MASTER'S THESIS

Ohua as an STM Alternative for Shared State Applications

Felix Wittwer

July 20, 2020
Version: Final Issue

MASTER'S THESIS

Ohua as an STM Alternative for Shared State Applications

Felix Wittwer

1. Reviewer	Prof. Dr.-Ing. Jeronimo Castrillon Chair for Compiler Construction TU Dresden
2. Reviewer	Dr.-Ing. Michael Roitzsch Chair for Operating Systems TU Dresden
Supervisor	Dr.-Ing. Sebastian Ertel

July 20, 2020

Felix Wittwer
Ohua as an STM Alternative for Shared State Applications
Compiler Construction, July 20, 2020
Reviewers: Prof. Dr.-Ing. Jeronimo Castrillon and Dr.-Ing. Michael Roitzsch
Supervisor: Dr.-Ing. Sebastian Ertel

TU Dresden
Chair for Compiler Construction
Institute of Computer Engineering
Department of Computer Science
01062 Dresden

Abstract

The efficient parallelization of shared state applications has been an ongoing research topic ever since the advent of multi-core processors. Due to the properties some of these programs exhibit, such as amorphous data parallelism and the use of pointer-based data structures, conventional concurrency control mechanisms like locking fail to uncover any meaningful parallelism. Speculative parallelism approaches have been developed as a result, the most prominent representative being *Software Transactional Memory*. But these frameworks have their own set of drawbacks like the lack of scalability and high overheads. It is unclear, whether approaches like implicit parallelism could be a feasible and more performant alternative to use for developing these applications.

This thesis tries to find out, whether *Ohua*, a framework for exploiting implicit parallelism, could be used for writing shared state programs and whether it could be an alternative to the by now aged STM. In order to do this, we tested in a preliminary study Ohua's usability in shared state environments. We then proposed a set of transformations for Ohua to run at compile-time to automatically recognize and exploit parallelism from such applications. To compare Ohua then against STM, we tested both in a series of benchmarks that ressemble real-world applications.

The evaluation shows that Ohua is not able to exploit parallelism in shared state applications to the same degree as STM does. Nonetheless, Ohua shows results that are widely on par with Software Transactional Memory and performs especially good in applications with amorphous data parallelism, validating our proposed transformations.

Contents

1 Introduction

At the beginning of the century, it became apparent that advancements in CPU architecture were hitting the power wall. Clock speeds of single cores could no longer be increased without also increasing the power dissipation of the unit beyond the capabilities of consumer-grade cooling solutions. Multi-core architectures and technologies like Intel's hyperthreading soon emerged as possible solutions to this problem, making exploiting parallelism a viable option to improve performance. However, this architectural change led to a new set of problems: As many programs operate on large data structures throughout the whole execution, their concurrent execution can create data races which can only be mitigated by using primitives that allow safe state sharing. The first conceptually simple solution to this problem that found wide adoption was synchronization using locks. By guarding sensitive blocks of code with a lock, one can ensure that only a single thread may enter such a critical section at a time. But this method of synchronization has a number of drawbacks. For one, locks do not compose [Lee06]. When using several locks or combining multiple libraries exposing locks, developers can easily produce deadlocks, situations in which program execution comes to a halt and is unable to proceed. Also, locking is a pessimistic parallelism approach as the concept behind it assumes that under no circumstances two or more threads may enter the same critical section if data races could occur. Yet, many real-world applications are based around pointer-based data structures. Exploiting parallelism from these irregular applications using locks is highly inefficient [Kul+09], as often several threads could manipulate the data structure at the same time without conflicting, e.g., when working on a large tree-based data structure. Instead, a more optimistic approach to parallelism is needed for this type of problem.

Software Transactional Memory [ST97] is a framework that allows state sharing without exposing low-level mechanisms like locks to the developer. Instead, the framework allows the definition of so-called atomic blocks, code sections in which all changes made to shared data are either written successfully or not at all, similar to database transactions. Using this framework for concurrency control yields so-called speculative or optimistic parallelism [Kul+07] as several threads execute transactions on the same shared data structure, assuming no conflicts will occur. Should a conflict be detected, the transaction is simply rolled back and retried. This framework has now been an integral tool for state sharing for a while, yet it has a number of problems that have been widely reported in research. These include a lack of scalability [Per+08] and too high framework overheads [Cas+08]. When we inspected and used STM code written as part of the widely used STAMP benchmark suite [Min+08], we also encountered issues: The code provided for some benchmarks does not always produce a correct result for the computation it implements [@Wit20b], serving as an example for how hard it is to correctly

write a program using Software Transactional Memory. Even worse, using STM actually makes memory management inside the application even harder, as several irregularly occurring memory management issues [@Wit20e] prove. Finally, even though Software Transactional Memory aimed to relieve the programmer of the burden to manage locks manually, the framework itself may introduce deadlocks into the application, as will be shown in this thesis. Although this may be traced back to a bug within the framework itself this shows impressively that the implementation of this framework is indeed not trivial and that proving its correctness is hard.

Ohua [EFF15], on the other hand, does not have these problems. Proposed by Ertel et al., the framework allows for the creation of implicitly parallel programs. Using a number of transformations, parallelism is extracted from an otherwise sequential piece of code. The resulting dataflow graph is then translated into a runtime which exploits the found parallelism. Both of these steps can be checked for correctness relatively easily, leaving only the sequential code correctness to the developer. Its underlying deterministic model additionally ensures that sporadically arising bugs as observed in the STAMP suite cannot occur. Unfortunately, Ohua achieves most of its guarantees because it only fosters local state to avoid the need for locking. Thus, it is incapable of handling shared state as of now.

In this thesis, we want to test the usability of Ohua in shared state applications. Therefore, we will look at the theoretical foundations of the framework to see, whether they allow an extension to shared state. In order to then properly evaluate Ohua in this context, we are going to compare its performance to the well-established shared state programming framework STM, to see whether it could indeed be a suitable alternative to use for this field of applications. We make the following contributions:

- Preliminary studies regarding the feasibility of applying Ohua to shared state scenarios where we tested, whether its theoretical foundations allow the introduction of parallelism in such applications.
- Descriptions and definitions of transformations for the Ohua compiler to run on input algorithms to enable it to automatically extract parallelism from shared state applications.
- A set of experiments where we applied Ohua to a number of such applications to test its performance in comparison to the STM framework.
- An evaluation of the question whether or not Ohua could be an alternative to the now-used STM framework for developing shared state applications.

The rest of this thesis is structured as follows: Firstly, we will introduce some basic notions and concepts necessary to understand this thesis as well as the motivation for possibly replacing STM in Chapter 2. Then, we will present our preliminary studies about whether or not Ohua's theoretical foundations allow shared state handling in Chapter 3. The resulting amendments that have to be made to the Ohua compiler will be defined in chapter 4 and applied manually to a set of benchmarks in

Chapter 5. Chapter 6 will then evaluate and interpret the results of the benchmarks. Related work on other possible STM replacements is presented in Chapter 7. The thesis will then close with future work presented in Chapter 8 before concluding in Chapter 9.

Background and Motivation

2

In this chapter, we are going to introduce both *Software Transactional Memory* and *Ohua* as a foundation for the following chapters. We will briefly present each frameworks' basic concepts as well as advantages and drawbacks to adduce for the following comparison. To better understand why STM's optimistic approach to parallelism gives it an advantage over other approaches in parallelizing shared state applications and to better understand their behavior, we will explain what causes this phenomenon. Therefore, we will introduce the class of *Irregular Applications* as well as the term *Amorphous Data Parallelism* which both describe important properties encountered in many shared state applications. We will also briefly discuss the difficulties that arise when reasoning about said application types and motivate, why further research into this topic could prove valuable.

2.1 Irregular Applications

In the past, most of the research conducted in the field of performance improvements for parallel programs has concerned itself with what is called regular applications. These are applications, whose degree of exploitable parallelism is simply determined by the program structure and the size of the input set. As both of there properties are known or can be inferred at compile time, any optimization potential is easy to uncover and exploit for compilers.

However, most applications that share state in some form, which are the main focus in this thesis, are *irregular applications*. This term describes applications, whose structure revolves around the manipulation of large, pointer-based data structures like graphs and trees. Due to this structural peculiarity, compiler analyses struggle to uncover any meaningful insights into the algorithm that would allow to exploit any parallelism hidden in it.

To better outline this, Kulkarni et al. [Kul+09] developed an abstract representation for irregular programs using a set of nodes and edges, as shown in Fig. 2.1. The nodes and edges represent the input data and their relationships. During the execution of an irregular algorithm, the program may perform computations on a (sub-)set of active nodes or edges, the work items. In Figure 2.1, these are highlighted in red. Performing said executions may involve reading from or writing to other nodes or edges in the graph, which form an active elements' neighborhood, shaded in blue. Which elements of the graph make up the neighborhood of an active element is not known beforehand and may encompass all direct neighbors in the graph (as seen for nodes n_2 and n_3), maybe only a single neighboring element (as seen for node n_4) or transitively the neighboring elements of its direct neighbors.

Usually, work items are not ordered and may be processed in any order. But processing one element may generate an arbitrary number of new active elements or remove others from the pool, depending on the algorithm. Also, some active elements may not be processed simultaneously due to overlapping neighborhoods, as the changes made by processing both items may conflict with each other.

Since these relationships and interdependencies are not known beforehand and depend on the input data, this information cannot be uncovered by compile time analyses. Yet, most shared state applications are irregular by design, making this a large and interesting field of research. Because static parallelization fails to uncover any viable parallelism opportunities in these programs, Kulkarni et al. [Kul+07] argue, that optimistic strategies need to be used to tackle this class of problems.

2.1.1 Amorphous Data Parallelism

Pingali et al. [Pin+09] argue, that most irregular applications additionally exhibit a behavior referred to as *amorphous data parallelism*:

Given a set of active nodes and an ordering on them, amorphous data parallelism is the parallelism that arises from simultaneously processing active nodes and is subject to neighborhood and ordering constraints. It is a generalization of standard data parallelism in which

1. concurrent operations may conflict with each other
2. activities can be created dynamically
3. activities may modify the underlying data structure

These characteristics have already been briefly discussed in connection with Fig. 2.1.

While irregular applications are hard to reason about due to their pointer usage, they are still a very interesting research topic, made more difficult by amorphous data parallelism. But as we will show in Chapter 5, applications of this type are common and widespread in different aspects of real world applications using shared state.

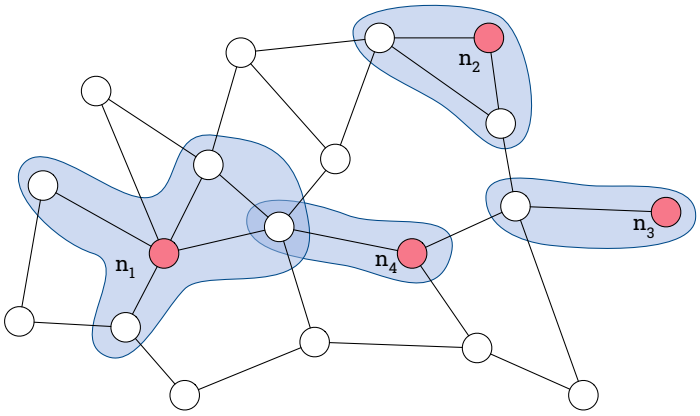


Fig. 2.1: Graph representation of an irregular application with active elements and their neighborhoods. Adapted from Kulkarni et al. [Kul+09]

2.2 Software Transactional Memory

With the invention of multi-threaded programming, a need for synchronization primitives arose to allow safe parallelization of data-processing applications. Usually, locks have been used by developers to guard access to pieces of shared data.

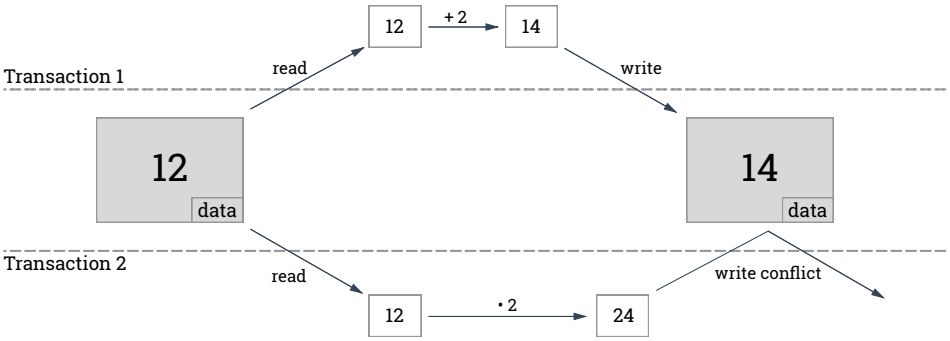
Lock-based programming however, has the fundamental drawback of easily producing deadlocks (i.e., a state where a number of threads may never again progress) when acquiring locks in the wrong order, forcing developers to employ great care when using them [Lee06]. This results in a lack of composability, as combining several small lock-based modules into a larger program would require the imposition of some sort of ordering on the locks, which frequently leads to problems. Even when written correctly, lock-based programs tend to quickly become hard to read and maintain due to the many rules that need to be enforced by the programmer herself without any external checks. Another problem is lock contention. If a lock is currently taken, all other threads seeking to acquire it have to wait. This opens up a new set of problems, as frequent waiting leads to prolonged execution times. Mitigating contention issues forces developers to consider the trade-off between the overhead introduced by many fine-grained locks, resulting in a higher chance of an accidental deadlock, and high contention.

Additionally, locking follows a pessimistic approach to parallelism, which is completely unsuitable for uncovering any viable parallelism in irregular applications. This is due to the fact that, as we have shown in the previous section, the heavy use of pointers in these programs hinders the efficient fine-grain use of locks, forcing developers to employ a coarse-grain locking strategy. But at the same time, conflicts due to parallelization in these programs may be rare, depending on the input data. It is possible to work on a large graph data structure in parallel with multiple threads without conflicts, when all threads work on non-overlapping portions of the program. In the example from Figure 2.1 for instance, the nodes n_1, n_2 and n_3 could be processed in parallel without conflicts. Locking however is unable to provide this type of speculative approach to parallel execution necessary to efficiently parallelize these applications.

Therefore, Shavit et al. [ST97] proposed a concept called *Software Transactional Memory*. This new approach to synchronization aimed to provide lock-free parallelism abstractions that allow multiple threads access to a shared variable without any form of blocking. Former critical sections are now regarded as *transactions*, which operate similar to their namesake in databases: They ensure atomicity, consistency and isolation for the code blocks they protect, only falling short in the durability databases offer, as this is not required for data residing in memory. Each transaction or atomic block, as they may also be referred to, ensures that the changes made in the guarded code block take all effect at once, eliminating possible race conditions. STM's operating principle is outlined in Figure 2.2 along with a code example in Fig. 2.2a. Every read and write operation to or from a piece of shared data is conducted inside a transaction block and gets initially saved to a local transaction log, ensuring each transaction runs isolated. When the transaction block

```rust
let data = TVar::new(12);   // data is a transaction variable
thread::spawn(move || {
    atomically(|trans| {    // closure for transaction blocks
        let local = data.read(trans)?;
        data.write(trans, local + 2)
    })});
thread::spawn(move || {
    atomically(|trans| {
        let local = data.read(trans)?;
        data.write(trans, local * 2)
    })});
```

(a) Rust code for both transactions.



(b) Sample execution of two parallel transactions where one transaction has to be rolled back due to a write conflict.

Fig. 2.2: Example of two transactions modifying the same shared value in different transactions.

comes to an end, the changes made to individual shared data sections are committed. Therefore, the changes in the log are applied to the original shared values. In case another transaction managed to commit in the meantime to a value our transaction also touched, the conflict is detected and instead of committing the changes the transaction is aborted and restarted. This is normally done until the transaction committed successfully.

In our example in Fig. 2.2, the transactions 1 and 2 both read the same value and modify it. But since the first transaction was able to commit earlier, the second transaction now stands in conflict and has to be aborted and scheduled for re-execution.

As the example already shows, a fundamental benefit of the Software Transactional Memory model is that it retains serializability [SDD16]: All transactions seem to execute serially, since the steps of one transaction never appear to be interleaved with the steps of another transaction. Therefore, the results of an execution must be equal to the result of a serial execution. This has been formalized by Swalens et al. [SDD16] in their proposed operational semantics for a language with transactions.

Overall, STM takes an optimistic approach [Kul+07] to parallelism, because it can be used to speculatively execute numerous computations in parallel, hoping for as few conflicts as possible. This is beneficial to irregular applications, as parts of these applications can only be efficiently parallelized using an optimistic approach. The result of this is an underlying non-determinism that ensues everytime transactions are employed in a multi-threaded environment. Problems in this strategy become apparent when applied to high-contention scenarios. Since STM may work in parallel over the same data structure or memory region, high contention always leads to a significantly increased number of retries for individual executions, which in turn leads to drastically reduced performance. Further shortcomings of this concept have been discussed in detail by Caşcaval et al. [Cas+08]: On the one hand, exception handling becomes impossible to do inside of a transaction without breaking its semantics. On the other hand, I/O operations cannot be transactionalized, as well as anything else that produces side effects outside of the transactions' scope as these effects may not be rolled back on error. Additionally, they reported large overheads of STM applications for smaller worksets as well as no debuggability, since the non-determinism makes a specific situation nearly irreproducible.

2.3 Ohua

As we have seen before, most solutions to state sharing in parallel programs offer developers certain abstractions or data structures such as locks or transactions for them to use when developing parallel programs. However, this forces programmers to think within the boundaries of their chosen tools and produces code that is tailored to a specific framework and therefore hard to migrate.

A completely different approach has been taken by Ertel et al. [Ert19] with the proposition of *Ohua*, which is a framework that allows for the development of implicitly parallel programs. It consists of three main components, as outlined in Fig. 2.3: *Algorithms*, which are used to describe the program part that shall be parallelized, a *compiler* that parses the algorithm and produces a parallelized version of it combined with a *runtime* that will execute the parallel code.

An Ohua algorithm is what might come closest to the abstractions and tools used by other paradigms. It describes the part of a program that should be parallelized.

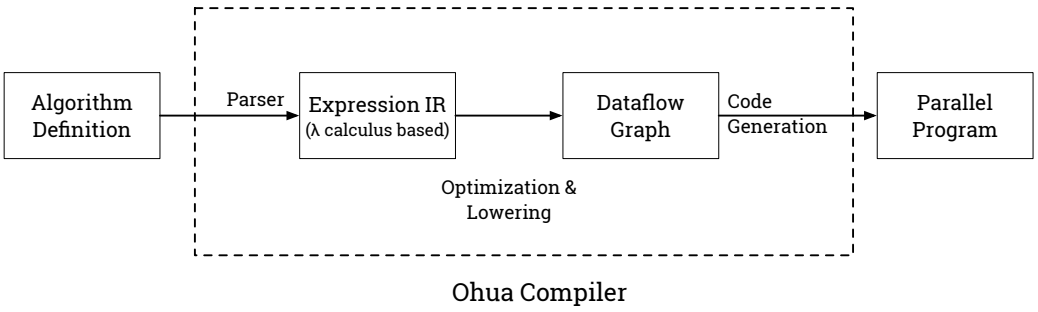


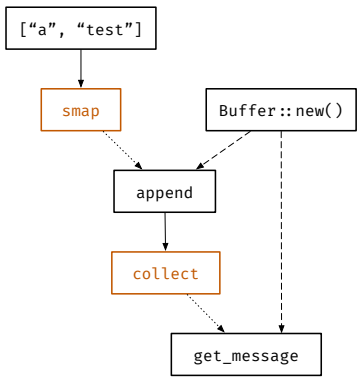
Fig. 2.3: Overview of the Ohua compiler and its components

In early versions [EFF15] these algorithms were written in separate files using a high-level language that closely resembled Lisp and OCaml. Over time, a Rust-like syntax [Ada19] had also been added to allow for more imperative algorithm definitions. By now, algorithms are not separately defined anymore and instead are functions that are written in a host language which is supported by Ohua, such as Rust and Go. Ohua comes with a standalone compiler called `ohuac`, which proceeds to parse the algorithm into an Expression IR, optimizes it and eventually transforms it into an optimized dataflow graph. These optimizations stem from a number of transformations that are applied to uncover more parallelism opportunities. A host language specific backend will then generate the parallel program together with the runtime code based on the DFG. This runtime executes as regular part of the program it is embedded in when the algorithm is called. As such, it may import and use any function from the host code, allowing the developer seamless integration of her existing code into the new parallel algorithm.

