Seventh International World-Wide Web Conference*, 1998

[19] Neil C. C. Brown. Communicating Haskell Processes: Composable Explicit Concurrency using Monads. In *Communicating Process Architectures*, pages 67–83, 2008

[20] David R. Butenhof. *Programming with POSIX threads.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997

[21] Caml.Inria.fr. OCaml Homepage.
`http://caml.inria.fr/ocaml/`

[22] Colin Campbell, Ralph Johnson, Ade Miller, and Stephen Toub. *Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures.* Microsoft Press, 2010

[23] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, DAMP '07, pages 10–18, New York, NY, USA, 2007. ACM

[24] Manuel M.T. Chakravarty, Gabriele Keller, Roman Lechtchinsky, and Wolf Pfannenstiel. Nepal – Nested Data-Parallelism in Haskell. In *EURO-PAR*, pages 524–534. Springer-Verlag, 2001

[25] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM

[26] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21:291–312, 2007

[27] Brad Chamberlain, Steve Deitz, Shannon Hoffswell, John Plevyak, Hans Zima, and Roxana Diaconescu. *Chapel Specification.* Cray Inc, 411 First Ave S, Suite 600, 0.4 edition, February 2005

[28] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation).* The MIT Press, 2007

[29] 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. *SIGPLAN Notes*, 40:519–538, October 2005

[30] Matthias Christen, Olaf Schenk, and Helmar Burkhart. PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures. In *IEEE International Parallel Distributed Processing Symposium*, pages 676–687, 2011

[31] Koen Claessen and John Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *International Conference on Functional Programming*, 2000

[32] Clojure.org. Concurrent Programming in Clojure.
`http://clojure.org/concurrent_programming`

[33] Clojure.org. Homepage.
`http://clojure.org/`

[34] E. F. Codd. *The relational model for database management: version 2.* Addison-Wesley Longman Publishing Co., Inc., 1990

[35] Common-Lisp.net. Bordeaux Threads – portable shared-state concurrency for Common Lisp.
`http://common-lisp.net/project/bordeaux-threads`

[36] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001

[37] Cray.com. Chapel homepage.
`http://chapel.cray.com/`

[38] Pierluigi Crescenzi and Viggo Kann. Approximation on the Web: A Compendium of NP Optimization Problems. In José D. P. Rolim, editor, *RANDOM*, volume 1269 of *Lecture Notes in Computer Science*, pages 111–118. Springer, 1997

[39] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM

[40] Kaushik Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms.* PhD thesis, EECS Department, University of California, Berkeley, 2009

[41] Reinhard Diestel. *Graph Theory.* Springer-Verlag, Heidelberg, 2005

[42] E. Dijkstra. *The humble programmer*, pages 111–125. Yourdon Press, Upper Saddle River, NJ, USA, 1979

[43] Anthony Discolo, Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Satnam Singh. Lock Free Data Structures Using STM in Haskell. In Masami Hagiya and Philip Wadler, editors, *FLOPS*, volume 3945 of *Lecture Notes in Computer Science*, pages 65–80. Springer, 2006

[44] Jack J. Dongarra, Steve W. Otto, Marc Snir, and David Walker. An Introduction to the MPI Standard. Technical report, Knoxville, TN, USA, 1995

[45] Ulrich Drepper. What every programmer should know about memory.
`www.akkadia.org/drepper/cpumemory.pdf`

[46] Hikmet Dursun, Ken-Ichi Nomura, Liu Peng, Richard Seymour, Weiqiang Wang, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. A multilevel parallelization framework for high-order stencil computations. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 642–653, Berlin, Heidelberg, 2009. Springer-Verlag

[47] Hesham El-Rewini and Ted G. Lewis. *Distributed and parallel computing.* Manning Publications Co., 1998

[48] Conal Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 45–56, New York, NY, USA, 2004. ACM

[49] Conal Elliott. Tangible functional programming. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 59–70, New York, NY, USA, 2007. ACM

[50] Erlang.org. The History of Erlang.
`http://www.erlang.org/course/history.html`

