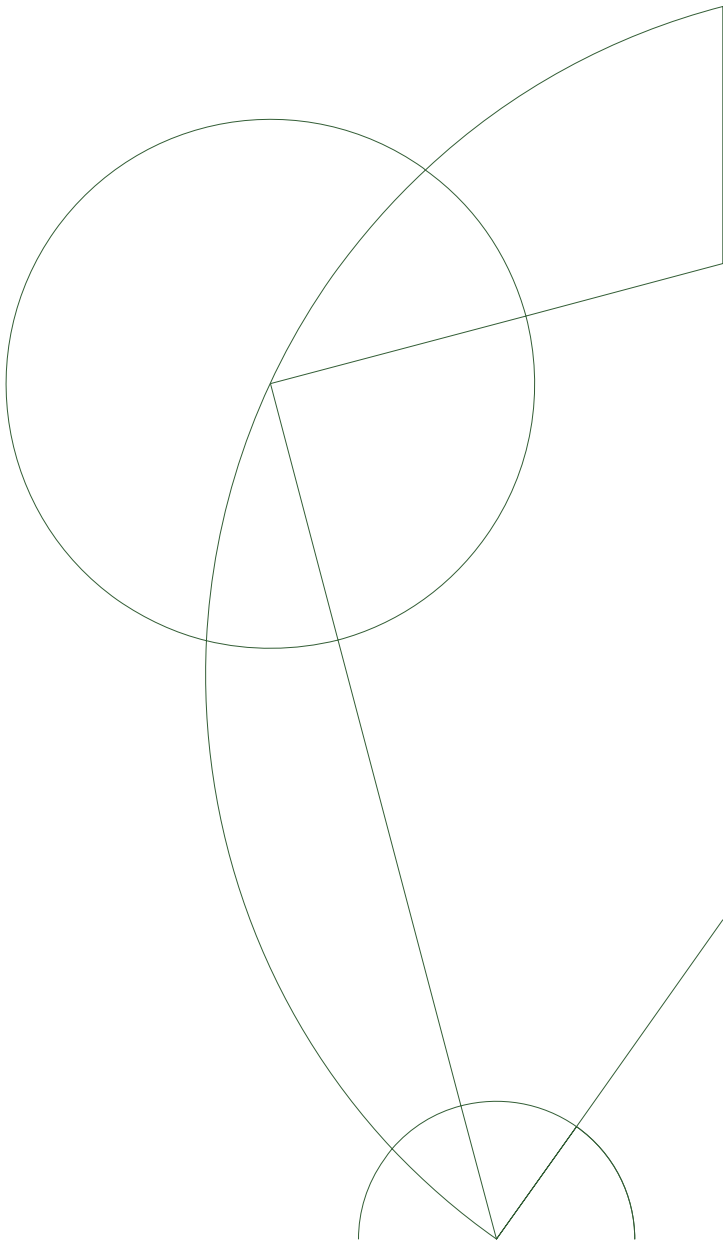
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

MSc Thesis in Computer Science

Duc Minh Tran <cwz688>

A multicore backend for Futhark



Supervisor: Troels Henriksen & Cosmin Eugen Oancea

30th of September 2020

Abstract

Over the last couple of decades, CPUs have seen a fundamental change from single-core to multicore design. This change has led to an increasing interest in parallel programming to take advantage of the multiple cores. As explicit parallel programming remains a challenging and error-prone task, a considerable effort has been made in implicit parallel languages and frameworks, which aims at hiding a lot of the tedious details of parallel programming and let the programmer write code with close to sequential semantics. However, current state-of-the-art languages and frameworks require that the programmer must perform granularity control through a tuning process to ensure good performance. Such a process is labour-intensive and generally not portable.

This thesis presents the design and implementation of a new back-end for the data-parallel language Futhark, targeting multicore CPUs. It aims to make implicit parallel programming simple by using *automatic granularity control*, which requires no tuning from the programmer. Our implementation uses an oracle-guided scheduling approach that through an online algorithm can predict the work (one-core run-time) of parallel for-loops. The oracle can determine the granularity, which amortizes the cost of parallelisation at run-time to effectively perform granularity control for both regular (nested) parallelism and irregular parallelism. To evaluate our approach, we extend the Futhark compiler to generate parallel C code and implement the algorithm in the run-time system of the new back-end.

We evaluate our implementation's performance against Futhark's sequential back-end and show that our implementation can provide a significant speed-up on the majority of existing Futhark programs. We also compare our implementation against established benchmark suites such as FinPar and Rodinia. Here we show that our implementation, in some cases, can provide similar or better performance compared to hand-optimized code without the need for manual tuning. However, our compiler does not consistently generate as efficient code as the hand-optimized benchmarks, which causes our implementation to be slower for some benchmarks.

Keywords: Automatic Granularity Control, Implicit parallelism, Parallel Languages, Futhark

Contents

1 Introduction **3**
1.1 Thesis structure 7

I Background and Related work **8**

2 Background **9**
2.1 Computer architectures and CPUs 9
2.2 Multi-threaded programming model 12
2.3 Parallelism and parallel constructs 16
2.4 Introduction to Futhark 19

3 Implicit parallelism **22**
3.1 Run-time system 22
3.2 Related work 23

II Methods and Implementation **31**

4 Parallel Algorithms for SOACs **32**
4.1 `map` 32
4.2 `reduce` 33
4.3 `scan` 36
4.4 `reduce_by_index` 38

5 Run-time system **44**
5.1 Automatic Granularity Control 44
5.2 Work management 52
5.3 Memory management 57

6 Implementation **58**
6.1 Compiler and Code generation 58
6.2 Scheduler 69

III Experimental evaluation **76**

7 Benchmarks **77**
7.1 Micro-benchmarks 78
7.2 Established benchmarks 81
7.3 Benchmarks against other implementations 91

IV Final remarks **95**

8 Future work and conclusion **96**
8.1 Limitations and future work 96
8.2 Conclusion 97

Chapter 1

Introduction

As CPU frequencies started to stagnate in the mid-2000s, using multicore CPUs has been a critical component in achieving better performance through shared-memory parallelism. However, it remains a difficult task to write explicit parallel programs, as it requires expert knowledge to ensure both correct and efficient programs. As such, a considerable amount of research has been put into implicit parallelism. Implicit parallelism aims to hide many of the tedious details in parallel programming, such as the scheduling of parallel work, and delegate these to the compiler and run-time system. Implicit parallelism has seen much work for both specialized languages and language extensions for parallel systems. Examples of languages which had support for implicit parallelism include Cilk[Blumofe u. a., 1995], Multilisp[Halstead, 1985], NESL[Blelloch u. a., 1993], while extensions include Fork/Join Java[Lea, 2000], OpenMP[OpenMP, 2008] and TBB[Intel, 2011]. These offer support for simple but powerful parallel constructs such as fork-join and parallel for-loops (`parfor`). For example, a "parallel map", which applies a function f to each element of an array `xs`, can be implemented using parallel for-loops. The function f may itself contain parallel for-loops, making it able to express nested parallelism as well.

Listing 1 A parallel map implementation using parallel for-loop

```
1 void parallel_map(T* xs, T* res, int n){
2   parfor (int i = 0; i < n; i++) {
3     res[i] = f(xs[i]);
4   }
5 }
```

To enable parallelization using parallel for-loops the run-time system must create subtasks[1] containing the closure of the function, which can be scheduled onto the computing cores for parallel execution. However, the creation and management of subtasks comes with a cost, i.e., the overheads of paral-

[1] Also known as tasks, threads, or fibers.

lelism. Such overhead can quickly become larger than the cost of execution. For example, consider a function f, which increments the values of `xs` by some constant. On modern computers, such an operation only takes a few cycles, while the creation and management of a single subtask can take on the order of thousands of cycles. If we were to naively create as many subtasks as there are iterations, the overhead would overwhelm the cost of execution by more than 100×. To control such overheads, one must tune the programs to perform *granularity control*, which aims at creating subtasks whose sequential work can amortize the cost of parallelization. For example, for a parallel loop, one must group the iterations into "chunks", which is executed sequentially. However, even when one can control such overheads, further problems arise when one considers *irregular* workloads, for example, when a call to f does not take constant time, but its work is data-dependent. This can lead to uneven workloads across subtasks and underutilization of the resources. Such a case further complicates the tuning process, as in addition to the overhead, it requires finding a partitioning of the iterations, which ensures load-balancing. Current state-of-the-art languages and frameworks require that the programmer must perform the tuning. Such a process is time-consuming and not portable as it depends on the machine, input data, and so on.

In this thesis, we present an implementation for writing implicit parallel programs in the data-parallel array language Futhark, targeting multicore CPUs. Our implementation provides *automatic granularity control*, which requires only an once-per-machine automatic tuning process, and no additional tuning is required from the programmer. Furthermore, our approach to granularity control supports both regular (nested) parallelism and irregular parallelism, such as the case above, where calls to f are not constant time.

Up until this thesis, Futhark was only able to target GPUs (through OpenCL or CUDA) or single-core CPUs (through C). The work in this thesis includes the design and implementation of a completely new back-end for Futhark, which involves the development of a code-generator for the compiler, as well as a run-time system capable of scheduling the generated code in parallel. As Futhark can target multiple platforms, we let the source language remain untouched in this thesis, such that programs can be used across the platforms, without target specific modifications. Instead, we rely on the compiler to perform multicore specific optimizations. Furthermore, as compiler optimization is a perpetual process, our goal is to design our new back-end such that either the compiler or run-time system can be changed in the future without the need to change the other.

We extend the compiler with a new internal representation for generating parallel C code for Futhark's Bird-Meertens operators (`map`, `reduce`, `scan`, etc)[Bird, 1989]. Our implementation use parallel for-loops for expressing parallelism and involves lifting local code to global functions, which can be encapsulated into a closure for parallel execution. Our compiler also generates multiple versions of the semantically-equivalent code, which incrementally ex-

ploits more levels of nested parallelism. For example, consider a program containing two nested parallel loops. If the outer-loop contains enough parallel work to saturate the machine, it would be more beneficial to perform the inner loop sequentially. On the other hand, if the outer-loop only contains a few iterations, it might be worth it to parallelize the inner-loop. Since such decisions cannot be made at compile-time, we let the compiler produce both versions for parallel computations and let the run-time system pick between them. The run-time system is then able to exploit different levels of parallelism depending on the specific input. The run-time system is implemented as a library and only exposes an API to the code-generator. The closure of the generated functions are then provided through an API call, which the run-time system schedules using automatic granularity control.

Our implementation for automatic granularity control is based on the online algorithm by Acar et al.[Acar u. a., 2019], which supports granularity control for regular (nested) parallelism. We adapt the algorithm to our implementation of parallel for-loops. Here we present the main idea behind their algorithm. A more detailed presentation of their work can be found in Section 3.2.2. Their work is based on an "oracle-guided scheduling" approach, which can predict the actual work (one-core run-time) from parallel code, e.g., a parallel map. In other words, the actual work denotes the run-time if we were to execute the parallel code purely sequentially on a single core. Under the assumption that the parallel code's actual work is *predictable* in the input size, n, one can define an abstract cost function $c(n)$, which computes a value proportional to actual work. In practice we use the asymptotic complexity, e.g. n, $n \log n$ or n^2 as cost functions. For example for a parallel map, assume an array of length n and that a call to $f()$ takes constant time C, then one can use a linear cost function, i.e., $c(n) = n$, to describe the asymptotic work of the parallel map. By the definition of "asymptotics", there exists a constant C, such that

$$C \cdot c(n) \approx T \tag{1.1}$$

where T denotes the sequential run-time from the parallel code on some input of size n. Through an iterative process, which samples sequential run-times, the algorithm is able to infer the value of C at run-time. The online algorithm leverages this estimate to predict the actual work of parallel codes to perform granularity control by only parallelising computations, which can amortize the cost of parallelisation. More precisely, let κ denote the cost of parallelisation (in units of time) i.e., κ is the smallest amount of work for a subtask, which makes parallel execution beneficial[2]. Continuing with our parallel map example; we can replace the right-hand side of Eq. 1.1 with κ and rewrite it to obtain the minimum number of iterations in a chunk which amortizes the cost, i.e., $n_{chunk} = \frac{\kappa}{C}$, which we use to perform granularity control. This approach to granularity control is flexible and requires no help from the compiler. Compiler approaches to granularity control often fall short in

[2] κ is a machine-dependent value, which can be computed offline once-per-machine

considering the environment the program is executed in and rely upon heuristics, which do not generalize well. This approach performs granularity control solely at run-time, by sampling sequential run-times of parallel codes and can be implemented as a library solution. The disadvantage is that it requires a cost function to describe the asymptotic work of parallel codes. In [Acar u. a., 2019], they let the programmer provide the cost function. However, as we do not wish to change the source language of Futhark, we use an assumption. Since we are using parallel for-loops to express parallelism, we assume that a linear cost function can describe the asymptotic work. We will discuss the implications this assumption might have in Section 5.1, where we more describe our approach in more detail.

For computations whose run-time are not predictable, i.e., results in irregular parallelism, the above approach does not apply, and our implementation handles this case differently. For handling this case, we present our own online algorithm for granularity control. As C is no longer a constant in this case, we use a moving average estimation of C during execution of the parallel work to estimate the chunk size, which amortizes the cost of subtask creation. Additionally, we implement a work-stealing strategy to enable workload balancing at run-time.

To evaluate our implementation, we present an extensive empirical evaluation, which is divided into three parts. First, we evaluate our new back-end against the sequential back-end of Futhark on a series of microbenchmarks, intended to show the performance of individual parallel constructs such as `reduce`, `scan`, etc. Second, we evaluate our new back-end against the sequential back-end of Futhark using real-world programs from the Futhark Benchmark suite, which shows how our implementation performs on programs containing flat-, nested- as well as irregular parallelism. On our test machine, which has 16 cores with 2-way multi-threading, we can achieve speed-ups ranging from 7.8 to 23.8, with an average of 15.9, across 13 benchmarks on the largest datasets. We only observed slow-downs in two cases, where the datasets is small, and the program contains very little work. For example, for LU matrix decomposition, we observed a slow-down of $\times 2.5$ for a 64×64 matrix. Finally, we compare our implementation to a set of hand-optimized programs from other benchmark suites, such as FinPar[Andreetta u. a., 2016] and Rodinia[Che u. a., 2009]. Our results show that our approach can eliminate the need for hand-tuning of programs and, in some cases, provide better performance. However, our compiler does not consistently generate as efficient code as the hand-optimized benchmarks, which causes our implementation to be slower for some benchmarks. We will discuss these results in more detail in Chapter 7, where we present our benchmarks.

Contributions

The specific contributions of this thesis are:

- Design and implementation of a new code-generator in the Futhark

compiler for compiling Futhark's Bird-Meertens operators (`map`, `reduce`, `scan`, etc) to parallel C code.

- Design and implementation of a run-time system for scheduling the generated C code in parallel. The run-time system provides workload balancing through a work-stealing strategy.
- Two online algorithms for automatic granularity control of parallel for-loops for regular (nested) parallelism and irregular parallelism, respectively. We implement the algorithms within the run-time system.
- An empirical evaluation using a wide variety of existing programs from the Futhark benchmark suite as well as a comparison of our approach against publicly available programs from established benchmark suites.

1.1 Thesis structure

The structure of this thesis is as follows:

Chapter 2 gives an introduction to multicore CPUs and the multi-threaded programming model along with its challenges. The chapter also presents parallel constructs as well as an introduction to the programming language Futhark. This chapter is meant to give a brief overview for readers who are unfamiliar with these topics.

Chapter 3 provides an overview of related work in implicit parallelism for specialized programming languages and extensions as well as approaches to automatic granularity control.

Chapter 4 presents how we translate Futhark's Bird-Meertens operators (`map`, `reduce`, `scan`, etc) into parallel algorithms using parallel for-loops.

Chapter 5 presents our methods for the run-time system, which includes our approach to automatic granularity control, work management as well as a memory management for the new back-end.

Chapter 6 presents the design and implementation of our new back-end, which gives a high-level overview of our new code-generator and run-time system.

Chapter 7 shows the result from our empirical evaluation for micro-benchmarks and established benchmarks, comparing the performance against the sequential back-end of Futhark. Finally, we present an evaluation against hand-optimized programs from other benchmark suites.

Chapter 8 briefly discusses the limitations or shortcomings of our work and suggest how future work can improve upon it.

Part I

Background and Related work

Chapter 2

Background

This chapter introduces the fundamental concepts this thesis is built on. First, we describe the basics of modern CPUs and multi-threaded programming along with its challenges. Second, we present the parallel construct, parallel for-loops, as a way of expressing parallel computations. Finally, we introduce the programming language, Futhark. While the reader might not become an expert in these topics from reading this chapter, they should gain enough knowledge to understand the renaming of this thesis.

2.1 Computer architectures and CPUs

Today computers are an essential part of modern society and can be found in the form of smartphones, laptops, servers, etc., and serve billions of people every day. While these can be different in size and in computing power, the majority of these are based on the von Neumann architecture[Neumann, 1945]. The architecture is based on the stored-program concept, keeping both data and instruction stored in the same shared memory. Before the architecture was introduced, each machine was designed and built for a single predetermined purpose, which required manual rewiring of circuits, a labour-intensive and error-prone process. The von Neumann architecture, shown in Figure 2.1, consists of three main components, CPU, memory, and I/O interfaces. The I/O interfaces allow the computer to interact with other devices. Such devices include hard disks, also known as persistent storage, or Graphics Processing Units (GPUs). The memory is used to store a program's instructions, which the machine executes, along with its data. A commonly used memory type is Random-Access memory (RAM), which, unlike hard disks, is non-persistent storage, i.e., the content is lost when the machine loses power. The CPU has three main components. First, the Control Unit, which determines the order in which instructions should be executed and controls the retrieval of necessary data from memory by interpreting the program's instructions. It does so by issuing a series of control signals through the control bus, a communication line for either the memory or I/O devices,

specified by signal send along the address bus. The data bus is a bidirectional signal line, where the data is sent or received through.

The arithmetic logic unit (ALU) is the execution unit of a CPU and performs mathematical operations such as addition, multiplication, or boolean operations. The registers are small, temporary storage locations. As they are located directly onto the CPU, they are much faster to access than memory, which usually is on the order of hundreds of cycles on a modern computer. In contrast, registers can be accessed with just a couple of cycles and are a critical component to improve CPUs' performance. A CPU that is implemented onto a single chip is also called a microprocessor.

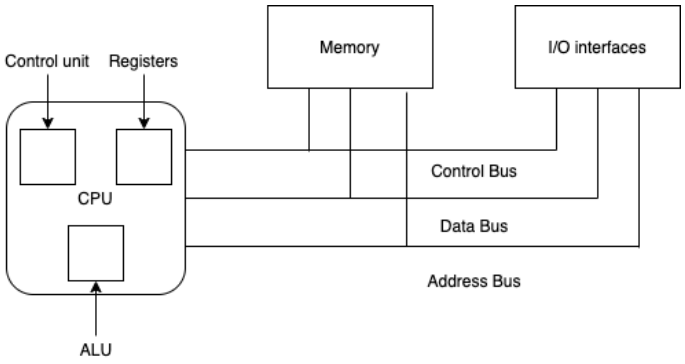


Figure 2.1: Diagram of the von Neumann Architecture and it's basic components (Source: `http://www2.cs.siu.edu/~cs401/Textbook/ch2.pdf`)

2.1.1 CPUs

CPUs are the hearts of computers and are intricate pieces of hardware, which can consist of billions of transistors all on fitted into a small integrated circuit. But back in November 1971, it was not the case. At that time, the first commercial produced microprocessor, 4004, was introduced by the American company, Intel Corporation. The 4004 had 2.300 transistors, could perform 60.000 operations per second and had 640 bytes of addressed memory. Since then, the computing power of microprocessors has grown massively. As predicted by Gordon Moore, the co-founder of Intel, would the number of transistors roughly double every two years, while the cost is halved. What this meant that we would observe an exponential growth in computing power every second year.

The doubling of transistors at the time resulted in higher clock frequencies, which made single-core applications run faster, the so-called "free lunch". In 2000 did the American semiconductor company Advanced Micro Devices (AMD) break a historical barrier as they introduced the Athlon microprocessor, which had a clock frequency of 1 GHz. However, from 2005, this rapid increase in clock frequencies changed. While we still observe higher transistor count, the clock frequencies have stagnated due to physical, thermal limitations, causing the microprocessors to heat up. Microprocessor designers had

to rethink how to make use of the transistors to obtain better performance. Instead, designers focused on increasing instructions per cycle (IPC) through various approaches:

Memory hierarchy: The problem with the Von Neumann architecture is that memory is placed far away from the chip and fetching data from memory can create a substantial delay. As microprocessors became faster and faster, they started to outperform the speed in which data could arrive, making memory a bottleneck in most systems.

To mitigate the bottleneck, microprocessors make use of a memory hierarchy in the form of caches. The memory hierarchy consists of several layers of small and fast memory embedded directly onto the chip. As more transistors could be fitted onto the single chip, the size and the efficiency of the caches grew as well.

Instruction-level parallelism: To further improve the IPC, modern processors make use of instruction-level parallelism. Out-Of-Order execution is a technique to avoid wasting clock cycles by executing later instructions, when the current instruction is stalling, and there are no data-dependencies between them. Another instruction-level parallelism technique is superscalar pipelines, enabling the execution of multiple instructions in parallel. Again, it requires that parallel instructions do not have data-dependencies.

Multiple cores: In 2005, AMD introduced the Athlon 64 X2 which featured a new architecture with two cores. This sparked a new trend in CPUs, which now could provide explicit parallelism onto the chip. The trend by adding additional cores is continuing today and for example, the Intel Xeon v2 Platinum 8290 come with 56 cores, and AMD provide 64 cores on their high-end Threadripper CPUs. Multicore core CPUs are not much different from the single-core counterpart, and each core still provides the same features above. Unlike the previous approaches, this does not provide the programmer with implicit performance gains but instead adds support for the programmer to better utilize the transistors in the form of writing parallel programs.