Ohua derives parallelism from executing independent nodes of the dataflow graph in parallel (task-level parallelism) and running multiple invocations of dependent nodes in a pipeline-parallel fashion. In order to rule out most of the common pitfalls in parallel programming, the use of shared state is forbidden in Ohua. Instead, data must be transferred explicitly by using arguments and function return values. This also rules out mutable global values which often represent state in programs. To still allow the convenient declaration of stateful algorithms, Ohua uses the concept of so-called *stateful functions* [Ert+19]. A stateful function is a host function that is associated with local state values which it is allowed to modify. Ohua encapsulates this operation and the associated state to ensure it is not leaked or used in a shared fashion.

```
1 let words = vec!["a", "test"];
2 let buffer = Buffer::new();
3 let _ = for w in words {
4     buffer.append(w);
5 };
6 buffer.get_message();
```

(a) Algorithm declaration in Rust



(b) The algorithm's dataflow graph. Dependencies in the control flow are dotted, state dependencies dashed.

Fig. 2.4: Example of an Ohua algorithm using the `smap` primitive. Adaption of [Ada19].

Ertel et al. additionally proposed `smap`, a stateful map primitive that applies an algorithm to a sequence of items [Ert+19] and is based on the insight, that loop operations modifying shared state can be considered as fold operations on the state. An example algorithm in Rust that outlines this is shown in Fig. 2.4. Functions highlighted in orange are builtin functions of the Ohua runtime, all other nodes

are host functions provided by the developer. In our example, the `smap` primitive is represented by the for loop and its use of the `buffer` variable in the body. The loop body is applied to a sequence of items, just as it would happen in a normal `map` statement. But additionally, `smap` ensures that changes made to the encapsulated state persist, as they would in a normal `for` loop. This allows the use of pipeline parallelism for longer loop bodies and – if no state is used in the loop at all – the use of data parallelism. So in Fig. 2.4, the newly initialized buffer is bound to the `append` function, which is called like a method on the state repeatedly for the individual loop iterations. Any results which would be produced by this loop, are then collected and upon completion of the loop passed on to the next node in the dataflow graph. In the small code snippet provided, an iteration simply produces a `()` literal[1] and hence the loop result itself is discarded. What remains, is the side effect of state alteration that occured during the loop processing. After the loop completes, the next function is executed. To avoid `get_message` being called too early, a control flow dependency is added to the (otherwise useless) `collect` function.

The result of Ohua's programming model is *deterministic parallelism*, as opposed to Software Transactional Memory which aims to improve execution times by scheduling possibly conflicting operations for parallel execution, completely forfeiting any determinism in the process. Another advantage is the way in which Ohua is integrated into existing code bases. In its current version, algorithms are defined as normal functions within the code, allowing developers to first test their implementations without any parallelism before compiling them with Ohua. This offers the advantage of producing framework agnostic code which can be easily reused in other applications as well.

Challenging however, is the ban the framework places on the use of shared state. Many applications encountered in the real world rely on the use of pointers and shared, global state. Since Ohua currently only fosters the use of local state with its stateful functions primitives, it is unclear whether it is possible to apply these concepts to shared state programs, which the Software Transactional Memory framework caters towards by design, and how good Ohua would perform compared to speculative parallelism which it itself does not exhibit.

[1] The `()` type is called Unit literal in Rust. It is the result of a computation that returns nothing and can be compared to the `void` type in C.

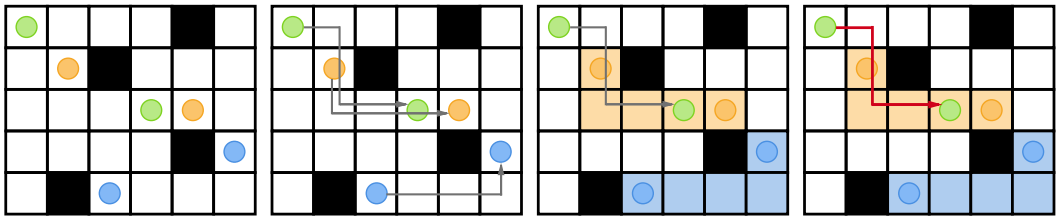
3 Preliminary Studies

In order to determine the usability of Ohua for implementing shared state applications and to identify any necessary modifications to the compiler or the runtime, we first tried to implement a single such application with it. After successful implementation we wanted to compare its performance against a Software Transactional Memory implementation of the same program and see, whether there are any shortcomings of Ohua in terms of performance. Based on this initial study we then wanted to find ways to improve Ohua's performance, if necessary, e.g., by introducing new compiler optimizations. The aim of these preliminary studies was to find out, whether Ohua was usable for writing programs relying on shared state and if it could be a viable alternative to STM in this setting.

3.1 Labyrinth Benchmark

As a first example application, we chose to implement the *labyrinth* benchmark as presented by Swalens et al. [SDD16]. This program is a variation of Lee's Algorithm [Lee61] which solves path-connection problems often encountered when searching for an optimal route or generating wiring diagrams where wires may not overlap. We base our own implementation on the descriptions of Watson et al. [WKL07], who presented an implementation for Transactional Memory.

Goal of the benchmark is to find a number of paths within a three-dimensional maze, as depicted in Fig. 3.1. As input, the algorithm is provided with a maze and a set of pairs of points, between which a path is to be found and mapped within the maze. During execution, one pair of coordinates is removed from the list of points (the worklist) and the program attempts to find a path between both points in the maze. This is done using a breadth-first search. The maze itself may also contain



(a) Initial grid with 3 point pairs.

(b) Possible paths between the points.

(c) First two paths are mapped into the maze.

(d) Conflict for the third path as it goes through a now-occupied segment.

Fig. 3.1: Illustration of the operation of the labyrinth benchmark, showing the (attempted) mapping of 3 paths in a 6×5 two-dimensional grid. Black squares represent walls that cannot be routed through.

```
1  let (worklist, grid) = /* read from file */
2
3  for points in worklist {
4      atomically(|trans| {
5          let local_grid = create_working_copy(&grid);
6
7          if let Some(path) = find_path(points, &local_grid) {
8              // if path is found, write back results
9              update_grid(&grid, &path, trans)?;
10         }
11
12         Ok(())
13     });
14 }
```

Listing 3.1: Simple implementation of the labyrinth benchmarks using Software Transactional Memory in Rust.

„walls“ (highlighted as black squares in Fig. 3.1), through which no path can be routed. Also, each point in the maze may only be occupied by a single path to rule out overlapping paths. The algorithm terminates once the worklist is empty and all paths have either been mapped successfully or deemed unmappable by the absence of a valid connection between both points.

This benchmark is a classic example for an *irregular application*: all operations happen on the shared maze data structure which is modified in the process. Additionally, any data parallelism obtainable in the application is *amorphous*, as mapping one path in the maze may render another pair of coordinates unroutable as both points get cut off from one another. An example for this can be seen in Fig. 3.1d, where the mapping of the orange path makes finding a path between the green coordinates impossible as one point is part of the orange path. As we have learned in Chapter 2.1, this specific class of problems can be parallelized easily using Software Transactional Memory. Searching for a single path can be compartmented into a transaction, treating all maze fields as transaction variables. Listing 3.1 shows the resulting transactional implementation of the labyrinth benchmark using the `rust-stm` library [@Ber20].

All path pairs are collected in a worklist, through which the algorithm iterates (line 3). Inside the transaction that is started for each item (line 4), a working copy of the maze is created as detailed in [SDD16] to reduce the number of repeated reads from individual transaction variables. Then, the breadth-first search commences in order to discover a route between the starting point and the target (line 7). Note that since we've made a copy of the grid beforehand, this happens completely locally. When a path is found, an update is run on the grid, inserting the path (line 9). Should another transaction, which also happened to alter one or more segments of the newly-found path, manage to commit in the meantime, the resulting conflict is detected and the transaction rolled back and re-run until either the update commits successfully or no path can be found anymore. Our transactional memory implementation is a mere adaption of the algorithm outlined above, augmented with

```
1  fn fill(maze: Maze, to_map: Vec<(Point, Point)>) -> Maze {
2      let paths = for pair in to_map {
3          find_path(maze, pair)
4      };
5
6      let (remap_paths, new_maze) = update_maze(maze, paths);
7
8      // recursively call `fill` as necessary
9  }
```

Listing 3.2: Simplified first implementation of a recursive Ohua algorithm for the labyrinth benchmark

concurrency by splitting the worklist into n parts, which are processed by n threads in parallel.

In our first Ohua implementation, we described idiomatically, what the algorithm should be doing. Listing 3.2 shows this simple program: First, all paths are searched for individually (lines 2-4), before they are written to the grid (line 6). If a path conflicts with a previously saved one (i.e., at least one segment of the path is not free anymore), it is scheduled for re-computation by adding it to the `remap_paths` vector. Until all paths have either been mapped or discarded as unroutable, these steps are repeated recursively, although this invocation has been omitted from the sample code for the sake of simplicity.

This implementation resembles an executable version of an Ohua algorithm that did compile and run on the initial Ohua compiler framework with Rust support[1] [@EWA19]. It deviated from the simple sequential implementation as well as STM's approach. But separating the update to the shared state from the search for paths was a deliberate choice to uncover parallelism opportunities for Ohua to begin with. We felt that this was a fair choice to make as it still describes the same algorithm as before, just slightly altered to match Ohua's current abilities and philosophy of fostering local state, as introduced in Chapter 2.3.

3.1.1 First Results

To establish a baseline for performance comparison, we measured the execution time of both benchmarks and calculated the speedup in relation to a sequential implementation, as explained in greater detail in Chapter 5.3. In an attempt to achieve comparable results, we used the same input data that had also been used previously by other authors and was originally proposed by Minh et al. [Min+08]. The chosen input maze for our test run had a size of $128 \times 128 \times 5$ cells and required the mapping of 128 paths, given as predefined coordinate pairs, into it. Resulting speedup figures for a varying number of worklist splits are shown in Fig. 3.2.

[1] The Ohua version that fully integrates with the host language as described in Chapter 2.3 did not exist when these preliminary studies were conducted and has been developed independently in parallel to this thesis.

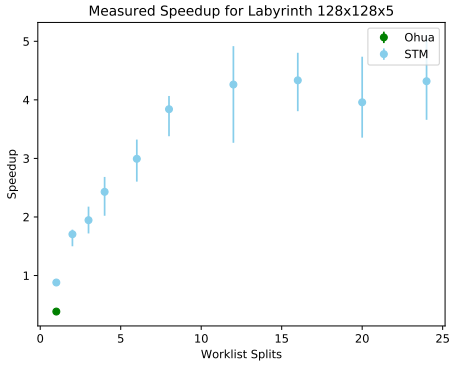


Fig. 3.2: Measured speedups of the labyrinth application for the STM and Ohua implementations.

As one can see in the plot, Software Transactional Memory is able to achieve a logistic growth in its speedup for 1 to 24 worklist splits, which converges at a mean maximum speedup of about 4.3. Ohua on the other hand recorded only a single measuring point for a run with no worklist splitting at all. This single run exhibited a speedup of less than 0.5, meaning it took more than twice as long to complete as the sequential reference implementation. The reason for this is the absence of any configuration options for e.g. worklist splits. Hence, there is no data parallelism in the Ohua algorithm, as the Rust runtime does not support parallel loops yet. Effectively, this single measurement shows the performance of a sequential algorithm executed with the Ohua runtime, revealing the overhead it has compared to a normal sequential implementation. Most of the overhead stems from the spawning of the threads each operator lives in and the resulting management and movement of data between these threads. Our hope is to show that these overheads amortise with increased data parallelism.

In order to achieve more data parallelism and an overall improved performance, we were looking for modifications that could be made to the Ohua algorithm to accomplish this. We first wanted to introduce manual changes to the algorithm without making amendments to the compiler to test different approaches easily. These manual changes simply emulate the changes a compiler pass would make on the algorithm layer. After identifying the optimizations that indeed provide a performance boost, we generalized them into compiler optimizations applicable to all algorithms. The resulting transformations will be discussed in Chapter 4.

3.2 Parallel Loop Implementation

As Fig. 3.2 clearly showed, the most pressing issue with the existing Ohua implementation for Rust was the absence of any data parallelism facilities like parallel loops. Our first assumption based on these results was, that most performance could be gained simply by processing multiple labyrinth paths in parallel, as this operation makes up for most of the execution time in the sequential implementation. The idea was to split the worklist into n equally sized chunks and process them

```
1  fn fill(maze: Maze, to_map: Vec<(Point, Point)>) -> Maze {
2      let (tm0, tm1) = splitup(to_map);
3      let part0 = for pair in tm0 {
4          find_path(maze, pair)
5      };
6      let part1 = for pair in tm1 {
7          find_path(maze, pair)
8      };
9
10     let paths = join(part0, part1);
11
12     let (remap_paths, new_maze) = update_maze(maze, paths);
13
14     // recursively call `fill` as necessary
15 }
```

Listing 3.3: Labyrinth implementation in Ohua using worklist splits for parallelism. Highlighted parts have been added in this iteration.

in parallel, which is similar to what STM does. Listing 3.3 shows an exemplary implementation for an algorithm splitting the worklist in two parts. Lines that were changed compared to the previous iteration are highlighted.

The main difference to the previous iteration of the algorithm is the introduction of two new operators, `splitup` and `join`. These are implemented in the user space for this example but would become part of the runtime when this alteration is translated into a compiler transformation. Their task is simply to split the worklist up into the required number of parts and merge them back together, once results have been produced. Data parallelism is introduced in a very verbose manner by the duplication of the loop (lines 3-8). This change can be scaled for an arbitrary number of threads by partitioning the worklist into as many parts as necessary for testing purposes.

Introducing a rudimentary form of data parallelism helped not only to obtain multiple meaningful measurements for Ohua, but also led to better performance, as can be seen in Fig. 3.3. It now also exhibits a performance growth behavior that can roughly be described as logistic, though it only manages to achieve half as much speedup as STM. Upon investigating this performance gap we determined two fundamental problems resulting from our current algorithm: Retries and Stragglers.

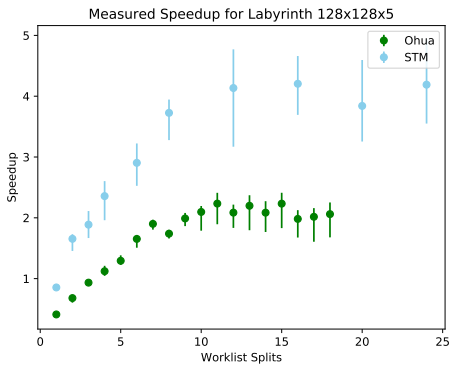


Fig. 3.3: Measured speedup for an Ohua implementation using worklist splits.

Besides data parallelism, another important factor influencing the execution time of the benchmark[2] is the number of repeated calculations that is necessary. As described before, the labyrinth benchmark is designed to repeatedly attempt to find a route between each coordinate pair until a path is either successfully mapped or no valid path can be found anymore. In the sequential implementation, this is not an issue as each path is searched for and updated individually without any concurrency, guaranteeing that each coordinate pair will only ever be evaluated once. Any concurrent implementation however, comes at the cost of potential write conflicts that need to be resolved. STM for example uses n threads, always working on finding n paths in parallel which are then written to the maze. This means that from the perspective of one thread, in between its last access to the maze and an attempted update, about $n - 1$ changes will ideally[3] have been made to the maze, each possibly introducing a conflict provoking a recomputation of the given path. So at worst, per iteration of n threads, $n - 1$ results may become invalid due to a write conflict, forcing recomputation.
For Ohua, this number is significantly higher. Its current approach is to update the shared state as late as possible, after computing all paths. The negative side effect of this is that in a run to map p paths, $p - 1$ results may require a recomputation per recursion step in the worst case, possibly leading to as much as $\sum_{0 \leq i < p} i$ conflicts, which may in part explain the bad performance of Ohua.

The second relevant performance bottleneck is the straggler problem. In research, when reasoning about parallel tasks it is often assumed that all tasks perform uniformly, i.e., require the same time to complete. Real applications however rarely fulfill this assumption. On the contrary, the tasks in these worklists are often wildly heterogeneous, each requiring a different amount of processing time. This is also the case in the labyrinth benchmark. The processing time for a single path depends solely on the number of nodes the path finder has to inspect which is related to the distance between the starting coordinates and the target. As result, the n threads of our algorithm each take differently long to process their worklist, which means that all threads finishing their work earlier have to wait at the synchronization point for the slower threads to finish. This forced slack time is also present in the STM algorithm implementation, yet not as severe as in the Ohua implementation, since it synchronizes all threads only once when terminating them, while Ohua's threads synchronize once per recursion step.

3.3 Lowering the Retry Count

Initially, we abandoned the idea of updating the maze after every state update in favor of a single update after processing all elements, to attempt solving the problem using only local state and a simple algorithm structure that still offers parallelism opportunities. Alas, these frequent updates are key to keep the execution time low because the quick propagation of changes ensures that fewer computations are

[2] In fact, this holds true for any irregular application following this *Calculate-Update Pattern*.
[3] Assuming that all theads proceed equally fast.

```
1  fn fill(maze: Maze, to_map: Vec<(Point, Point)>, frequency: usize) -> Maze {
2      let (points, still_to_map) = take_n(to_map, frequency);
3      let (tm0, tm1) = splitup(points);
4      let part0 = for pair in tm0 {
5          find_path(maze, pair)
6      };
7      let part1 = for pair in tm1 {
8          find_path(maze, pair)
9      };
10
11     let paths = join(part0, part1);
12
13     let (remap_paths, new_maze) = update_maze(maze, paths);
14     let to_remap = join(remap_paths, still_to_map);
15
16     // recursively call `fill` as necessary
17 }
```

Listing 3.4: Ohua algorithm using a fixed update frequency to lower the number of write conflicts. Highlighted parts have been newly added in this iteration.

carried out based on outdated information. Therefore, we altered the algorithm such that updates to the shared state are conducted after processing a fixed number of elements, which we will refer to as the *update frequency* f. In our algorithm, which is shown in Listing 3.4, this is reflected by introducing a new operator, `take_n`. Making use of the existing recursion semantics, it caps the number of elements to process per step to at most f elements (line 2). As before, all computations in a single step are conducted on the same state and updated all at once after finishing mapping the paths. Once the updates have been written to the maze or rejected due to a collision, the list of failed updates gets merged again with the rest of the worklist that has not been processed in this recursion step (line 14). Using this approach, the number of possibly denied state updates is dramatically reduced to only maximally $f - 1$ elements. With decreasing values for f, the overall probability of a write conflict is also reduced.