[51] Erlang.org. Homepage.
`http://www.erlang.org/`

[52] Eitan Farchi, Yarden Nir, and Shmuel Ur. Concurrent bug patterns and how to test them. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, pages 286.2–, Washington, DC, USA, 2003. IEEE Computer Society

[53] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59, New York, NY, USA, 2004. ACM

[54] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A study of the internal and external effects of concurrency bugs. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 221–230. IEEE, 2010

[55] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Longman Publishing Co., Inc., 1995

[56] Martin Fowler. *Domain-Specific Languages.* Addison-Wesley Professional, 2010

[57] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25, 2007

[58] Free Software Foundation. The Gnuplot Homepage.
`http://www.gnuplot.info/`

[59] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-Oblivious Algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285–, Washington, DC, USA, 1999. IEEE

[60] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995

[61] Jens Gerlach, Peter Gottschling, and Uwe Der. A Generic C++ Framework for Parallel Mesh-Based Scientific Applications. In *Proceedings of the 6th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 45–54, London, UK, 2001. Springer-Verlag

[62] Anwar Ghuloum. What makes parallel programming hard?
`http://blogs.intel.com/research/2007/08/03/what_makes_parallel_programmin`

[63] Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, David Holmes, and Tim Peierls. *Java Concurrency in Practice.* Addison-Wesley Longman, 2006

[64] Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, and Thanasis Tsantilas. Portable and Efficient Parallel Computing Using the BSP Model. *IEEE Transactions on Computers*, 48, 1999

[65] Paul Graham. *On LISP: Advanced Techniques for Common LISP.* Prentice Hall, September 1993

[66] Ananth Grama, George Karypis, Vipin Kumar, and Anshul Gupta. *Introduction to Parallel Computing (2nd Edition).* Addison Wesley, 2nd edition, 2003

[67] Clemens Grelck and Sven-Bodo Scholz. SAC: a functional array language for efficient multi-threaded execution. *International Journal of Parallel Programming*, 34:383–427, August 2006

[68] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir. *MPI - The Complete Reference: Volume 2, The MPI-2 Extensions.* MIT Press, Cambridge, MA, USA, 1998

[69] John L. Gustafson. Reevaluating Amdahl's law. *Communications of ACM*, 31: 532–533, May 1988

[70] Stuart Halloway. *Programming Clojure.* Pragmatic Programmers. Pragmatic Bookshelf, 1 edition, 2009

[71] Kevin Hammond and Greg Michelson, editors. *Research Directions in Parallel Functional Programming.* Springer-Verlag, London, UK, 2000

[72] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM

[73] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag

[74] Haskell.org. Documentation: Control.Concurrent.STM.
`http://hackage.haskell.org/packages/archive/base/latest/doc/html/Control-Concurrent-STM.html`

[75] Haskell.org. Documentation: Control.Monad.
http://hackage.haskell.org/packages/archive/base/latest/doc/html/Control-Monad.html

[76] Haskell.org. GHC - Frequently Asked Questions.
http://www.haskell.org/haskellwiki/GHC:FAQ

[77] Haskell.org. GHC Runtime Parameters for controlling the Garbage Collector.
http://www.haskell.org/ghc/docs/7.0.3/html/users_guide/runtime-control.html

[78] Haskell.org. Brief information about NUMA architectures and GHC.
http://www.haskell.org/haskellwiki/GHC/Data_Parallel_Haskell

[79] Haskell.org. Hackage – The Haskell Package Repository.
http://hackage.haskell.org

[80] Haskell.org. An Introduction to Haskell Arrays.
http://en.wikibooks.org/wiki/Haskell/Hierarchical_libraries/Arrays

[81] Haskell.org. The arrays package on Hackage.
http://hackage.haskell.org/package/array-0.4.0.0

[82] Haskell.org. Documentation: Control.Concurrent.
http://hackage.haskell.org/packages/archive/base/latest/doc/html/Control-Concurrent.html

[83] Haskell.org. IRC Channel.
http://www.haskell.org/haskellwiki/IRC_channel

[84] Haskell.org. Haskell Mailing Lists.
http://www.haskell.org/haskellwiki/Mailing_Lists