Hyper-threading: As memory bandwidth continued to be a bottleneck on most systems, Intel introduced the notion of hyper-threading in 2002 or more generally called Multithreading. Multithreading adds the notion of virtual cores. With Multithreading, one physical core acts as two cores while still sharing many of the core resources. In simple terms, Multithreading enables each virtual core to have its own architectural state, i.e., registers and the front-end of the pipeline (fetching and decoding), while still sharing the execution units, such as the ALU. This enables more efficient concurrent execution on the same physical core, as each thread can take over whenever the other is stalling, usually wait-

ing for data to arrive. According to Intel, Multithreading adds up to 30% performance gain on a single core.

While all of these optimizations can significantly increase performance, it remains challenging to write programs that can take advantage of the multiple cores. For example, in C, which has no notion of multi-threaded programming, the programmer must explicitly manage threads and synchronization on shared memory to ensure correctness. Such a process is notoriously known to be error-prone and time-consuming. The next section provides an insight into the multi-threaded programming model and its challenges.

2.2 Multi-threaded programming model

This section introduces POSIX threads as a way of creating parallel work-flows and introduces its programming model. This section also discusses some of the difficulties of writing correct multi-threaded programs and presents the mechanisms to ensure that a program remains correct using threads.

2.2.1 Operating System threads

OS threads (hereafter threads) allow programs to execute using multiple parallel work-flows within a process. Threads can be scheduled onto different cores by the operating system and run as separate entities. This is accomplished since they only duplicate the bare essential resources of its process. Each thread maintains it is own stack pointer, registers, and program counter while sharing other process properties, such as the heap, global data, environment, file descriptors, and signal handlers. As such, threads can be thought of as lightweight processes, which can achieve parallelism without the heavy management that comes with processes.

POSIX Threads (`pthreads`) was initially created for providing a standardized programming interface for the creation and managing of threads across hardware on UNIX systems. `pthreads` presents has its own execution model, which is independent of the programming language. For example, the execution model of `pthreads` does not provide any guarantee about the execution order of threads, i.e., the execution of concurrent threads may interleave arbitrarily. Thus a great amount of attention must be paid towards ensuring thread-safeness. Thread-safeness refers to the programs ability to execute multiple threads simultaneously while ensuring that shared data is “consistent“ and avoid creating “race“ conditions. For example, consider a small program where we try to increment a shared counter using two threads. A minimal program is shown in Listing 2 using the POSIX threads library. The program creates two threads through lines 9 and 10, which calls `thread_increment` and increments the shared variable `counter`. It then waits for both threads to return from the function through lines 12 and 13. Finally is the value of outputted.

Listing 2 A unsafe program using threads

```
int counter = 0;
void *thread_increment(void *args) {
  counter++;
  return NULL;
}

int main() {
  pthread_t tid1, tid2;
  pthread_create(&tid1, NULL, thread_increment, NULL);
  pthread_create(&tid1, NULL, thread_increment, NULL);

  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  printf("%d\n", counter);
}
```

With a sequential execution model, one might expect that the counter would contain the value of two, but since POSIX does not provide any guarantee about the execution order, both threads might read zero and increment it and write back one. Without ensuring that such an operation is thread-safe, data corruption can occur. We call such operation on shared data critical, and when more operations are needed to modify some shared data, we call it a critical section. There are several ways of ensuring that data-corruption does not occur through synchronization mechanisms by ensuring that only a single thread accesses to shared data at the time.

2.2.2 Mutual exclusion

We can avoid race-conditions to occur by using explicit locking, thereby serializing critical sections. With `pthreads` such locks are available, called `mutex`es. The extended code in Listing 3 shows an example usage, where `PTHEREAD_MUTEX_INITIALIZER` is a macro that initializes the `mutex`.

Listing 3 Extended program, which ensures updates are thread-safe using `mutex`es

```
...
pthread_mutex_t lock = PTHEREAD_MUTEX_INITIALIZER;
void thread_increment(void *args) {
  pthread_mutex_lock(&lock);
  counter++
  pthread_mutex_unlock(&lock);
  return NULL;
}
...
```

The use of `mutex`es are very flexible and can be used to lock arbitrary sized critical sections. Another advantage of `mutexes` is the usage in combination

with condition variables, allowing threads to sleep and wake up only when a certain condition is full-filled. Such a pattern is especially useful when working with producer-consumer problems, allowing threads not to waste CPU resources when there is no work. In modern operating systems, `mutexes` are implemented using *futexes*, short for "Fast user-space mutex", which only invokes a kernel call whenever there is contention for the lock. Whenever a thread is trying to acquire an unlocked lock, the operation is performed using an atomic operation (Section 2.2.3), which happens in user-space. Only when the requested lock is locked, a kernel call is made, requiring a context switch to kernel space, which is relatively expensive.
When using locks, one must also beware of deadlocks. Deadlocks occur whenever within a set of threads, each thread is waiting for a lock to be released, which is controlled by another thread. Deadlocks are the source of many bugs in multi-threaded programming and can be challenging to resolve, as they might only occur on rare occasions.

2.2.3 Atomic operations

At a more primitive level are atomic operations, which forms the basis of many locks implementation, including the `mutex` presented above. Atomic operations provide a guarantee that an operation cannot be interrupted by other threads. For example, can an atomic addition of integers, read the old value, perform the addition, and write the result back atomically.

```
1 ...
2 void thread_increment(void *args) {
3   atomic_add(counter , 1)
4   return NULL;
5 }
6 ...
```

Atomic operations require special machine language instructions available through runtime libraries. The library implementation depends on the concrete machine currently executing on. Luckily, most mainstream compilers, such as GCC, already provide atomic operation functions, such that the programmer does not need to worry about the concrete machine the program is executed on. It is usually a very efficient way of dealing with race-condition but is not applicable in some cases. For example, they do not support floating-point operations. Compared to the use of `mutexes` are atomic operations less flexible (unless used as locks), only providing atomic operations on few primitives and unable to be used in combination with condition variables, for example. Furthermore, as many compilers and processors rearrange instructions to obtain better performance they usually only think about its dependencies from a sequential point-of-view. Whenever concurrent programs are executed, we usually need to explicitly tell the compiler or processor that specific instructions cannot be rearranged. The next section introduces the notion of memory models and how we can use barriers to prevent such rearranging instructions.

2.2.3.1 Memory models and Barriers

When dealing with lock-free concurrent programming, the notion of memory ordering needs to be handled carefully. As an optimizing compiler has the freedom to reorder instructions, it can cause unexpected behavior on multi-threaded programs, where the order can be crucial. The reordering of instructions is not only confined to optimizing compilers. Processors today make use of reordering instructions at run-time, e.g., out-of-order execution, to increase performance. We first introduce the notion of memory models. We usually distinguish between strong and weak memory models. A memory model is said to be strong if a core/thread performs a series of writes, every other core/thread sees those writes in the same order as they were performed. A weak memory model is then a processor which does not guarantee such ordering of writes and reads. From a hardware perspective, an example of a strong memory model are microprocessors from the x86/64[1] architecture family. Architectures based on Alpha or ARMv7 are examples of weak hardware memory models. From a software perspective, programming languages like C/C++ also falls into the category of weak memory models, as the compiler is free to reorder instructions. To better understand the problem of a weak memory model in a multi-threaded setting, consider the simple two threaded program below; thread 1 waits as long as `f` equals 0 and prints the value of `x` afterwards, while thread 2 writes to `x` and then sets `f`. Without restricting the compiler from reordering, it might reorder the two instructions on thread 2, resulting in thread 1 printing 0.

```
1  // thread 1
2  while (f == 0);
3  print x
```

```
1  // thread 2
2  x = 42;
3  f = 1;
```

To ensure correct program behavior, we need to make use of *memory barriers*, which prevents the compiler from rearranging instructions. A memory barrier instruction tells the compiler that everything before the barrier cannot be reordered onto the other side and vice versa. The modified example program would now look like:

```
1  // thread 1
2  while (f == 0);
3  memory_barrier();
4  print x
```

```
1  // thread 2
2  x = 42;
3  memory_barrier();
4  f = 1;
```

Note that we also need a memory barrier on thread 1 as it might read the value of `x` before reading `f`. Such reordering can also cause unexpected program behavior. While such a full barrier prevents reordering, we can relax the memory model a bit. We distinguish between `acquire` and `release` semantics. An `acquire` semantic prevents memory reordering of the read-acquire with any read or write operation that follows it. On the other hand,

[1] except for a few exotic cases

a `release` semantic prevents reordering of the write-release with any read or write operation that precedes it in program order. In our example, thread 1 only require an `acquire` semantic on the variable `f`, while the write to `f` on thread 2 require a `release` semantic.

2.2.3.2 Atomic functions

Compilers such as GCC and Clang provide built-in atomic function, which makes use of memory models described above. We list some of them here, where `type` can be any integral scalar or pointer type of size 1, 2, 4, 8 or 16 bytes

`type __atomic_op_fetch(type *ptr, type val, int memmodel)`
: where `op` can be any of `add`, `sub`, `and`, `xor`, `or` or `nand`. Semantically this function performs the atomic operations `{*ptr op= val; return *ptr}`

`type __atomic_load_n(type *ptr, int memmodel)`:
: Returns the content of `*ptr`

`void __atomic_store_n(type *ptr, type val, int memmodel)`:
: Writes the content of `val` to `*ptr`

`type __atomic_exchange_n (type *ptr, type val, int memmodel)`:
: Writes `val` into `*ptr` and returns the previous content of `*ptr`.

`bool __atomic_compare_exchange(type *ptr, type*expected, type desired, bool weak, int succes_memmodel, int failure_memmodel)`
: which atomically reads the content of `*ptr`, compares it to `*expected` and if they are equal writes `desired` into `*ptr` and returns `true`. Otherwise the content of `*ptr` is written into `*expected` and the function returns `false`.

The implementation of these depends on the specific target architecture, but if the architecture presents no lock-free instruction sequence, a fallback to an explicit locking routine is used.

2.3 Parallelism and parallel constructs

This section formally presents the notion of parallelism and introduces some of the most important theoretical results within the subject. Finally, we present the parallel construct, parallel for-loops, which can be used to express (nested) parallelism.

2.3.1 Types of parallelism

We first define the notion of parallelism. Parallelism refers to *simultaneous* execution of work. The definition of parallelism is often confused with concurrency, which refers to different computations' ability to interleave with each other. Parallelism is also concurrent, but not necessarily vice versa. We also distinguish between two forms of parallelism:

- `Task parallelism` refers to simultaneously executing different operations on potentially different types of data. As such can task parallelism perform asynchronous computations. One example of task parallelism is the one performed by operating systems, which are able to perform different tasks simultaneously. Task parallelism can also be found in languages, such as Cilk.

- `Data parallelism` refers to simultaneous execution using the same operations, but on different pieces of the same data. Examples of data parallelism are Single Instruction Multiple Data (SIMD). Futhark is based on data-parallel computation as well. In contrast to task parallelism are data-parallel operations performed synchronously.

Task- and data parallelism are not mutually exclusive; they can naturally coexist within an application. Finally, we introduce the notion of work-efficiency as a formal tool to reason about parallel algorithms efficiency. We define the work of an algorithm as the number of operations performed by it. Which leads to the definition of *work efficiency* of parallel algorithms:

Definition 2.3.1 (Work efficiency)**.** A parallel algorithm is said to be *work efficient* if it performs no more work asymptotically than it's sequential counterpart.

2.3.2 Fork-join model

The fork-join model[Conway, 1963] is a simple and powerful parallel design pattern for setting up a program for parallel execution. It is used in languages like Cilk[Blumofe u. a., 1995] as the main model of parallel execution. The fork-join model is conceptually simple and uses the two keywords, `fork` and `join` to express parallelism. The `fork` keyword specify that a section of code is suitable for parallelism, which will spawn parallel computations. Each `fork` is then accommodated by a `join`, which is a synchronization mechanism. The join ensures that the main computation cannot continue until all parallel computations have finished. Parallel computations may recursively use fork-joins and these two keywords suffice to express nested parallelism, which makes the fork-join a powerful way of structuring parallel programs. A concrete instance of fork-join is the `fork2join`, usually used with divide-and-conquer algorithms, where each fork generates two sub-branches, which can be executed in parallel. Another concrete application of a fork-join is the

parallel for-loop. With a parallel for-loop, the iterations can then be divided out among the cores for parallel execution. An implicit join point is set just after the for-loop ends. Here we show how a parallel map, which applies some function f to each element of an array, `xs`, can be implemented using a parallel for-loop.

```
1  void parallel_map(T* xs, T* res, int n){
2    parfor (i = 0; i < n; i++) {
3      res[i] = f(xs[i]);
4    }
5  }
6  // Implicit join
```

Such an implicit parallel construct is powerful as it can represent various computations while using the same generic interface. For example, we use parallel for-loops to express parallelism for Futhark's operators. The next section presents the programming language Futhark and the semantics of these operators.

2.4 Introduction to Futhark

Before we introduce Futhark, we provide some mathematical definitions, which is used by the language.

2.4.1 Mathematical Definitions

Definition 2.4.1 (Semigroups)**.** Let $\mathcal{X}$ be a non-empty set and $\oplus$ be a binary operator on $\mathcal{X}$. The pair $(\mathcal{X}, \oplus)$ is called a *semigroup* if the operation is associative, i.e. for all $x, y, z \in \mathcal{X}$ we have

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z \tag{2.1}$$

Examples of a semigroup is multiplication over the real-numbers denoted as the pair $(\mathbb{R}, *)$

Definition 2.4.2 (Commutative)**.** A semigroup $(\mathcal{X}, \oplus)$ is called commutative if for all $x, y \in \mathcal{X}$ we have

$$x \oplus y = y \oplus x \tag{2.2}$$

Definition 2.4.3 (Neutral element)**.** An element $e \in \mathcal{X}$ is said to be the neutral element of a semigroup $(\mathcal{X}, \oplus)$ if for all $x \in \mathcal{X}$ it satisfies the following property

$$e \oplus x = x \oplus e = x \tag{2.3}$$

A semigroup which has a neutral element is also called a *monoid*.

2.4.2 Futhark

Futhark is a pure functional data-parallel array language from the ML-family, which uses a few generic Second-Order Array Combinators (SOACs) to obtains its parallel code. Futhark focuses less on expressivity and elaborate type system, and more on compilation to high-performance parallel code, through its heavily optimizing ahead-of-time compiler. While the primary target of the optimizing compiler is to generate efficient GPU code via OpenCL and CUDA, the language is hardware agnostic. With this thesis, Futhark will also be capable of generating parallel C code to be executed on a multicore CPU.

Futhark supports regular nested data-parallelism, as well as imperative-style in-place updates of arrays, but maintains its purely functional style through a uniqueness type system, which prohibits the use of an array after it has been updated in-place. Like most ML-family languages, Futhark also has parametric polymorphic and uses type parameters, which allows functions and types to be polymorphic. Type parameters are written as a name preceded by an apostrophe. For example Listing 4 shows a type abbreviation *number* with a type parameter t, which is instantiated with concrete types in line 2, where $f32$ denotes 32-bit floating point.

Listing 4 Example of type abbreviation with type parameter in Futhark

```
type number 't = t
type float = number f32
```

Lastly, it is also possible to specify a parametric module, meaning that the module can take another module as an argument (i.e., module-level functions), which allows for abstraction over modules.

Futhark achieves much of its data-parallelism through its Second-Order Array Combinators (SOACs), `map`, `reduce`, `scan` and `reduce_by_index`. The use of parallel SOACs lets the programmer write programs with sequential semantics as in conventional function language, but permit parallel execution through Futhark's compiler and run-time system. An overview of the four SOACs and its semantics are shown below, where we use $[x_1, x_2, \ldots, x_3]$ to denote an array of n elements and $[n]\alpha$ to denote the type of an array of length n with α.

- `map`: $(\alpha \to \beta) \to [n]\,\alpha \to [n]\,\beta$
  `map` $f\,[x_1, x_2, \ldots, x_n] \equiv [f(x_1), f(x_2), \ldots, f(x_n)]$
  where the function f is applied to every element of the array of xs.

- `reduce`: $(\alpha \to \alpha \to \alpha) \to \alpha \to [n]\,\alpha \to \alpha$
  `reduce` $f\ ne\ [x_1, x_2, \ldots, x_n] \equiv f(\ldots f(f(ne, x_1), x_2), \ldots,)x_n)$
  where every element in the array is collapsed (or reduced) using the binary operator f. Here we require that $(\mathcal{X}_f, f)$ is a monoid with neutral element ne, where $\mathcal{X}_f$ is the set of input f takes.

- `scan`: $(\alpha \to \alpha \to \alpha) \to \alpha \to [n]\,\alpha \to [n]\,\alpha$
  `scan` $f\ ne\ [x_1, x_2, \ldots, x_n] \equiv$
  $[f(ne, x_1), f(f(ne, x1), x2), \ldots, f(\ldots f(f(ne, x_1), x_2), \ldots), \ldots, x_n)]$.
  The `scan` SOAC produces an array of same size as the input. The binary operator f is applied to every element, where the first argument is the result of the preceding prefixes. Again we require that $(\mathcal{X}_f, f)$ is a monoid with neutral element ne.

- `reduce_by_index`: $[m]\alpha \to (\alpha \to \alpha \to \alpha) \to \alpha \to [n]i32 \to [n]\alpha \to [m]\alpha$
  `reduce_by_index` $dest\ f\ ne\ is\ vs$
  where f is an associative and commutative function with neutral element ne. Here is and vs must be same length. We describe the semantics using imperative code:

```
for j < length is:
  i = is[j]
  v = vs[j]
  if i >= 0  &&  i < length dest:
    dest[i] = f(dest[i], v)
```

The `reduce_by_index` reads each value in is and vs and updates the value at $dest[i]$ with the value of v using the function f. Note that updates are ignored if the index is out-of-bounds.

These SOACs permits the Futhark compiler to generate parallel code, which means that Futhark programs are commonly written as bulk operations on arrays. Through the SOACs, the Futhark compiler can provide optimizations. For example is the composition of nested `map-reduce` computation efficiently supported based on fusion and code generation [Henriksen u. a., 2016; Larsen und Henriksen, 2017] and the compiler also provide support for 1D and 2D tiling[Henriksen u. a., 2017]. However, the compiler also has some limitations when it comes to irregular parallelism. The Futhark program below shows a simple example of irregular parallelism, where the work of the nested `reduce` depends on the value of i, which is varying from 1 to n.

```
1        map (\i -> reduce (+) 0 [1...i]) [1...n]
```

Even though the Futhark language supports irregular nested parallelism, the compiler is unable to exploit it. Currently, when targetting GPUs, the compiler sequentialize the nested reduce, which results in a skewed work-load across the hardware threads. This has been a historical downside of Futhark. For example, many irregular problems, such as BFS, needed to be manually written into a regular parallel form, e.g. by applying flattening techniques, to exploit the inner parallelism at the cost of its efficiency. For the new back-end, we aim to provide support for exploiting irregular parallelism through work-load balancing techniques without the need for a manual rewrite.

Chapter 3

Implicit parallelism

For the past decade there have been a development in providing specialized languages and frameworks, which enable programmers to write parallel programs with close to sequential semantics using parallel constructs such as parallel for-loops. These languages and frameworks raise the level of abstraction from the tedious details of parallel programming and delegate these to the compiler and run-time system. This chapter first introduces run-time systems and then gives an overview of the related work within the field of implicit parallelism and approaches to granularity control.

3.1 Run-time system

The run-time system defines the execution behavior of a program, which is not directly attributable to the program itself. The run-time system can, for example, aid in freeing up resources when they are no longer needed, also known as the garbage collector. Other tasks include stack and heap management, bounds checking, exception handling, communication with the kernel, etc. An example of a simple run-time system is the one implemented by the C language. C's run-time system can be regarded (by modern standards) as a barebone run-time system, where, no bounds checking is performed nor garbage collection. Instead, the run-time system puts these tasks onto the programmer. As the new back-end compiles to C code, we use the already in place run-time system while providing additional features such as bounds checking and implicit memory management. We did not have to deal with these parts of the run-time system a lot in our work, but instead, we mainly focused our work on the scheduler.

An intrinsic part of the run-time system for implicit parallel languages is the scheduler. The scheduler is responsible for delegating parallel work onto the computing resources through various means. The main goal of a scheduler in parallel languages is to maximize throughput, i.e. the amount of work completed pr. unit time. Schedulers usually use threads, also called workers, to obtain higher throughput through parallelism. To obtain parallelism, the

scheduler divides the work into smaller subtasks, which contains the closure of the work and delegates these to the workers, which are then scheduled onto the hardware resources for parallel execution. However, for a scheduler to achieve its goal, it must consider the overheads of parallelism. To ensure that such overheads do not overwhelm the benefits of parallelism, one must perform granularity control, which controls the amount of work a single parallel subtask contains. However, determining how much work is needed to amortize the cost depends on the specific input, machine, and so on, and is one of the main challenges in implicit parallelism. As such, researchers have provided several approaches to granularity control, as choosing an appropriate granularity is essential in achieving good performance in parallel applications.

3.2 Related work

This section provides some insight into the field of implicit parallelism and is two-fold. First, we provide an overview of some of the most popular implementations for writing implicit parallel programs, which include stand-alone languages and language extensions. Second, we investigate the research field within automatic granularity control, giving us ideas on how to approach the problem with Futhark.

3.2.1 Languages and frameworks

We first gain a high-level view of some popular languages and frameworks implementations for achieving parallelism on a CPU. The implementations presented here share the common goal of making parallel computing simple, hiding a lot of the complexity of parallel programming away. A goal also shared by Futhark.

3.2.1.1 Cilk

Perhaps the most known programming language for implicit parallelism is Cilk[Blumofe u. a., 1995]. Cilk is a task-parallel language and was originally developed at Massachusetts Institute of Technology in the 1990s as an extension to accommodate existing C code. Cilk initially extended the C language by a few keywords for identifying parallelism, which can safely be executed. The idea was that the programmer should be responsible for exposing parallelism to the compiler and run-time system. A small Cilk program computing the nth Fibonacci number is shown in Listing 5, where `spawn` indicates that the computation following can be computed in parallel, while `sync` denotes that computation cannot continue until all previous spawned work has completed. Such a parallel construct is an example of the fork-join model, more specifically a `fork2join`, often used for divide-and-conquer algorithms. A key property of Cilk was that removing all of such keywords from the program would result in a valid C program, called the *sequential elision*.