In addition to the number of worklist splits, this change introduced the *frequency* f as second parameter to the algorithm, which we wanted to fix to a specific value to

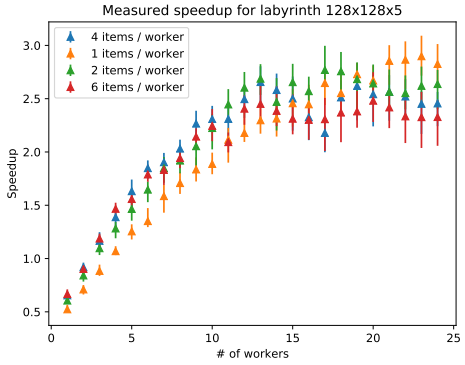


Fig. 3.4: Performance of various configurations of the Ohua-frequency algorithm.

find a middle ground between processing as many elements as possible and keeping the number of retries low. Small update frequencies imply fewer conflicts, but also more recursions and hence more negative side effects from stragglers. Choosing a too large value for f on the other hand might improve the parallel performance but comes at the cost of more repeated calculations. We tested our implementation with varying update frequencies. The results are shown in Fig. 3.4, which shows the performances for Ohua algorithms using a update frequency of 1, 2, 4, or 6 times the thread count, meaning that each thread will receive 1, 2, 4, or 6 work items. As the plot shows, it is better to keep the update frequency low, as we have speculated before. Best performance for larger thread counts is achieved by $f = threadcount$, although this run exhibited the worst speedup for smaller thread counts, probably due to the overhead we discussed as a potential limiting factor. Hence, we opted for using a frequency of $f = 2 \cdot threadcount$ as this configuration continuously shows an average to top performance in the pool of examined frequencies and we believe it provides the best trade-off between update conflicts and straggler performance penalties for the current benchmark.

3.4 Improving Resource Utilization

Stragglers are a common problem in applications processing data in parallel. Hence, a lot of solutions have been proposed already to tackle this problem using different techniques. One solution to this problem are work-stealing scheduling strategies, which have been discussed as early as 1981 by Burton and Sleep [BS81] and later by Halstead [Hal84]. The basic idea of a traditional work-stealing scheduler is that each processor in a computer system is assigned a set of work items to process. Each item consists of an isolated stream of instructions that is executed, possibly spawning new items in the process. Work items are unordered and may be processed in any order and in parallel. Should a processor run out of work, it can „steal“ work items from other queues to avoid idle time. This concept has also been implemented in software, usually providing a runtime consisting of a thread pool, a scheduler and a set of tasks. Fig. 3.5 sketches how this approach could reduce the slack time for all threads in a system by stealing work from the longest-running thread. By reducing the time all threads spend idling, the overall execution time and resource utilization improve.

To mitigate our straggler problem, we decided to move all data-parallel processing to a work stealing runtime. Rust's ecosystem offers multiple well-matured runtimes for this purpose. We chose to use *tokio* [@con20] as it was the most popular runtime at the time we implemented this, but we kept the code mostly library-agnostic to make a later switch in libraries as easy as possible.

Listing 3.5 shows the Ohua algorithm after adding the work-stealing runtime. Now, its setup in the algorithm `run_algo` (line 2) forms the initial step before running any part of the algorithm itself. This runtime is then reused throughout all recursion steps of `fill` to keep the added overhead low. Similar to previous iterations, f items are first taken from the worklist and then split for processing (lines 8-10). These

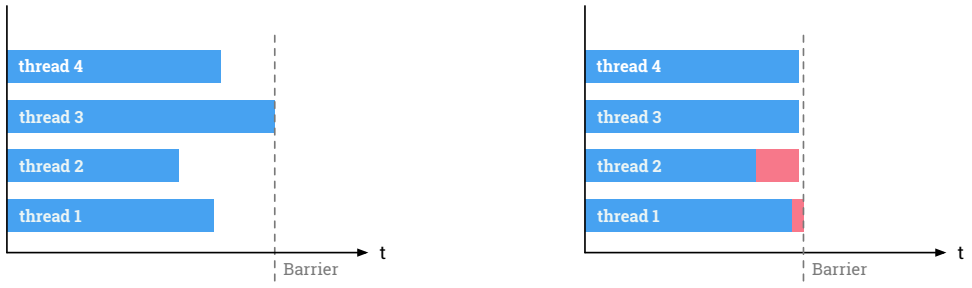


Fig. 3.5: Illustration of the straggler problem and how work-stealing scheduling can significantly reduce this problem. In the second illustration, the slack time is almost completely removed by threads 1 and 2 stealing work from thread 3.

```
1  fn run_algo(maze: Maze, to_map: Vec<(Point, Point)>, frequency: usize, threadcount:
   ↪  usize) -> Maze {
2      let rt = create_runtime(threadcount);
3
4      fill(maze, to_map, frequency, threadcount, rt)
5  }
6
7  fn fill(maze: Maze, to_map: Vec<(Point, Point)>, frequency: usize, threadcount:
   ↪  usize, rt: Arc<Runtime>) -> Maze {
8      let (points, still_to_map) = take_n(to_map, frequency);
9
10     let worklist = split_evenly(points, taskcount);
11     let task_handles = spawn_onto_pool(worklist, maze, rt);
12     let paths = collect_work(task_handles);
13
14     let (remap_paths, new_maze) = update_maze(maze, paths);
15     let to_remap = join(remap_paths, still_to_map);
16
17     // recursively call `fill` as necessary
18 }
```

Listing 3.5: Ohua algorithm using a work-stealing runtime to schedule its tasks. Highlighted sections have been altered or added in the current iteration.

work sets are then scheduled as tasks for execution on the threadpool (line 11). Following the findings of Ousterhout et al. [Ous+13] we make individual tasks as small as possible, each consisting of only a single coordinate pair for optimal load balancing between all threads. After collecting the results from the runtime (line 12), the algorithm will proceed as in previous versions.

In employing this runtime, we hoped to reduce the slack time seen in our first parallel implementation by moving work away from the longer-running threads to ones that finished quicker. Figure 3.6 confirms this. Ohua now manages to achieve a threefold speedup compared to a sequential implementation of the benchmark, closing in on STM's performance. It also benefits from its deterministic execution model, exhibiting a lower variation in the measured values than STM. The higher variance in STM's results is mainly due to the non-deterministic execution model, leading to a varying commit order for the mapped paths with each execution. This influences, which paths can be mapped successfully in a single run and how many write conflicts

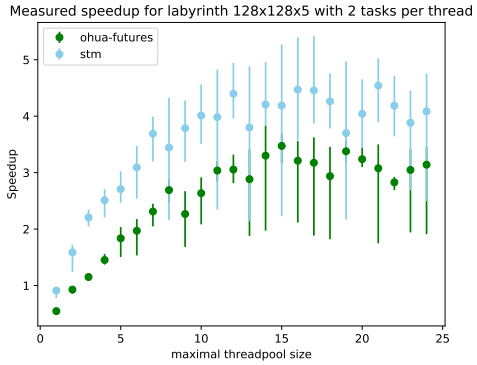


Fig. 3.6: Measured speedup for an Ohua implementation using worklist splits, an update frequency of $2 \cdot threadcount$ and a work-stealing runtime.

occur during execution which in turn has a direct effect on the predictability of the execution time. Ohua on the other hand is more deterministic. All paths are mapped in parallel, but they are merged in the same order they were in before being processed, yielding reproducible results that can be easily debugged and do not underlie a similarly high deviation. Due to the scattering of STM's results, one can not with definitive certainty say whether it indeed outperforms Ohua for every thread count value as the variances sometimes overlap, e.g., for 11, 13 and 15 threads.

All in all, these preliminary studies showed that in irregular applications, Ohua can achieve performances similar to Software Transactional Memory if the right transformations are applied to the algorithm. This warrants further investigation by trying to find generalized dataflow graph transformations which can be applied to Ohua algorithms in order to boost their performance at runtime.

Compiler Transformations

4

After discussing possible optimizations to the labyrinth algorithm in Chapter 3, we will now attempt to generalize the changes we made into transformations to be applied at compile time. The resulting transformations will be formally described and their correctness discussed in this chapter, using an Expression IR that is based on the lambda calculus. An implementation of the presented transformations will be left for future work.

4.1 Expression IR Definition

During compilation, the Ohua compiler framework parses algorithms provided as inputs into an Expression IR on which it then runs a number of optimizations [Ert+18], as we have shown in Figure 2.3 in Chapter 2.3. We are going to describe our transformations in this intermediate representation, which we will therefore present now briefly. The Expression IR we use is based on the call-by-need lambda calculus [AF97; Ari+95] which prevents duplicated computations. Figure 4.1 defines our expression language, which is building atop the language used in previous research on Ohua optimizations [Ert+18] by Ertel et al. The language defines the basic terms of the call-by-need lambda calculus for variables, abstractions, application and lexical scoping. We additionally define conditionals and fixed-point combinators to realize recursive expressions as well as free and state-modifying foreign functions. Using the combinator ff_f, one can express the application of a function f which is not defined as part of the calculus to an arbitrary number of arguments. This allows us to integrate code written in other languages like Rust into the algorithm, which is a key premise for Ohua's concept as Embedded DSL. Furthermore, we expand the definitions used in previous work by adding the combinator sf_f, which applies a method f to a state value s and an arbitrary number of additional values. We made this addition in order to model the state manipulations usually found in shared state applications. Methods that are executed on a state value may alter it but are usually also allowed to return another value (e.g., when reading from a piece of state). This behavior is reflected in the `sf` combinator producing a list with two values as result, where the first value is the the altered state value and the second value is the ordinary value produced by the function f.

In order to complete the inclusion of legacy code into the Expression IR, values may not only be abstractions or lists of values but also values in V_h, the value domain of the host language. Aside from recursion, we also define the well-known higher-order function `nth` to retrieve a particular element from a list of values. The function `map` applies a term to a list of values. Depending on whether the term applied contains a stateful function or not, the definition of the function differs as state updates need to be applied during loop execution. Both the definition of the `sf` combinator and the

Terms:

t	$::=$	x	variable
	$\mid$	$\lambda x.t$	abstraction
	$\mid$	$t\ t$	application
	$\mid$	$\texttt{let}\ x = t\ \texttt{in}\ t$	lexical scope (variable binding)
	$\mid$	$\texttt{if}(t\ t\ t)$	conditionals
	$\mid$	$\texttt{let}\ f = \lambda x.t\ \texttt{in}\ t$	fixed-point combinator
	$\mid$	$\texttt{ff}_f(x_1 \ldots x_n)$	apply the free foreign function f to $x_1 \ldots x_n$ with $n \geq 0$.
	$\mid$	$\texttt{sf}_f(s\ x_1 \ldots x_n)$	apply the state-modifying foreign function f to state s and $x_1 \ldots x_n$ with $n \geq 0$.

Values:

v	$::=$	$o \in V_\text{h}$	value in host language
	$\mid$	$\lambda x.t$	abstraction
	$\mid$	$[v_1 \ldots v_n]$	list of n values

Predefined Functions:

$$\texttt{map}(\lambda x.t\ [v_1 \ldots v_n]) \equiv [(\lambda x.t)\ v_1\ \ldots\ (\lambda x.t)\ v_n] \quad \text{given that } \texttt{sf}_f \notin t$$

$$\texttt{map}(\lambda s_0\ x.t\ [v_1 \ldots v_n]) \equiv \texttt{let}\ s_1\ y_1 = (\lambda s_0\ x.t)\ v_1\ \texttt{in}\ \ldots\ \texttt{let}\ s_n\ y_n = (\lambda s_{n-1}\ x.t)\ v_n\ \texttt{in}\ [s_n\ [y_1 \ldots y_n]] \quad \text{given that } \texttt{sf}_f \in t$$

$$\texttt{nth}(n\ [v_1 \ldots v_n \ldots v_p]) \equiv v_n$$

$$\texttt{split}(n\ [v_1 \ldots v_p]) \equiv [[v_1 \ldots v_{\frac{p}{n}}]\ \ldots\ [v_{\frac{(n-1)p}{n}+1} \ldots v_p]]$$

$$\texttt{join}([[v_1 \ldots v_{\frac{p}{n}}]\ \ldots\ [v_{\frac{(n-1)p}{n}+1} \ldots v_p]]) \equiv [v_1 \ldots v_p]$$

$$\texttt{take\_n}(n\ [v_1 \ldots v_n \ldots v_p]) \equiv [[v_1 \ldots v_n]\ [v_{n+1} \ldots v_p]]$$

$$\texttt{len}([v_1 \ldots v_n]) \equiv n$$

Fig. 4.1: Language definition of the Expression IR.

$$\begin{aligned}
&\texttt{let}\ f = \lambda x_\text{maze}\ x_\text{points}.\\
&\ \texttt{let}\ f_\text{body} = \lambda y\ z.\\
&\quad \texttt{let}\ x_\text{path} = \texttt{ff}_\text{find_path}(y\ z)\ \texttt{in}\\
&\qquad \texttt{sf}_\text{update}(y\ x_\text{path})\ \texttt{in}\\
&\quad \texttt{let}\ x_\text{resulting_maze}\ x_\text{map_results} = \texttt{map}(f_\text{body}\ x_\text{maze}\ x_\text{points})\ \texttt{in}\\
&\qquad \texttt{let}\ x_\text{unmapped} = \texttt{ff}_\text{get_unmapped}(x_\text{map_results})\ \texttt{in}\\
&\qquad\quad \texttt{if}\ (\texttt{ff}_\text{count} x_\text{unmapped} = 0\\
&\qquad\qquad x_\text{resulting_maze}\\
&\qquad\qquad f\ x_\text{resulting_maze}\ x_\text{not_mapped})\ \texttt{in}
\end{aligned}$$

Fig. 4.2: Expression for our labyrinth algorithm.

handling of it inside a map operation are based on the notion of state threads which were introduced to Ohua by Ertel et al. in recent work [Ert+19]. The function `split` separates an input list into equally sized chunks, while the function `join` reverses this operation, flattening a list of lists into a single large list. Both functions cancel each other out:

$$\texttt{join}(\texttt{split}(n\ [v_1 \dots v_p])) \equiv [v_1 \dots v_p]$$

The function `take_n` splits the input list into two parts, where the first list contains the first n elements (or the whole input list, whichever is smaller) and the second list forms the remainder of the input list. `len` is a simple function to determine the length of a list.

As a shorthand for writing more concise terms, we introduce a simple destructuring syntax which has already been used in the definition of `map` and is defined as follows:

$$\begin{aligned} \texttt{let}\ y\ z = \texttt{ff}_f()\ \texttt{in} \quad \equiv \quad & \texttt{let}\ x_{\text{result}} = \texttt{ff}_f()\ \texttt{in} \\ & \texttt{let}\ y = \texttt{nth}(1\ x_{\text{res}})\ \texttt{in} \\ & \texttt{let}\ z = \texttt{nth}(2\ x_{\text{res}})\ \texttt{in} \end{aligned}$$

Using our defined calculus, we can now define an expression that describes our labyrinth benchmark from Chapter 3.1. As point of origin, we use an idiomatic declaration of the labyrinth algorithm which is more compact than any version in the aforementioned chapter and resembles the algorithm as it would be written by a developer. This version is shown in Listing 4.1. Notably, we removed the loop split that allowed us to exploit some initial parallelism in Chapter 3. In the corresponding Lambda Expression in Fig. 4.2 we bind the `fill` algorithm to a fixed-point combinator f. In it, we bind the body of the loop to f_{body}, to which we then apply the initial state value x_{maze}. The resulting partial binding is then mapped onto the list of input values, namely x_{points}.

```
1  fn fill(maze: Maze, points: Vec<(Point, Point)>) -> Maze {
2      let map_results = for pts in points {
3          let path = find_path(maze, pts);
4          maze.update(path)
5      }
6
7      let unmapped = get_unmapped(map_results);
8      if unmapped.len() == 0 {
9          maze
10     } else {
11         fill(maze, unmapped)
12     }
13 }
```

Listing 4.1: Idiomatic definition of the labyrinth algorithm.

4.2 Transformation 1: Map Parallelization

The base idea of our first optimization developed in Chapter 3.2 was to execute the path finding in parallel to make use of the multi-threading functionalities of modern commodity CPUs. This was possible because the path-finding loop was altered to not contain any state update and the loop iterations themselves were therefore independent of one another. Since we want to have as few application logic as possible in the code generation and the runtime, it makes sense to move this optimization into the Expression IR optimization stage. To achieve a low-cost parallelization that does not require any knowledge about parallel loops in the runtime, we simply split the loop in question into a number of smaller loops. As these small loops do not exhibit any data dependencies between one another, the execution runtime can run them in parallel without having to understand, what these operators are.

Although we did this in our specific example for the problem of pathfinding, this can indeed be generalized. Each `map` combinator which does not modify state internally, i.e., does not entail `sf` combinators, may be parallelized in this way. Using the predefined functions `split` and `join` we define the transformation to adapt a map operation for p threads:

$$\texttt{let } r = \texttt{map}(t\ [v_1 \dots v_n])\ \texttt{in} \quad \xrightarrow[\text{p threads}]{} \quad \begin{array}{l} \texttt{let } m_1 \dots m_p = \texttt{split}(p\ [v_1 \dots v_n])\ \texttt{in} \\ \quad \texttt{let } r_1 = \texttt{map}(t\ m_1)\ \texttt{in} \\ \quad\quad \dots \\ \quad\quad\quad \texttt{let } r_p = \texttt{map}(t\ m_p)\ \texttt{in} \\ \quad\quad\quad\quad \texttt{let } r = \texttt{join}([r_1 \dots r_p])\ \texttt{in} \end{array}$$

Fig. 4.3: Transformation 1: Map Parallelization for p threads.

This transformation turns a single stateless map operation into p independent map operations which can then be individually scheduled and executed. To prove the semantic correctness of this transformation, we can show that the left and the right expression are indeed equivalent by resolving the right expression bottom-up:

$$\begin{aligned} \texttt{join}([r_1 \dots r_p]) &\equiv \texttt{join}([\texttt{map}(t\ m_1) \dots \texttt{map}(t\ m_p)]) \\ &\equiv \texttt{join}([[t\ v_1 \dots t\ v_{\frac{p}{n}}] \dots [t\ v_{\frac{(n-1)p}{n}+1} \dots t\ v_p]]) \quad \text{by map definition} \\ &\equiv \texttt{join}(\texttt{split}(p\ [t\ v_1 \dots t\ v_n])) \\ &\equiv [t\ v_1 \dots t\ v_n] \\ &\equiv \texttt{map}(t\ [v_1 \dots v_n]) \end{aligned}$$

Using this rather simple transformation we are now able to split state-free loops into smaller chunks of work that can be executed in parallel due to the absence of data dependencies.

4.3 Transformation 2: State Decoupling

In its initial state, a loop containing a state update, i.e., a `smap`, cannot be executed in parallel as the state update must occur sequentially to avoid locking and guarantee a deterministic execution. In our labyrinth example, we circumvented this by splitting the state update off and running it after the parallel path search from the start. But to allow any parallelization and later transformations to occur in unaltered applications like the algorithm in Listing 4.1, the state-free parts of such a loop must be decoupled from the state update first, using a transaction which we will define and discuss in this intermediate step first.