[85] Haskell.org. Introduction to Monads.
http://www.haskell.org/haskellwiki/Monad

[86] Haskell.org. Documentation: Data.Sequence.
http://hackage.haskell.org/packages/archive/containers/latest/doc/html/Data-Sequence.html

[87] Haskell.org. Documentation: Data.Set.
http://hackage.haskell.org/packages/archive/containers/latest/doc/html/Data-Set.html

[88] Haskell.org. Documentation: Control.Monad.State.
http://hackage.haskell.org/packages/archive/mtl/latest/doc/html/Control-Monad-State-Lazy.html

[89] Haskell.org. The Haskell Homepage.
`http://www.haskell.org`

[90] Haskell.org. Documentation: Data.IORef.
`http://hackage.haskell.org/packages/archive/base/latest/doc/html/Data-IORef.html`

[91] Haskell.org. Documentation: Data.List.
`http://hackage.haskell.org/packages/archive/base/latest/doc/html/Data-List.html`

[92] Haskell.org. The Glasgow Haskell Compiler.
`http://www.haskell.org/ghc/`

[93] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, New York, NY, USA, 2 edition, 2008

[94] Roger Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969

[95] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16:197–217, March 2006

[96] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 1978

[97] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21:359–411, September 1989

[98] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 134–142. IEEE Computer Society, 1998

[99] Paul Hudak, John Peterson, and Joseph Fasel. A gentle introduction to Haskell 98.
`http://www.haskell.org/tutorial/`

[100] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM

[101] John Hughes. Why functional programming matters. *Computer Journal*, 32(2): 98–107, April 1989

[102] Galen C. Hunt, Maged M. Michael, Srinivasan Parthasarathy, and Michael L. Scott. An efficient algorithm for concurrent priority queue heaps. *Information Processing Letters*, 60:151–157, November 1996

[103] Graham Hutton. *Programming in Haskell*. Cambridge University Press, January 2007

[104] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill Higher Education, 1st edition, 1992

[105] IBM. IBM 650 Model 4 announcement.
`http://www-03.ibm.com/ibm/history/exhibits/650/650_pr4.html`

[106] Intel.com. Platform 2015: Intel Processor and Platform Evolution for the Next Decade.

[107] Intel.com. C++ Compiler Homepage.
`http://software.intel.com/en-us/articles/intel-compilers/`

[108] Intel.com. Intel Manycore Testing Lab.
`http://software.intel.com/en-us/articles/intel-many-core-testing-lab/`

[109] Don Jones, Simon Marlow, and Satnam Singh. Parallel Performance Tuning for Haskell. In *Haskell '09: Proceedings of the second ACM SIGPLAN symposium on Haskell*. ACM, 2009

[110] Mark P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, 1999

[111] Simon L. Peyton Jones, Roman Leshchinskiy, Gabriele Keller, and Manuel M. T. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *FSTTCS*, volume 2 of *LIPIcs*, pages 383–414. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008

[112] Simon Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Tony Hoare, Manfred Broy, and Ralf Steinbruggen, editors, *Engineering theories of software construction*, pages 47–96. IOS Press, 2001

[113] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. September 2002

[114] Simon Peyton Jones and Satnam Singh. A Tutorial on Parallel and Concurrent Programming in Haskell. In *Proceedings of the 6th international conference on Advanced functional programming*, AFP'08, pages 267–305, Berlin, Heidelberg, 2009. Springer-Verlag

[115] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 280–292, New York, NY, USA, 2000. ACM

[116] Michael Kanellos. Intel scientists find wall for moore's law.
`http://news.cnet.com/2100-1008-5112061.html`

[117] Henry Kasim, Verdi March, Rita Zhang, and Simon See. Survey on parallel programming model. In Jian Cao, Minglu Li, Min-You Wu, and Jinjun Chen, editors, *Network and Parallel Computing*, volume 5245 of *Lecture Notes in Computer Science*, pages 266–275. Springer Berlin / Heidelberg, 2008