Listing 5 A parallel program computing the n'th Fibonacci number in Cilk

```
cilk int fib(int n) {
  if (n < 2) {
    return n;
  }
  else {
    int x, y;
    x = spawn fib(n - 1);
    y = spawn fib(n - 2);
    sync;
    return x + y;
  }
}
```

Much of the theoretical work of scheduling multi-threaded computation comes from Cilk. When the first version of Cilk was released in 1995, it introduced a distributed randomized work-stealing scheduling approach, where idle threads try to steal work from other threads[Blumofe u. a., 1995]. The Cilk scheduler provided theoretical bounds on space, time, and communication overhead, providing asymptotic performance guarantees.
Due to the rise of mainstream parallel computing in the mid-2000s, Cilk Arts was created to lead the compiler's development. Cilk Arts released Cilk++ version in 2008, which extended C++ and added native support for generic lock-free reductions and parallel loops over a fixed number of entries. In 2009 Cilk Arts were acquired by Intel to continue the development of the compiler, and Cilk++ was implemented into Intel's C++ compiler. However, the additional work of maintaining the compiler slowly became less beneficial as more portable alternatives started to provide similar performance. In 2017, Cilk depreciated, and recommended users to use Intel's Threading Building Blocks(TBB) or OpenMP instead. TBB is a library solution and provides basic building blocks for writing parallel programs using standard C++ code without the need for a special compiler. TBB provides generic parallel algorithms, task schedulers, synchronization primitives, and much more. TBB draws much of its inspiration from Cilk; for example, TBB also uses a randomized work-stealing approach. However, where TBB often falls short is its ease of use. The program below shows a C++ program using TBB to compute the nth Fibonacci number. Compared to Cilk is the TBB version much more verbose and requires more work from the programmer. Note that for both Cilk and TBB, the programmer must perform granularity control manually by providing a cut-off.

Listing 6 A parallel program computing the n'th Fibonacci number in TBB

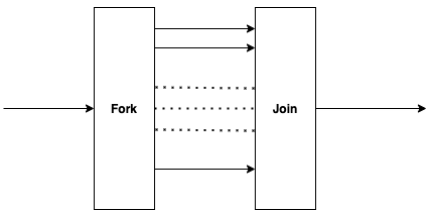
```
class FibTask: public task {
public:
  const long n;
  long* const sum;
  FibTask( long n_, long* sum_ ) :
    n(n_), sum(sum_)
  {}
  task* execute() {
    if( n<CutOff ) {
      *sum = SerialFib(n);
    } else {
      long x, y;
      FibTask& a = *new( allocate_child() ) FibTask(n-1,&x);
      FibTask& b = *new( allocate_child() ) FibTask(n-2,&y);
      // Set ref_count to 'two children plus one for the wait".
      set_ref_count(3);
      // Start b running.
      spawn( b );
      // Start a running and wait for all children (a and b).
      spawn_and_wait_for_all(a);
      // Do the sum
      *sum = x+y;
    }
    return NULL;
  }
};
```

3.2.1.2 OpenMP

So far, we have seen how implicit parallel programs can be expressed using a specialized programming language or as a library. A third approach is to use a directive-based approach with direct support from the compiler and a runtime library to transform sections of programs into parallel computation. Open Multi-Processing (OpenMP)[OpenMP, 2008] is a language extension for supporting multi-threaded programming, with compiler support for C/C++ and Fortran. OpenMP is supported by most mainstream compilers such as GCC. As with Cilk, OpenMP uses keywords to express that specific parts of a computation can be parallelized. More specifically, OpenMP uses the compiler-specific directive, `#pragma omp <flags>` with several extension flags for specifying thread creation, workload distribution, synchronization and much more. The example program in Figure 3.1a, shows how one can parallelise a for-loop with OpenMP by using the flag `parallel for`. OpenMP also uses fork-join model where an implicit join is inserted right after the loop.

```
1  #pragma omp parallel for
2  for (int i = 0; i < n; i++) {
3    // Do parallel work
4  }
5  // Implicit join insertion
```

(a) A parallel-forloop using OpenMP



(b) Illustration of a parallel-for loop using fork-join model

Figure 3.1: Example and illustration of parallel for-loop in OpenMP

When a directive is encountered during run-time, a *team* of threads are taken from a thread-pool managed by the OpenMP run-time system, and the work is delegated among these. As parallel computations are executed on a request basis, the run-time system needs to fetch the number of needed threads and prepare them for usage every time, which causes overhead. Furthermore, as noted in [Bull u. a., 2012] can OpenMP employ a relatively high overhead due to it is granularity control strategies, where often too many subtasks are created. Available controls for granularity control in OpenMP are static, dynamic, guided, or auto. A static approach refers to a fixed distribution of iterations, which is decided before the computation is started. When static is chosen, the iterations are evenly divided among the threads, which is preferred when the workload is the same across iterations.

Alternatively, there is a dynamic approach where the distribution is not fixed before the execution is started, and can be readjusted during the parallel computation. Dynamic scheduling is usually used when the work-load across iterations is not balanced to achieve load-balancing. When dynamic scheduling is specified with a statically chosen chunk-size, each thread will then take chunk-size work at the time. The thread can then request more when it has finished its chunk, which results in better load-balancing. When no chunks-size is specified, it defaults to one. Another approach to achieve load-balancing is the guided approach, which is based on a heuristic, which aims at reducing subtask creation while still achieving load-balancing. Assume $n_{threads}$ threads are fetched. The scheduler will initially create a $\frac{n}{n_{threads}}$ chunk size for the first subtask, while the second subtask is assigned half of that as so on, i.e., it gradually halves the work-load of each subtask. A lower threshold can be provided for the minimum chunk size. This approach works well when the majority of the work-load is towards the end of the computation. A final approach is that the compiler can decide which to use during compilation. It was not possible to find the exact heuristic used. As with Cilk and TBB, deciding between these approaches and finding an appropriate chunk size puts most of the burden onto the programmer. While different scheduling approaches are needed, we want to hide such choices from the programmer with Futhark.

3.2.1.3 MaPLe

The final language is MaPLe (MPL), which is a parallelism-oriented fork of MLton, a whole-program optimizing compiler for the functional programming language Standard ML. Among the related languages, MPL is most similar to Futhark; in fact, Futhark heavily inspired by Standard ML and provide many common features, such as parametric polymorphism and a module system. MPL uses two fork-join constructs for expressing parallelism

```
1 val par: (unit -> 'a) * (unit -> 'b) -> 'a * 'b
2 val parfor: int -> (int * int) -> (int -> unit) -> unit
```

The first one is `par`, which takes two functions to execute in parallel and returns the result of them. This is equivalent to the use of `spawn` and `sync` in Cilk. The second one is `parfor`, which is a parallel for-loop, as seen in OpenMP. The `parfor` takes a chunk-size g, a range (i, j) and a function f. It executes $f(k)$, $i \leq k < j$ in parallel using g as the granularity control parameters, i.e. it creates approximately $(j - i)/g$ continuous chunks. Again, we want to hide the underlying parallelism parameters away from the programmer. However, the simplicity of the `parfor` can give us an idea on how to structure our implementation using parallel for-loops.

3.2.1.4 Summary

While we have only touched upon a small subset of the specialized languages and frameworks for parallel computing, they provide us with a good idea of the landscape. The presented implementations use high-level parallel constructs to obtain parallelism, similar to how Futhark uses SOACs. However, all of the above use approaches to granularity control, which requires manual tuning by the programmer. The whole background for Futhark is that we want to make parallel programming convenient, and manual tuning of programs does not suit that definition. The next section provides some insight into automatic granularity control and the related work within the field.

3.2.2 Automatic Granularity Control

In this thesis, we aim at providing automatic granularity control through the compiler and run-time system. Stand-alone compiler techniques often fall short as they fail to consider the environment the program is executed in. For example, Iwasaki et al.[Iwasaki und Taura, 2016] use a synthesizing static technique for determining cut-offs, for when it becomes more beneficial to perform sequential execution for Cilk style divide-and-conquer programs. However, it relies on having accurate static analysis for determining the cost of execution at compile-time. Such an approach does not generalize well as those cut-offs depend on the host-machine, where execution time on modern processors depends, among others, on caching and pipelining, which are

difficult to determine the effects of at compile-time. Thus should an implementation for automatic granularity control at the very least involves the run-time system, which can adapt to the environment.

This section will investigate automatic granularity control approaches, which involves the run-time system.

3.2.2.1 Adaptive Granularity Control Using Multi-versioning

In 2013, P. Thoman et al.[Thoman u. a., 2013] presented a compiler and run-time approach for granularity control using multi-versioning. The rough idea was that the compiler generates multiple versions of parallel codes, each of which was exploiting different levels of parallelism. Their approach was based on *unrolling* of nested parallelism, thereby increasingly sequentializing larger and large sections of the program. At run-time, the scheduler then decides which version to use based on *task demand*, which is an indicator of how (under)utilized threads are. When there are many idle-threads, the run-time system will try to schedule the version spawning the most parallel subtasks, while when the system is fully utilized, the sequential versions will be chosen. Such an approach tries to exhaust the maximum amount of parallelism in the system. Though one pitfall of such a run-time approach is that, even if there are many idle threads, it does not ensure that parallelization is beneficial. In fact, such parallelization might run slower than simply executing the program sequentially due to the cost of subtask creation and management. Such costs must be taken into consideration when designing the run-time system.

3.2.2.2 Oracle-based Granularity control

The next approach is from Acar et al.[Acar u. a., 2019], which provides granularity control in the context of `fork2join` constructs such as in Cilk. Their approach supports nested parallelism and uses a "oracle-guided scheduling" approach, which is capable of predicting the sequential run-time of parallel codes. Their key contribution is an online algorithm for implementing the oracle, which we present below.

For each parallel code, they ask the programmer to provide a parallel body, which is a lambda function $f()$ that performs a parallel computation. They also ask the programmer for a sequential alternative, i.e., a sequential function $g()$, which delivers the same results as the parallel body. In Cilk, the sequential alternative can be obtained by removing all parallel keywords. Finally, for every such pair, they ask the programmer to provide a cost function $c(n)$, which gives an abstract measure of the work performed by $g(n)$, where n denotes input size. In other words, $c(n)$ computes a value proportional to the sequential execution time of $g(n)$. In practice, good choices for the cost function are asymptotic complexities, e.g. n, $n \log n$ or n^2. Hence their work assumes that the run-times are predictable and can asymptotically be described by the cost function. By the definition of "asymptotics", there exist

a constant C such that

$$C \cdot c(n) \approx T_{g(n)} \tag{3.1}$$

where $T_{g(n)}$ denotes the execution time of $g(n)$ for input of size n. Their online algorithm aims to compute C by sampling execution times of $g()$ and use Eq. 3.1 to exploit the estimate of C to predict the sequential run-time for future inputs. To perform granularity control, they introduce the quantity κ, which denotes the smallest amount of work, in units of time, that would be profitable to parallelize on the host machine. κ should be large enough to amortize the cost of creating and managing a single parallel subtask. On modern machines, the practical values of κ are on the order of tens to hundreds of microseconds. Their run-time policy is simple; if the result of $f(n)$ can be obtained by executing $g(n)$ in less time than κ then $g(n)$ should be used. However, to make such a decision, the algorithm needs an estimate of C. Without an sufficiently accurate estimate of C the algorithm might end up invoking $g()$ on a large input, and thereby destroying all available parallelism. However, to estimate C it needs to run $g()$, which creates an initial circular dependency. To solve the dependency the algorithm initially only execute $g(n)$ with small input sizes, where it initially only sequentialises the "base" case. In the Fibonacci example from Listing 5 this corresponds to only sequentializing $fib(2)$, which gives $T_{g(2)}$ and is used to compute an early estimate of C by computing the ratio $\frac{T_{g(2)}}{2}$. The estimate may be subsequently used to predict that another slightly larger input may execute in less time than κ. As long as $T_{g(n)}$ is smaller than κ, the algorithm tries to gradually sequentialise larger and larger computation, i.e. increase n, which gives a better estimate of C. The algorithm stops increasing n whenever $T_{g(n)}$ reaches the threshold of κ, which then indicates the smallest amount of work that is worth parallelizing.

This approach takes into account the cost of parallelization, something that the previous approach did not do. Furthermore, it only relies on the run-time system being able to map measurements onto parallel codes, and can be implemented as a library solution. Such an approach is favorable, making it easier to perform changes to either the compiler or run-time system down the road. However, it involves the programmer, as they need to provide a sequential version and a cost-function. Luckily, Futhark's parallel operators can naturally be expressed as a sequential operation, so such a limitation is not an issue. The cost-function is more troublesome. As we want to leave the source language of Futhark unchanged, we cannot let the programmer provide us with the cost function. While a linear cost function applies in most cases and can be used as default, there exist cases that follow different asymptotics. In some cases, such as when the run-time is not predictable, defining a proper cost function is impossible. We will need to account for such cases.

3.2.2.3 Hearbeat scheduling

What we have seen so far are schedulers using granularity control with a greedy scheduling approach, where threads are kept busy as long as there is work to do. Another type is lazy scheduling or lazy task creation, which postpones parallel computations until the cost can be amortized. The aim is to reduce the total overhead of subtask creation, while granularity control tries to reduce the same overhead by switching to sequential code. Heartbeat scheduling[Acar u. a., 2018] is an example of such a lazy task creation technique. It involves each thread maintaining a call stack, which the thread pushes frames onto. At periodic intervals, N, the thread will check its call frame, and if it finds parallel work, it will promote the frame, creating a subtask. The subtask may then be subject to load-balancing, e.g., stealing. In between promotions, the threads perform the sequential work, popping of work from the call stack. Such a scheduler's performance requires proper tuning of the promotion parameter, N, which decides the frequency parallel frames are promoted. If N is too small, too many subtasks are potentially created, resulting in over parallelization, while a too-large N creates under parallelization. Even if one can find a good promoting value, lazy scheduling can at best reduce the overhead of subtask creation, while granularity control can switch between sequential and parallel algorithms, where sequential algorithms are usually both asymptotically and practically more work-efficient. While both lazy task creation and granularity control could co-exist within a system, there is, to our knowledge, no work that tries to combine both methods. We believe that such a system can quickly become too complex to be practically efficient.

Part II

Methods and Implementation

Chapter 4

Parallel Algorithms for SOACs

In this chapter, we describe how each of the SOACs can be translated into parallel algorithms using parallel for-loops. We present the parallel algorithm using C-like code. We will use the notation `parfor` to denote a parallel loop, while a `seqfor` denotes a sequential for-loop. For now, we omit how we decide the chunk size for a parallel for-loop, and we assume that it is a fixed number. For the analysis of work efficiency of our parallel algorithms, we assume that it takes constant time to decide the chunk size and the chunk size in the interval of $[1, n]$ where n is the number of iterations.

4.1 `map`

The idea of translating a `map` is fairly simple, which semantically applies some function `f` to each element of an array. We demonstrate how the operation is compiled using the generic Futhark function in Listing 7, which takes an n-length array as input and applies the function f to each of the elements.

Listing 7 A map function in Futhark using a generic function f over xs

```
1 let mapf [n] 't (xs:[n]t) = map f xs
```

The semantically equivalent sequential C version can be written as:

Listing 8 C code for a sequential map

```
1 void sequential_mapf (int32_t* xs, int32_t* res, int n) {
2   seqfor (int i=0; i<n; i++) {
3     res[i] = f(xs[i]) ;
4   }
5 }
```

where the result is written to an array `res`, passed through a function parameter. Below we show how the map function is translated into a parallel for-loop, where we group the iterations in the chunks to amortize the cost of

parallelisation. Note that it is simple to convert the parallel for-loop into a sequential one, by letting the chunk size be equal to n.

Listing 9 C code for a parallel map using a parallel for-loop

```
void parallel_mapf (int32_t* xs, int32_t* res, int n) {
  int chunk = ... //  to be determined
  int n_chunks = (n  + chunk - 1) / chunk;
  parfor (i = 0; i < n_chunks; i++) {
    seqfor (int j = chunk * i ; j < min(n, (i + 1) · chunk); j++) {
      res[j] = f(xs[j]);
    }
  }
}
```

4.1.1 Work efficient

We can easily see that the work performed in this parallel algorithm is work-efficient, as the number of operations is the same as in the sequential version.

4.2 `reduce`

The implementation of `reduce` requires a bit more work. We rely on the assumption that the operator is associative and that the programmer provides the operator's neutral element. We use a concrete `sum`-function to show how we can translate a Futhark `reduce` into an equivalent sequential C version.

Listing 10 A sum program in Futhark

```
let sum  [n] (xs:[n]i32) = reduce (+) 0  xs
```

The equivalent sequential C-version can be written as:

Listing 11 C code for a sequential reduction

```
void sequential_sum (int32_t* xs, int32_t* res, int n) {
  int32_t sum = 0;
  seqfor (int i=0; i<n; i++) {
    sum += xs[i];
  }
  *res = sum;
}
```

Unlike with a `map`, each iteration is no longer independent of each other as the next value of `sum` depends on the previous, so we cannot just easily divide the iteration space out into chunks. If we were to do so, we would need to synchronize the access to the variable `sum`, which can cause the computation to be equivalent to the sequential one due to contention. Furthermore, unless

the operator is commutative, there is no way to ensure that updates are performed using the correct ordering.
In order to use a parallel loop, we divide the computation into two stages. The first stage divides the iterations into chunks, each of which has a unique id. To avoid synchronization, we allocate an intermediate array before the parallel loop starts. We can then reduce each chunk in parallel, where each chunk then writes to it's "private" place in the intermediate array. When all chunks are processed, the calling thread then reduces over the intermediate array, giving us the final result. The parallel pseudo-code is shown in Listing 12

Listing 12 C code for a parallel sum function using parallel for-loop

```
1  void parallel_sum (int32_t* xs, int32_t* res, int n) {
2    int chunk = ... //  to be determined
3    int n_chunks = (n  + chunk - 1) / chunk;
4    // Stage 1
5    // Allocate chunk_sum[n_chunks] = {0};
6    parfor (i = 0; i < n_chunks; i++) {
7      int id = get_chunk_id();
8      seqfor (int j = chunk * i ; j < min(n, (i + 1) · chunk); j++) {
9        chunk_sum[id] += xs[j];
10     }
11   }
12   // Stage 2 - Perform a reduction over chunk_sum
13   int32_t sum = 0;
14   seqfor (int i = 0; i < n_chunks; i++) {
15     sum += chunk_sum[i];
16   }
17   *res = sum;
18 }
```

The parallel reduction is illustrated in Figure 4.1. Since the number of chunks is expected to be small, the second stage is expected to be fast, and hence we perform it sequentially. Note the use of a unique chunk id, which ensures that the reduction is performed in the correct order. Whenever the operator is communicative as well, we can lift such a restriction, which can give us more freedom when scheduling the parallel for-loop.

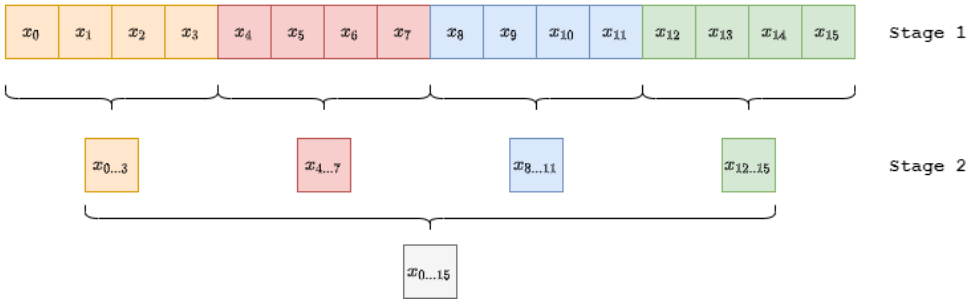


Figure 4.1: Illustration of a parallel reduction on an array with 16 elements, which is divided into 4 evenly sized chunks

4.2.1 Work efficient

It is relatively easy to see that the parallel algorithm is work-efficient. The sequential algorithm has $\mathcal{O}(n)$ operations. The first stage performs exactly n operations in total. The second stage has at most n operations as each chunk is at least of size one. Thus we have that $\mathcal{O}(2n) = \mathcal{O}(n)$ operations are performed.

4.2.2 Segmented reduce

A segmented reduce is handled slightly differently. A segmented reduce is where the input data is grouped into segments, and a reduction is performed over each segment. One can think of the reduction above as a segmented reduce with one segment. But whenever the number of segments is larger than one, the above method does not work directly. Instead, we perform a sequential reduction over each segment, parallelizing only over the number of segments. While there exist approaches which can parallelize over each segment as well, such an implementation requires a slightly more complicated technique. As our benchmarks do not benefit from this optimization, we will leave it as future work to improve our implementation for exploiting more parallelism. Note that the segmented reduction is regular, meaning that all segments are of the same size. Assuming we have n segments with m elements in each segment, the pseudo-code in Listing 13 shows how we are able to parallelize a segmented reduction.

Listing 13 C code for parallel segmented reduction

```
1  void parallel_segmented_sum (int32_t** xs, int32_t* res, int n, int m) {
2    int chunk = ... //  to be determined
3    int n_chunks = (n  + chunk - 1) / chunk;
4    parfor (int k = 0; k < n_chunks; k++) {
5      seqfor (int i = chunk * k ; i < min(n, (k + 1) · chunk); i++) {
6        res[i] = 0; // Neutral element
7        seqfor (int j = 0; j < m; j++) { // sequential reduce over segment i
8          res[i] += xs[i][j];
9        }
10     }
11   }
12 }
```

The sequential algorithm has $\mathcal{O}(n \cdot m)$ operations. We iterate over each segment exactly once through lines 4 and 5. The inner most `seqfor` performs exactly m operations and hence is the total work $\mathcal{O}(n{\cdot}m)$ of parallel algorithm.

4.3 `scan`

The parallel algorithm for `scan` is slightly more complicated than the `reduce`. We use a pre-fix sum as a concrete instance of a `scan`, to explain the parallel algorithm, whose sequential version is shown below.