To formalize this, we let t be a non-empty term that does not contain any state-modifying foreign functions. We require non-emptiness here because otherwise the `map` function would only contain a `sf` combinator in which case no parallelization can occur. Furthermore, we define a state-modifying foreign function $\texttt{sf}_f$ and values $v_1 \dots v_n$ which are argument to `sf` and are bound either by the lambda expression mapped over of within t. The initial state value s is bound outside the `map` combinator. We can now define this preliminary transformation as follows:

$$
\begin{array}{l}
\texttt{map}((\lambda s\ x.t \\
\qquad \texttt{sf}_f(s\ v_1 \dots v_n)) \\
\quad [x_1 \dots x_n])
\end{array}
\quad \longrightarrow \quad
\begin{array}{l}
\texttt{let}\ \ x_{\text{intermediate}} = \texttt{map}((\lambda s\ x.t\ [v_1 \dots v_n]) \\
\qquad\qquad\qquad\qquad [x_1 \dots x_n])\ \texttt{in} \\
\quad \texttt{map}((\lambda s\ y.\texttt{let}\ \ v_1 \dots v_n = y\ \texttt{in} \\
\qquad\qquad \texttt{sf}_f(s\ v_1 \dots v_n))\ s \\
\qquad x_{\text{intermediate}})
\end{array}
$$

This transformation splits the `sf` combinator off from the rest of the loop body, allowing the first to be parallelized using Transformation 1 from Section 4.2. It represents a frequent pattern in shared state scenarios, where such loops often act as a fold operation on the state value.

But while this transformation enables us to further optimize state loops, we also have to discuss whether this is a legitimate optimization to make or whether it may alter the semantics of the program in question. For the previous transformation, showing the equivalence of both terms was simple and could therefore be done. For this alteration however, this would unwind into a lengthy proof which would be out of scope for this thesis and shall therefore be left for future work. Moreover, one can see easily, that an expression can be constructed that, given the same input, will not produce the same results before and after the transformation. An example that immediately comes to mind is the labyrinth algorithm. When searching a path, the `find_path` function reads the current state value which will ater the application of the transformation at no point contain any of the previously found paths. This produces numerous write conflicts, forcing recomputations and generating commits in a different order than in the sequential version, eventually leading to different results. However, this is also the case for other parallelism approaches like *Software Transactional Memory*, which also does not preserve equivalence to a sequential execution. This is due to the amorphous data parallelism often encountered in

these shared state programs. There is just no way to efficiently parallelize these applications without encountering their side effects and in particular while retaining sequential equivalence. Hence, we argue that this transformation is nonetheless valid, especially when migrating the application from a STM background. What's more, this approach manages to retain one of the core promises of Ohua: Determinism. Even though the results from before and after applying the transformation are not equivalent, they are both deterministic, which cannot be said about STM whose model is founded on non-deterministic execution. On the other hand, when attempting to parallelize such an application for the first time, developers should be well aware of the fact that parallelizing amorphous data parallel programs comes with trade-offs, making our approach still seem a good fit.

4.4 Transformation 3: Batch Updates

The second optimization we put forth in Chapter 3.3 considered itself with improving execution times by batching state updates. Our general idea was that the state update forms a sequential bottleneck which severely decreases performance. After the application of the transformation described in Section 4.3 however, we found that the isolated state updates gave way to the negative side effects of applications exhibiting amorphous data parallelism which mainly manifested in increased write conflicts due to infrequent updates. Hence, we wanted to introduce a way to vary the frequency of state updates in order to have a way to reduce the number of conflicts.

In our preliminary studies, we tackled this problem by only ever processing an arbitrary but fixed number of elements before updating the state. Although we mixed the retry semantics of the algorithm itself with the update frequency approach in Chapter 3.3 by immediately appending any failed updates to the back of the work set instead of putting them in a separate list, we can distill a generic transformation from this approach. Basic idea for this transformation is to turn the stateful loop into a recursive combinator that always takes up to n elements per recursion step from the input set and processes them until all elements have been processed. To formalize this we let n be the number of items to be processed per step. As for the previous transformation, s defines the initial state value and is bound outside of the `map` expression, while $[x_1 \dots x_p]$ is the set of input values for the map operation. The term t does not contain any `sf` combinators and the values $v_1 \dots v_p$ are all bound within the scope of the lambda expression mapped over.

The transformation is then defined as follows:

$$\texttt{map}((\lambda s\ y.t\ \texttt{sf}_f(s\ v_1 \ldots v_p))\ s\ [x_1 \ldots x_p])$$

$$\downarrow$$

$$\begin{aligned}
&\texttt{let}\ r = \lambda s_0\ x. \\
&\quad \texttt{let}\ x_{\text{input}}\ x_{\text{rest}} = \texttt{take_n}(n\ x)\ \texttt{in} \\
&\quad\quad \texttt{let}\ s_{\text{new}}\ x_{\text{result}} = \texttt{map}((\lambda s\ y.t\ \texttt{sf}_f(s\ v_1 \ldots v_n))\ s_0\ x_{\text{input}})\ \texttt{in} \\
&\quad\quad\quad \texttt{if}(\texttt{len}(x_{\text{rest}}) = 0 \\
&\quad\quad\quad\quad [s_{\text{new}}\ x_{\text{result}}] \\
&\quad\quad\quad\quad (\texttt{let}\ s_{\text{res}}\ x_{\text{res}} = r\ s_{\text{new}}\ x_{\text{rest}}\ \texttt{in} \\
&\quad\quad\quad\quad\quad s_{\text{res}}\ \texttt{join}(x_{\text{result}} x_{\text{res}})\ \texttt{in} \\
&\quad r\ s\ [x_1 \ldots x_p]
\end{aligned}$$

Note that we used in this transformation a non-decoupled stateful loop for the sake of brevity only. Transformation 2 may be applied either before or after this step to expose the necessary loop parallelism that warrants this transformation in the first place.

One can see easily, that this transformation also retains Ohua's determinstic approach: All elements are processed in the same order and state updates are also applied in the same, fixed order for each value for n. But, as was the case for Transformation 2 in Section 4.3 and has been exhaustively discussed there, semantic equivalence is not preserved by this operation either but is admissible as this is due to the basic properties of amorphous data parallel programs.

4.5 Transformation 4: Straggler Reduction using Work Stealing

The third modification we presented in Chapter 3 was the improving of resource utilization by tackling the straggler problem we discovered after parallelizing state-free map operations. Underlying cause for the straggler problem was the static assignment of work to specific threads. Due to the non-uniform nature of most map operations encountered in real-world applications and noise introduced by the operating system itself [LWH16], static work set assignments can produce wildly varying execution times per thread, as seen in Fig. 3.5. Hence, we applied a work-stealing task scheduling runtime to improve performance.

Admittedly, this optimization step is hardly a transformation but a mere engineering solution. It can be applied by changing the code generated for the set of `map` combinators produced by Transformation 1 presented in Section 4.2. Instead of generating a number of loop operators as originally intended, we can instead create a work-stealing runtime and schedule the loop operations as tasks in the runtime.

5 Experimental Setup

In order to compare the performance of Software Transactional Memory against Ohua in the context of shared state applications, we employ a set of benchmarks originally proposed by Minh et al. [Min+08]. In this chapter, we will categorize the benchmarks introduced by the authors and present a representative selection of applications which we will use to compare Ohua's performance against STM. To be able to better understand the behavior of the different implementations later on, we will briefly analyze, where each application has potential for parallelization and how this could be leveraged using Ohua and STM. Additionally, we will explain, which values we measured during execution of the benchmarks and how they are relevant for our evaluation.

5.1 Benchmark Choice

After presenting our transformations for Ohua in Chapter 4, we now wanted to compare its performance against STM in order to evaluate if Ohua could indeed be used as a suitable replacement for developing shared state applications. To provide a comprehensive comparison, we chose to use the *Stanford Transactional Applications for Multi-Processing* suite [Min+08]. Introduced by Minh et al., it was designed as a set of benchmarks for testing software transactional memory frameworks. The authors included 8 applications from different application areas in their suite, which are supposed to resemble the diverse landscape of parallelism in applications developers might face. In particular, the STAMP suite contains examples from different application domains and varying use cases for transactional memory such as high-contention and low-contention scenarios.

Application	**Instructions per tx** *(mean)*	**Time spent in transactions**
labyrinth	219,571	100%
bayes	60,584	83%
yada	9,795	100%
vacation	3,223	86%
genome	1,717	97%
intruder	330	33%
kmeans	117	7%
ssca2	50	17%

Tab. 5.1: A basic characterization of STAMP applications, comparing the mean number of instructions per transaction and the overall percentage of time the application spends in transactions. These numbers stem from a C implementation and have been adapted from Minh et al. [Min+08]

Application	tx length	r/w set	tx time	Contention
labyrinth	long	large	high	high
bayes	long	large	high	high
yada	long	large	high	medium
vacation	medium	medium	high	low/medium
genome	medium	medium	high	low
intruder	short	medium	medium	high
kmeans	short	small	low	low
ssca2	short	small	low	low

Tab. 5.2: A qualitative summary of each STAMP application's runtime transactional characteristics. The length of a transaction is determined by the number of instructions it encompasses. The characteristics are ranked relative to the other applications in the suite. Adapted from Minh et al. [Min+08]

The tables 5.1 and 5.2 give a basic characterization of the benchmarks in terms of their usage of transactions. As can be seen in table 5.1, the length of individual transactions varies greatly per application, as does the overall time that is spent by the benchmark executing transactions. Even though the numbers in the table have been adapted from Minh et al. and represent values measured for their C-based implementation, they still outline the general characteristics of the respective STM-based algorithms. Some applications suffer so badly from the irregular properties outlined in Chapter 2.1 that exploiting their parallelism requires them to spend more than 80 % of their overall execution time in transactions. This is for example the case in the *labyrinth* benchmark, where fields of a dense 3-dimensional matrix have to be continuously updated, as we explained in Chapter 3.1. Other applications such as *kmeans* or *ssca2* have relatively short transactions, meaning their data parallelism is easier to exploit or they generally do not feature as much opportunities for parallelism as other applications.

Another relevant and perhaps the most limiting factor for programs relying on optimistic parallelism principles such as STM is contention. This characteristic is visualized in table 5.2 along with other properties. When facing high contention scenarios, STM implementations are usually unable to achieve the near-linear speedups Minh et al. reported for other benchmarks. Contention is a byproduct of frequent reading and writing accesses to the shared data structure that inevitably lead to frequent conflicts which require a rollback of all affected transactions except for the one that committed its changes first. Hence, the relative amount of reads and writes per transaction is also reported in table 5.2. Long transactions, large read/write sets, more time spent in transactions and high contention are all factors promoting conflicts. The results of conflicts are frequent rollbacks and accompanying recomputations, which reduce the overall performance.

Based on the analysis provided by the authors, we selected a representative range of benchmarks for our comparison between Ohua and STM. We chose applications with varying transaction lengths and frequency of transaction use as well as different

levels of contention. The next sections will briefly outline the details of the chosen benchmarks and explain, how they were implemented.

Since the authors only provided a C-based reference implementation for their programs, we had to re-implement them in Rust to rule out language-specific performance changes when comparing to Ohua's Rust implementation. We chose to use the `rust-stm` library written by Bergmann et al. [@Ber20] for these implementations. Upon inspecting the original source code, it turned out that the authors adapted the code in order to improve the performance of STM in some benchmarks. For instance, they provided their own implementation of a `HashMap`[1] that offered the use of transactions on a per-bucket basis, effectively exposing fine-granular parallelization opportunities that normal HashMaps cannot provide. In Rust, no corresponding STM-specific data structures existed prior to this work. We debated, whether or not we should use these optimizations in our own implementation but ultimately decided in favor of doing so. First, one could argue that these optimizations would be made by developers anyway after deciding to use the STM framework in an attempt to tailor the program code towards the library used. Secondly, we wanted to remain as close as possible to the original implementations from Minh et al. in the hopes of achieving similar results for STM as they did. Therefore, we contributed a small library [@Wit20f] which provides data structures like HashMaps and HashSets augmented for the use with transactions. Both the library and the benchmarks were implemented from scratch based on the descriptions provided by the authors, literature they cited and the code they supplied. We did so in an idiomatic way, applying both concepts to the problems using the tools the frameworks and the STM data structure library provide natively. As implementing the Ohua transformations proposed in Chapter 4 has been left for future work, we applied these transformations manually to the algorithms. The code for all benchmarks we wrote for this thesis may be found online [@Wit20c].

When implementing the *labyrinth* benchmark, we found that two transactions may deadlock. We have reported this issue [@Wit20g] and resorted to forking and patching the library [@Wit20d] in question to move on with our tests.

5.1.1 Parallelism Opportunities

When presenting each benchmark, we will provide a short assessment of the opportunities for parallelism it provides and how they can be exploited when using Ohua and STM. Goal of this analysis is to detect structural limitations within the benchmarks which one of the frameworks cannot overcome to exploit the full available parallelism. For this analysis, we produced an abstract prepresentation of each selected application which aims to point out a program's structural parallelism opportunities. To keep this representation concise and simple, we simplified each algorithm by removing loop conditions, abort conditions and any unnecessary computations. The resulting abstracted code snippets only contain the main operations performed on the input data to produce the end result as well as any shared state used.

[1] A dynamic key-value store allowing fast lookups by hashing the keys and organizing them in different buckets.

```
let mut maze = Maze::initialize();
for pair in points {
    let path = find_path(maze, pair);
    maze.update(path);
}
```

Listing 5.1: Abstract description of the *labyrinth* algorithm

5.1.2 Labyrinth Path Mapping

The labyrinth benchmark we presented in Chapter 3.1 based on the work of Swalens et al. [SDD16] was originally proposed by Minh et al. as possible benchmark for Software Transactional Memory. Since we have already analyzed and implemented this application, it was apparent for us to adduce it for our concluding comparison. Additionally, it is one of few benchmarks in the suite exhibiting amorphous data parallelism, a trait the authors did not consider when compiling their benchmark list but which they happened to include by coincidence, as it is frequently encountered in real-life applications using shared state, the main use case for transactional memory applications. Another interesting property of the benchmark was its frequent use of transactions (the whole algorithm is executed within transactions) and the high contention on the shared data structure, which puts the synchronization primitives under heavy stress, as described before.

As can be seen in Listing 5.1, the *labyrinth* application consists of a state-modifying loop. After finding a path for each input coordinate pair, the maze gets updated accordingly. This intertwining of calculation and state update is customary to programs exhibiting amorphous data parallelism, as it creates these dependencies between separate calculations in the first place. When using STM for concurrency control, one can execute the whole loop in parallel, guarding each individual iteration using a transaction, resulting in the 100 % transaction coverage reported in Table 5.1. Ohua is also able to exploit the parallelism in this loop due to the Transformations 2 and 3, allowing both approaches to possibly exploit the maximal available parallelism.

5.1.3 Intruder Detection

The intruder application implements a signature-based Network Intrusion Detection System (NIDS) and is used in networking to detect attacks or malicious activities in an active network as well as policy violations. It is based on design proposal number five of Haagdorens et al.'s work on „Improving the Performance of Signature-Based Network Intrusion Detection Sensors by Multi-Threading“ [HVG04].

The basic function of this application is to scan incoming network packets and match them against known intrusion signatures. This happens in three distinct stages, as outlined in Fig. 5.1. Incoming network traffic is captured and queued for inspection. Due to the architecture of modern network protocols, individual data flows have to be split into several packets that are transmitted individually and may reach the recipient out of order. Attackers have used this in the past by splitting malicious flows and sending them out of order to avoid detection. To counter this, the second

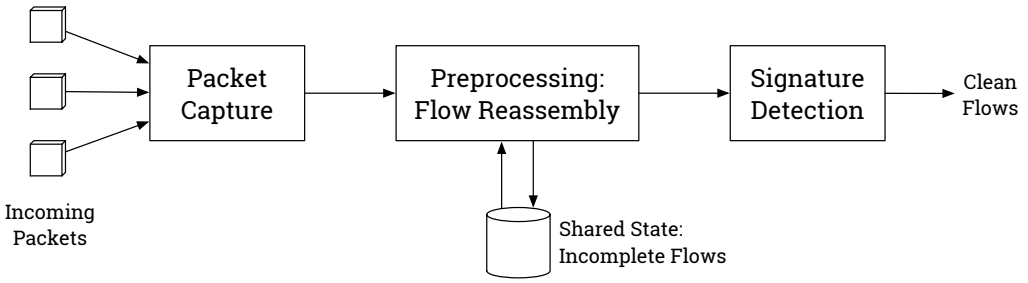


Fig. 5.1: Workflow of the *intruder* benchmark. Processing of incoming data is conducted in three stages.

```
let mut flows = State::new();
for packet in input {
    flows.add(packet);
}

for flow in flows {
    analyze(flow);
}
```

Listing 5.2: Abstract description of the *intruder* algorithm

step in the algorithm is to deploy stateful detection avoidance countermeasures, which involve preprocessing the received packets and reassembling the original flows. These flows can then finally be matched against known attack patterns, filtering any malicious packets from the stream of incoming data.

In this application, no amorphous data parallelism may be found. Instead, we encounter two loops of which one is stateful and the other state-free, as shown in Listing 5.2. The flow reassembly phase (step two) happens in parallel in the STM implementation, using a transaction-aware HashMap. Since the loop body does not contain any other state-free functions, there are no parallelization opportunities for Ohua as no transformations from Chapter 4 are applicable. Also, it would not make sense to attempt to exploit parallelism from this loop using any workarounds, so it is executed sequentially. This shows that Ohua may only extract non-trivial parallelism from irregular applications that fit certain criteria, i.e., contain loops that do not solely consist of state-modifying operations, as its main approach is to exploit data parallelism by the use of certain transformations to uncover it. Both approaches manage to implement the state-free analysis loop (step three) in parallel as again no concurrency control is necessary. Ohua does this using Transformations 1 and 4. Our evaluation will show, if it is more performant to attempt exploiting parallelism from the stateful loop or if a sequential approach as done in Ohua performs better.

This benchmark was chosen because it is mostly similar in its properties to the labyrinth application as it also features short transactions and high contention on the shared data, but does only spend about 33 % of the overall execution time in transactions. We mainly wanted to see, how this slight difference in properties is reflected in the performance results.

```
let mut centers = initialize(input);
loop {
    // data parallelism
    for item in input {
        item.find_center(centers);
    }
    // fold
    centers = recompute(input);
}
```

Listing 5.3: Abstract description of the *kmeans* algorithm

5.1.4 K-means Clustering

Proposed as benchmark by Narayanan et al. [Nar+06] the k-means clustering algorithm partitions a set of n observations into k clusters. It was originally put forth by MacQueen et al. in 1967 [Mac+67] and still is a very popular algorithm for cluster analysis in data mining which is often used to classify data.

The inner workings of the algorithm are rather simple: It takes a set of n observations and a desired number of clusters to sort the observations into. Then the k cluster centroids are initialized. Many ways exist to realize this but we chose, akin to our C reference implementation, to select the coordinates for each cluster centroid randomly from the coordinates preset by the input data set. Following this initialization, each observation is assigned to its nearest cluster based on the squared multi-dimensional spatial euclidian distance between both points. Afterwards, new centroids are computed by calculating the means of all observations now assigned to a certain cluster. These last two steps now get repeated iteratively until the algorithm either reaches an upper bound of iterations or converges, i.e., less than a certain percentage of observations change per iteration.