[118] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM

[119] David J King and John Launchbury. Lazy Depth-First Search and Linear Graph Algorithms in Haskell. *Glasgow Workshop on Functional Programming*, pages 145–155, 1994

[120] Donald E. Knuth. *The art of computer programming*, volume 3. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 2nd edition, 1998

[121] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, LFP '86, pages 151–161, New York, NY, USA, 1986. ACM

[122] Matthias Korch and Thomas Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms. *Concurrency and Computation: Practice and Experience*, 16:1–47, January 2004

[123] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing: design and analysis of algorithms.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994

[124] Claudia Leopold. *Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches.* John Wiley & Sons, Inc., New York, NY, USA, 2001

[125] Michael Lesniak. Pastha: parallelizing stencil calculations in haskell. In Leaf Petersen and Enrico Pontelli, editors, *Proceedings of the POPL 2010 Workshop on Declarative Aspects of Multicore Programming*, pages 5–14. ACM, 2010

[126] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999

[127] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide.* No Starch Press, 2011

[128] Ben Lippmeier and Gabriele Keller. Efficient parallel stencil convolution in haskell. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 59–70, New York, NY, USA, 2011. ACM

[129] Hans-Wolfgang Loidl, Fernando Rubio, Norman Scaife, Kevin Hammond, Susumu Horiguchi, Ulrike Klusik, Rita Loogen, Greg Michaelson, Ricardo Pena, Steffen Priebe, Álvaro J. Rebón Portillo, and Philip W. Trinder. Comparing Parallel Functional Languages: Programming and Performance. *Higher-Order and Symbolic Computation*, 16(3):203–251, 2003

[130] Rita Loogen, Yolanda Ortega-mallén, and Ricardo Peña marí. Parallel functional programming in Eden. *Journal of Functional Programming*, 15:431–475, May 2005

[131] I. Lotan and N. Shavit. Skiplist-Based Concurrent Priority Queues. In *Proc. of the 14th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 263–268, 2000

[132] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. *SIGARCH Computer Architecture News*, 36:329–339, March 2008

[133] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM

[134] Simon Marlow. Parallel and Concurrent Programming in Haskell. `{http://community.haskell.org/~simonmar/par-tutorial.pdf}`

[135] Simon Marlow. Haskell 2010 Language Report

[136] Simon Marlow. Mailinglist discussion about new CAS primitives. `http://permalink.gmane.org/gmane.comp.lang.haskell.glasgow.user/20585`

[137] Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime support for multicore haskell. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 65–78, New York, NY, USA, 2009. ACM

[138] Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K. Aswad, and Phil Trinder. Seq no more: better strategies for parallel Haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 91–102, New York, NY, USA, 2010. ACM

[139] Simon Marlow, Ryan Newton, and Simon Peyton Jones. A monad for deterministic parallelism. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 71–82, New York, NY, USA, 2011. ACM

[140] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for parallel programming.* Addison-Wesley Professional, 2004

[141] John McCarthy. History of LISP. *SIGPLAN Notices*, 13:217–223, August 1978

[142] Paul McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* Kernel.org, Corvallis, OR, USA, 2011

[143] Microsoft.com. Homepage of Visual F#.
`http://msdn.microsoft.com/de-de/library/dd233154.aspx`

[144] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978

[145] Yaron Minsky. OCaml for the masses. *Communications of ACM*, 54(11):53–58, November 2011

[146] M. Moir and N. Shavit. Concurrent Data Structures. In *Handbook of Data Structures and Applications, D. Metha and S. Sahni Editors*, pages 47–14 — 47–30, 2007. Chapman and Hall/CRC Press

[147] G. E. Moore. Progress in digital integrated electronics. In *IEEE International Electron Devices Meeting*, volume 21, pages 11–13. IEEE, 1975

[148] Scott Oaks and Henry Wong. *Java Threads.* O'Reilly, Sebastopol, CA, 3. edition, 2004

[149] Chris Okasaki. *Purely functional data structures.* Cambridge University Press, Cambridge U.K., 1998