Listing 14 C code for sequential prefix-sum

```
1  void sequential_prefix_sum (int32_t* xs, int32_t* res, int n) {
2    int32_t accum = xs[0];
3    res[0] =  accum;
4    seqfor (int i = 1; i < n; i++) {
5      accum = accum + xs[i];
6      res[i] = accum;
7    }
8  }
```

The parallel algorithm uses three stages. We omit the (verbose) pseudo-code here and describe the three stages instead. For an input array **xs** and resulting array **res**, we describe the three stages as:

1. In the first stage, we divide the input into *fixed* sized chunks. Then in parallel, each chunk is scanned sequentially. The prefix-sum results from each chunk are written to **res**.

2. In the second stage, we perform a sequential scan over the *last* element from each chunk of **res**, except for the first chunk, which reads the neutral element and the last one, which isn't involved in this stage. The result is written to **res**. This stage gives us the accumulative values up until each chunk.

3. Finally, using the same chunk sizes as in stage 1, we can compute the final stage in parallel. Each chunk first reads the last value from the previous chunk and uses it as its "starting element", except for the first chunk, which again reads the neutral element. Each chunk can then be scanned in parallel as in stage 1, giving us the final result.

The three stages are illustrated in Figure 4.2, which shows how the prefix-sum over an integer array is computed.

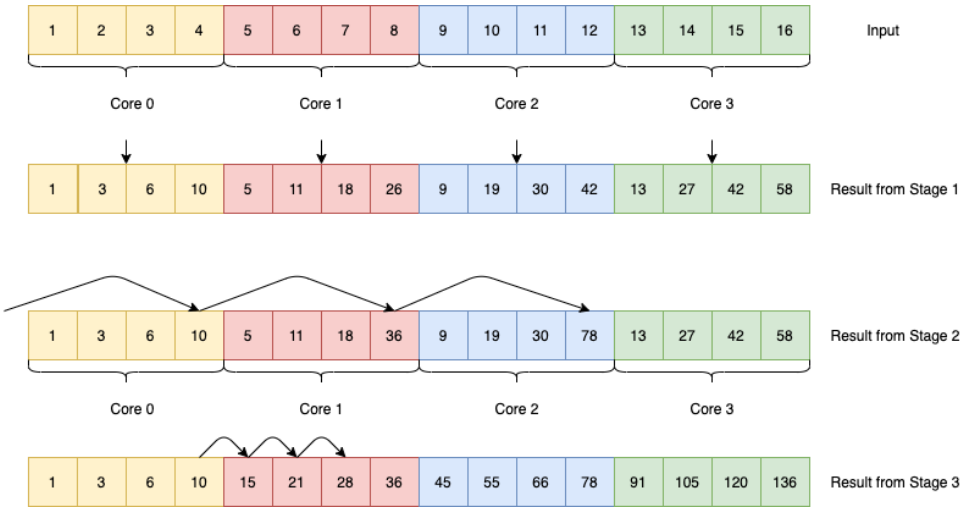


Figure 4.2: Illustration of the parallel scan algorithm for computing the prefix-sum of a 16 element array, which is divided into 4 evenly sized chunks

4.3.1 Work efficiency

The asymptotic work of a sequential scan is $\mathcal{O}(n)$. By looking at the work of each of the three stages, we can see if our algorithm is work-efficient. Stage 1 applies the operator exactly n times as in the sequential algorithm, so it is $\mathcal{O}(n)$. The second stage can, at most, apply the operator n times, since each chunk is at least of size 1. The third and final stage applies the operator n times again. Hence we find that we apply the operator at most $3n$ times, and we have that $\mathcal{O}(3 \cdot n) = \mathcal{O}(n)$ and it's work-efficient.

4.3.2 Segmented Scan

The conversion of a segmented scan is similar to the one of a segmented reduction. We perform a sequential scan over each segment, where we only parallelize over the segments. Likewise, our algorithm for a segmented scan is suboptimal, as we cannot exploit parallelism within a segment. We will also leave such improvements for future work. For brevity, we also omit the pseudo-code for a segmented scan. By using the same deduction method as with the segmented reduce, can one see that the parallel algorithm is work efficient.

4.4 `reduce_by_index`

Finally is the implementation of `reduce_by_index`. We remind the reader that it has the following type and semantics:

`reduce_by_index`: $[m]\alpha \to (\alpha \to \alpha \to \alpha) \to \alpha \to [n]i32 \to [n]\alpha \to [m]\alpha$
`reduce_by_index` *dest f ne is vs*

Listing 15 The semantics of `reduce_by_index` using imperative pseudo-code

```
1 for j < length is:
2   i = is[j]
3   v = vs[j]
4   if i >= 0  &&  i < length dest:
5     dest[i] = f(dest[i], v)
```

Where f is an associative and commutative function with neutral element ne. Here is and vs must be of the same length. `reduce_by_index` is also called a generalized reduction, which reduces a collection of data into k buckets where there is no pattern in the input data, in contrast to a segmented reduce, where the data is grouped into segments. We use a concrete instance of a `reduce_by_index`, called a histogram, for simplicity. A histogram has the semantics shown by the imperative code. Note that we omit the bounds checking of indices.

Listing 16 Imperative pseudo-code for a histogram computation

```
1 for j < length is:
2   i = is[j]
3   v = vs[j]
4   histogram[i] += v;
```

We provide two different parallel algorithms, which can be chosen between at run-time depending on the input sizes. The parallel algorithms are based on Subhistogramming and Atomic updates.

4.4.1 Subhistogramming

In Subhistogramming, each parallel computation is given its own sub-histogram, of the same size as the histogram, along with a chunk of the indices and values used to update the histogram. The idea is that each chunk can work independently on its local histogram, which avoids synchronization of updates on the same cells. A segmented reduction is then performed across the sub-histograms to obtain the final result using the parallel algorithm described in Section 4.2.2. Let $n_{subhistos}$ be the number of sub-histograms used and let m denote the histogram's width. Again we have that n elements are processed.

The pseudo-code for the parallel sub-histogram computation is shown Listing 17, where we omit the code for performing the segmented reduction for brevity. First, we allocate the sub-histograms in line 4. Then each chunk first retrieves its unique id in line 7. Then in line 9 through 11, the sub-histogram is initialized with the neutral element. In lines 13 through 17, the chunk is processed.

Listing 17 Parallel algorithm for histogram using Subhistogramming

```
1  void parallel_subhistogramming (int32_t* dest, int32_t* is, int32_t* vs, int n, int m) {
2    int chunk = ... //  to be determined
3    int n_subhistos = (n  + chunk - 1) / chunk;
4    // Allocate subhisto[n_subhistos, m];
5    // Stage 1
6    parfor (i = 0; i < n_subhistos; i++) {
7      int id = get_chunk_id();
8      // Part 1: Initialize subhistogram belonging to chunk
9      seqfor(j = 0; j < m; j++) {
10       subhisto[id, j] = 0
11     }
12     // Part 2: Perform the updates
13     seqfor (int j = chunk * i ; j < min(n, (i + 1) · chunk); j++) {
14       int32_t ind = is[j];
15       int32_t val = vs[i];
16       subhisto[id, ind] += val;
17     }
18   }
19   // Stage 2
20   Reduce n_subhistos histograms using segmented reduction
21 }
```

4.4.1.1 Work efficient

The sequential algorithm performs $\mathcal{O}(n)$ work. The parallel algorithm first initializes each sub-histogram, which creates $\mathcal{O}(n_{subhistos} \cdot m)$ work in total initialization work. The second part of stage 1 performs $\mathcal{O}(n)$ work in total as in the sequential algorithm. The segmented reduction also performs $\mathcal{O}(n_{subhistos} \cdot m)$ work as shown in Section 4.2.2. In total, we have that the parallel algorithm performs $\mathcal{O}(n_{subhistos} \cdot m + n)$ work, and hence is the algorithm not work-efficient. As we usually only create a small number of sub-histograms, we can assume that $n_{subhistos}$ is a small constant, which is at most the number of cores. However, when m is large, this parallel algorithm can become slow. We initially only implemented this approach but found that the performance was too slow on some benchmarks, even slower than the sequential version. When m is large, initializing each sub-histogram and the reduction afterward adds significant overhead and caused too much additional work, causing the slowdown. Instead, we try to provide an alternative whenever such cases occur.

4.4.2 Atomic Updates

With an Atomic updates approach, all threads perform updates on the same histogram, but each update is wrapped inside an atomic operation. The algorithm presented here is based on the work by S. Hellfritzsch[Hellfritzsch, 2018] in his master's thesis. The generic atomic approach is shown below, which uses an `atomic_operation` to update the value.

Listing 18 Generic atomic histogram approach

```
1  void parallel_atomic_hist (int32_t* dest, int32_t* is, int32_t* vs, int n, int m) {
2    int chunk = ... //  to be determined
3    int n_chunks = (n  + chunk - 1) / chunk;
4    parfor (i = 0; i < n_chunks; i++) {
5      seqfor (int j = chunk * i ; j < min(n, (i + 1) · chunk); j++) {
6        int32_t ind = is[j];
7        int32_t val = vs[i];
8        atomic_operation { histo[ind] += val }
9      }
10   }
11 }
```

Note that there is no overhead in allocating additional memory, nor is there a need for an additional pass with this strategy. This approach is then clearly work-efficient. However, as there is no free lunch, this approach can become relatively slow in practice when parallel computations access the same bucket, causing the writes to become sequential. Furthermore, there is not an explicit atomic operation for all binary operators. Since we need to support user-defined operations, we need to ensure that we provide an atomic update operation for all cases. We provide three levels of strategies, each of which becomes less and less efficient. We start with the most efficient, where the hardware supports the atomic operation.

4.4.2.1 Binary operation on integral scalar values

We start of when the operator is defined on integral scalar type values where the operation is one of `add`, `sub`, `and`, `xor`, `or` or `nand` and the size is 1, 2, 4, 8 or 16 bytes in length. When we identify such a case, we can use the `__atomic_op_fetch` presented in Section 2.2.3.1. Following the example from the previous section, our atomic update approach becomes:

Listing 19 Atomic histogram approach on scalar values

```
1  void parallel_atomic_hist (int32_t* dest, int32_t* is, int32_t* vs, int n, int m){
2    int chunk = ... //  to be determined
3    parfor (i = 0; i < (n  + chunk - 1) / chunk; i++) {
4      seqfor (int j = chunk * i ; j < min(n, (i + 1) · chunk); j++) {
5        int32_t ind = is[j];
6        int32_t val = vs[i];
7        atomic_add_fetch(&histo[ind], val);
8      }
9    }
10 }
```

4.4.2.2 Binary operators on one memory location

The next approach is based on the usage of atomic compare-and-swap (CAS) operations. Recall that a CAS operation atomically compares the expected value to the current value, and if they are equal, perform the swap with the desired value. Otherwise, the current value is returned in the expected value. Upon a successful swap, the CAS operation returns true. We can leverage such an insight to obtain an atomic update by merely casting any primitive type to an integral scalar value of the same size, e.g. for `float` we can cast it to `int32`.

Listing 20 Atomic histogram approach on floats

```
1  void parallel_atomic_hist (float* dest, int32_t* is, float* vs, int n, int m) {
2    int chunk = ... //  to be determined
3    parfor (i = 0; i < (n  + chunk - 1) / chunk; i++) {
4      seqfor (int j = chunk * i ; j < min(n, (i + 1) · chunk); j++) {
5        int32_t ind = is[j];
6        float val = vs[i];
7        float expected = histo[ind]
8        int done = 0;
9        while (!done) {
10         float new_val = val + expected;
11         done = atomic_cas((int32_t*)&histo[ind],
12                           (int32_t*)&expected, to_bits_32(new_val));
13       }
14     }
15   }
16 }
```

We describe the update in three steps:

1. The current value is stored in the variable `expected` in line 7. We perform the update in line 10, which is a simple addition as before.

2. In line 11-12, we try to perform the update using a CAS operation. A few things to note. We are directly casting the pointer values to integer values as this won't change the bits when the CAS operation eventually

reads them. Since the `new_val` is passed by value, we can not perform a direct cast without changing the bits and thus the value. Instead, we use an explicit cast function, which converts it to a type of `int32` without changing the bits.

3. If the CAS operation was successful, then we are done, and the while loop exits. Otherwise, the current value is written into `expected` and we perform the operation again until we eventually succeed.

This approach can be used to implement all binary operators, which works on single memory locations, e.g., floats, doubles, and so on, as long as the size if one of $1, 2, 4, 8, 16$ bytes.

4.4.2.3 Locking based approach

When neither of the above two methods is applicable, we fallback to a locking approach. For brevity, we only provide the updating code, omitting the `parfor` and `seqfor` notation. We also continue the example of using 32-bit integers as our histogram type, but it's easy to see that this approach can be used on any type.

Listing 21 Snippet of an Atomic histogram update using a locking based approach

```
1  ...
2  int32_t ind = is[j];
3  int32_t val = vs[j];
4  int done = 0;
5  while (!done) {
6    if (atomicExch ((int*)&locks[ind], 1) == 0) { // Acquire lock
7      // Critical section - start
8      histo[ind] += val
9      // Critical section - end
10     memory_barrier();
11     atomic_write(&locks[ind], 0); // Release lock
12     done = 1;
13   }
14 }
15 ...
```

We use an array of integers called `locks`, which initially contains all zeros, indicating that all locks are unlocked. In practice, we allocate a static array of a fixed size and then use the modulo operation to obtain a specific lock index. By using an explicit locking mechanism, we can perform any operation within the critical section. We acquire the lock by using an `atomicExch` operation, which returns the previous value of `locks[ind]`. When the previous value was 0, we know that we have acquired the lock successfully. When we are done with our update, we then release the lock using an atomic write operation.

Note the use of a memory barrier, which ensures that we do not release the lock before we are done with the update.

The reader may ask why we are not using `mutex`es instead, which can provide the same mechanism. Mutexes adds memory overhead, since each `mutex` uses 40 bytes on 64-bit machines running Linux, compared to the 4 bytes used by an integer. When the histogram is large, which we can expect since we are using Atomic updates rather than Subhistogramming, we would need relatively many locks to avoid contention, causing large memory overhead. Such overhead can be limited by using the more primitive integer array as locks combined with an atomic exchange operation.

4.4.3 Choosing a cut-off

For choosing between Subhistogramming and the Atomic update approach, we use a simple heuristic. Whenever the condition $n_{subhistos} \cdot m \leq n$ holds, we choose Subhistogramming; otherwise, we use the Atomic update approach. The heuristic is based on the result from the asymptotic bound from Subhistogramming, i.e., $\mathcal{O}(n_{subhistos} \cdot m + n)$, which when the condition holds the n term dominates and Subhistogramming is work-efficient. Our results show that such a simple heuristic is sufficient on our micro-benchmarks, but more work is required on optimizing the implementation. We will leave further optimizations for future work.

Chapter 5

Run-time system

This chapter presents our approach to the run-time system in Futhark. We start with automatic granularity control, which presents our online algorithms for flat- and nested parallelism as well as irregular parallelism. It then presents our approach to work management within our run-time system. Finally, we briefly present a few memory policies, which makes memory management easier.

5.1 Automatic Granularity Control

In this section, we present our online algorithms for automatic granularity control. We will use parallel loops to express parallel tasks, where the iterations can be divide into continuous chunks and distributed onto the hardware cores for parallel execution. Our approach handles the scheduling of regular- and irregular parallelism as two separate cases. To distinguish between the scheduling approaches of regular- and irregular parallelism, we will call the scheduling of regular (nested) parallelism as static scheduling, while the scheduling of irregular parallelism is called dynamic scheduling. We first present our approach for regular parallel computations, which might contain nested parallelism.

The simplest scheduler implementation creates as many subtasks as there are cores and partitions the iterations evenly out among these subtasks. We will call such an approach for *even partitioning*. When the work of the task is much greater than the overhead of creating these subtasks, this is an easy way of exploiting as much parallelism as possible, with minimal overhead. But when the work is not greater than the overhead, this simple method can create too small chunk sizes to amortize the cost of parallelization. In such a case we want to determine a chunk size, which amortizes the cost.

Our approach to automatic granularity control is inspired by the online prediction algorithm in [Acar u. a., 2019], which performs automatic granularity control using an oracle-guided scheduling approach. While their algorithm is for `fork2join` parallel constructs, we extend their approach to parallel for-

loops, which is more suitable in the context of Futhark.

As in [Acar u. a., 2019], we use an abstract notion of "work", which ties the sequential execution time of a task to a cost function, describing the asymptotic work. For a given task, at a high-level, our approach use samples of execution times from previous runs to predict future runs' work (of the same task). Under a few assumptions, the algorithm can predict the work of a task by estimating the constant associated with the asymptotic work through an iterative online algorithm. By quantifying the cost of parallelization, we can compute the granularity of a task, i.e., the chunk size, which amortizes the overhead. But in order to perform granularity control, our algorithm needs to know the constant associated with the task; however, the constant is estimated from running the tasks. This creates an initial circular dependency. We cannot run the task sequentially as the time penalty on a large task is potentially huge. On the other hand, the potential time penalty for executing a small task in parallel is much less. Our algorithm breaks this interdependency by performing an even partitioning initially, to obtain sequential execution time samples and use these to estimate the constant. The estimated constant is then subsequently used to predict future runs' work and perform granularity control. When the task is executed again, more samples are collected, and our estimate becomes better and better. The algorithm is then able to gradually determine the granularity, which amortizes the overhead more precisely. Even partitioning comes with a small overhead but is much safer when the task turns out to be large. The algorithm most likely only performs the mistake of executing small tasks in parallel a few times. As after it has done it, it will realize that it should have executed the task sequentially. The next section presents our algorithm more formally.

5.1.1 An online algorithm for regular parallelism

In the following we let $f(i, j)$ denote a parloop function, which takes an iteration range (i, j), where $i, j \in \{0, ..., n-1\} \wedge i < j$. For example from our parallel algorithm for `reduce` from Chapter 4, $f()$ is a function containing the sequential loop in stage 1. Here n denotes the input size, i.e., the number of iterations the parallel loop shall execute. We call a parallel for-loop involving a parloop function $f()$ a task, which the scheduler can create subtasks for. A subtask contains the closure of the parloop function along with an iteration range, which can later be distributed among the cores for parallel execution. Additionally, let $g(n)$ denote the corresponding sequential function of a task, i.e. the result of executing $g()$ or a parallel task using $f()$ is the same. In our implementation $g()$ is the same function as $f()$, i.e. $g(n) \equiv f(0, n-1)$, but just executed differently. Finally we state our assumptions.

First, we assume that the execution time of $g(n)$ is predictable in its input size n. Second, we assume that the execution time of $g(n)$ is linear in it's input size, i.e. it has the asymptotic $\mathcal{O}(n)$. The latter is different from [Acar u. a., 2019], which used the programmer provided cost functions to describe

asymptotic work. In Futhark, we want to avoid such involvement from the programmer, and we rely on a linear relationship assumption instead. While such an assumption is suitable in many cases, there are certainly cases where this is not true - we will discuss the implications our assumptions might have later.

The question is now, how do we decide how many subtasks (if any) should be created for a given task? The next section first handles the edge case of deciding between sequential or parallel execution.

5.1.1.1 Sequential execution

The main part of the algorithm presented is trying to quantify whether it's worth to parallelise the given task or sequentialise it when it contains no more than a small amount of work. To formally quantify the cost of parallelisation, let κ denote the smallest amount of work (in units of time) that would profitable to parallelize on a given machine. The value of κ should be large enough to amortize the cost of creating and managing a single subtask. As in Acar u. a. [2019], we employ the policy: On some input of size n, if we can obtain the result of a task by executing it sequentially, i.e., execute $g(n)$, in less time than κ, then we should run $g(n)$. Let $T_{g(n)}$ denote the running time of $g(n)$, then by our linearity assumption we have the relationship

$$C \cdot n \approx T_{g(n)} \tag{5.1}$$

for some constant $C > 0$. The online algorithm aims to estimate C, through the ratio $\frac{T_{g(n)}}{n}$, by sampling runs of $g(n)$, and use the above relationship to predict whether a call to $g(n)$ takes less time κ. To obtain an initial estimate of C we perform even partitioning as described above. The penalty is roughly κ times the number of subtasks created, if the task turned out to be small, which is manageable. Finally, we describe how to obtain a measurement of $T_{g(n)}$ from parallel subtasks. Recall that $g()$ is the same as $f()$, so we can measure the time of each subtask and sum them up to obtain our estimate of $T_{g(n)}$. More precisely, let $T_{f(n_i)}$ denote the time to execute a subtask with n_i iterations for the i'th subtask using the parloop function $f()$. We then have the relationship

$$\sum_{i=0}^{n_{subtasks}} T_{f(n_i)} \approx T_{g(n)} \tag{5.2}$$

Note that the time measurement $T_{f(n_i)}$ include the time to perform the function call. This gives us a slight overestimate of the computation time of $T_{g(n)}$ and hence C. The algorithm uses this to make *safe* sequentialisations, i.e. only sequentialise computations which are surely less than κ. Subsequent runs of the same task (on perhaps different input of n), will exploit C to predict whether a call to $g(n)$ will take less time than κ, and if so, will run $g(n)$.

5.1.1.2 General Granularity control

Whenever the run-time system receives a task and decides it to be scheduled in parallel, it also has to figure out how many subtasks are worth creating. In the case when no estimate of C is present, we use the procedure described above, i.e. perform even partitioning. But having an estimate of C we can estimate the minimum amount of work is needed to justify creating a subtask. Let n denote the number of iterations for a parallel task using the parloop function $f()$ with estimated constant C, then we can estimate the minimum amount of iterations for each subtask by replacing the right-hand side of Eq. 5.1 with κ. By rewriting it, we then have the minimum number iterations is:

$$n_{chunk} = \max\left(\frac{\kappa}{C}, 1\right) \tag{5.3}$$

where we floor the minimum number of iterations to one. Given this, we can estimate the number of subtasks to create as

$$n_{subtasks} = \left\lfloor \frac{n}{n_{chunk}} \right\rfloor \tag{5.4}$$

which makes it worth while to parallelise the computation. Note that $n_{subtasks}$ is greater than one; otherwise, the task would be executed sequentially. Furthermore, under the assumption that the workload is regular, we cap $n_{subtasks}$ to the number of cores, as this would be the maximum amount of parallelism obtainable on the particular machine with minimal overhead. We then have that

$$n_{subtasks} = \min\left(n_{cores}, \left\lfloor \frac{n}{n_{chunk}} \right\rfloor\right) \tag{5.5}$$

are created for any given task when its scheduled for parallel execution.