In the *kmeans* application, all parallelizable calculations happen within the single state-free innermost loop, which assigns all observations to a new centroid before the new centroids are calculated sequentially. This is shown in Listing 5.3. The STM implementation introduces some shared state for deciding on the convergence of the algorithm (not depicted in the abstract representation) and for updating the centroids during the inner loop iteration, executing the whole body of the outermost loop in parallel using transactions. Ohua exploits the data parallelism of the inner loop using Transformations 1 and 4 to exploit data parallelism. As both approaches are able to exploit all opportunities for parallelism, though each does it differently, we expect similar performances for both implementations.

k-means is very much like the intruder benchmark in the regard that its parallelism opportunities are limited to a single state-free loop but unlike the aforementioned benchmark k-means features low contention on shared data structures while presenting a similar transaction utilization, making this an interesting benchmark to look at.

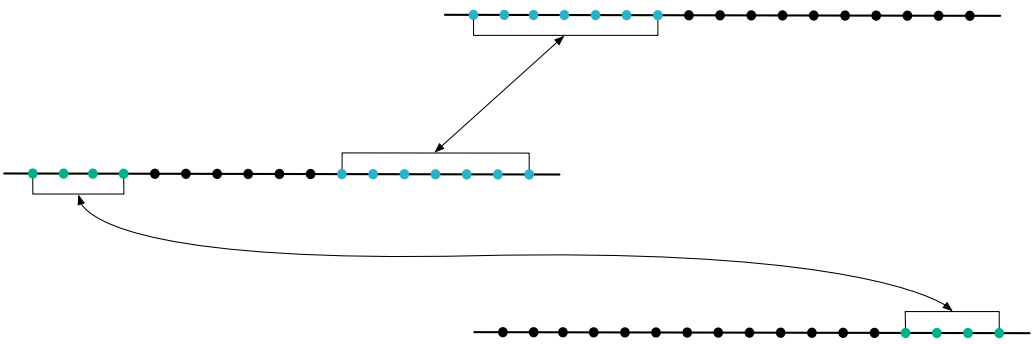


Fig. 5.2: Visualization of the overlap matching found in the *genome* benchmark. The blue match is stronger as it has seven matching elements and has hence been found first.

```
let mut nucleotides = Hashset::new();
for segment in input {
    // deduplication
    nucleotides.insert(segment);
}

loop {
    for item in nucleotides {
        item.find_neighbor(nucleotides);
    }
}
```

Listing 5.4: Abstract description of the *genome* algorithm

5.1.5 Genome Sequencing

This benchmark implements a *„whole-genome shotgun sequencing“* algorithm as outlined by Pop et al. [PSS02] in their work. Its goal is to sequence a (fictional) genome, i.e. to reassemble a nucleotide sequence from a set of snippets which is something frequently done in genetics.

The first step in the algorithm is to deduplicate the DNA segments that have been provided as inputs, since there are usually many duplications. The second phase is then concerned with finding neighboring segments in the remaining pool of DNA parts by utilizing overlap matching. By reducing the overlap size each iteration, the best possible fit is chosen for each neighbor search. Figure 5.2 provides visualization of how this matching works. Starting from a match length of $n - 1$, the algorithm attempts to find a matching predecessor-successor pair for all loose ends. With each iteration, the match length is reduced, until it ends with an overlap of one. In our example, the blue overlap match is found first due to seven matching nucleotides. Three iterations later, the green match is established, fully connecting the centered genome segment. After this phase has finished, all but two segments have a predecessor and successor assigned. Starting from the one element in the set with no predecessor, the chain of nucleotides can be rebuilt by simply following the links.

genome consists, as outlined in Listing 5.4, mainly of two separate loops. The first loop expression to deduplicate the input set is again a purely state-modifying loop, as seen before in the *intruder* benchmark. All operations in the loop body are directly accessing the shared state, making parallelization difficult. The second loop on the other hand is state-free, allowing for the simple exploiting of the parallelism within without the need for concurrency control.
Providing a parallel implementation for the first loop however, is more challenging. Only by using a transaction-aware HashMap with k buckets, the STM implementation is able to exploit the loop's parallelism for up to k conflict-free accesses. Ohua, again unable to use any transformations for leveraging parallelism, may also exploit a certain part of parallelism by emulating the same behavior as STM and partitioning the input set beforehand into k parts, so that the compiler may break the resulting state-free loop using the first transformation:

```
1 fn dedup(segments: Segments, threadcount: usize) ->
  ↪ Vec<SequencerItem> {
2     let parts = partition(segments, threadcount);
3     for p in parts {
4         deduplicate(p)
5     }
6 }
```

This application example shows, that even when using Ohua, code sometimes has to be written in a certain way to expose the parallelism opportunities of a computation to the compiler. Nevertheless, the resulting Ohua algorithms can still be executed as a sequential program while the resulting STM code, which is even more optimized by the use of special data structures, is unable to do so. Overall, we expect differing results for both implementations, as they try to exploit the parallelism hidden in the first stateful loop in different ways, probably with varying degrees of success.

All in all we chose this application to complete our benchmark selection as it provides similar properties as the kmeans benchmark in terms of transaction length and contention but spends nearly all its execution time in transactions, a stark change compared to the 7 % of transaction time in kmeans.

5.1.6 Summary

In summary, we found in our analysis that only the labyrinth application contains amorphous data parallelism and is the only application where Ohua's Transformations 2 and 3 are applicable. All other benchmarks contain loops that are state-free, while possibly occuring state modifications exist in separate loops. While STM can be used to exploit parallelism from the latter, Ohua has to execute this type of loop sequentially to avoid handling shared state. Therefore, we expect to see different behavior in the labyrinth application than in the other applications for Ohua's performance compared to STM.

5.2 Reference Measurements

Minh et al. proposed their STAMP suite in 2008. Since then, hardware (e.g., clock speeds of CPUs) as well as compile-time optimizations have evolved significantly, rendering the results provided in their paper outdated. Hence, we decided to run the benchmarks relevant to our investigation again, on the same hardware and under the same conditions as our Rust benchmarks to have a reference we can compare our STM implementations to. Although the authors themselves only measured results for the smaller two of their three suggested input sets, we tested all three to have a more comprehensive set of results to compare our STM implementation against.

Upon building and executing the benchmarks with the included `tl2` STM implementation [@Min13], a number of issues with the STAMP suite became apparent, which we already briefly touched upon in Chapter 1. The genome benchmark could not be built from the original source code, due to conflicts caused by compile-time memory allocation patching. We reported this issue [@Wit20a] and employed a workaround to be able to use the benchmark nonetheless. Additionally, we found that the implementation of the labyrinth benchmark sometimes does not yield a correct solution, causing the program to crash [@Wit20b]. Similar behavior was found for the intruder application, although we did not investigate this further, instead discarding failed runs in both cases, lowering the effective number of test results slightly below 30. These issues cement some of the drawbacks of STAMP and STM in general, which we discussed in the beginning of this thesis.

For comparability and in order to provide a more complete performance graph, we intended to run the selected four benchmarks for the same range of threads as our own applications but the suite only supports thread counts which are a power of two, leaving us with 1, 2, 4, 8 and 16 threads as test parameters. Due to this limited result range, we will only be able to draw vague general comparisons between STAMPs STM and our own STM implementation as the sparse result coverage for higher thread counts leaves us unable to reliably identify trends in the performance of STAMP.

5.3 Measurements

In our experiments, we tried to achieve reproducible and plausible results so that we can make an educated comparison between both STM and Ohua. This section will briefly explain our benchmarking setup to enable others to reproduce our results.

5.3.1 Input Data

For each STAMP application, Minh et al. [Min+08] additionally provided three sets of parameters to model small, medium and large workloads. We adopted these parameters with only minor modifications, as intended by the authors. A change was made to the parameters of the *genome* benchmark as the input data that was randomly created using the original input parameters proved faulty in our

Application	Arguments	Description
genome	-g 256 -s 16 -n 16384	n gene segments of length s are first sampled from a gene consisting of g nucleotides and then reassembled again.
genome+	-g 510 -s 32 -n 32768	
genome++	-g 16384 -s 64 -n 16777216	
intruder	-a 10 -l 4 -n 2048 -s 1	From seed s, n traffic flows are generated, $a\%$ of which contain attacks. Each flow consists of up to l packets, which are assembled and inspected by the program.
intruder+	-a 10 -l 16 -n 4096 -s 1	
intruder++	-a 10 -l 128 -n 262144 -s 1	
kmeans-high	-n 15 -t 0.05 -i random-n2048-d16-c16.txt	The input file i containing n points in d dimensions generated about c centers is loaded and then clustered into n clusters. A convergence threshold of t is used.
kmeans-high+	-n 15 -t 0.05 -i random-n16384-d24-c16.txt	
kmeans-high++	-n 15 -t 0.00001 -i random-n65536-d32-c16.txt	
kmeans-low	-n 40 -t 0.05 -i random-n2048-d16-c16.txt	
kmeans-low+	-n 40 -t 0.05 -i random-n16384-d24-c16.txt	
kmeans-low++	-n 40 -t 0.00001 -i random-n65536-d32-c16.txt	
labyrinth	-i random-x32-y32-z3-n96.txt	The input file i describes a maze of dimensions $x \times y \times z$ and n paths to map.
labyrinth+	-i random-x48-y48-z3-n64.txt	
labyrinth++	-i random-x512-y512-z7-n512.txt	

Tab. 5.3: Input data sets for the benchmarks presented in this thesis. Adapted from Minh et al. [Min+08] and adjusted to mitigate flaws in the original algorithms.

re-implementation. Also, a unnecessary parameter was removed from the *kmeans* benchmark.

Table 5.3 gives a full overview over all parameters we used. Input sets marked with a + indicate a larger input and an appended ++ marks the largest of the three input sets for a benchmark. The *kmeans* benchmark inputs are additionally labeled as *high* and *low*, which refers to the relative amount of contention produced by the inputs.

5.3.2 Measured Values

For the purpose of our comparison between Ohua and STM, two values are of importance and have hence been measured. As we are calculating the speedups of both implementations in reference to a sequential implementation, the execution time in milliseconds, i.e., the time it takes the algorithm to complete, is relevant. Moreover, we were interested in the power consumption of both algorithms. But since measuring the power consumption of the algorithms would have been to complicated and time-consuming, we opted for measuring the total CPU time used by the programs, as the amount of power used while processing is resulting from the utilization of a PCs individual components. Since all these programs are performing purely in-memory computations, we figured that measuring the CPU time would be a sufficient approximation to draw some general conclusions regarding the power usage.

For both measured values, the setup phase (parsing input arguments and reading input files) and teardown phase (writing the results to a file) where not included in the measurements. This was done in order to reduce potential noise from Operating System calls resulting from the I/O operations performed in these stages. Research has shown in the past that this noise, as well as different contention scenarios can severely impact the measured time values, leading to variations in the measured time values. To take this into account and produce more realistic results, each measurement has been done 30 times, so that statistical outliers do not carry as much weight. From the resulting measured values the geometric mean was calculated per data point to compile all measurements into a single data point.

Using the measured execution times in Milliseconds, we calculated the speedup using the following equation:

$$S_c = \frac{t_{seq}}{t_f}$$

where S_c is the speedup of the configuration under investigation, t_{seq} is the mean execution time of the sequential implementation and t_f the mean execution time of the framework in question.

5.3.3 Running Configuration

All benchmarks conducted for this work were run on an Ubuntu 18.04 server with 128GB RAM and two Intel Xeon E5-2630 v2 processors which have a base frequency of 2.60 GHz and offer combined 12 cores with 24 threads. At all times, the program under inspection was running exclusively on the machine, meaning that aside from background operating system jobs, no other tasks ran on the system simultaneously.

Results and Evaluation

6

In this section, we will present and interpret the results of our experiments as presented in Chapter 5. This involves firstly a brief analysis of the results of the original STAMP benchmark to establish them as a baseline to legitimize or discuss the results of our Rust-based STM implementation of the algorithm as we will use the latter to evaluate Ohua's performance. Subsequently, we will analyze the results of our Ohua benchmarks in detail to see, whether it could be used as a suiting replacement for STM in terms of performance.

6.1 Reference Measurement Results

Benchmark results for the reference measurement runs we conducted can be found in Figure 6.1. We will compare our achieved speedups to the original benchmark runs from Minh et al. [Min+08], namely to the results of their Eager STM implementation to see, how performance of the framework changed over the years. This allows us to set realistic expectations for the performance of our STM-based implementations.

In the *labyrinth* benchmark, we observed an increasing speedup for up to 8 threads in the small and medium sized problem sets, maxing at about 3.0 speedup and followed by declining performance for 16 threads. The largest input data set on the other hand exhibits a steadily increasing curve, achieving a speedup of about 6.0 for 16 threads. This deviates from the original results, where both the labyrinth and the labyrinth+ benchmark showed behavior similar to our labyrinth++ curve form: A continuously increasing performance, achieving a speedup of about 4.0 at best. The form of the curve and the overall reduced performance may indicate that either the sequential implementation is performing better now than it did back in 2008 or that the overhead of the `tl2` STM framework increased. Our first theory is backed as possible cause by the fact that compiler optimizations have improved in the past years along with CPU clock speeds, making a better sequential performance not unlikely. An increased framework overhead on the other hand would also support the performance drop for 16 threads, which the better sequential performance does not explain. This was also discussed previously by Perfumo et al. [Per+08] who reported similar performance drops for STM applications with higher thread counts. Another explanation for this decline would be our used hardware architecture, as it consists of two CPUs. While all runs up to 8 threads can be executed on a single CPU, the 16 thread version requires the utilization of both cores, requiring more complex memory and resource management which in turn takes more time and may also be an explanation since we do not know if Minh et al. had similar hardware. But since we only made this and the following measurements as reference points for the data we acquired in our own experiments, we did not investigate the causes for this behavior beyond speculations to possible reasons and leave this to future work.

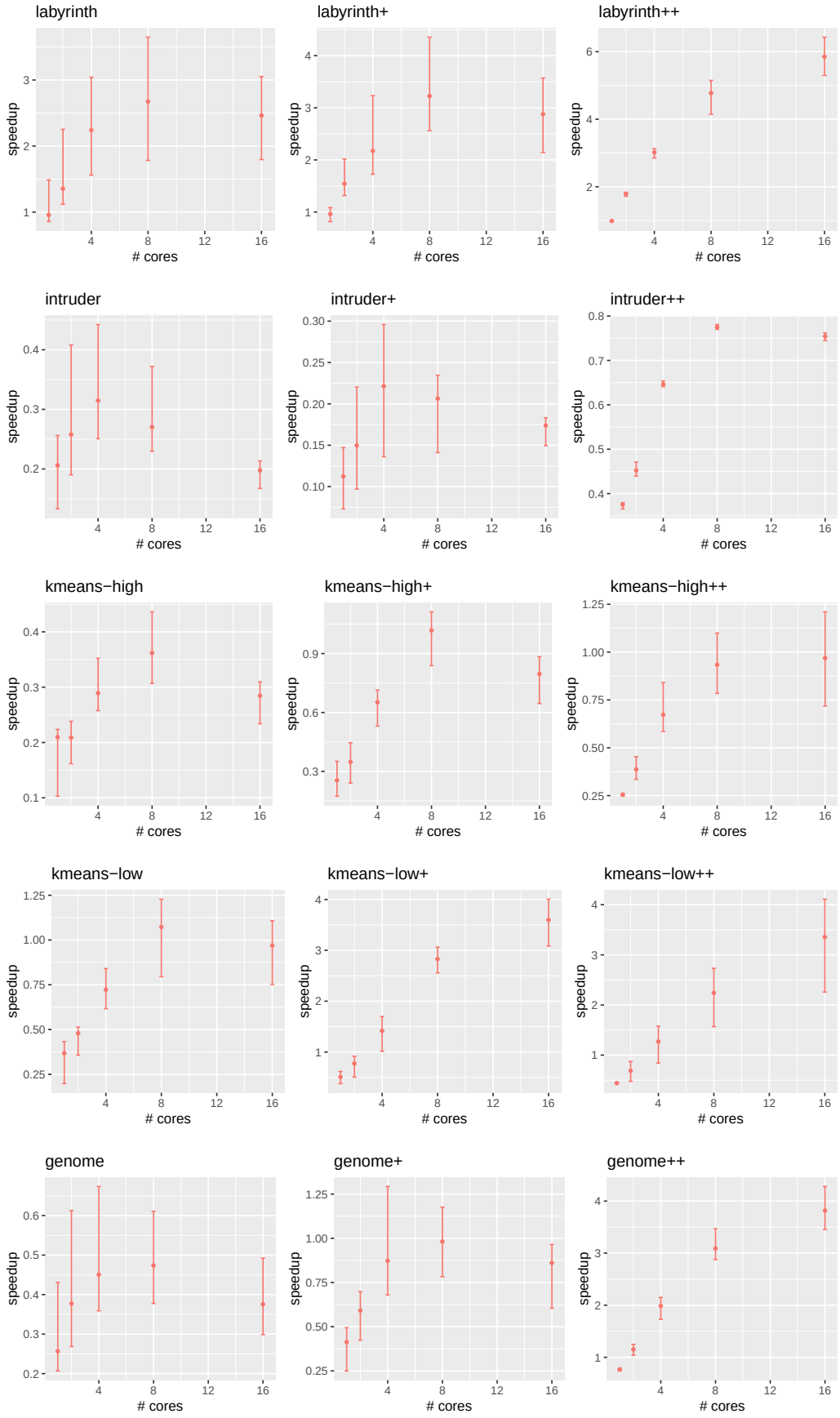


Fig. 6.1: Speedup achieved by STM in the original STAMP benchmarks.

A similar deviation from the original measurements can be seen for the *intruder* application. Our own results show a curve that peaks at 4 or 8 threads respectively, before declining again. Maximum speedups are as low as 0.3 for the small, about 0.2 for the medium and 0.8 for the largest input set. The original results on the other hand show Eager STM achieving a speedup slightly below 1 for both intruder and intruder+[1] meaning that our results are again remotely similar, taking into account our previous assumptions about possible changes in the execution environment compared to the original benchmarks.

For its high-contention scenario, our *kmeans* measurements show a similar behavior as the previous applications. Both smaller benchmarks achieve peak performance for 8 threads while performance decreases for 16 threads or stagnates in the case of kmeans-high++. In the original measurements however, the authors achieved steadily increasing near-linear speedups like our high++ version did with speedups of up to 4.0 for both input sets while we only achieved a maximal speedup of 1.0 for both high+ and high++.
A more stark deviation is visible in the low-contention variant of the benchmark: The original speedup grew linearly with increasing number of threads used due to the low contention, maxing out at execution times 9 or 10 times faster than the sequential version respectively. We again have a peak performance at 8 threads for the kmeans-low variant but steadily increasing speedups for low+ and low++, achieving a 3.5 speedup.

In *genome*, we achieved for the first time curve shapes similar to the original results from Minh et al., although the original results showed peak performance at 4 threads while our results peak at 8 threads before decreasing again. Also, our total achieved speedup is slightly below the originally reported numbers. The solid performance of genome++ might be due to the fact that the overhead of the used framework amortized for this large input set.