[150] OpenMP Architecture Review Board. OpenMP Specifications 3.1.
`http://www.openmp.org/mp-documents/OpenMP3.1.pdf`

[151] OpenMP.org. Homepage.
`http://www.openmp.org`

[152] Oracle.com. JavaDoc: java.lang.Thread.
`http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Thread.html`

[153] Oracle.com. java.util.concurrent.ConcurrentSkipListMap.
`http://download.oracle.com/javase/6/docs/api/java/util/concurrent/ConcurrentSkipListMap.html`

[154] Andy Oram and Greg Wilson. *Beautiful code.* O'Reilly, first edition, 2007

[155] Dominic A. Orchard, Max Bolingbroke, and Alan Mycroft. Ypnos: declarative, parallel structured grid programming. In *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming*, DAMP '10, pages 15–24, New York, NY, USA, 2010. ACM

[156] Bryan O'Sullivan and Johan Tibell. Scalable I/O Event Handling for GHC. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 103–108, New York, NY, USA, 2010. ACM

[157] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell.* O'Reilly Media, Inc., 1st edition, 2008

[158] Marios Papaefthymiou and Joseph Rodrigue. Implementing parallel shortest-paths algorithms. In Sandeep N. Bhatt, editor, *Parallel Algorithms*, volume 30 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 59–68. American Mathematical Society, 1997

[159] Will Partain. The nofib Benchmark Suite of Haskell Programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, 1993. Springer-Verlag

[160] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 295–308, New York, NY, USA, 1996. ACM

[161] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987

[162] Gregory F. Pfister. *In search of clusters (2nd ed.).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998

[163] Robert F. Pointon, Philip W. Trinder, and Hans-Wolfgang Loidl. The Design and Implementation of Glasgow Distributed Haskell. In *Selected Papers from the 12th International Workshop on Implementation of Functional Languages*, IFL '00, pages 53–70, London, UK, 2001. Springer-Verlag

[164] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of ACM*, 33:668–676, June 1990

[165] William Pugh. Concurrent maintenance of skip lists. Technical report, College Park, MD, USA, 1990

[166] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill Education Group, 2003

[167] Fethi A. Rabhi and Sergei Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing.* Springer, 2002

[168] Fethi A. Rabhi and Sergei Gorlatch, editors. *Patterns and skeletons for parallel and distributed computing.* Springer-Verlag, London, UK, 2003

[169] Racket-Lang.org. Homepage of Racket (Scheme).
`http://racket-lang.org/`

[170] T. Rauber and G. Rünger. *Parallel Programming for Multicore and Cluster Systems*. Springer Verlag, 2010

[171] James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007

[172] Paul Roe. Parallel Programming using Functional Languages

[173] Ian Ross. The Invention of the Transistor. *Proceedings of the IEEE*, 86(1):7–28, January 1998

[174] SAC-Home.org. Homepage.
`http://www.sac-home.org/`

[175] Vijay Saraswat. Report on the Programming Language X10. Language specification, IBM, January 2010

[176] Wolfgang Schreiner. Parallel Functional Programming – An Annotated Bibliography (2nd Edition). Technical report, 1993

[177] Peter Seibel. *Practical Common Lisp*. APress, 2004

[178] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 1–16, New York, NY, USA, 2002. ACM

[179] A.B. Shiflet and G.W. Shiflet. *Introduction to computational science: modeling and simulation for the sciences*. Princeton University Press, 2006

[180] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, July 1985

[181] Chris Smith. *Programming F# - a comprehensive guide for writing simple code to solve complex problems*. O'Reilly, 2009

[182] Joshua B. Smith. *Practical OCaml*. Apress, Berkely, CA, USA, 2006

[183] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI – The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998

[184] Matthew Sottile, Timothy G. Mattson, and Craig E. Rasmussen. *Introduction to Concurrency in Programming Languages*. Chapman & Hall/CRC, 1st edition, 2009

[185] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, and Jacob Matthews. *Revised [6] Report on the Algorithmic Language Scheme*. Cambridge University Press, New York, NY, USA, 1st edition, 2010