5.1.1.3 Nested parallelism

In the case of nested parallelism, each level maintains its own estimate of C. For example, if we have two tasks with the parloop functions f_{outer} and f_{nested}, then they will have their own estimates C_{outer} and C_{nested}, respectively. Note that C_{outer} contains the work performed by f_{nested} in its estimate.

For a single level of parallelism, the online algorithm described above suffices for automatic granularity control. However, when we measure the execution time of f_{outer}, the description so far does not take into account that some nested computations might be executed in parallel. If f_{nested} is executed in parallel and if we are not careful with our time measurements, then C_{outer} could contain time measurements, including the overhead of parallelization from scheduling f_{nested}. Hence would time measurements stemming from computations containing nested parallelism more likely exceed κ and f_{outer} would be executed in parallel, when it should have been executed sequentially. Instead, we must exclude any timings that involve creating and

managing parallel subtasks, only including the times from executing the par-loop functions. We finally sum up all sequential execution times stemming from the nested computations, giving us an estimate of C_{outer}, which only contains the actual work performed. This allows us to perform granularity control for nested parallelism.

Note that our algorithm does not consider the depth of the parallelism nor the system's current state; it merely computes how many subtasks to create using Eq. 5.5 at each level. If the machine is saturated, it can become suboptimal to create these subtasks, as there might not be any available threads to take the work. The thread that created the subtasks might have avoided subtask creation overhead and finished the task slightly faster by doing it sequentially. However, maintaining the current state of the system, e.g., keeping track of which threads are idle, also comes with an overhead. Further, finding a good heuristic for when to avoid subtask creation and when to create subtasks is difficult in practice. Instead, our approach aims to keep it simple and avoid such heuristics.

5.1.1.4 Discussion

In this section, we discuss some of the shortcomings, our approach might encounter due to our assumptions.

First, our assumption that there exist a constant, which can describe the work for any input size, i.e. $T_{g(n)} \approx C \cdot n$, holds for any n. However, in practice, one can observe that an estimation of C depends on the input size. For example, for two different input sizes, one might be small enough to fit into the caches, while the other might not. This can create different estimates of C. Even two identical calls to the same function might result in different run-times if one call already has cached data available. In the worst case, an underestimation of C could result in running a task sequentially, which could be executed faster in parallel. To minimize such effects, we accumulate run-times and iterations across all runs of the same task, which gives us a more robust estimate of C.

Second, in cases where our linearity assumption does not hold, we might, in theory, end up creating too many or too few subtasks, resulting in over-parallelization or under-parallelization, respectively. For example, if the true cost function of a parallel computation is $\mathcal{O}(n^2)$, then our algorithm will estimate too-small granularities, as it will overestimate the constant C, which results in over-parallelization. However, such over-parallelization is limited in practice, since we cap the number of subtasks. Our linearity assumption cannot result in under-parallelization, since we are using parallel loops, which has a minimum complexity of $\mathcal{O}(n)$.

Third, in the case of nested parallelism, we might also face some issues. For example, consider a function which sums up the row vectors for a $n \times m$ matrix of integers. In Futhark such a function can implemented using a `reduce` with a nested `map` (Listing 22).

Listing 22 A program in Futhark which sums up the row vectors

```
1 let vector_sum [n][m] (xss:[n][m]i32) = reduce  (map2 (+))  (replicate m 0) xss
```

Our algorithm will correctly estimate the constant associated with the `map`, C_{map}, since the complexity is linear in the dimension of the vector, i.e., $\mathcal{O}(m)$. However, the `reduce` has no knowledge about the work complexity of the inner `map` and assumes that its work can be estimated through its constant C_{reduce}. In other words, the `reduce` only considers the complexity with respect to n, i.e., the number of vectors. If the dimension of the vector, m, changes during run-time our estimation of C_{reduce} will become invalid. In such a case, the complexity of the `reduce` should really be $\mathcal{O}(n \cdot m)$, which includes the complexity of the nested parallelism. While we did not observe any current issues because of this, a program in the future might suffer from our simplification. To solve such an issue, one could let the compiler provide the cost function based on the levels of nested parallelism or even use static analysis to infer the cost function[Jost u. a., 2010]. If these methods are not precise enough, one could allow the programmer to provide the cost function as a last resort. However, even if we were to let the programmer provide a cost function, further problems arise. Our compiler might perform transformations such as combining adjacent `map`s, i.e., fusion[Blelloch u. a., 1993], which can provide benefits by increasing the granularity and reduce the need for synchronization points. But if the two adjacent `map`s have different cost functions, it is not clear what the cost function of the new fused `map` should be. We could prevent such transformations in these cases but at a cost of losing the benefits of the transformation.

In summary, our assumptions are associated with a couple of open issues.

5.1.2 Granularity control for irregular parallelism

The above method assumed that the workload is predictable, where the execution time could be described as a function of the input size n. When the workload of each iteration is not predictable, e.g. when the work is data-dependent, the discrepancy between the work across cores can be large and result in under-utilization. One can imagine a worst-case scenario where a single thread might end up doing the majority of the work, while the remaining threads finish quickly and become idle. Hence when we suspect that there might be an uneven workload across each subtask, we want to load balance the work during execution dynamically.

In our implementation, we use a compile-time approach to determine if a particular task contains irregular work. The information is then given to the scheduler, which will use a dynamic scheduling approach.

5.1.2.1 Approaches to dynamic scheduling

For dynamic scheduling, we tried to employ several automatic granularity control approaches to achieve load-balancing. The initial idea was to employ a strategy similar to how TCP handles congestion control. The idea was that whenever a task is marked as being potentially irregular, we start by chunking the iterations evenly out onto each core as in even partitioning. When threads initially start performing work, it creates a subtask containing only one iteration initially and executes it. The next time it takes two, and then four and so on. In the case that work ended being approximately load-balanced, we would not endure a large penalty for the additional subtask creations. When a thread finished its work, it will try to acquire more work. Whenever a new subtask was acquired, implying that the task indeed was irregular; we would halve the chunk size from the acquired subtask, similar to how TCP backs off whenever it detects congestion on the network, e.g., when packages are lost. Unfortunately, we observed that whenever a task was heavily irregular, the iteration counter would quickly go towards one as a lot of acquiring was happening. This resulted in poor performance due to the overhead of subtask creation. Other approaches to irregular parallelism is the *guided* scheduling method used by OpenMP, briefly described in Section 3.2.1.2. But such a heuristic only works on a subset of irregular programs, more specifically on those programs where the majority of the workload is at the end of the iterations. A property we can not predict for Futhark programs and hence not applicable.

Instead, we use an online algorithm which ensures, on average, that subtasks created justify the cost of creating and managing a single parallel subtask. More specifically, we use κ as a guideline on how much work is needed to amortize the cost by continuously adjusting the chunk-size based on the value of C. We still use our linearity assumption here, which is a crude simplification, as it does not make sense to define a cost function in this case.

5.1.2.2 An online algorithm for irregular parallelism

As with the initial idea, we initially perform an even partitioning of the iterations. When the thread initially starts executing the subtask it only takes a small chunk at the time. The chunk is initially a single iteration, which is used to gather initial information about C. Note also that C is reset before each irregular task. As the work-load is data-dependent, C estimates might vary between irregular tasks and cannot be reliably used again as in the previous algorithm. As we sample more information from previous runs, we accumulate the sampled run-times and iterations and our C estimation becomes closer and closer to the *average* run-time per iteration. Accordingly, we readjust the number of iterations per chunk. Before each subtask is executed we recompute the number of iterations needed to amortize the cost of subtask creation, based on the same method described in Section 5.1.1, i.e. $n_{chunk} = \frac{\kappa}{C}$. This becomes our chunk size, leaving the remaining iterations for

later execution. The process is illustrated in Figure 5.1 from the perspective of a single thread, which is assigned 10 iterations initially and uses 4 steps to process its work. In the first step, it will take 1 iteration to obtain an estimate of C. In the second step, it uses $n_{chunk} = \frac{\kappa}{C}$ to find it's chunk size and finds that it needs to consume 4 iterations to amortize the cost of subtask creation. This again gives an updated estimate of C, and at the third step, C has grown, so it only takes 2 iterations. Finally, is there only 1 iteration left and the thread processes that one. When the thread runs out of work, it will try to acquire more work from other threads, e.g., through stealing (See Section 5.2), which ensures work-load balancing. While we only showed the process from a single thread's perspective, note that C is *shared* among other threads working on the same task, and updates are done using atomic operations. There is an important point to be made here. If we were to chunk the

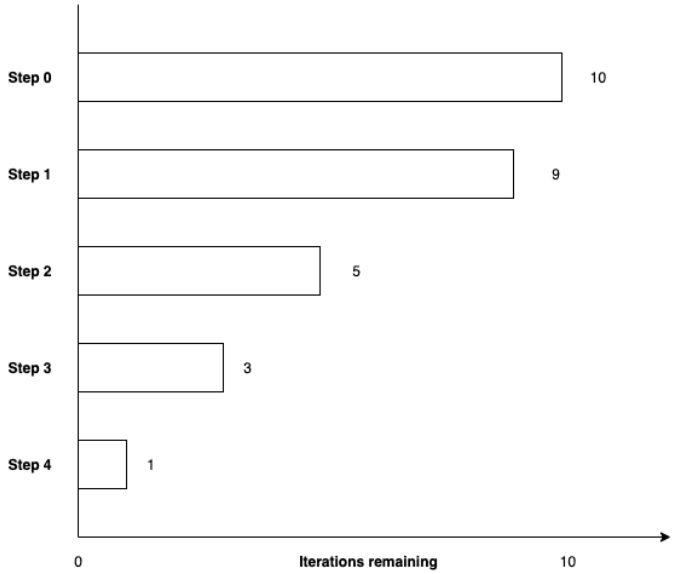


Figure 5.1: Illustration of our online algorithm for irregular parallelism from a single thread's perspective

iterations using some small chunk size before execution starts, there would be no (easy) way of combining those into continuous chunks again. By keeping large blocks of iterations, we have the opportunity for a subtask to consume a large chunk if needed, making this algorithm similar to lazy task creation.

Finally, note that irregular parallelism can only be scheduled for `map`s and `reduce`es, where the latter requires that the operator is commutative. As iterations might get moved around from core to core, we cannot ensure that the operator is applied in the correct order when it is only associative. For `scan`, will our parallel algorithm not work using a dynamic scheduling approach, as it requires that the two stages use the same fixed chunk sizes. Such a requirement is difficult to maintain in a dynamic environment. Though it is uncommon to use irregular code patterns with these operators, so such

a limitation is not a problem in practice.

5.2 Work management

We will need a way of managing the work which needs to be performed for a given task. An initial approach is to have a centralized work queue that all threads enqueue work onto and all threads deques from. While such an approach can be easy to maintain and implement, it scales poorly as more and more threads try to access the same queue. Instead, it is more scalable to have each thread have its own local work queue, which it can manage independently. However, by letting each thread have its own local queue, another problem arises. How should work be distributed among the threads? One approach is to use *work-sharing*. When one's own queue gets "large", subtasks are sent to other threads. This can ensure that work is distributed evenly among other threads. But if all threads have a lot of work, they will all try to send work to each other, creating communication congestion, where threads spend a lot of time trying to off-load work. It is also difficult to estimate when it should off-load work as it depends on the state of other threads.
Instead, one can use a *work stealing* approach, where an idle thread becomes a "thief" and try to steal work from another thread, called the "victim". This approach has much better properties than work-sharing when the system is under load. For example, communication only happens whenever a thread is idle. Such a scheme can enable workload balancing at a much lower cost, as the thieves were idle anyways. Many different work-stealing strategies exist. A key property of a good work-stealing algorithm is that it is lightweight. It should not spend much time figuring out which queue to steal from. The simplest one, while also working well in practice, is a randomized work-stealing strategy. Here whenever a thread runs out of work, it randomly selects a victim. If it fails to find any work in the victim's queue; it will try to select another victim randomly again until it finds some work. However, one problem is that work can sometimes fail to be distributed evenly among the cores as they fail to steal from the correct victim, which holds a lot of work.

Work-sharing and work-stealing can be complementary and can easily co-exist. In our implementation, we make use of both, depending on the current state of the scheduler. When all threads are idle, and a task initially arrives, it is subtasks will be distributed using a work-sharing approach, which ensures that work is distributed consistently. If the task contains any nested parallelism, the nested task will not be distributed using work-sharing as this will create communication congestion. Instead, the nested subtasks will be pushed onto the thread's own queue and distributed through work-stealing. To better understand our choice, we should look at how queues for work management is implemented.

5.2.1 Data-structures

We will need a safe and efficient way of storing and distributing work among the threads. We will follow the classical producer-consumer pattern, where one or more producers create work, which is later consumed by a consumer. Before diving into some popular implementations, we should think about which properties such a data-structure should have.

Thread-safe As the data-structure is shared among threads, we would want to ensure that it's thread-safe, for example, using one of the mechanisms described in Section 2.2.1.

Space-efficient It should not use more space than necessary, but should not need to allocate memory frequently either. A common choice is to use a circular array buffer, which is a more efficient use of an array.

Locality of work It is common that recent inserted work provide better locality for the thread that inserted it, i.e., it's desirable to let the latest work be executed onto the same core. To optimize for such behavior, a LIFO behavior is appropriate, while stealing threads consume from the "back" making it a FIFO for them.

For work-stealing we would want it to support the following operations, `insert`, `consume` and `steal`.

5.2.1.1 Job queue using mutexes and condition variables

The first implementation shown is a classical job-queue based on the use of `mutexes` and condition variables. As described in Section 2.2.1 can the combination be resource-efficient. The common design pattern for ensuring mutual exclusion is shown in Figure 5.2 using function names from the POSIX thread library. We also use a generic name Object as the element that our job queue manages. Note that `insert` and `consume` are *blocking* functions. Such behavior can both be an advantage or disadvantage. Whenever there is no work to perform, the thread can sleep, freeing up computing resources to the system, only waking up whenever some work has been assigned to the thread. However, in a work-stealing environment, it is usually not beneficial to block, but instead, threads should try to look for work. Finally, thieves should not be blocking others' queues. Instead, they immediately return whenever the queue is empty.

```
1  void insert(Object o) {
2    pthread_mutex_lock()
3    while (queue_is_full) {
4      pthread_cond_wait()
5    }
6    // Now there's room - insert work
7    pthread_cond_broadcast()
8    pthread_mutex_unlock()
9  }
```

(a) Producer `insert`

```
1  Object consume() {
2    pthread_mutex_lock()
3    while (queue_is_empty) {
4      pthread_cond_wait()
5    }
6    // Now there's work - consume
7    pthread_cond_broadcast()
8    pthread_mutex_unlock()
9    return work;
10 }
```

(b) Consumer `consume`

```
1  Object steal() {
2    pthread_mutex_lock()
3    if (queue_is_empty) {
4      pthread_mutex_unlock();
5      return NULL;
6    }
7    // Now there's work - consume
8    pthread_cond_broadcast()
9    pthread_mutex_unlock()
10   return work;
11 }
```

(c) Thieves `steal`

Figure 5.2: Common design pattern for ensuring mutual exclusion with `mutex` and condition variables

5.2.1.2 Chase-Lev's lock-free deque

A very popular data-structures for storing work is a concurrent deque[Chase und Lev, 2005; Lê u. a., 2013]. The deque is specially designed for work-stealing environments and provides concurrent access for threads without the need for locking to ensure thread-safety. The highly efficient scheme is based on a collection of array-based double-ended queues (deque), where the owner pushes and pops from the bottom, while stealing threads steals from the top. Note that *only* the owner is allowed to push work onto its deque, and hence is the data-structure mostly applicable in a distributed setting, where each thread is managing its own deque. This type of deque has shown to become an industry-standard in terms of efficiency due to its limited need for synchronization across threads to ensure consistency, making it very efficient when many tasks are produced. The deque has three main operations:

- `push_back(Object o)`: Pushes o onto the bottom of the queue
- `Object pop_back()`: Pops an object from the bottom of the deque if the deque is not empty. Otherwise returns `Empty`

- `Object steal()`: If the deque is empty, returns `Empty`. Otherwise return the element stolen from the top of deque. But if the thread loses the race it returns `Abort`.

The deque only keeps track of two indexes, the `top` and `bottom`, which indicates the two ends of deque. If `bottom` is less than or equal `top` then the deque is empty. On a `push_back()` operation the `bottom` index is incremented, while it's decremented on a `pop_back()`. A *successful* `steal` operation increments the `top` index. Note that the bottom of the deque works like a stack (LIFO) data-structure while the top works a queue (FIFO). This increases the locality as newly pushed subtasks are more likely to use the already cached data. Under the constraint that only the owner of the deque is allowed to invoke `push_back()` and `pop_back()`, synchronization is only needed in two cases. Whenever an owner invokes a `pop_back()` *and*, there is exactly one object left in the deque; it has to ensure that it did not lose a race to a stealing thread. Similarly, whenever a thread steals from the top, it also has to ensure that it has won the race with another potential stealing thread. Such condition checking can be done through a single atomic compare-and-swap (CAS) operation. Finally, to ensure a sequentially consistent memory, one needs to make use of memory barriers as described in Section 2.2.3.1, ensuring that the instruction orders are not reordered during execution.

5.2.2 Discussion

The two presented data-structure implementations, while achieving the same goal, differ widely. We list the advantages and disadvantages of each below:

Job-queue

+ Efficient as threads can be woken up only when work is present, not consuming CPU resources otherwise
+ Easy to distribute work evenly among cores as any thread can push work to any queue.
- Slow when many threads try to access same queue, as they are blocking each-other making failed attempts expensive as each stealing thread can only check the queue one at time.
- Requires kernel intervention for waking up threads or if there is lock contention.

Chase-lev's deque

+ Lightweight as synchronization is only required in two cases and uses fast atomic hardware supported operations.
+ Efficient when many threads are trying to steal from same queue as they can concurrently race for the work, making failed attempts cheaper than the with job-queue.

- Difficult to ensure proper distribution as we rely on threads are stealing from the correct queue, i.e. threads acquire work through a steal.
- Resource inefficient as threads will run "full tilt" all the time, since there is no mechanism for de-scheduling and waking up threads in a consistent way.

We can see that the job-queue can be used in both work-sharing and work-stealing schemes, while Chase-lev's deque is only applicable in the work-stealing scheme. In our work, we provide implementations for both data-structures but use the job-queue for several reasons. In Futhark, the majority of the tasks will be scheduled statically with one level of parallelism, where we cap the number of subtasks created to the number of threads. As a result, are there most likely only one subtask present in each queue at the time, which contains approximately the same amount of work, and stealing is rarely required. For such programs, we observed that the job-queue performed better than Chase-lev deque. This performance gain is due to the job-queue's ability to distribute work directly onto other thread's job-queue. We saw a much larger variation in benchmarks for programs when using Chase-lev's deque as threads failed to steal from the right queue consistently. Another issue with Chase-lev's deque is that only the owner can push work onto its deque, which results in the main thread's deque essentially becomes centralized, which all threads initially steal from. Chase-lev's deque is more appropriate for `fork2join` style programs, which can avoid letting a deque becoming centralized. While you can implement parallel loops using `fork2join`, we did not explore such an option in this thesis.

The results with the job-queue showed much more consistent performance for such programs. As such is a work-sharing approach more suitable for the majority of tasks in Futhark. Furthermore, since Futhark is used to accelerate compute-intensive parts of a larger program, we do not want to consume resources in between such computations. While we did explore the option of using UNIX signals to only wake up threads when needed, the above disadvantages out favored the need to use Chase-lev's deque.
In our implementation, we only enable work-stealing in case of nested- or irregular parallelism. For nested computations, the subtasks created are pushed onto the thread's own queue, and distributed through work-stealing.

For irregular workloads, the subtasks are initially distrusted using work-sharing. When threads run out of work, they will try to steal more work from other threads. Here we saw a small performance penalty when using the job-queue. With a small modification to the work-stealing strategy, we were able to limit the need for stealing in this case. We use a half-work stealing strategy, where we steal half of the iterations at a time, while previously, we either stole the entire subtask or only a small chunk of it. When stealing an entire subtask, it would often leave the victim with no work, and it would be forced to steal again, increasing the communication overhead. On the

other hand, stealing a small chunk would quickly leave the thief with no work, again increasing overhead. A half-work stealing approach is hence a compromise between the two.

5.3 Memory management

As memory needs to be handled manually in C, it's relevant to set up some memory management policies. As Futhark programs are meant to be alive for extended periods of time, it is important to release memory back to the system whenever it is not needed any more. The run-time system for the sequential C back-end already employs a memory management system, which ensures that allocated memory is released back to the system whenever there are no more references to the memory. This is handled by wrapping every memory allocation into a `memblock` struct, which then keeps track of the number of references to the memory allocated. But memory management with the multicore back-end can become much more difficult to manage when different threads are allocating and sharing memory. For the multicore back-end we employ the following memory policies

1. Whenever a task or subtask is executed, where the result is returned back to the caller, the memory must already be allocated before the task or subtask is started.

2. A subtask should never free any memory not allocated by it self.

The first policy ensures that the memory allocations cannot be a "side-effect" of running some task or subtasks. This follows naturally for a pure functional language. The second policy ensures that for every memory allocation within a subtask is also freed within the subtask. This policy make it easier for the compiler to keep track of allocations as it can just insert the corresponding free at the end of the scope that the allocation was made in. These policies enable simple memory management.

Chapter 6

Implementation

This chapter describes the implementation of the new multicore back-end, which includes our work for both the compiler and run-time system. A key design goal is that the compiler and the run-time system should be two separate components, such that either can be subject to future changes while avoiding changes to the other. We design an API for the run-time system, which the generated C program interacts with. This creates an abstraction between them, which hides away the implementation details of the run-time system from the compiler. Before presenting the new code-generator and scheduler, we first give an overview of the new multicore pipeline.