Overall, our results are remotely similar to the original results, although we consistently observed lower speedups than reported in the original. We identified either a better sequential performance, increased framework overhead or hardware overhead as possible sources, in reality it may even be a combination of all these factors. Due to the fact that both result sets vary greatly, we opted to discard the original results reported by Minh et al. in favor of using our own measurements as reference for our Rust-STM implementations.

6.2 Rust-based Benchmark Results

labyrinth. We already discussed in detail the performance of the *labyrinth* application for Ohua and STM, which is shown in Fig. 6.2, in Chapter 3. Notably, Ohua performs slightly worse than STM, but shows the same overall scaling behavior for more cores while exhibiting less variance in the results. For labyrinth++, one can not state clearly, which benchmark performs better due to the comparatively large

[1] Hardware-based Transactional Memory approaches perform generally better in this benchmark.

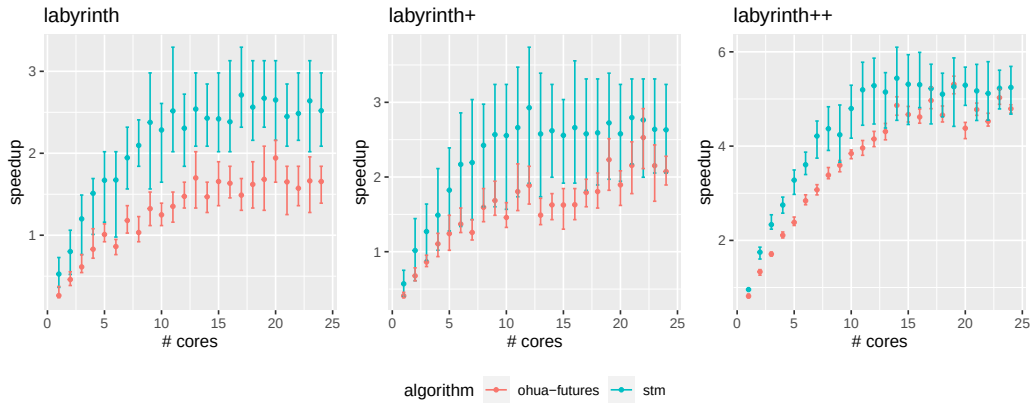


Fig. 6.2: Speedup in the labyrinth application relative to a sequential implementation.

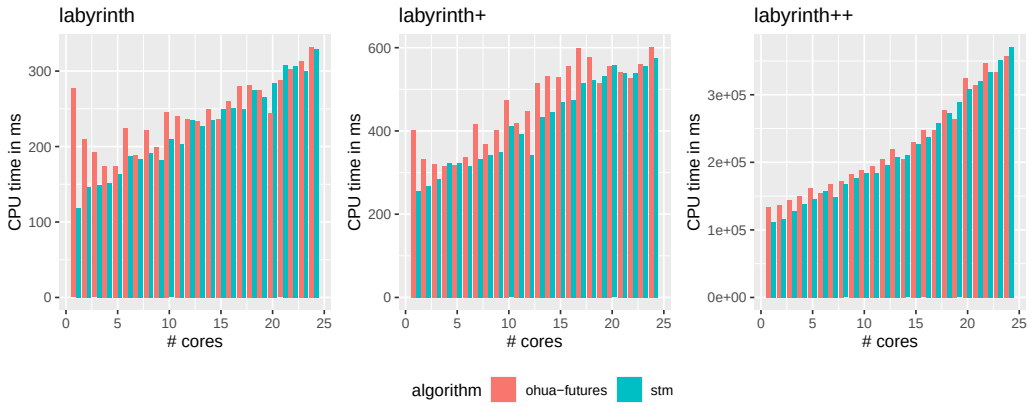


Fig. 6.3: CPU time used by both frameworks in the labyrinth application.

variance in the STM results. The Software Transactional Memory implementation itself performs just as the original STAMP implementation by Minh et al. [Min+08], reinforcing the validity of our implementation.

When comparing the CPU time used by both Ohua and STM as shown in Figure 6.3, we unsurprisingly see an overall growing demand for computation time as creating more threads and moving data between them in itself takes more time. One can observe the correlation between the time spikes for both applications in the labyrinth+ benchmark and a degraded performance for the respective thread counts. Due to Ohua's algorithm structure, low thread counts become even less performant for smaller input sets, as fewer threads require more loops of the algorithm, creating a non-negligible runtime overhead. This becomes irrelevant for larger inputs such as labyrinth++, though.

intruder. For the *intruder* program, both frameworks deliver extremely different results: STM performance for the small input set (as seen in Fig. 6.4) is similar to the performance showcased in the reference measurements in Fig. 6.1, although the curve shape is different, the Rust version showing a slow and steady increase in performance. The medium sized input set however, produces a rise so flat it almost

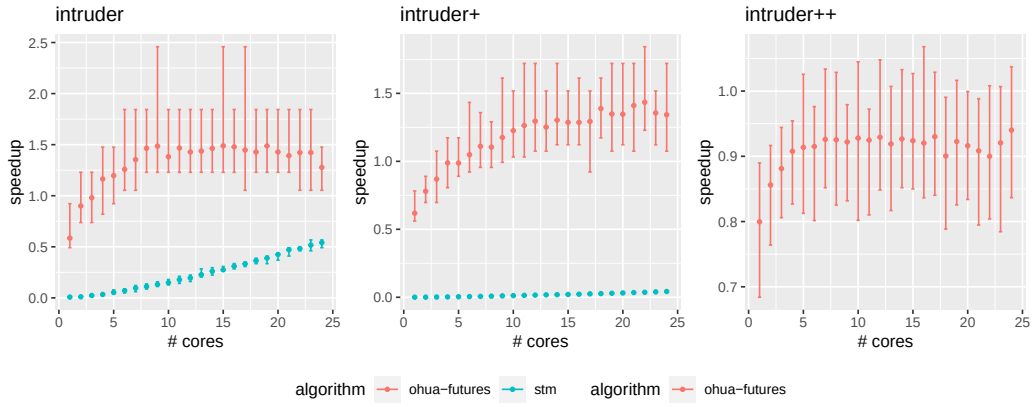


Fig. 6.4: Speedup in the intruder application relative to a sequential implementation.

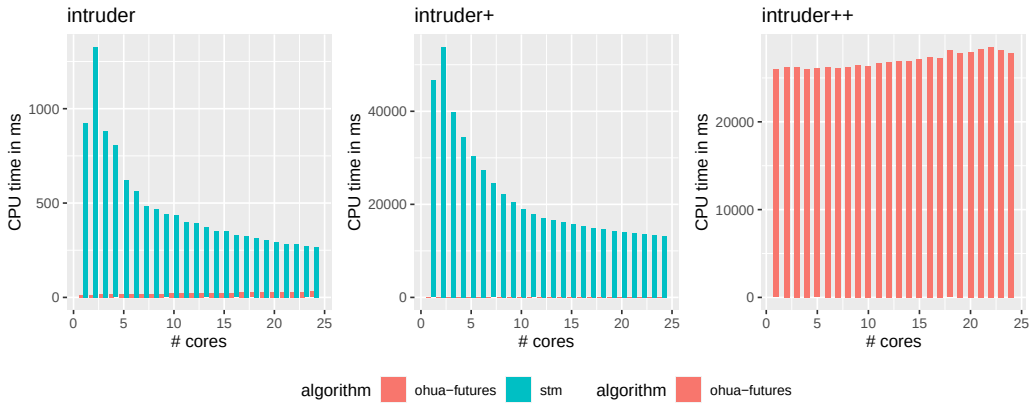


Fig. 6.5: CPU time used by both frameworks in the intruder application.

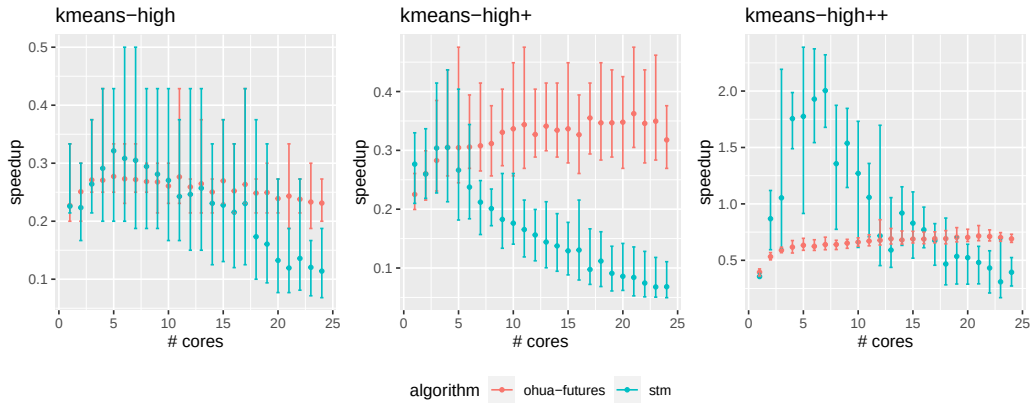


Fig. 6.6: Speedup in the kmeans-high application relative to a sequential implementation.

becomes invisible due to the graph scaling. We suspect this to be caused by the slow reassembly phase of the benchmark, which operates on a shared HashMap. Even though we augmented the standard libraries' HashMap implementation with basic transaction handling capabilities like Minh et al. did, there is still a lot of contention on these shared data instances, impacting execution times. This assumption is supported by the immense amount of CPU time used by the STM implementation. Because of these long execution times we did not measure STM's speedups for the significantly larger intruder++ input set.

Ohua on the other hand achieves in both smaller cases relatively good speedups of about 1.5 and 1.3 respectively, outperforming Software Transactional Memory. Particularly interesting is the performance plateau that is reached by Ohua for a medium amount of threads and which is best visible in the smallest input set. Source of this is the fact that a not insignificant portion of the algorithm runs sequentially in Ohua, as explained in Chapter 5.1.3. Hence, only a certain speedup may be achieved by parallelizing the application, thus creating a plateau that transitions into declining performance later on when the framework overhead becomes too large. But as was the case with STM, the intruder+ version performs slightly worse than the smaller input set and Ohua's intruder++ results are even worse, below 1.0. This can be attributed to the increasing input set sizes, which take longer to process during the sequential flow reassembly phase, impairing the speedup achieved by the parallel detection phase. The aforementioned performance difference is also clearly seen in the use of CPU time which is shown in Figure 6.5, where Ohua uses orders of magnitude less CPU time than STM.

kmeans Our *kmeans* results show no clear winner in terms of performance. Although there were differences in the performance behavior between the high contention and low contention versions of the STAMP benchmark, the STM version written in Rust showcases in Figures 6.6 and 6.8 similar curve shapes for both sets of input data which are completely different from the results in Fig. 6.1. This is most likely due to differences in the implementations of both benchmark versions. C-based

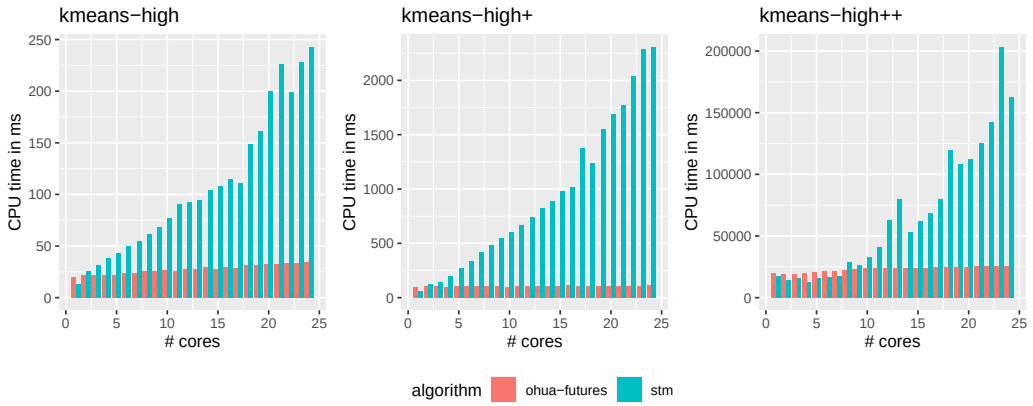


Fig. 6.7: CPU time used by both frameworks in the kmeans-high application.

```
unsigned long data[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

pid_t pid = fork();
if (pid != 0) {
    int lower = 0;
    int upper = 4;
    // changes elements at indices 0 to 4
    modify_elements(data, lower, upper);
} else {
    int lower = 5;
    int upper = 9;
    // changes elements at indices 5 to 9
    modify_elements(data, lower, upper);
}
```

Listing 6.1: Example for memory sharing that is possible in C, but impossible in Rust.

programs allow ways of memory sharing that are irreproducible in Rust, forcing us to change some aspects of the program, probably causing these differences.

An example for the differences in what is considered legal memory sharing can be seen in the snippet of C code shown in Listing 6.1. There, an array of data (in this example integers, but in *kmeans* it would be observations to classify) is to be manipulated by two concurrent threads. Since the developer wants to split the work evenly between both threads, she can assign both threads non-overlapping ranges to iterate over and let both threads work directly on the array. Without any locking, this is risky, as there is no control mechanism ensuring that both threads don't alter the memory regions of the other thread due to a bug in an index calculation or by writing too large portions of data to the array. Nonetheless, this is the fastest possible implementation (due to the absence of safeguards). In Rust however, this is impossible. The language itself does not allow this type of fast yet unsafe memory sharing, as the type system ensures that data shared across threads without locks is read-only. So, if one wants to implement the same algorithm shown in above listing

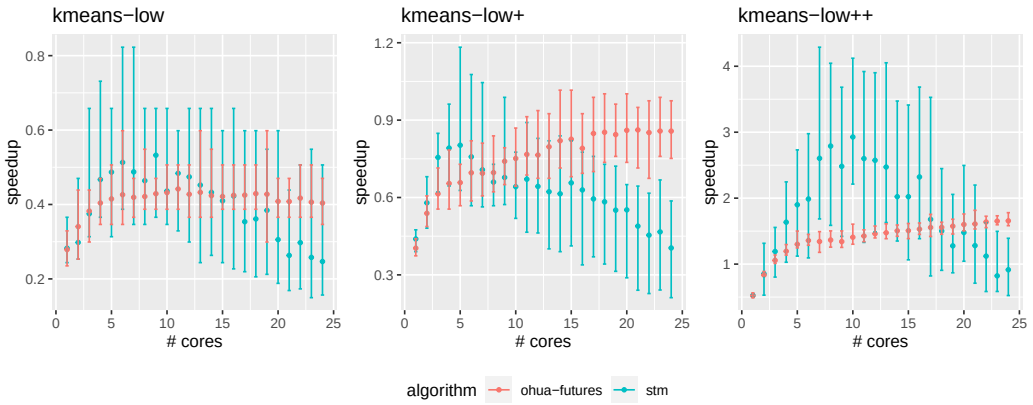


Fig. 6.8: Speedup in the kmeans-low application relative to a sequential implementation.

without locks in Rust, the data must either be copied[2] and sent to the respective threads (making a later consolidation of the changes made necessary) or the vector has to be split apart, each part moved into the scope of the respective thread. Both approaches are time-consuming, as in both cases new memory must be allocated for the new vectors and the data has to be copied or moved, not to mention the time it takes to consolidate the changes made afterwards.

Overall, both benchmark sets show that there is no clear pattern for Ohua's performance with respect to the size of the input set. Each individual set of input data requires a specific number of iterations before the algorithm converges. In the Software Transactional Memory version this number is completely unpredictable, fluctuating due to the order in which the elements are processed, as floating point addition offers only a limited precision and is hence not commutative. Minor differences in the calculation of new centroids therefore can lead to missing the convergence threshold which in turn leads to more computations. Ohua does this in a fixed amount of iterations as it performs all additions deterministically. This becomes visible in Ohua's fairly level performance for each input set. As its determinism guarantees that the computations performed are always in the same order, stripped of any fluctuations caused by non-determinism, speedups and slowdowns mark where the framework either helps to improve performance or where it weighs down execution times.

When comparing the utilized CPU times by both frameworks in Figs. 6.7 and 6.9, we see that Ohua uses a relatively steady amount of computation time throughout all measurements for kmeans, while STM needs linearly more computation times with increasing thread count. This can be attributed to an increased number of conflicts due to higher contention, the latter also showing in degrading speedups with increasing thread counts. All in all, Ohua is on par or outperforms Software Transactional Memory in the smaller two of both kmeans input sets and performs steadily for both large inputs, all while utilizing only a fraction of the CPU time STM needs, meaning it is also using significantly less energy to produce its results.

[2] In Rust jargon, this is called a *clone*.

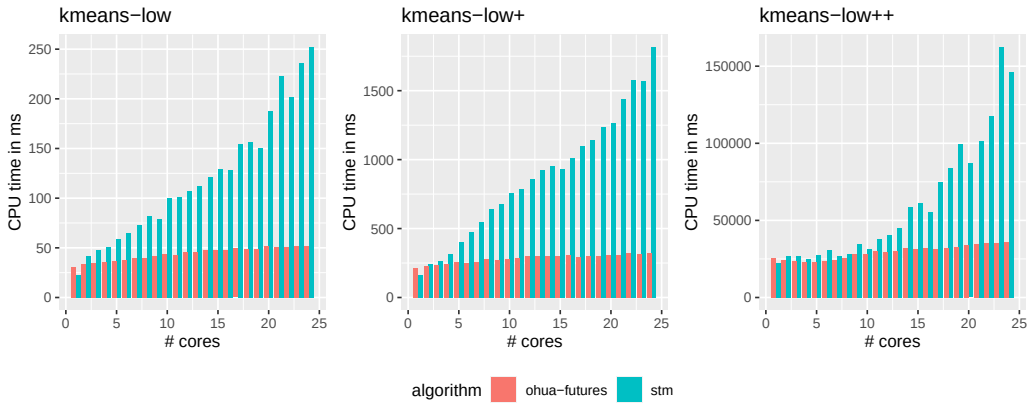


Fig. 6.9: CPU time used by both frameworks in the kmeans-low application.

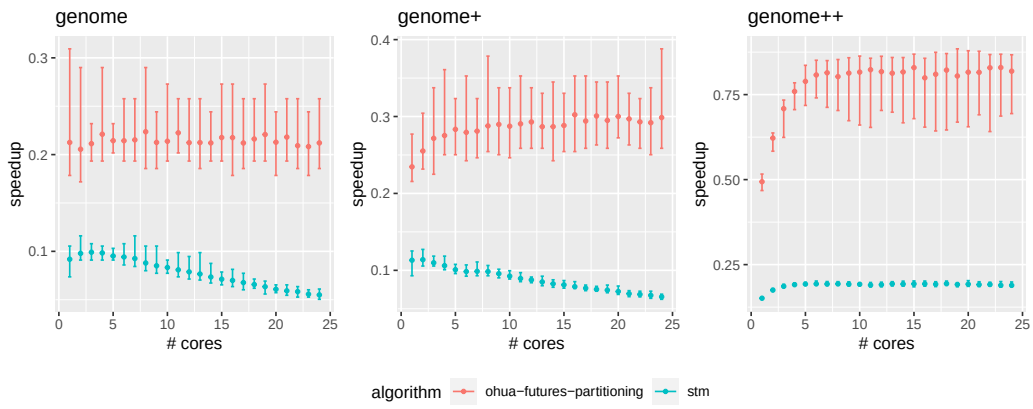


Fig. 6.10: Speedup in the genome application relative to a sequential implementation.