[186] Guy L. Steele. *Common LISP: The Language*. Digital Press, Bedford, MA, 2. edition, 1990

[187] Volker Stolz and Frank Huch. Runtime verification of Concurrent Haskell programs. In *In Proceedings of the Fourth Workshop on Runtime Verification*, pages 201–216. Elsevier Science Publishers, 2004

[188] Sairam Subramanian. Parallel and Dynamic Shortest-Path Algorithms for Sparse Graphs. Technical report, Providence, RI, USA, 1995

[189] Michael Suess. What Makes Parallel Programming Hard? `http://www.thinkingparallel.com/2007/08/06/what-makes-parallel-programming-hard`

[190] Aater Suleman. What makes parallel programming hard? `http://www.futurechips.org/tips-for-power-coders/parallel-programming.html`

[191] Martin Sulzmann, Edmund S.L. Lam, and Simon Marlow. Comparing the performance of concurrent linked-list implementations in Haskell. In *DAMP '09: Proceedings of the 4th workshop on Declarative aspects of multicore programming*, pages 37–46, New York, NY, USA, 2008. ACM

[192] Håkan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65:609–627, 2005

[193] Michael Süß and Claudia Leopold. Common Mistakes in OpenMP and How to Avoid Them - A Collection of Best Practices. In Matthias S. Müller, Barbara M. Chapman, Bronis R. de Supinski, Allen D. Malony, and Michael Voss, editors, *IWOMP*, volume 4315 of *Lecture Notes in Computer Science*, pages 312–323. Springer, 2006

[194] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007

[195] Andrew S. Tanenbaum and James R. Goodman. *Structured Computer Organization*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition, 1998

[196] Donald E. Thomas and Philip R. Moorby. *The Verilog hardware description language (4th ed.)*. Kluwer Academic Publishers, Norwell, MA, USA, 1998

[197] P. W. Trinder, K. Hammond, W, S. L, and Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8:23–60, 1998

[198] D Turner. An overview of Miranda. *SIGPLAN Notices*, 21:158–166, December 1986

[199] Valgrind.org. Helgrind: a thread error detector.
`http://valgrind.org/docs/manual/hg-manual.html`

[200] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of ACM*, 33(8):103–111, 1990

[201] Leslie G Valiant. A bridging model for multi-core computing. *Journal of Computer and System Sciences*, 77(1):154–166, 2008

[202] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 214–222, New York, NY, USA, 1995. ACM

[203] Wikipedia. Instructions per Second.
`http://en.wikipedia.org/wiki/Instructions_per_second`

[204] Wikipedia. Lambda-Calculus.
`http://en.wikipedia.org/wiki/Lambda_calculus`

[205] Wikipedia. POSIX Threads.
`{http://en.wikipedia.org/wiki/POSIX_Threads}`

[206] Wikipedia.org. Compare-and-swap Instruction.
`http://en.wikipedia.org/wiki/Compare-and-swap`

[207] Gregory V. Wilson. *Parallel Programming Using C++*. MIT Press, Cambridge, MA, USA, 1996

[208] Andreas Wirz, Michael Süß, and Claudia Leopold. A Comparison of Task Pool Variants in OpenMP and a Proposal for a Solution to the Busy Waiting Problem. *International Workshop on OpenMP*, 2006

[209] X10. Homepage.
`http://x10-lang.org/`

[210] V. V. Zhirnov, R. K. Cavin, J. A. Hutchby, and G. I. Bourianoff. Limits to binary logic switch scaling-a gedanken model. *Proceedings of the IEEE*, 9(11):1934–1939, November 2003

Statement

Erklärung

Hiermit versichere ich, dass ich die vorliegende Dissertation selbstständig und ohne unerlaubte Hilfe angefertigt habe und andere als die in der Dissertation angegebenen Hilfsmittel nicht benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder unveröffentlichten Schriften entnommen sind, habe ich als solche kenntlich gemacht. Kein Teil dieser Arbeit ist in einem anderen Promotions- oder Habilitationsverfahren verwendet worden.

Kassel, 14. März, 2012 Michael Lesniak