6.1 Compiler and Code generation

The Futhark compiler's pipeline is shown in Figure 6.1.

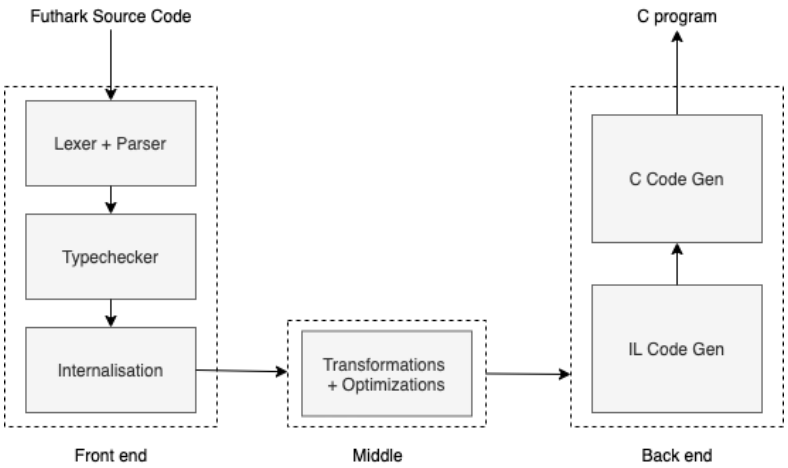


Figure 6.1: Data flow diagram of the Compiler pipeline for Multicore compilation

We put our focus on the back-end, which is responsible for generating

code based on the translations to parallel constructs presented in Chapter 4. A little focus will be made on transformations and optimizations, while the front-end will not be discussed in this chapter. The Futhark compiler is written in Haskell and is, including our contribution, publicly available under a free software license at:

`https://github.com/diku-dk/futhark/`

Below we give a short description of the compiler pipeline.

6.1.1 Compiler pipeline

The front-end brings the Futhark program to a SOAC IR representation, which resembles A-normal form, i.e., three-address code of statement-like bindings[Henriksen u. a., 2014]. The middle-end takes the SOAC IR representation and performs optimizations and transformation to program, such as fusion[Blelloch u. a., 1993]. For our multicore back-end, the middle-end outputs the program using an internal representation called Multicore Memory (`MCMem`), which has applied all the optimizations and transformations as well as inserted explicit statements for memory allocations. The new representation is similar to the SOAC IR, but introduces four new constructs `SegMap`, `SegRed`, `SegScan` and `SegHist`, which express parallel execution. Finally, we reach the back-end, which generates multi-threaded code for our target language C, using the four new constructs. The compiler outputs the C code within a single file, which is compiled using GCC to either an executable or a library. The run-time system's code is already embedded in the C file; however, can the run-time system be regarded merely as a library, which is linked to C code at compile-time. We start this section with an overview of the optimizations and transformations made by the compiler.

6.1.2 Middle-end

This part of the compiler takes the SOAC IR representation from the Futhark program as input and is responsible for applying transformations and optimizations. Optimizations include code inlining, copy propagation, constant folding, common-subexpression and dead-code elimination, and hoisting of invariant terms outside of loops and SOACs[Henriksen u. a., 2014]. Furthermore can simplification rules be applied, such as producer-consumer fusion. For example can a composition of `reduce`/`scan` and `map` be fused together by the compiler, which we denote as `redomap`/`scanomap`, respectively, and is defined as:

$$\texttt{redomap} \odot f\ ne\ xs \equiv \texttt{reduce} \odot ne\ (\texttt{map}\ f\ xs)$$
$$\texttt{scanomap} \odot f\ ne\ xs \equiv \texttt{scan} \odot ne\ (\texttt{map}\ f\ xs)$$

Fusion can eliminate the need for synchronization points, i.e., joins and reduce the need for intermediate arrays, which can ultimately reduce the need

for memory copies. After performing its transformations and optimizations, it finally ensures that memory is allocated by inserting explicit memory allocations statements into the program. We did not have to deal with this in our implementation, and a lot of the memory management is leveraged from previous work.
After this stage, we use a new internal representation, which is similar to the SOAC IR representation, but with a couple of subtle changes. The new representation introduces four new constructs `SegMap`, `SegRed`, `SegScan` and `SegHist`, which express parallel execution. Internally we call these new constructs for `SegOp`s. These four `SegOp`s corresponds to perfect parallel nests where, semantically, the innermost parallel construct is a `map`, `redomap`, `scanomap` or a `histogram`[1], respectively and the rest are `map`s. These four `SegOp`s suffices to represent all the parallel constructs in Futhark, e.g. a `scatter` is compiled to a `SegMap`. Our program now contains a collection of `SegOp`s, which is compiled to the parallel algorithms presented in Chapter 4. The `SegOp`s constructs provide flexibility for our code-generator. For example can the middle-end output different `SegOp`s, which can be used to generate semantically equivalent code versions, where each can exploit different levels of parallelism. Listing 6.2 shows the information carried by each `SegOp`. For example, a `SegRed` represents a reduction, where `SegBinOp` is the binary operator and `KernelBody` contains the statements of `SegOp` which could be statements for preparing the data to use with the operator or potentially any nested `SegOp`s. Finally is there `SegSpace`, which represents the iteration space for the parallel loops. The new representation of the program is internally called `MCMem`.

```
1  data SegOp lvl lore
2    = SegMap lvl SegSpace [Type] (KernelBody lore)
3    | SegRed lvl SegSpace [SegBinOp lore] [Type] (KernelBody lore)
4    | SegScan lvl SegSpace [SegBinOp lore] [Type] (KernelBody lore)
5    | SegHist lvl SegSpace [HistOp lore] [Type] (KernelBody lore)
```

Figure 6.2: Internal `SegOp` representation

6.1.2.1 Transformations and Optimizations for the multicore back-end

As Futhark is data-parallel array language, many programs can easily saturate CPUs, which has relatively few cores, using only the outer-most parallel level. We use this observation as a heuristic in our compiler to sequentialise nested `SegOp`s. An advantage by letting the compiler sequentialise nested `SegOp`s is that it can perform other optimizations such as reducing explicit memory copies. For example consider two nested `map`s. When the nested `map` is compiled to a nested `SegMap` we require a memory copy to write the result back into the outer `SegMap`'s resulting array. With sequentialisation we can

[1] Also called `reduce_by_index` in the source language

use in-place lowering, meaning that the inner `SegMap`'s now sequential code can write directly to the outer resulting array. This can in memory bounded programs, increase performance up to 2×. As such, we let the compiler sequentialise nested operations, which generally leads to better generated code and performance. However, a downside of only producing a sequential version of our `SegOp` is that if the outer-most parallel level does not exhaust the machine, we cannot exploit potential nested parallelism. The middle-end thus outputs two semantically equivalent `SegOp`s for each parallel computation. A version where all nested `SegOp`s (if any) have been sequentialised as well as a version where the `SegOp`s are left untouched, i.e. contains nested `SegOp`s. The latter is only generated if there are any nested `SegOp`s; otherwise, the versions would be equivalent. To avoid confusion between program structures produced from this part of the compiler, we distinguish between `top-level form` and `nested form`. The `top-level form` is the one where every nested `SegOp` has been sequentialised, while the `nested form` is the program structure containing nested `SegOp`s. By having both versions available, the scheduler is able to select between either at run-time depending on the input sizes, making it possible to exploit potential nested parallelism if needed.

6.1.3 Back-End

Given a program with `MCMem` representation, the back-end first translate each of the `SegOp`s into the parallel algorithms presented in Section 4 using an imperative Intermediate Language (IL). We then translate the `SegOp` from the Intermediate Language to C code. Here we use a technique similar to *lambda lifting*[Johnsson, 1985], which transforms local functions to global functions, by extending the function parameters to include the environment. However, in our implementation, we do not actually extend the function parameters as it would lead to function parameter explosion and instead use closures.

Every `SegOp` generates one C function, `segop_fn`, which has the type shown in Listing 23. The control flow of the parallel algorithm is encapsulated within a `segop` function. For example for a `SegRed`, which is translated to the parallel algorithm for `reduce`, the `segop` function will contain code for execution the first stage in parallel, as well as the sequential code for performing stage 2.

Listing 23 `SegOp` function type

```
1 typedef int (*segop_fn)(void* args, int64_t iterations, int tid, struct scheduling_info info)
```

The `segop` function takes four arguments. First is a pointer to a struct containing the function environment. Besides the environment, the task function also takes the number of iterations, the id of the thread that executed the function, and lastly, the scheduling information. We will return to the latter argument in the next section, where we present the scheduler's API.

Having a distinct `segop` function creates a level of abstraction, in which each function corresponds to a `SegOp`, making it possible to generate a function for the `top-level form` and one for the `nested form`.

For a parallel computation within a `SegOp`, we also lift the code body into a C function, `parloop_fn`, which has the type and template layout shown in Listing 24.

Listing 24 Task function type and template

```
typedef int (*parloop_fn)(void* args, int64_t start, int64_t end, int subtask_id, int tid);

// parloop function template
int parloop_fn(void *args, int64_t start, int64_t end, ...) {
  int err = 0;
  struct parloop_struct *free_vars = (struct parloop_struct *) args;
  // Unpack free variables into local variables
  // prebody
  for (int64_t i = start; i < end; i++) {
    // function body
  }
  // postbody
  return err;
}
```

The `parloop_fn` function takes 5 arguments. Similar to the `segop` function, a pointer containing the environment of the function. It also takes the start and end of the iterations space and a subtask id, which is used to identify which location the specific subtask should write its results to. Lastly, is the id of the thread executing the function, which is used for logging purposes. The template for a `parloop_fn` function is also shown. It first casts its environment into the struct belonging to the function and unpacks the environment into local variables. A `prebody` can optionally be supplied, which usually initializes values before the for-loop is entered. Then the sequential for-loop is entered, which iterates over the range (`start`, `end`) executing the function body. Finally, a `postbody` can optionally be provided to write the results of the for-loop back. The motivation for the `prebody` and `postbody` is that, in some cases, it is beneficial to generate code where the function body within the for-loop only uses variables that are local and write the result back after the loop has finished - We will show concrete examples later of such benefits. The `prebody` and `postbody` are not always used and can be empty, but are included so we can use a single representation of a parallel for-loop internally, which is both simple and flexible. The implementation by lifting both the `segop` and `parloop_fn` allows us to pass the function pointer along with environment to the scheduler. The next section describes the scheduler API and how the scheduler uses these functions.

6.1.3.1 Scheduler API

For brevity, in this section, we omit non-relevant information or fields when showing code or definitions. Examples of such can be debugging or logging information, which is not relevant to the run-time system's functionality. Furthermore, as C only has one namespace, it's common to prefix library functions and struct identifiers with the library's name. For example, in our implementation for the scheduler, we use the prefix `scheduler_`. We will omit such prefixes here for simplicity.

The scheduling of a `segop` is divided into two phases, decision, and execution. We start by describing the decision phase. Whenever a `segop` needs to be executed, a query will first be send to the scheduler through the API function `decide()`. The function decides how the given `segop` should be executed, i.e., sequentially or in parallel and if so, how many subtasks are created. Our compiler generates code for invoking the call to `decide()` and fills out the struct `segop`, such that the scheduler can make the decision (Listing 25). The struct contains the closure of the `segop` functions, as well as information on which scheduling approach should be used for the `segop`, i.e., dynamically or statically, which is decided at compile-time. Lastly, it contains pointers to the fields that contain time measurements associated with the `segop`. These fields will be used to compute estimate C and decide how the particular `segop` should be executed along with the number of iterations using one of the procedures described in Section 5.1. Note that we do not actually store the estimate of C but rather the total time and total iterations from previous runs. This allows us easier accumulate time measurements.

The scheduler fills out the `scheduling_info` struct and invokes either the `top-level` or `nested` `segop` function with it, such that the initialization code within the `segop` function can prepare for parallel execution, e.g., by allocating the memory needed depending on the the number of subtasks created. Whenever a parallel section is reached within the `segop` function, another call to the scheduler is made, supplying the parloop function created by the compiler from Listing 24. This is the execution part of the scheduler and is invoked using the scheduler function `execute_parloop()` (See Listing 26). Our compiler generates the necessary code for invoking `execute_parloop()`, by filling out the struct `parloop`, which contains scheduling information passed down from the decision phase. Similarly to a `segop`, we encapsulate the environment of the `parloop` function through the pointer, `args`. If the number of subtasks was one then `execute_parloop()` executes the parloop function sequentially, otherwise it invokes `execute_parallel()`, which creates the subtasks and distributes the work, and blocks until all the subtasks are finished. We will return to later on how we distribute the work in more detail in Section 6.2. This concludes our presentation of the scheduler's API, which only exposes two functions, decision, and execution. While we could just settle with one API call, which included both decision and execution, our ap-

Listing 25 Scheduler task function and struct definitions

```
enum scheduling {
  DYNAMIC,
  STATIC
};

// Struct definition of segop closure
// and segop information
struct segop {
  void *args;
  segop_fn top_level_fn;
  segop_fn nested_fn;
  int64_t iterations;
  enum scheduling sched;

  // segop time measurements
  int64_t *total_time;
  int64_t *total_iter;
};

int decide(struct segop *);

struct scheduling_info {
  int64_t iter_pr_subtask;
  int64_t remainder;
  int nsubtasks;
  enum scheduling sched;

  int64_t *total_time;
  int64_t *total_iter;
};
```

Listing 26 Executing a parallel loop

```
thread_local int thread_id;

struct parloop {
  parloop_fn fn;
  void* args;
  int64_t iterations;
  struct scheduler_info info;
};
...
int execute_parloop(struct parloop *task)
{
  int err = 0;
  if (task->info.nsubtasks == 1) {
    // Execute sequentially
    err = task->fn(task->args, 0, task->iterations, 0, thread_id);
  } else {
    // Execute in parallel
    err = execute_parallel(task);
  }
  return err;
}
```

proach comes with several advantages. It is possible to pre-query scheduling information before a `SegOp` is run. This could include getting information on how much additional memory is needed for a particular parallel task, which depends on the particular scheduling. For example, if the scheduler decides to run the given task sequentially, we do not need to allocate additional memory, which can be more memory efficient for some `SegOp`s, such as Subhistogramming. It is also possible to explicitly ensure that multi-stage `SegOp`s use the same scheduling. For example, recall that the parallel scan algorithm has two parallel stages and that we required that the scheduling of them must be the same. Since these phases are encapsulated inside the same `segop` function, both phases use the same scheduling decision to ensure such a requirement. Lastly, it is possible to hand-over multiple `segop` functions to the scheduler, which is semantically equivalent, i.e. produces the same result, but exploits different levels of parallelism. In our work, we only provide the `top-level-` and `nested segop` functions, but future work might try to incorporate more versioning of the `SegOp`s, giving the scheduler more options to choose from depending on the input, current state of the run-time system, and so on.

6.1.4 Back-end continued

Having shown the run-time system's API, it should now be easier to understand what is generated by the compiler. For every `SegOp` we generate a `segop` function along with a struct containing the fields of the environment, i.e. its free variables. Additionally, if the `SegOp` contains nested `SegOp`s, we

generate an additional `segop` function, which can exploit the nested parallelism. The scheduler can then choose between these two. As the intersection of both `segop` functions' free variables is usually large, we only generate one struct. Furthermore for each `segop` function one or more `parloop` functions are generated, which themselves create their own struct containing the environment of the function. For example, a non-segmented `SegRed` creates one `parloop` functions, for the first stage. The second stage is sequential, and the corresponding code is generated directly inside the `segop` function. The bodies of the `parloop` functions may themselves create `segop` functions for exploiting nested parallelism. The reader may notice the similarity of the C code from Section 4 to our implementation. A `segop` function corresponds to a parallel algorithm, while the sequential body of the `parfors` are lifted into `parloop` function.

Our approach to multi-versioning has it's downsides, as it can lead to linear code size in the depth of the nested parallelism. However, in practice, are the levels of nested parallelism not that deep and the code size is still manageable.

6.1.4.1 Efficient sequential elisions

Recall that our scheduler can obtain the sequential elision by executing the parloop function sequentially. However, we can do better in some cases. If a `segop` is executed sequentially, we want to use the more practically work-efficient sequential algorithm. In our compiler, we generate code for the parallel algorithms, such that they are overloaded and can be executed as efficiently as their sequential version. For example, for `scan`, when the first stage is executed sequentially, it reduces to a sequential scan. We use the information passed from the scheduling information in our code-generator and insert checks into our generated code to bypass stage 2 and 3, which are not needed in this case. This overloading creates a small dependency between the compiler and scheduler, where a hard-coded check is inserted into the code-generator. For `map` and `reduce` the sequential elision is already as efficient as the sequential version, so we do not need additional checks. Unfortunately, is our sequential elision for `reduce_by_index`, not as efficient in practice as the purely sequential version. If we use Subhistogramming, then we require an explicit memory copy to write the result from the local histogram to the resulting array while the Atomic approach suffers from the overhead of using atomic operations.

6.1.4.2 Determining if a task is irregular

The compiler determines whether a parallel computation could contain any uneven workload by inspecting the statements of the `parloop` function body. The only current source of potential imbalance used is if the function body contains a while-loop. Such statements are usually used when the termination criteria are not known beforehand. Another interesting source of imbalance is

if-else statements. One can imagine a scenario where the two branches' workload is widely different. However, determining if the workload of the branches is different at compile-time is difficult and requires more precise analysis to do properly. We leave such a case and other potential sources for future work to investigate. Finally are there cases where the inner-parallelism's work depends on it's arguments, such as:

```
map (\i -> reduce (+) 0 [1...i]) [1...n]
```

where the work of the nested `reduce` depends on the value of i, which is varying from 1 to n. At the time of writing, we do not currently check for such cases in our compiler either.

6.1.5 Memory optimizations

In multicore systems, memory bandwidth can quickly become a bottleneck. Here we present the memory optimizations we performed to minimize such effects.

6.1.5.1 Memory access

During our work, we found that GCC is much more capable of generating efficient code whenever the sequential for-loop in Listing 24 used variables, which do not perform memory accesses. More specifically, we found that GCC can put these variables in registers, requiring less spilling and re-loads of data from memory. Currently, we only do such optimizations whenever the operator is on primitive values for `reduce`s and `scan`s, but future work should look into generating code, which uses fewer memory accesses in general.

In our implementation, we first load necessary data in the `prebody` and write them to local variables, which is used inside the for-loop. Whenever the loop exits, we write the result back using the `postbody`. An example is shown in Listing 27, which optimizes the sum program from Listing 12. Here we declare a local variable, `local_sum`, to be used within the for-loop, which can be put into a register. We observed that such a simple optimization can provide orders of magnitude better performance on our micro-benchmarks, but minimizing memory access is not the whole story.

6.1.5.2 False sharing

False sharing occurs when two or more cores work on independent data that share the same cache line. When a write is made to the cache-line, the system's cache-coherence protocol will force an update to the other cores using the same cache line. This will cause memory stalls and wasteful usage of memory bandwidth, which will result in performance degradation, especially on systems with non-uniform memory access (NUMA) behavior. The perfor-

Listing 27 Optimization to use local variables for a scalar reduction

```
1  int32_t parallel_sum (int32_t* xs, int32_t* res, int n) {
2    int chunk = n / n_cores;
3    int n_chunks = (n  + chunk - 1) / chunk;
4    // Stage 1
5    // Allocate chunk_sum[n_chunks] = {0};
6    parfor (i = 0; i < n_chunks; i++) {
7      int id = get_subtask_id();
8      // prebody
9      int32_t local_sum = chunk_sum[id];
10     seqfor (int j = chunk * i ; j < min(n, (i + 1) · chunk); j++) {
11       local_sum += xs[j];
12     }
13     // postbody
14     chunk_sum[id] = local_sum;
15   }
16   ... // Stage 2 of reduce
17 }
```

mance speed-up above is also due to the removal of false sharing[2]. Before the optimization in Listing 27, the shared intermediate array `chunk_sum` would cause false sharing, as the cache lines on our system, are 64-bytes, which can store 16 values of 32-bit each. To minimize this effect, we try to avoid letting threads work on shared memory allocations, which is currently only supported for `reduce` and `scan`. When the operator is on primitive values, we use the optimization above, while on vectorized values, we allocate a local array for the accumulator. Our Subhistogramming implementation could also benefit from it, since we currently allocate the sub-histograms as a large contiguous memory block, which is shared among the threads.

6.1.5.3 Memory reuse

This optimization enables reuse of memory allocations by hoisting allocations and avoids putting pressure on the memory allocator. Whenever we encounter an immediate memory allocation inside the sequential for-loop in our parloop function, we move such allocations outside. Listing 28 shows an example of an allocation, which safely can be moved outside to the `prebody`. The `free` is also moved to the `postbody`.

6.1.5.4 Lexical memory usage

Recall that every memory allocation is wrapped inside a `memblock`, which keeps track of references to the memory. Memory blocks which are only used lexically, i.e., do not leave the scope that it was defined in, does not need reference counting and can be handled in a stack-like fashion avoiding the, albeit

[2]We believe the drastic performance increase is mainly due to the elimination of false sharing, though it is difficult to measure the individual effects

Listing 28 Example of memory allocation that can safely be moved to the `prebody`

```
1  int parloop_fn(void *args, int64_t start, int64_t end, ...) {
2      ... // Initialization code
3      for (int64_t i = start; i < n; i++) {
4          type* intermediate_res = malloc()
5          // execute function body, writing result to intermediate_res
6          // write intermediate_res to res
7          free(intermediate_res)
8      }
9      ...
10 }
```

small, overhead of reference counting. For example, all allocations within a subtask, i.e., within a parloop function, do not need reference counting, which follows from our second memory policy.