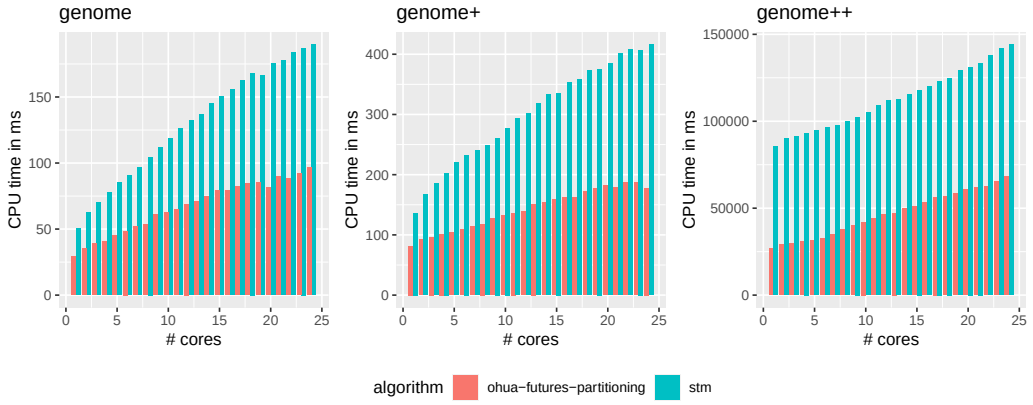


Fig. 6.11: CPU time used by both frameworks in the genome application.

genome. In *genome*, the curve shapes of our STAMP results from Fig. 6.1 and the results from the STM version in Rust in Fig. 6.10 are similar in the way that they exhibit peak performance for a medium amount of threads before it declines. What's striking is the difference in absolute speedups achieved by our Rust implementation. Likely causes for this include a by comparison more performant sequential implementation, which is possible as we developed these binaries independently from the other implementations which use one of the two frameworks while Minh et al. basically reuse the STM implementation, only removing any TM and threading code from it. Alternatively, it could be because of higher overheads of the `rust-stm` library, for reasons we have outlined above. To test this theory, we profiled a STM `genome+` run using 8 threads and analyzed the resulting flamegraph [Gre16]. We found that the cloning of data amounts for about 23,75 % of the whole execution time of the benchmark[3]. While a more efficient library implementation using Rust's *unsafe* features that allow circumventing certain safeguards could may help to improve performance, we deemed time-consuming improvements on a STM framework out of scope for this work. Ohua manages to achieve clearly better speedups in all three test cases. The execution times remain relatively similar, indicating a steady amount or overhead, although we see an increasing CPU usage in Figure 6.11, indicating the increasing framework overhead. But due to the large portions of still sequential code, i.e., the nucleotide sequence deduplication, no real performance gains can be made.

6.3 Summary

In general, the experiments we've conducted have shown that Ohua's deterministic execution model does indeed lead to less variance in execution times. Our achieved results are on par with those of STM in applications like the labyrinth benchmark, where it was able to break up the stateful loop using Transformation 2 and 3. Technically, we also outperformed Software Transactional Memory in the intruder and genome benchmarks but as STM achieved in both cases subpar results, it should be evaluated whether the STM library used can be optimized for a performance comparable to the C-based version or whether this is not an implementation-related but rather a framework-related issue before judging about this.

With the kmeans and genome benchmarks, it became apparent that not all shared state applications offer the same amount of parallelism exploitable using the transformations we proposed in Chapter 4. In fact, it became apparent that of the various forms of irregular applications tested, the one exhibiting amorphous data parallelism could be parallelized best: the labyrinth benchmark. This is the only one of the four algorithms we implemented that displays the characteristics of amorphous data parallelism we explained in Chapter 2.1.1. That may indicate that Ohua would be most effective when targeting this type of applications in particular, as our batching

[3] For this analysis we profiled the whole application execution. Of the overall execution time, only 46,06 % of all samples were attributed to the execution of the benchmark itself. 10,94 % of the total sample count were attributed to data cloning and also within the benchmark execution part of the program. Hence, roughly 23,75 % of the benchmark execution time is spent duplicating data.

Transformation 2 and 3 presented in chapters 4.3 and 4.4 enable us to handle the state updates found in this type of applications. To verify this, future work should implement other programs from the STAMP suite that contain amorphous data parallelism such as the *yada* and *ssca2* algorithms. But even in the non-amorphous cases we tested Ohua on, we saw a significantly lower CPU usage, and by extension power consumption, for Ohua than for STM. The absence of contention in Ohua's model is the key reason for this conservation of resources. Hence, the fact that the framework is more limited in what parallelism may be exploited does not seem to be a downside but rather an advantage. Such state-modification-only loops contain usually more contention as transactions are much shorter and commits occur more frequently. Exploiting parallelism from these loops hence has the potential of yielding a negative instead of a positive performance impact, as more write conflicts and recomputations occur.

Overall, Ohua would indeed seem to be a promising replacement for STM-based shared state applications. While it is clearly not suited to be used for every type of shared state program, so is STM. Performance-wise Ohua will not always outperform Software Transactional Memory but manages to at least be on par with it, while yielding other positive properties like a deterministic execution model, often better energy efficiency, and easier to work with code bases due to the elimination of parallelism abstractions. Based on the benchmark results we assume, that Ohua will showcase the best performance behavior when used in applications with amorphous data parallelism as this allows it to not just parallelize state-free loops but also stateful loops using Transformation 2 and 3.

7 Related Work

Before Ohua, various other frameworks for parallel programming have been proposed that could serve as a replacement for *Software Transactional Memory* in a shared state scenario to mitigate some of the shortcomings of the original proposal by Shavit et al. [ST97]. This chapter will briefly present some of this related work.

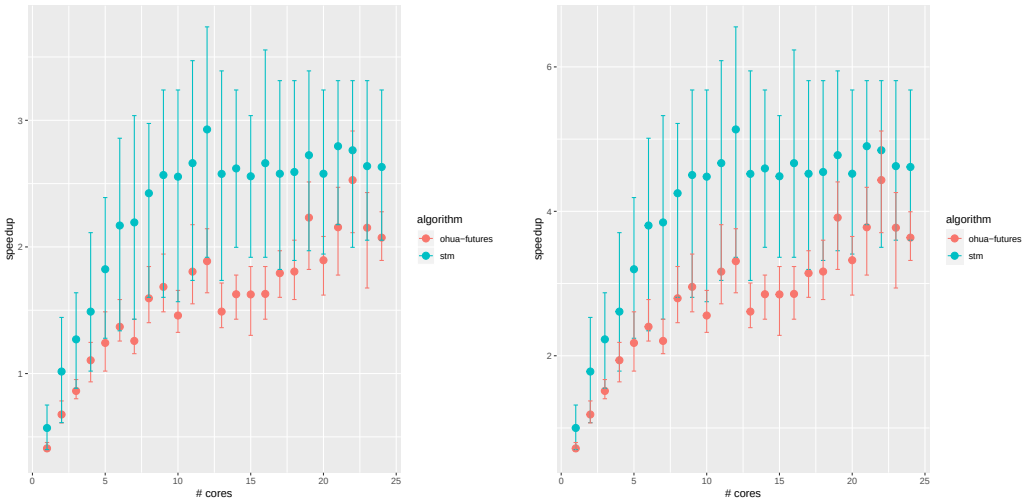
7.1 Chocola: Combining multiple concurrency models

According to Van Roy et al. [VH+04], most concurrency models can be grouped into three different categories: deterministic, message-passing and shared memory models. In a study, Swalens et al. [SDD18] found that developers often employ multiple concepts from different categories to solve their tasks. They regard this interleaving of multiple concepts as highly dangerous, as each concurrency model comes with its own set of restrictions on what may or may not be done in the program in order to uphold their individual guarantees. Mixing two or more models in an application may void some of these guarantees, a fact most developers are not aware of, as the semantics of this nesting are usually not well-defined. For example, transactions (a shared-memory model) provide isolation[1] as a guarantee. However, if futures (a deterministic model) are used inside of transactions, this guarantee is voided, unbeknownst to the developer.

As a solution, and to account for this culture of mixing several concurrency models, the authors propose *Chocola* [SDD18], a unified framework of futures, transactions and actors (a message-passing model). Chocola is a fork of the Clojure programming language that comes with native support for all three concepts. In their work, Swalens et al. provide well-defined semantics for the combination of two concurrency models. When defining and implementing their language, they attempted to uphold as much of the original guarantees the individual models offered as possible when combining them, e.g., by altering how futures behave inside a transaction. For the most part they succeeded, although determinacy was a guarantee they could often not retain.

To evaluate their work the authors also re-implemented a subset of the STAMP benchmark suite using Chocola. However, they only tested 4 out of 8 benchmarks, as they argue that only the four selected ones offered any potential for enhancements by combining transactions with another concurrency model. Swalens et al. reported speedups of 2.3 for a Chocola-based labyrinth implementation, as compared to a

[1] Isolation ensures, that one transaction can never see the changes made by another transaction until the latter has committed. Various levels of isolation have been defined, but Swalens et al. [SDD18] were only interested in serializability.



(a) Sequential baseline.

(b) Single-threaded STM as baseline.

Fig. 7.1: Speedups of Ohua and STM for the labyrinth benchmark with different baselines.

speedup of 1.3 for a conventional STM implementation. Nevertheless, it remains unclear how the authors achieved these measurements, as this is not discussed in their work. Judging from their references to previous work [SDD16; SDD17], these speedups most likely refer to a single-threaded run of their STM implementation. But as McSherry et al. [MIM15] pointed out in their work, choosing the performance of another implementation as reference point gives no indication of the real performance of both approaches. In real application scenarios, parallelism is introduced to supercede an existing sequential implementation because the latter can usually only be optimized to a certain point before reaching its performance limit. When setting a single-threaded run of any framework as baseline, one can only make a comparison between relative speedups to other frameworks, leading to distorted plots as can be seen in Fig. 7.1.

Both plots show the results of the labyrinth+ benchmark as explained in Chapter 5.1.2 and evaluated in Chapter 6.2. Figure 7.1a shows our original speedups, comparing both implementations to a seuential baseline implementation. Fig. 7.1b on the other hand shows the same results but with the single-threaded STM run as baseline, nearly doubling the apparent speedups. Hence, to obtain realistic speedups and account for the potential overheads of various frameworks, it is favorable to compare against a sequential baseline. Swalens et al. probably decided against doing so as it would have required them to write a third, sequential version of their benchmarks. This demonstrates another strength of Ohua: Due to its implicit nature, any algorithm definition may simultaneously serve as sequential implementation by simply executing the application as-is, without invoking the Ohua compiler first. But due to this bad baseline choice, we cannot compare our results to those Chocola achieves.

A significant advantage of the authors' approach is, that it allows the combination of several different concurrency models while offering well-defined and formalized semantics. This makes the development of complex software easier for developers as

they do not have to put too much thought in the ramifications of this combination. On the other hand side is this offering of multiple concurrency models also the frameworks' hugest disadvantage. Developers now don't have to learn how to correctly use one, but three different concepts with different guarantees attached. Resulting code is so extremely tailored towards the use of these different concepts and their interaction between one another that migrating the code to another framework becomes virtually impossible, depending on how many of Chocola's features are used. Applications developed with this framework also contain many (potentially different) concurrency abstractions. In that regard, developing parallel programs might become easier, but understanding and reasoning about them becomes much harder, especially compared to Ohua, which does not expose any abstractions and shields the developer completely from having to reason about concurrency, as this is exploited at compile-time.

7.2 Software Lock Elision

Roy et al. [RHH09] detail in their work the problems of migrating lock-based code to a Transactional Memory framework. If done incorrectly, the semantics of the code may change due to the different behavior of transactions compared to locks, leading to different results. Hence, they propose a Software Lock Elision runtime that builds upon lock-based code and allows threads to speculatively execute lock-based critical sections in parallel. This framework features an optimistic execution model and detects conflicts between accesses by concurrent threads dynamically. In this regard, it functions similar to Software Transactional Memory. But additionally, Software Lock Elision takes a best effort approach on its implementation: A fallback to acquiring locks without any speculation is possible, e.g., when the speculative state overflows the cache, when using nested locks or when performing an operation that only works non-speculatively (such as waiting for a condition variable). Roy et al. state that their system is only tailored towards workloads with high contention and a low number of conflicts, as only then using locks becomes a liability, producing too much overhead.

As can be seen in Fig. 7.2, their design features both the automatic and the manual use of SLE. A developer may either annotate her lock-based code to explicitly use Software Lock Elision at certain points or profile the locks in her binary and apply binary rewriting systems to add a SLE runtime to the application[2]. The runtime is designed to retain maximal fairness towards non-speculative locks by prioritizing them over threads holding the same lock speculatively. The authors also tried to ensure the isolation between speculative and non-speculative work and to add support for application-specific lock types.

[2] In their work, Roy et al. discuss the design for an automated SLE runtime but left an actual implementation to future work. As of today, no follow-up work detailing an actual implementation has been provided.

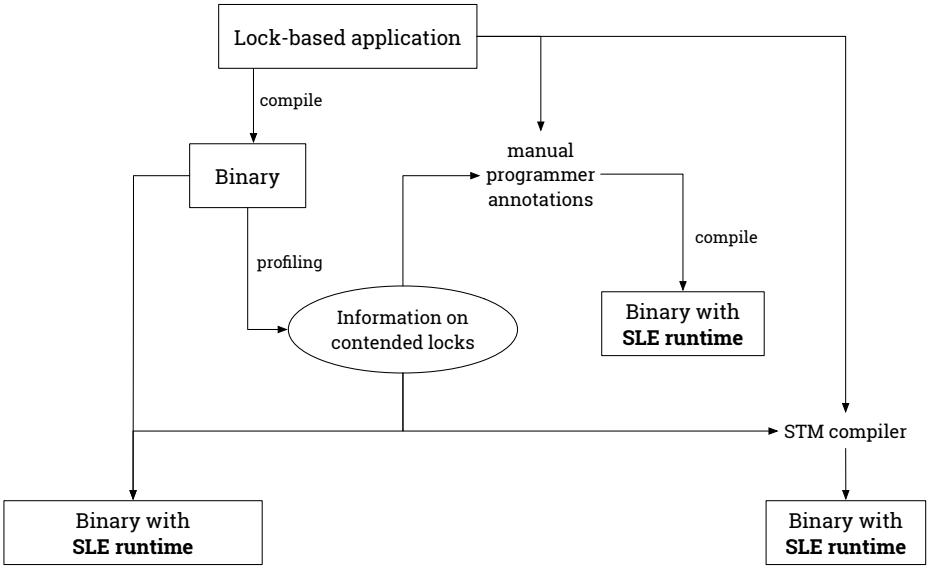


Fig. 7.2: The workflow of the Software Lock Elision runtime. Adapted from Roy et al. [RHH09].

To evaluate their work, Roy et al. implemented 6 out of 8 STAMP benchmarks[3] using manually-annotated SLE code. For the *kmeans*, *intruder* and *ssca2* benchmarks the authors reported speedups 1.5 to 2 times higher than those of the original STM implementation. However, the SLE runtime failed to achieve good results for *genome*, *labyrinth* and *vacation*, sometimes only reaching 25-50 % of the STM speedup. Interestingly, the framework only achieved good performances in benchmarks with short transactions and at most a medium amount of time spent in transactions, as can be seen in table 5.2.

Overall, this type of automated approach to enhance lock-based code with speculation looks very promising. It allows the use of STM concepts within lock-based code, which enables developers to easily improve performance of their existing code. But this requires the use of lock-based code. The authors make the correct usage of locks a prerequisite in their work, although this is the hardest aspect of lock-based programming, as we discussed in Chapter 2.2. Therefore, an abstraction-free approach to concurrent programming, like Ohua, might be preferable to avoid having developers write locking code manually.

7.3 Heterogeneous Parallelism

Fluet et al. [Flu+07b] argue, that currently, no parallel programming language exists, that can be adequately used for general purpose programming, as most of these languages are tailored towards research purposes. Additionally, they lament the missing parallelism features in general purpose programming languages. In their opinion, parallel languages need to provide mechanisms for multiple levels of parallelism, since most applications exhibit parallelism at multiple levels that

[3] They reported 2 benchmarks, *yada* & *bayes*, to not run on their systems anymore.

could be exploited for performance and since most commodity hardware achieves optimal performance only when exploiting these multiple parallelism levels (e.g., by employing both threads and SIMD parallelism). In response, the authors propose *Manticore* [Flu+07b; Flu+07a; Flu+09; Flu+10], a parallel language for heterogeneous parallelism.

The language belongs to the family of statically-typed strict functional languages and bases its concurrency mechanisms on Concurrent ML. Coarse-grain parallelism in Manticore is achieved using explicit mechanisms like `spawn` for thread creation and typed channels. An important feature of the language is the absence of shared state. Instead, threads may only use message passing via aforementioned channels for communication and synchronization. As opposed to coarse-grain parallelism, the fine-grain parallelism is achieved using implicit mechanisms, because Fluet et al. argue that this type of parallelism is cumbersome to implement by hand and can easily pose large overheads when done incorrectly. For this, they use constructs like parallel arrays (immutable sequences that can be computed in parallel), which are all created or invoked using special annotations.

```
1 fun imgToGray img =
2     [: [: rgbToG pix | pix in row :]
3         | row in img :]
4 fun convert img = let
5     val replCh = channel()
6     in
7         spawn (send (replCh, imgToGray img));
8         recvEvt replCh
9     end
```

Listing 7.1: Code example showcasing implicit and explicit parallelism in Manticore, adapted from [Flu+07b].

Listing 7.1 features parallel comprehension syntax as an example for such annotations in the function `imgToGray`, also showing that these parallel list comprehensions can also be nested. The function `convert` on the other hand shows the use of explicit parallelism for coarse-grain parallelism. To efficiently harness the resulting parallel code, Manticore includes a continuation-based runtime that caters to the different scheduling needs of explicit and implicit threads.

Although Fluet et al. have been continuously working on Manticore and published several follow-up works, no benchmarks have been conducted with the language yet. Nonetheless is the author's proposition very interesting as it is also relying on finding and extracting implicit parallelism, like Ohua. Unfortunately, they only employ this idea for lower-level parallelism, leaving much of the parallelization effort to the developer, who may not be able to parallelize the code as efficient as automated approaches. Another shared approach with Ohua is the elimination of shared state, by which many types of errors in parallel code can be ruled out.

7.4 Flexible Parallel Execution

The approach that is perhaps the most similar to Ohua's approach is *Parcae* [Ram+12] by Raman et al. Like us, they opted for an automatic approach for their work, eliminating the need for explicitly managing threads. Though, their motivation lies within the fact that most parallelization efforts target only a static, anticipated number of environments, outside which a programs performance may decrease drastically. Parcae was designed to circumvent that issue by allowing it to automatically tune the program dynamically to react to any changes in the environment at runtime.