6.2 Scheduler

The final implementation is the scheduler. Our scheduler launches $n_{vcores} - 1$ POSIX threads, which acts as worker threads. Including the main thread, we have one thread running per (virtual) core. Each thread maintains its own worker struct which contains a work-queue and thread specific variables such as its id. Before the program starts, the worker threads are spawned using a thread function and kept alive during the program's duration. After the threads are spawned, they are immediately put to sleep and only woken up whenever there is work. Whenever the main thread starts execution of the program, and it encounters a parallel computation it will call `decide()`, which decides how it should be scheduled as described in Section 6.1.3.1. Whenever a parallel computation with more than one subtask is decided, `execute_parallel()` is called, which handles subtask creations and distribution. A subtask (Listing 29) contains the closure of the `parloop_fn` similarly to the `parloop` struct from Listing 26. It also contains its iterations range, subtask id, shared information for timings used for measuring the sequential work of the task the subtasks belongs to and auxiliary information used whenever the task was dynamically scheduled. Lastly, a join counter is used to track how many subtasks are yet to be finished. Each subtask maintains a reference to the join counter and atomically decrements it upon completion of a subtask. An outline of the process for subtask creation and distribution is shown in Listing 30. Lines 9 through 12 read the scheduling information from the previous decision phase. Line 13 creates the shared join counter. Lines 17 through 26 creates the subtasks using the scheduling information. Since we are using the job-queue, we are able to distribute the subtasks onto the other workers queue directly, which are then woken up. After creating the subtasks, the main thread will perform it's own subtask first using the

wrapper function `run_subtask()`. We will show the function's content later on when we present time measurements in Section 6.2.2. After executing its subtask, the main thread then loops, waiting for the join counter to become zero. This is outlined in lines 29-36. If the main thread has already finished its work, and the join counter is still not zero, it will try to steal some work in the meantime. Note that upon successful stealing, the stolen subtask might be part of another concurrent task and as such is the stolen subtask executed immediately, i.e. the join counter is not checked between a successful steal and the execution. Note also that both `decide()` and `execute_parallel()` are re-entrant if there are any nested `segop`s and can be called by any thread. In the case of nested `segop`s, the same procedure is performed as above, just where the calling thread becomes responsible for creating subtasks. Furthermore, are subtasks pushed onto the calling thread's queue only. The subtasks are then distributed using work-stealing.

Listing 29 Subtask struct definition

```
1  struct subtask {
2    parloop_fn fn;
3    void* args;
4    int64_t start, end;
5    int id;
6
7    // Shared variables across subtasks
8    volatile int *join_counter; // Counter for ongoing subtasks
9    int64_t *task_timer;
10   int64_t *task_iter;
11
12   // Parameters used when task is dynamically scheduled
13   int chunkable;
14   int64_t chunk_size;
15 };
```

6.2.1 Work-stealing on demand

We observed that programs which have load-balanced work, that enabling work-stealing would decrease the performance of the program. When work-stealing were enabled, threads were busy trying to steal work, and not immediately ready for the next subtask to arrive at their job-queue. As most Futhark programs do not require work-stealing, we disable work-stealing by default and only enable it whenever an irregular task is encountered or a task with nested parallelism is used, i.e., the scheduler uses the `nested form`. In our implementation, we use a simple heuristic for selecting between the `top-level form` and the `nested form`. We only use the `nested form` whenever the task does not have enough iterations to exhaust all threads; otherwise, we use the `top-level form`. As some threads will still be sleeping in such a case, we push a dummy subtask to the sleeping threads' job-queue to wake it up.

Listing 30 Outline of `execute_parallel`

```
// A thread local reference to it's own worker struct
thread_local struct worker* worker;
// global reference to other workers
struct workers* workers;

int execute_parallel(struct parloop *parloop, int64_t *timer)
{
  // Load scheduling information
  struct scheduler_info info = task->info;
  int64_t iter_pr_subtask = info.iter_pr_subtask;
  int64_t remainder = info.remainder;
  int nsubtasks = info.nsubtasks;
  volatile int shared_counter = nsubtasks;
  ...

  // Create subtasks and distribute them
  int subtask_id = 0;
  int64_t start = 0;
  int64_t end = iter_pr_subtask + (int)(remainder != 0);
  for (subtask_id = 0; subtask_id < nsubtasks; subtask_id++) {
    struct subtask *subtask = create_subtask(task->fn, task->args, start, end, &shared_counter, ...);
    push_subtask(workers[subtask_id], subtask);
    // Update range variables
    start = end;
    end += iter_pr_subtask + ((subtask_id + 1) < remainder);
  }

  // Join (wait for subtasks to finish)
  while(shared_counter != 0) {
    if (!subtask_queue_is_empty(worker->q)) {
      struct subtask *subtask = subtask_queue_dequeue(worker);
      if (subtask != NULL) run_subtask(worker, subtask);
    } else {
       // Try to steal some work while waiting for subtasks to finish
    }
  }
  return 0;
}
```

To keep treads awake, we set a global variable `active_work`, which indicates that there is work that can be stolen. While the variable is greater than zero, threads will stay awake and try to steal. Note that our heuristic does not take into account the current state of the system (how many idle threads, etc.) nor how much actual work the nested parallelism has. For example, if there is not much nested parallel work, it can become inefficient to wake up the threads just to handle the small amount. Future work should improve our simple heuristic, by considering the work of the nested parallelism.

6.2.2 Time measurements

The procedure for time measurement is shown in Listing 31, where we for simplicity exclude code, which is not relevant to time measurements. For our implementation of time measurements, we assume that a function `now()` returns the current time. The function should measure time at a couple of magnitudes smaller than that of κ. Usually, such function can be implemented using hardware cycle counters, which are cheap to query e.g., by reading from the time stamp register `rdtsc`. But we found that converting cycles to time was not consistent on our machine as it relies on the core frequency, which can sometimes boost, causing unreliable measurements. Furthermore, `rdtsc` is also not guaranteed to be synchronized across cores, which can cause problems if a thread is migrated to another core during execution. Instead, we use the UNIX function `clock_gettime`, which can measure the current time with nanosecond precision[3]. It is slightly more expensive than using cycle counters but is sufficiently precise for our purpose. For proper time measurement, we use two thread-local variables, `total` and `timer`, which has the following invariants. `timer` either contains the time of the beginning of the innermost subtask started or a point in time posterior after it. `total` holds the time of all the sequential work performed starting from the beginning of the innermost subtask run plus the time stored in the variable `timer`. We also use a thread-local variable `nested` to keep track of if we are within a nested computation. We first explain time measurement in the absence of nested parallelism. In this case `nested` is zero when entering `execute_parloop()`.

If the parloop function is executed sequentially, the time measurement is straight forward. When the task is executed in parallel, the process requires a couple of more steps. `execute_parloop()` will call `execute_parallel()`, which generates the subtasks and distributes them among the cores as described in Section 6.2. For proper time measurements, we only want to measure the time used executing the sequential work of the subtasks and exclude any of the overhead from parallelization. To do so, `execute_parallel()` allocate a `task_timer` which initially is zero. Whenever a thread starts executing a subtask, it will call `run_subtask()`, which executes a subtask and measures how long it took. The function `total_now()` computes the total amount of

[3]The resolution depends on the specific CPU, which sometimes cannot provide nanosecond precision

sequential work since the beginning of the innermost call to `run_subtask`. It's easy to see that when the subtask contains no nested parallelism, `total_now()` computes the duration of the sequential subtask. Upon finishing a subtask, `run_subtask()` will atomically add it's time measurement to the `task_timer`. When all subtasks have finished `task_timer` contains the sum of the sequential work for each subtask, giving us the approximation from Eq. 5.2. Finally as `execute_parallel()` returns the timings associated with the task is updated using the function `report()`. Whenever a irregular computation is encountered, in addition to `task_timer` we also use a `task_iter`, which keeps track of how many iterations the time in `task_timer` is based on. In conjunction with `task_timer`, we can then continuously compute our C estimate used to adjust our chunk sizes. For a correct implementation, both of these need to be updated atomically, which can be done by packing both values into one variable and perform an atomic update using a CAS operation. However, at the time of writing the updates are done as two separate atomic addition operations.

6.2.2.1 Nested parallelism

Whenever there is nested parallelism, we must exclude the time taken inside `execute_parallel()`. Whenever the nested computation is executed sequentially, we require no additional action, and we can just compute the time of the nested computation. But when it's executed in parallel, we need to save the time up until entering `execute_parallel()`; otherwise, the parallelization overhead would be included in the current parallel computation. As we unfold back to the outer parallel level(s) we add the `time_before` to the `total`, giving us the result of all the sequential work only, excluding the time used to create and manage the parallel subtasks.

Listing 31 Time measurement accounting for nested parallelism

```
thread_local time total = 0;
thread_local time timer = 0;
thread_local int nested = 0;

time total_now(time total, time t) {
  return total + (now() - t)
}
// Workers call run_subtask when they are executing subtasks
void run_subtask(subtask) {
  total = 0;
  timer = now();
  nested++;
  run(subtask);
  nested--;
  time elapsed = total_now(total, timer);
  // Report time elapsed to shared timer across subtask
  Atomic {task_timer += elapsed}
}

time execute_parallel(task) {
  time task_timer = 0;
  // Create and distribute subtasks
  // Join (wait for subtasks to finish)
  return task_timer;
}

void execute_parloop(task) {
  ...
  if (task->info.nsubtasks == 1) {
    time t = now()
    // execute task sequentially
    time elapsed = now() - t
    report(task, elapsed)
  } else {
    time t_before = total_now(timer)
    t_task = scheduler_execute_parallel(task)
    report(task, t_task)
    total = t_task
    if (nested) {
      total += t_before
    }
    timer = now()
  }
}
```

6.2.3 Testing

We use Futhark's extensive suite of test programs for testing our implementation, which contains more than 2500 tests across 1400 programs. Our implementation passes all these tests. Furthermore, we also pass on all the

programs in the Futhark benchmark suite, which contains large and complex programs. This is a good indication of the correctness of our implementation.

6.2.4 Summary

In this section, we've presented the implementation of the code-generator and the scheduler. Our implementation creates an abstraction between these components, where the scheduler only exposes an API to the code-generator consisting of two functions. Future tweaks for either can thus be done independently of each other.

Note that our run-time system implementation only uses locks for synchronizing the job-queues, otherwise relying on atomic operations to synchronize reads and writes to shared resources. This is to reduce the number of kernel calls, which in turn reduces the overhead of potential context switches to and from kernel-space. We only require involvement from the kernel when waking up threads and whenever there is contention for a job-queue lock, i.e., when there is work-stealing involved. In the latter case, we've reduced the overhead of stealing by using a half-work, work-stealing algorithm.

Part III

Experimental evaluation

Chapter 7

Benchmarks

For evaluating our work, we conduct a series of benchmarks, which is divided into three sections. First, we use micro-benchmarks to see how our implementations can perform on small, simple programs compared to the sequential back-end. This will give us an early indication of the performance of our implementations. The micro-benchmark for `reduce_by_index` is also used to see how well our heuristic performs in practice for a variety of input sizes.

Second, using more real-world applications, we compare the performance of our implementation against the sequential back-end. These benchmarks include programs from the Futhark benchmark suite[1].

Finally, we benchmark our implementation against programs from three publicly available benchmark suites, namely FinPar, Accelerate, and Rodinia.

While we cannot present all the benchmarks in the suite, we select a few who have different characteristics. We then analyze them to see why we either perform well or why we do not perform well and where we can improve. All of our reported benchmarks from our implementation uses the average of 10 runs and is obtained using the in-built benchmarking tool in Futhark.

Experimental setup

Our test machine has two Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz processors, each with 8 cores and 16 threads. The machine has 132 GB of 1600 MHz DDR3 RAM shared between the processors. Each core has two 32 KB L1 caches for data and instructions respectively, one 256 KB L2 cache and a shared 20MB L3 cache per processor. The operating system is Red Hat Enterprise Linux (RHEL) V. 7.8 running GNU/Linux kernel v3.10.0. To compile our generated C code, we use GCC 4.8.5. Since the machine has two processors, it has Non-Uniform Memory Access (NUMA) behaviour.

[1]The Futhark benchmark suite can found at `https://github.com/diku-dk/futhark-benchmarks`

Tuning process

For finding our parameter κ, we use a one-time per machine automatic tuning process, similarly to [Acar u. a., 2019]. The tuning process uses a small reduction program over an array of random 32-bit integers containing 10^8 elements. We observed that we needed to use so many elements to obtain a repeatable result on the same machine. To pick our values for κ, we initially first run our reduction sequentially without the involvement of the scheduler. This gives us an estimate of C.

Then with the involvement of the scheduler, we execute a series of reductions using only one core, where we create subtasks according to

$$n_{subtasks} = \left\lfloor \frac{n}{n_{chunk}} \right\rfloor \tag{7.1}$$

where $n_{chunk} = \frac{\kappa}{C}$ and n is 10^8. Note that we do not cap the number of subtasks created in this tuning process, as we want to measure the the overhead of subtask creation and management. We start initially with a small value of $\kappa = 1\mu s$. Using only one core, we progressively try to increase κ until we reach the first value of κ upon our reduction runs slightly above 5% slower than the sequential run, which becomes our overhead of parallelisation. Intuitively, the 5% is the overhead we allow from creating and managing subtasks, compared to the optimal time from the sequential execution. On our test machine, our tuning process found that $\kappa = 5.1\mu s$.

Theoretical vs practical speed-ups

While we might expect that using P processors would result in P times speed-up, in reality, there are several reasons that one does not observe such speed-ups. First are parallel algorithms often less practically efficient than their sequential counter-part. An example is the parallel `scan` algorithm. While it is work-efficient, it usually does not show a linear speed-up due to the additional pass. Second, memory bandwidth usually does not scale up to P processors, and performance might be affected as threads compete for the shared memory bandwidth. Finally, there might not be enough work to exhaust all cores; for example, our scheduler might only estimate that there is enough work to use a small portion of the cores, taking into consideration the overhead of parallelization.

7.1 Micro-benchmarks

The following sections presents the evaluation of our multicore back-end using a series of micro-benchmarks for our SOACs for `reduce`, `scan` and `reduce_by_index`. We omit `map` as it's just a parallel-loop and is not an explicit parallel algorithm as the others. This section aims to understand which kind of speed-up we can expect for simple programs compared to sequential C

back-end and give us a good baseline for the speed-up we can expect from larger programs.

7.1.1 Reduce

For the microbenchmarks for `reduce`, we use a summation over an array of randomly generated numbers. For data types, we use 32-bit integers and single-precision floating points as they are the most commonly used types in Futhark.

Type	Size	Sequential	Sequential elision	Multicore
`i32`	10^6	$477\mu s$	$\times 0.95$	$\times 2.2$
	10^7	$4801\mu s$	$\times 0.97$	$\times 3.0$
	10^8	$48548\mu s$	$\times 0.99$	$\times 3.6$
`f32`	10^6	$1189\mu s$	$\times 0.94$	$\times 5.3$
	10^7	$11947\mu s$	$\times 0.96$	$\times 7.7$
	10^8	$119541\mu s$	$\times 0.97$	$\times 8.2$

Table 7.1: Benchmark results of a summation program using an array of 32-bit data types. **Sequential** denotes the sequential back-end, while **Sequential elision** is our implementation when executed sequentially. **Multicore** is our implementation executed in parallel. Speed-ups are with respect to **Sequential**.

The results show that the more elements, the better the speed-up we get. This should not come as a surprise. We also see that our **Sequential elision** is comparable to **Sequential** in terms of execution times.

But we are experiencing relatively low speed-ups, considering that we have 16 cores with 2 threads each available. The programs are heavily memory-bound, causing cores to wait longer for data to arrive than actually performing work. For single precision, the speed-up is better as floating-point operations are more expensive and hence perform slightly more work. Even with hardware optimizations such as pre-fetching of data, integer addition is so fast that it caused around 50% more cycles to be stalled in the pipeline compared to floating-point. We can see that for programs that are memory bound, we should not expect a lot of speed-up compared to the sequential back-end.

7.1.2 Scan

The parallel `scan` is interesting since, while it is work-efficient, it requires an additional pass. We want to make sure that it's beneficial to use such an algorithm in practice. We here use a prefix-sum program over an array of random generated 32-bit integers and single-precision floating points.

Type	Size	Sequential	Sequential elision	Multicore$_{elision}$
i32	10^6	$3237\mu s$	$796\mu s$	$\times 1.8$
	10^7	$32599\mu s$	$24893\mu s$	$\times 5.8$
	10^8	$573026\mu s$	$246405\mu s$	$\times 6.3$
f32	10^6	$4782\mu s$	$1438\mu s$	$\times 3.8$
	10^7	$61233\mu s$	$29463\mu s$	$\times 6.2$
	10^8	$614452\mu s$	$275742\mu s$	$\times 6.2$

Table 7.2: Benchmark results of a prefix-sum program using an array of 32-bit data types. **Sequential** denotes the sequential back-end, while **Sequential elision** is our implementation when executed sequentially. **Multicore** is our implementation executed in parallel. Note that the speed-up is with respect to **Sequential elision**

First, we see that **Sequential elision** is faster than our **Sequential** by a good margin. The reason for that is the inefficient code from the sequential back-end, which performs the `scan` on an intermediate array. It then performs a memory copy to the resulting array, while our implementation performs the `scan` directly on the resulting array. We assume that such inefficiencies will be removed in the future. Hence, we compare the speed-ups against the sequential elision. Again is the program heavily memory bound and we only observe a small speed-up similar to `reduce`. Nevertheless, this shows that the parallel scan algorithm can provide speed-ups despite the additional pass.

7.1.3 Reduce_by_index

Our micro-benchmark for `reduce_by_index` adds an additional dimension. Recall that we have two different implementations of `reduce_by_index`, Subhistogramming, and Atomic updates, where we use a heuristic to select between them at run-time. We want to make sure that our heuristic is relatively robust. We use a histogram computation as it is the most common application of `reduce_by_index` in Futhark. We use an input size of 10^6 and 10^8 of 32-bit integer elements, which are randomly generated. We also vary the destination array size from 10^2 up to 10^8, where the random indexes are mapped into using a modulo operation. The results are shown in Table 7.3. As expected, when **Dest** is small, there are a lot of conflicting writes to the same location, which causes **Atomic** to become slow, significantly slower than the sequential version. **Subhistogramming** performs much better in these cases. It is first when **Dest** is quite large, 10^6, that we see that **Atomic** starts to provide speed-up, while **Subhistogramming** starts to become slower.

As in the case of `scan`, the code from **Sequential** performs a memory copy to write the result back to the resulting array. This causes an increase in run-time for larger destination arrays.

The slow-down for the **Sequential elision** compared to **Sequential** when **Subhistogramming** was chosen is due to poor branch prediction according to

Input	Dest	Sequential	Sequential elision	Subhistogramming	Atomic
10^6	10^2	$3788\mu s$	$7557\mu s$	$\times\mathbf{4.2}$	$\times 0.3$
	10^4	$3606\mu s$	$7250\mu s$	$\times\mathbf{4.5}$	$\times 0.87$
	10^6	$9942\mu s$	$15154\mu s$	$\times 0.5$	$\times\mathbf{3.0}$
	10^8	$517955\mu s$	$146940\mu s$	$\times 0.27$	$\times\mathbf{4.2}$
10^8	10^2	$382472\mu s$	$783646\mu s$	$\times\mathbf{4.8}$	$\times 0.29$
	10^4	$397964\mu s$	$788686\mu s$	$\times\mathbf{6.3}$	$\times 0.95$
	10^6	$718781\mu s$	$817539\mu s$	$\times\mathbf{3.2}$	$\times 2.7$
	10^8	$2810741\mu s$	$5939206\mu s$	$\times 1.17$	$\times\mathbf{3.6}$

Table 7.3: Micro benchmark using a histogram computation over 32-bit integers, where **Input** denotes the number of elements to process and **Dest** denotes the size of the histogram. The approach chosen by our heuristic is in bold and the speed-up shown is with respect to **Sequential**. The times for the **Sequential elision** is the one chosen by our heuristic.

the profiling tool `perf`, even though both use the same data. We have not been able to figure out why they behave so differently. When **Atomic** was chosen, **Sequential elision** is slower due to the overhead of using atomic operations, even though the **Atomic** does not perform a memory copy to return the result. While our heuristic can choose the best performing implementation, in this case, we note that our micro-benchmarks here are far from exhaustive, nor did we investigate adversarial cases. For example when `dest` is large *and* all updates are on the same location. Such a program is going to be slow using either method.

7.1.4 Summary

In this section, we have benchmarked a set of micro-programs. This section showed that the implemented parallel algorithms along with the scheduler is capable of providing speed-up. While the speed-up was not particularly impressive, the micro-benchmarks performs little work and are heavily memory bound. The next section presents much more work heavy programs, using benchmarks from the Futhark-benchmark suite. For these programs, we should expect more respectable speed-ups.

7.2 Established benchmarks

This section benchmarks a series of programs from the Futhark benchmark-suite. It aims to show how the multicore back-end performs on more realistic real-world programs. We generally have three types of programs, which we want to benchmark:

- `Flat regular programs`: These programs are by far the most common in Futhark. Recall that the compiler generates `top-level form`

`SegOp`s, which means that we only have one level of parallelism. Furthermore is the workload regular and these programs are scheduled using a static scheduling approach, where we divide the iterations out evenly out among each subtask.

- `Flat irregular programs`: These programs are similar to programs on `top-level form`, but with the difference that each iteration's workload is no longer predictable. These programs are less common in Futhark but are nevertheless interesting to see how well our online algorithm from Section 5.1.2.2 performs on such programs.

- `Nested regular programs`: The final program type is the nested parallel program types. Such programs should show us how well our scheduler is capable of handling nested parallelism. We remind the reader, that we only select the `nested form` whenever the parallel loop has fewer iterations than threads.

The next sections benchmarks are hand-picked to show a different variety of such programs. We aim to select a few benchmarks with different underlying characteristics and highlight what makes the program perform well or not so well. In the latter case, we discuss the main reasons and what can be improved to make the program perform better in the future. The Futhark benchmarks are ported from three benchmark suits, FinPar, Accelerate, and Rodinia. The Parallel Financial Benchmark (FinPar)[Andreetta u. a., 2016] is a suite containing real-life financial applications, such as Option pricing of European Call-options (OptionPricing) and Local Volatility Calibration (LocVolCalib). Accelerate is a suite containing several implementations in the Haskell-embedded language for data-parallel array programming by the same name. Benchmarks from Accelerate include FFT, Smoothlife, and N-Body. Rodinia is a well-known benchmark containing more than 20 benchmarks such as K-Means and Back-propagation. These benchmark suites have implementations, targetting multicore CPUs. We will compare our implementation against these later in this chapter.