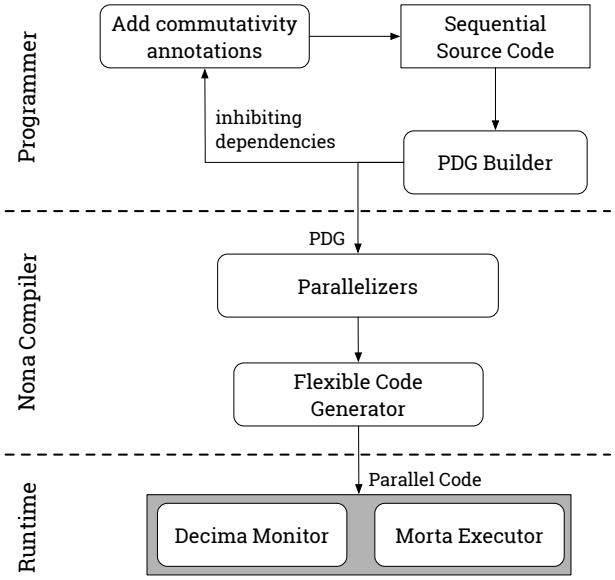


Fig. 7.3: Architecture and Workflow of the Parcae framework, adapted from [Ram+12].

These flexible parallel programs, as the authors call them, are generated by the Nona compiler, one of the three components comprising the Parcae framework, which is shown in Fig. 7.3. The compiler takes a sequential program as input and identifies parallelizable regions within it by building a program dependence graph, to which some node reordering and regrouping operations are applied, which might require further annotations from the developer to highlight commutative operations. It then applies the parallelizing transforms to reach of these regions, although currently only loop nests are in scope for this step. Such transformations encompass data-parallel transforms with critical sections and pipeline transforms but developers may expand this list at their choosing. Nona then creates several tasks, each containing a loop body, which are marked either for parallel or sequential execution. The resulting program contains numerous hooks for profiling and a set of tasks which are to be executed. At runtime, the Morta executor takes care of executing the program and finding a parallelism configuration that is optimal for the execution environment. This happens by first executing a sequential version of the parallel regions and monitoring the execution using the third component, the Decima Monitor. Based on the gathered information, the execution scheme of the program is adjusted to optimize performance, which becomes a continuous process throughout execution.

To evaluate their work, Raman et al. chose applications from a variety of benchmark suites and implemented them using Parcae. In the *kmeans* benchmark, which was chosen as representative from the STAMP suite, Parcae was able to improve performance by about 42 % compared to a baseline parallel execution while only consuming 84 % of the power a baseline parallel implementation consumed.

When comparing Parcae with Ohua, one can find many similarities in the languages approaches, although the initial motivation behind both concepts is different. One of the most prominent differences between both frameworks on the other hand is that Parcae requires much scheduler knowledge and has a high degree of dynamicity in the runtime, imposing non-negligible overheads during execution, which the authors also acknowledged. Ohua solves most of these issues as compile time by creating a runtime where data dependencies dictate when a tasks has to be executed sequentially or whether a parallel execution with other operators is possible. For scheduling, we currently mostly rely on the Operating System scheduler, avoiding costly reconfigurations at runtime. Also, we do not require annotations of any sort in the source code, but rather deduce all necessary information from the Data Flow Graph. While we do not achieve as good performance improvements for the *kmeans* application, we manage to only use about 5-20 % of the CPU time STM uses, indicating higher energy savings than reported for Parcae.

8 Future Work

In this work, we put the Rust backend to the Ohua framework to test for the first time, to see whether it could be developed into a sustainable alternative to Software Transactional Memory. This chapter will list some interesting directions for future work we have discovered in this thesis.

First and foremost, our work provided a theoretical description of four transformations that were only implemented manually for the benchmarks shown in the Chapters 5 and 6. Hence, as a first step these transformations should be incorporated in the compiler framework to allow further work building on them.

One of the first main discoveries we have made was that Ohua introduces overheads into the runtime that can become quite significant when an application only contains little parallelism to exploit. When large portions of the data flow graph are sequential, it does not make sense to introduce a new operator for each of these sequential steps. Hence, future work should focus on identifying these sequential groups of operators at compile time and potentially fuse them into a single operator in the Ohua runtime, thus reducing the number of created threads. This could directly help to reduce overhead of the Ohua runtime while also addressing a problem that Cascaval et al. [Cas+08] put forth in their article as they argued that high framework overheads among other things make STM only suitable as „a research toy“.

In Chapter 4 we presented a number of transformations for Ohua to conduct on Expression IR level. We briefly discussed the correctness of our propositions but future work should also work to provide a formal proof for these transformations where possible in order to verify their properness.

Using the Transformations 2 and 3 we presented in Chapter 4.3 and 4.4 we were able to uncover opportunities for implicit parallelism in applications by isolating state-free loops. Our results showed that this was key to improve Ohua's performance in applications exhibiting amorphous data parallelism but was not applicable to other irregular applications. Therefore, future work could investigate whether it is possible to extract more implicit parallelism opportunities from these non-amorphous irregular programs to allow Ohua to achieve better results in applications like *kmeans* or *genome*. We believe that further research in this direction can improve the understanding for Ohua's limits as well as its model of implicit parallelism and local state as many of the targeted applications build heavily on shared state which hinders simple parallelization efforts.

Another direction for future research should be the reinforcement of the results we saw in Chapter 6.2. The STAMP suite contains more applications with amorphous data parallelism like *ssca2* and *yada*, which should be implemented in Ohua to attempt to reproduce the performance we saw in the labyrinth benchmark and verify

the applicability of our Transformations 2 and 3. At the same time, one could also check, whether the rust-stm library [@Ber20] used by us could be optimized to yield performances comparable to the C implementation in benchmarks like *intruder* and *genome*.

Finally, future work could discuss the question whether the STAMP benchmark suite is still adequate to use today. Many other researchers draw on these benchmarks when evaluating their work on transaction-related research or irregular applications. But as we have found, many of these applications today yield speedups below 1, even when using the original code base for testing. This raises the question whether it actually still makes sense to use this suite as is. Minh et al. prided themselves on having chosen real-world applications to compile STAMP, which is a good idea and the selection made sense back then as STM and HTM were indeed able to overcome the more limited clock speeds and speed up program execution. Today however, no one would use STM for most of these algorithms anymore, given the latest benchmark results. Hence, we propose that a new benchmark suite should be compiled, building on the principal idea of Minh et al. and potentially salvaging some of their applications. Special consideration should be put into the fact that most, if not all, shared state applications are irregular applications, meaning that some of them will exhibit amorphous data parallelism. To allow for better test coverage, this property should be incorporated into a potential new suite.

9 Conclusion

The Ohua framework presents a novel approach to writing parallel programs. By relieving the developer of the burden of having to use abstractions for introducing parallelism into the program, she is able to write concise and less complex algorithms which are easier to maintain and verify. Parallelism is instead introduced into the application by the Ohua compiler itself, which extracts it from the algorithm using a set of transformations. State sharing however poses a challenge for Ohua, as its programming model only fosters local state and offers no synchronization primitives whatsoever.

In this thesis, we have shown that the framework nonetheless may also be used for implementing shared state applications. After showing that Ohua is able to extract non-trivial parallelism from such a program in a preliminary study, we proposed a set of compiler transformations that could help leverage parallelism in stateful loop operations which are often encountered in shared state environments. Using these transformations, we implemented a selection of benchmarks from the STAMP benchmark suite. We chose a representative subset of applications based on the authors' categorization and used them to compare Ohua against Software Transactional Memory, a widely adopted framework for writing shared state applications. Our benchmark results showed that Ohua could indeed be a viable alternative to STM. Its performance was widely on par with STM, although results varied per application, while offering advantages in terms of code conciseness and verifiability. We found that in many examined shared state applications Ohua is unable to break up loops using shared state to exploit parallelism beyond state-free loops, as these loops are often only consisting of state modifications. STM is able to break these patterns up using its speculative nature, but often at the cost of many retried computations. Ohua's determinism and reduced contention improved the applications' execution times, managing to compensate the missing parallelism in these sections. However, the most performance gains and the best scaling behavior for Ohua were seen in applications exhibiting amorphous data parallelism. As these programs change their behavior when parallelizing them anyway, Ohua may apply slightly more aggressive transformations, allowing it to extract more parallelism than in non-amorphous applications. Hence, we hypothesize that Ohua will be the most effective when used on amorphous data parallel shared state applications.

Future work should mainly focus on validating this theory while also proving the formal correctness of our proposed transformations.

Acknowledgements

This work would not exist, if it were not for the extensive support I received while writing it. First and foremost, I want to thank my supervisor, Sebastian Ertel, for supporting me while I researched this topic, encouraging me to continuously question the results I achieved and getting me back on track when I got lost in situations where nothing seemed to work.

My supervisors, Jeronimo Castrillon and Michael Roitzsch, who both encouraged me to take the time I needed to finish this thesis when I fell into the low that was the unexpected *working from home* situation that costed me more time than I had anticipated.

Nancy for her continuous unconditional support while I worked through many nights when creativity struck me and I remained absent from numerous joint evenings.

And finally, the people at the chair for compiler construction for offering refreshing and creative inputs to my thesis, for showing me that research is fun and also for providing me with an office space to work in.

Bibliography

[Ada19] Justus Adam. „Ohua-powered, Semi-transparent UDF's in the Noria Database". MA thesis. TU Dresden, Nov. 2019 (cit. on p. 9).

[AF97] ZENA M. ARIOLA and MATTHIAS FELLEISEN. „The call-by-need lambda calculus". In: *Journal of Functional Programming* 7.3 (1997), pp. 265–301 (cit. on p. 21).

[Ari+95] Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. „A Call-by-Need Lambda Calculus". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: Association for Computing Machinery, 1995, pp. 233–246 (cit. on p. 21).

[BS81] F Warren Burton and M Ronan Sleep. „Executing functional programs on a virtual tree of processors". In: *Proceedings of the 1981 conference on Functional programming languages and computer architecture*. 1981, pp. 187–194 (cit. on p. 18).

[Cas+08] Calin Cascaval, Colin Blundell, Maged Michael, et al. „Software transactional memory: Why is it only a research toy?" In: *Queue* 6.5 (2008), pp. 46–58 (cit. on pp. 1, 8, 57).

[EFF15] Sebastian Ertel, Christof Fetzer, and Pascal Felber. „Ohua: Implicit Dataflow Programming for Concurrent Systems". In: *Proceedings of the Principles and Practices of Programming on The Java Platform*. PPPJ '15. Melbourne, FL, USA: ACM, 2015, pp. 51–64 (cit. on pp. 2, 9).

[Ert+18] Sebastian Ertel, Andrés Goens, Justus Adam, and Jeronimo Castrillon. „Compiling for concise code and efficient I/O". In: *Proceedings of the 27th International Conference on Compiler Construction*. 2018, pp. 104–115 (cit. on p. 21).

[Ert+19] Sebastian Ertel, Justus Adam, Norman A. Rink, Andrés Goens, and Jeronimo Castrillon. „STCLang: State Thread Composition as a Foundation for Monadic Dataflow Parallelism". In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*. Haskell 2019. Berlin, Germany: ACM, Aug. 2019, pp. 146–161 (cit. on pp. 9, 23).

[Ert19] Sebastian Ertel. *Towards Implicit Parallel Programming for Systems*. Dresden, 2019 (cit. on p. 8).

[Flu+07a] Matthew Fluet, Nic Ford, Mike Rainey, et al. „Status report: the manticore project". In: *Proceedings of the 2007 workshop on Workshop on ML*. 2007, pp. 15–24 (cit. on p. 54).

[Flu+07b] Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. „Manticore: A heterogeneous parallel language“. In: *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*. 2007, pp. 37–44 (cit. on pp. 53, 54, 68).

[Flu+09] Matthew Fluet, Lars Bergstrom, Nic Ford, et al. „Programming in Manticore, a heterogenous parallel functional language“. In: *Central European Functional Programming School*. Springer. 2009, pp. 94–145 (cit. on p. 54).

[Flu+10] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. „Implicitly threaded parallelism in Manticore“. In: *Journal of functional programming* 20.5-6 (2010), pp. 537–576 (cit. on p. 54).

[Gre16] Brendan Gregg. „The flame graph“. In: *Communications of the ACM* 59.6 (2016), pp. 48–57 (cit. on p. 48).

[Hal84] Robert H Halstead Jr. „Implementation of Multilisp: Lisp on a multiprocessor“. In: *Proceedings of the 1984 ACM Symposium on LISP and functional programming*. 1984, pp. 9–17 (cit. on p. 18).

[HVG04] Bart Haagdorens, Tim Vermeiren, and Marnix Goossens. „Improving the performance of signature-based network intrusion detection sensors by multithreading“. In: *International Workshop on Information Security Applications*. Springer. 2004, pp. 188–203 (cit. on p. 31).

[Kul+07] Milind Kulkarni, Keshav Pingali, Bruce Walter, et al. „Optimistic parallelism requires abstractions“. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2007, pp. 211–222 (cit. on pp. 1, 5, 8).

[Kul+09] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. „How much parallelism is there in irregular applications?“ In: *ACM sigplan notices* 44.4 (2009), pp. 3–14 (cit. on pp. 1, 4, 5).

[Lee06] Edward A Lee. „The problem with threads“. In: *Computer* 39.5 (2006), pp. 33–42 (cit. on pp. 1, 6).

[Lee61] Chin Yang Lee. „An algorithm for path connections and its applications“. In: *IRE transactions on electronic computers* 3 (1961), pp. 346–365 (cit. on p. 11).

[LWH16] Adam Lackorzynski, Carsten Weinhold, and Hermann Härtig. „Decoupled: low-effort noise-free execution on commodity systems“. In: *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*. 2016, pp. 1–8 (cit. on p. 27).

[Mac+67] James MacQueen et al. „Some methods for classification and analysis of multivariate observations“. In: *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*. Vol. 1. 14. Oakland, CA, USA. 1967, pp. 281–297 (cit. on p. 33).

[MIM15] Frank McSherry, Michael Isard, and Derek G. Murray. „Scalability! But at what COST?“ In: *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, May 2015 (cit. on p. 51).

[Min+08] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. „STAMP: Stanford transactional applications for multi-processing“. In: *2008 IEEE International Symposium on Workload Characterization*. IEEE. 2008, pp. 35–46 (cit. on pp. 1, 13, 28, 29, 36, 37, 39, 42).

[Nar+06] Ramanathan Narayanan, Berkin Ozisikyilmaz, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. „Minebench: A benchmark suite for data mining workloads“. In: *2006 IEEE International Symposium on Workload Characterization*. IEEE. 2006, pp. 182–188 (cit. on p. 33).

[Ous+13] Kay Ousterhout, Aurojit Panda, Joshua Rosen, et al. „The case for tiny tasks in compute clusters“. In: *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*. 2013 (cit. on p. 19).

[Per+08] Cristian Perfumo, Nehir Sönmez, Srdjan Stipic, et al. „The limits of software transactional memory (STM) dissecting Haskell STM applications on a many-core environment“. In: *Proceedings of the 5th conference on Computing frontiers*. 2008, pp. 67–78 (cit. on pp. 1, 39).

[Pin+09] Keshav Pingali, Milind Kulkarni, Donald Nguyen, et al. „Amorphous data-parallelism in irregular algorithms“. In: *The University of Texas at Austin, Department of Computer Sciences, Austin, TX, USA* (2009) (cit. on p. 5).

[PSS02] Mihai Pop, Steven L Salzberg, and Martin Shumway. „Genome sequence assembly: Algorithms and issues“. In: *Computer* 35.7 (2002), pp. 47–54 (cit. on p. 34).

[Ram+12] Arun Raman, Ayal Zaks, Jae W Lee, and David I August. „Parcae: a system for flexible parallel execution“. In: *ACM SIGPLAN Notices* 47.6 (2012), pp. 133–144 (cit. on p. 55).

[RHH09] Amitabha Roy, Steven Hand, and Tim Harris. „A runtime system for software lock elision“. In: *Proceedings of the 4th ACM European conference on Computer systems*. 2009, pp. 261–274 (cit. on pp. 52, 53).

[SDD16] Janwillem Swalens, Joeri De Koster, and Wolfgang De Meuter. „Transactional Tasks: Parallelism in Software Transactions“. In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2016 (cit. on pp. 7, 11, 12, 31, 51).

[SDD17] Janwillem Swalens, Joeri De Koster, and Wolfgang De Meuter. „Transactional actors: communication in transactions“. In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems*. 2017, pp. 31–41 (cit. on p. 51).

[SDD18] Janwillem Swalens, Joeri De Koster, and Wolfgang De Meuter. „Chocola: integrating futures, actors, and transactions“. In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. 2018, pp. 33–43 (cit. on p. 50).

[ST97] Nir Shavit and Dan Touitou. „Software transactional memory“. In: *Distributed Computing* 10.2 (1997), pp. 99–116 (cit. on pp. 1, 6, 50).

[VH+04] Peter Van-Roy, Seif Haridi, et al. *Concepts, techniques, and models of computer programming*. MIT press, 2004 (cit. on p. 50).

[WKL07] Ian Watson, Chris Kirkham, and Mikel Luján. „A study of a transactional parallel routing algorithm“. In: *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*. IEEE. 2007, pp. 388–400 (cit. on p. 11).

Web pages

[@Ber20] Gunnar Bergmann. *rust-stm: Software transactional memory*. July 2020. URL: `https://github.com/Marthog/rust-stm` (cit. on pp. 12, 30, 58).

[@con20] The tokio contributors. *Tokio - The asynchronous run-time for the Rust programming language.* Apr. 2020. URL: `https://tokio.rs` (visited on Apr. 24, 2020) (cit. on p. 18).

[@EWA19] Sebastian Ertel, Felix Wittwer, and Justus Adam. *ohua-rust-runtime: A runtime for Ohua in Rust.* July 2019. URL: `https://github.com/ohua-dev/ohua-rust-runtime` (cit. on p. 13).

[@Min13] Chi Cao Minh. *stm: STAMP source code and tl2 library*. Sept. 2013. URL: `https://github.com/robert-schmidtke/stm` (cit. on p. 36).

[@Wit20a] Felix Wittwer. *Genome benchmark from STAMP suite not working (anymore)*. May 2020. URL: `https://github.com/robert-schmidtke/stm/issues/1` (cit. on p. 36).

[@Wit20b] Felix Wittwer. *Labyrinth application sometimes yields incorrect results*. July 2020. URL: `https://github.com/kozyraki/stamp/issues/2` (cit. on pp. 1, 36).

[@Wit20c] Felix Wittwer. *ohua-rust-benchmarks: Benchmarks to measure the performance of the ohua-rust runtime*. Apr. 2020. URL: `https://github.com/Feliix42/ohua-rust-benchmarks` (cit. on p. 30).

[@Wit20d] Felix Wittwer. *rust-stm: Software transactional memory*. Mar. 2020. URL: `https://github.com/Feliix42/rust-stm` (cit. on p. 30).

[@Wit20e] Felix Wittwer. *Segmentation fault within the intruder benchmark*. July 2020. URL: `https://github.com/kozyraki/stamp/issues/3` (cit. on p. 2).

[@Wit20f] Felix Wittwer. *stm-datastructures: Datastructures specifically tailored for use with STM.* Mar. 2020. URL: `https://github.com/feliix42/stm-datastructures` (cit. on p. 30).

[@Wit20g] Felix Wittwer. *Two transactions may deadlock*. June 2020. URL: `https://github.com/Marthog/rust-stm/issues/17` (cit. on p. 30).

List of Figures

List of Tables

List of Listings

Declaration

I hereby certify that this thesis has been composed by me and is based on my own work, unless stated otherwise. No other person's work has been used without due acknowledgement in this thesis. All references and verbatim extracts have been quoted, and all sources of information, including graphs and data sets, have been specifically acknowledged.

Dresden, July 20, 2020

Felix Wittwer