7.2.1 Flat regular programs

The programs presented in this section are the most common types in Futhark, which are scheduled statically. For most of these programs we only exploit top-level parallelism, when the dataset contains enough work to saturate our machine. The results are shown in Table 7.4.

7.2.1.1 FinPar

The FinPar programs enables the compiler to aggressively fuse both `map-map` and `map-reduce` compositions. We use three datasets for each benchmark, small, medium, and large. OptionPricing is a `map-reduce` composition. For OptionPricing, we observe a similar speed-up for all datasets. However, we

Original suite	Program	Dataset	Sequential	Multicore
FinPar	OptionPricing	Small	1556 ms	×17.2
		Medium	1583 ms	×17.3
		Large	12596 ms	×17.3
	LocVolCalib	Small	3723 ms	×17.9
		Medium	7975 ms	×20.7
		Large	181232 ms	×21.2
Accelerate	Smoothlife	128×128	1264 ms	×9.1
		256×256	5912 ms	×12.7
		512×512	26409 ms	×17.5
		1024×1024	120883 ms	×16.4
	N-body	1000	6 ms	×9.2
		10000	632 ms	×15.2
		100000	65011 ms	×16.9
	Tunnel	1000	1527 ms	×17.9
		4000	22143 ms	×19.1
		8000	86553 ms	×18.7
Rodinia	BFS (heuristic)	4096 nodes	1.2ms	×2.0
		512n_high_var	3.8 ms	×4.2
		64k.n_skew	250 ms	×12.6
		1M nodes	346 ms	×11.3
	K-Means	8/204800	3166 ms	×14.5
		5/494019	3771 ms	×11.2
	Backprop	medium	831 ms	×12.1

Table 7.4: Benchmark result for flat regular parallel programs from Futhark Benchmark Suite. Here **Original suite** denotes the suite the program was ported from. All times are shown in ms and speed-up is against **Sequential**

observed that the program had an irregular workload through an if-else statement, causing threads to perform different amounts of work. This could benefit from a dynamic scheduling approach. However, as described in Section 6.1.4.2, determining the workload of the branches in if-else statements is difficult at compile-time. One could let the programmer supply directives to help the compiler in such cases.

LocVolCalib is an outer `map` containing a sequential for-loop, which contains several nested `map`. The nested `map`s are then sequentialised by the compiler in the `top-level form`. However, the small dataset does not provide enough outer-most parallelism to exhaust all cores on our machine when using the `top-level form`. In this case, our scheduler uses the `nested form`. Table 7.5 show show our run-time system leverages the `nested form` to exploit the nested parallelism, gaining a speed-up of ×1.41.

	LocVolCalib		
Dataset	`Top-level form`	`Nested form`	Speed-up
Small	295 ms	208 ms	$\times 1.41$

Table 7.5: Comparison of LocVolCalib performance on small dataset using the `top-level form` vs. the `nested form`

7.2.1.2 Accelerate

The benchmarks from Accelerate shown here are Smoothlife, N-Body, and Tunnel.

Smoothlife is a simulation of Conway's Game of Life[2], which is a cellular automaton over a two-dimensional grid of cells. The program is structured as an outer-loop with inner `maps`. The dataset denotes the height and width of the grid in Table 7.10. As Smoothlife contains small `maps`, we see that we need a large grid size to obtain a great speed-up.
N-Body is a simulation of a dynamic system often used in physics. The program is structured as a sequential outer-loop with inner `maps` as Smoothlife. Here we use a naive direct algorithm, which has the complexity $\mathcal{O}(n^2)$. Here the dataset denotes the number of bodies to simulate. As the actual work in N-Body is not that large, we observe that we need a large dataset to obtain great speed-up as well. In Section 7.2.2, we use the more efficient Barnes-Hut algorithm for computing N-Body, which contains irregular parallelism. Tunnel contains a straight forward `map` with a sequentialised inner `map-reduce`. The datasets denote the size of the outer-map. Here we observe consistent speed-up for all datasets.

7.2.1.3 Rodinia

For Rodinia, we only have three programs which are different from those already shown from Accelerate and FinPar. More precisely, we benchmark BFS, K-Means, and Back-propagation from Rodinia.
The Futhark Benchmark suite has several BFS implementations. Here we choose the fastest sequential (and parallel) version, BFS (heuristic). BFS is one of the few programs in the suite containing parallel computations that cannot saturate our machine, i.e., we do not hit the cap in Eq. 5.5. This was observed for the two smaller datasets. Here our algorithm realizes that there is not enough work and only creates a few subtasks. Table 7.6 shows the difference if we were to naively perform an even partitioning across all cores of the iterations on the smaller datasets, which results in over-parallelization.

[2] `https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life`

	BFS (heuristic)		
Dataset	**Naive**	**Our**	Speed-up
4096 nodes	4.1 ms	0.6 ms	$\times$6.8
512n_high_var	2.3 ms	0.9 ms	$\times$2.6

Table 7.6: Comparison of BFS with a naive even partitioning and our implementation for automatic granularity control

K-Means is an iterative algorithm used in unsupervised learning for clustering related d-dimensional points into K-clusters. K-Means contains a `map` over the number of points, which for each point computes the distance to the k-clusters' centroids. The program also has two `reduce_by_index`s over the number of points for updating the k cluster centroids. One of these is over integer values, and the other is over vectorized values with dimension $d = 35$. The datasets are denoted on the form k/n, where k is the number of clusters, and n is the number of points used. As the number of clusters used here is small, $k = 5$, and $k = 8$, the run-time system chooses to use Subhistogramming. Here we observe respectable speed-ups for both datasets; however, we are limited by our `reduce_by_index` implementation because of false sharing. This is best seen for the 5/494019 dataset, which has a smaller speed-up. Since the number of clusters is small and we allocate the sub-histograms as a shared contiguous array, multiple sub-histograms can fit into the same 64-byte cache line for the integer computation. Future work might improve upon our implementation by using private sub-histograms.

Finally, there is back-propagation, a very popular algorithm used in Deep Learning to train Neural Networks. The result for Backprop shows a decent speed-up compared to the input size, but the parallel program has unfulfilled potential. Programs such as Backprop uses fairly basic matrix operations, which can leverage vector instructions. When we used OpenMP compiler directives such as `vectorize`, we observed almost a doubling in performance. But as a proper implementation of vectorization cannot rely on the use of compiler directives, we did not dig deeper into using such optimizations in this thesis.

7.2.2 Irregular programs

This section presents the few benchmarks from our suite, which have irregular parallelism, namely, N-Body and Mandelbrot. Finally, we also show benchmarks for a ray-tracer, which naturally contains irregular parallelism. The main results are shown in Table 7.7 and 7.9.
As before, N-Body (BH) is an outer sequential-loop with inner `map`s, but here we use the Barnes-Hut algorithm, which is based on the use of octrees and has asymptotic $\mathcal{O}(n \log n)$. The traversal of the tree is implemented using a while-loop, which results in irregular parallelism. Mandelbrot is a simple `map` containing a while-loop, and hence also contains irregular parallelism.

Original suite	Program	Dataset	Sequential	Multicore
Accelerate	N-body(BH)	1000	3 ms	$\times$0.5
		10000	58 ms	$\times$3.2
		100000	808 ms	$\times$7.8
	Mandelbrot	2000	714 ms	$\times$23.1
		4000	2893 ms	$\times$23.3
		8000	11708 ms	$\times$23.8

Table 7.7: Benchmark results for irregular parallel programs from Futhark Benchmark Suite. Here **Original suite** denotes the suite the program was ported from. **Parallel** is the speed-up with respect to **Sequential**

Mandelbrot shows great speed-up due to our online algorithm for irregular computations along with work-stealing, which can load-balance the work. Unfortunately, can our algorithm not perform great on all irregular programs. On N-body (BH), we observe that our algorithm is unable to create a speed-up for the smallest data-set. Table 7.8 shows how the benchmark performs if we were to schedule the programs statically instead. We observe an increase of $\times$1.6 in performance by using our online algorithm for Mandelbrot while observing a slow-down for N-body (BH) compared to a static scheduling approach.

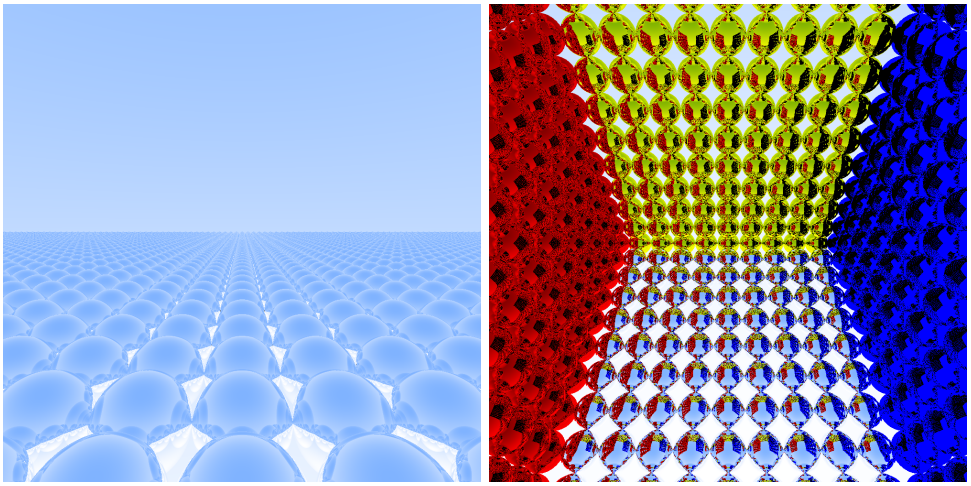
Program	Dataset	Static	Dynamic	Speed-up
Mandelbrot	2000	49 ms	31 ms	$\times$1.6
	4000	201 ms	124 ms	$\times$1.6
	8000	799 ms	492 ms	$\times$1.6
N-Body(BH)	1000	1.5 ms	7.6 ms	$\times$0.2
	10000	11 ms	18 ms	$\times$0.6
	100000	66 ms	104 ms	$\times$0.6

Table 7.8: Comparison of programs containing irregular parallelism using static and dynamic scheduling

The slow-down can be explained. When the estimate of C is greater than κ, our algorithm chooses a granularity of 1 for each subtask, which is not an issue in itself; for Mandelbrot, this is also the case. However, as the constant C for N-Body(BH) is only *slightly* larger than κ, the overhead of subtask creation cancels out its benefits. The small workload per subtask meant that the program spent as much time creating subtasks as actually doing work. Furthermore, the irregular work is small for N-Body(BH), so there is less to gain in trying to load-balance the work, which causes slow-downs.

7.2.2.1 Ray tracers

To demonstrate the scheduler's ability to deal with programs with heavily irregular parallelism we use a ray tracer[3]. Ray tracing is a rendering technique used in 3D computer graphics, usually used for generating 2D images. As the name suggests, every pixel in the image traces a ray to intersect with objects in a scene, which can be done in parallel using a `map`. A ray may recursively be traced by scattering 0 to n times of object surfaces, where n is a threshold to avoid infinite scattering. As such are raytracers potentially highly irregular programs depending on the scene. For our benchmarks, we use two irregular scenes, shown in Figure 7.1. The `irreg` scene is heavily irregular as for the top half of the image, the rays are scattered 0 times, while the bottom half, the rays are scattered multiple times. The `rgbbox` scene is less irregular, but nevertheless, are rays scattered a non-predictable number of times. The results are shown in Table 7.9



(a) Scene `irreg` (b) Scene `rgbbox`

Figure 7.1: The irregular scenes used for benchmark with a resolution of 1000 $\times$ 1000 pixels

For our ray-tracing benchmarks, we see that our algorithm performs quite well. Table 7.9 also compares the performance of the ray-tracer when a statically scheduling approach is used to see the effect of our algorithm and work-stealing. We see that our algorithm provides a performance gain for such heavily irregular programs; with larger performance gains, the more irregular the programs are. The reason for the speed-ups can be explained. The fastest pixels, i.e., the ones who do not recurse, finish first. This gives us an underestimation of C and way below κ in this case. As such, the algorithm will consume large amounts of the "cheap" iterations very quickly in the be-

[3]The program implementation and benchmark can be found at `https://github.com/athas/raytracers`

ginning. But as the more expensive pixels finish, our estimate of C starts to increase, and we begin to consume smaller amounts of iterations. The latter stages of the computation is the part that ensures load-balancing taking only a small amount at the time. Finally, we see a smaller difference between the static and dynamic scheduling approach for `rgbbox` as there are less irregular work and less to gain from work-load balancing.

	`Ray-tracer`		
Scene	**Sequential**	**Static**	**Dynamic**
`rgbbox`	4827 ms	$\times$14.1	$\times$17.7
`irreg`	1759 ms	$\times$8.1	$\times$16.7

Table 7.9: Benchmark results from the two irregular scene used for our ray-tracer. Here **Static** denotes the speed-up to generate the images using a static scheduling approach, while **Dynamic** denotes the use of our algorithm. Both speed-ups is with respect to **Sequential**

7.2.3 Nested parallel programs

This section presents the final type of programs, which can utilize nested parallelism. We only have a couple of programs that we can show how our implementation can exploit nested parallelism. Crystal is a `map` containing an inner `map` over an array. Here the outer-map's iterations are lower than the number of cores, so we need to exploit the nested parallelism from the inner `map` to saturate our machine. Below, the datasets are denoted on the form k/m where k is the number of outer-iterations, and m denotes the inner array's size.

LU decomposition is the decomposition of a matrix to a Lower and Upper triangular matrix. It's interesting as it can contain the possibility of nested parallelism for smaller matrices.

Original suite	Program	Dataset	Sequential	Multicore
Accelerate	Crystal	1/200	10 ms	$\times$12.8
		1/2000	11248 ms	$\times$27.3
		1/4000	44517 ms	$\times$26.7
		6/2000	67338 ms	$\times$22.9
		16/2000	174110 ms	$\times$24.4
		32/2000	340430 ms	$\times$21.1
Rodinia	LUD	64	0.1 ms	$\times$0.4
		256	6 ms	$\times$1.9
		512	44 ms	$\times$3.7
		2048	2760 ms	$\times$11.2

Table 7.10: Benchmark results for programs with nested parallelism from Futhark Benchmark Suite. Here **Original suite** denotes the suite the program was ported from. **Multicore** is the speed-up with respect to **Sequential**

For Crystal, we see that our scheduler can handle nested parallelism quite well. We observe remarkable speed-ups due to our scheduler can use the `nested form` instead and distribute work to the other cores through work-stealing. When $k = 32$ the scheduler uses the `top-level form` instead where the inner `map` is sequentialised. The great speed-ups for Crystal is, its lack access to shared memory. The program only uses private stack-allocated variables for its computation, which fits into the faster caches. The only time the parallel computation accesses shared memory is when it has to write its result back.

The result for LUD with dataset 64 is an example where our heuristic for nested parallelism fails. As the outer-most parallel only contains a couple of iterations, we spend a relatively large amount of time waking up the remaining threads to handle the nested parallelism. However, there is only a small amount of nested parallel work, so we observe a slow-down. It is first when there is enough nested parallelism that our overhead of waking up threads becomes amortized. To fix such cases, future work could take into consideration how much nested work there is only to wake up an appropriate amount of threads to handle the nested parallelism.

7.2.4 Scaling in hardware threads

Finally, we shortly present how our programs scale in the number of hardware threads. Figure 7.2 shows the three different trends for our programs as we adjusted the number of hardware threads used. First is the (almost) linear scaling for Crystal due to its small amount of memory access, which means that the memory bandwidth becomes less of a bottleneck for this program. Then there is LocVolCalib, which initially scales as well as Crystal, but becomes slightly worse for larger hardware threads due to memory bandwidth. This is by far the most common trend for our programs. Finally, there is

BFS (heuristic), which does not have a lot of work. There is a slight increase for a smaller number of hardware threads, but from 16 hardware threads, the speed-up is slightly decreasing, due to over-parallelisation. This is due to how we measure execution time, i.e., include the function call in the time measurements, which can give an overestimate of C. We did this to only sequentialise computations, which are surely greater than κ, but as we increase the number of hardware threads and we initially perform even partitioning across the threads, this overestimation becomes relatively larger. This, in turn, creates the slight over-parallelisation. For programs with smaller workloads, this gives some slow-downs in the number of hardware threads. However, as this trend was only observed on programs with small workloads, the relative impact is small.

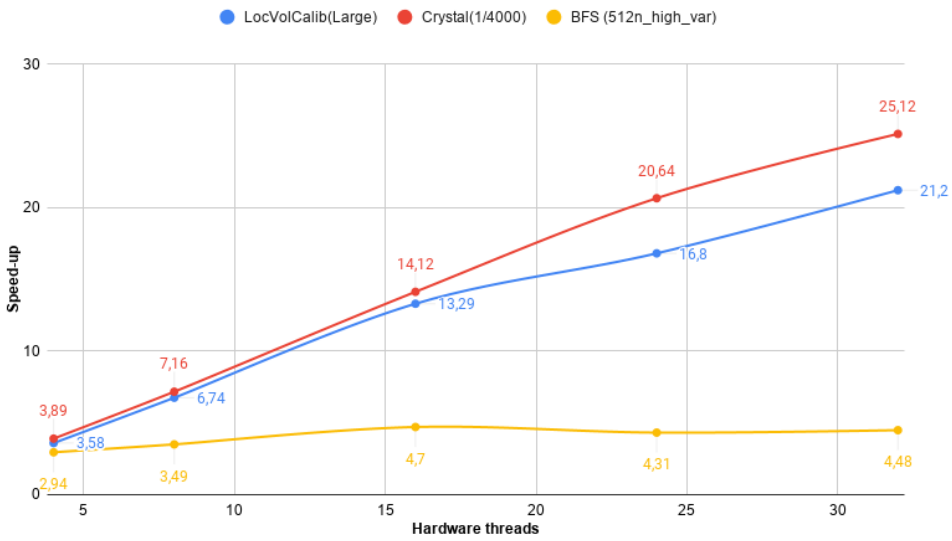


Figure 7.2: Three trends of the scaling of speed-up (compared to sequential back-end) in the number of hardware threads. The number of hardware threads shown range from 4 to 32. The dataset used is denoted in parenthesis.

7.2.5 Summary

This section showed how our implementation performed on a series of benchmarks. We saw that the speed-up is proportional to the amount of work that can be parallelized for regular flat-parallel programs. For example, for LocVolCalib, we saw a speed-up of $\times 20$ compared to the sequential back-end for larger datasets. We also saw that for programs that perform a small amount of access to shared memory can achieve greater speed-ups as the shared memory bandwidth did not become a bottleneck in these programs. Furthermore, when there is enough nested parallelism, our scheduler is able to handle nested programs quite well using a work-stealing approach. While

our implementation can sometimes result in slower times than the sequential back-end, these cases were only observed on smaller inputs, where the absolute times are small. These cases show that our multicore back-end cannot guarantee that all programs will run at least as fast as the sequential back-end.

For irregular parallelism, we saw mixed results depending on the program. For N-body(BH), which had an estimated C close to κ and little irregular work, we saw that our online algorithm was too conservative with its granularities, resulting in large overhead and poor performance. While for programs with much more irregular work, such as Mandelbrot and the Ray-tracer, we saw much better performance. Our results indicate that a one-size-fits-all solution is difficult and requires more work in the future. Perhaps, we need help from the programmer in such cases to set the chunk size manually.

Finally, we touched upon how code-generation can affect the performance of the program. For example, can the use of vectorized instructions, which are natural to exploit in data-parallel languages, be used to achieve even better performance.

7.3 Benchmarks against other implementations

In this section, we present a comparison of our implementation against established benchmark implementations from other suites, namely FinPar, Accelerate, and Rodinia.

7.3.1 FinPar

The benchmarks in the FinPar suite are hand-written C++ programs using OpenMP as the driver for parallelism. We report the average of 10 runs of the FinPar programs. The results are shown in Table 7.11. We see that for LocVolCalib, our implementation can give better performance for the medium and small datasets. Only for the large dataset is our implementation slower, with about 10%. For OptionPricing, our implementation is slower across all datasets, which is due to inefficient code generation, which is not as optimized as the hand-written implementation. Furthermore, the FinPar implementation uses a dynamic scheduling approach, which can load-balance the work to obtain a small speed-up, which we don't do. This advantage is best seen for the smallest dataset, which has relatively more irregular work. If we hard-code our compiler to schedule OptionPricing as dynamic, we unfortunately don't observe a speed-up; in fact, it becomes orders of magnitude slower. This is because the generated OptionPricing code within the parloop function performs many small allocations before entering the sequential for-loop. These allocations become a huge overhead since we generate many small subtasks when using dynamic scheduling. If we want to dynamically schedule OptionPrincing, we would first need to hoist these allocations outside the parloop function first.

Program	Dataset	FinPar	Our	Speed-up
`LocVolCalib`	Small	314 ms	208 ms	$\times$1.5
	Medium	421 ms	385 ms	$\times$1.1
	Large	7686 ms	8549 ms	$\times$0.9
`OptionPricing`	Small	43 ms	91 ms	$\times$0.47
	Medium	60 ms	91 ms	$\times$0.66
	Large	523 ms	724 ms	$\times$0.73

Table 7.11: Comparison of benchmarks from **FinPar** against **Our** implementation

7.3.2 Accelerate

Here we compare benchmark results from Accelerate with our implementation. The Accelerate benchmarks use the criterion library for measuring time. We note that Accelerate uses a much more modern compiler, LLVM 9, than we have available on our test machine, which uses GCC v.4.8.5. Furthermore, Accelerate uses the LLVM equivalent `-ffast-math` flag available on GCC to speed-up computations, which we do not use as it breaks IEEE compliance for floating points, disables error checking for mathematical operations, and much more. Here we benchmark our implementations against Crystal, Tunnel, Mandelbrot, N-Body, and Smoothlife. The results are shown in Table 7.12.

Program	Dataset	Accelerate	Our	Speed-up
`Crystal`	1/2000	378 ms	425 ms	$\times$0.89
	1/4000	1400 ms	1700 ms	$\times$0.82
`Tunnel`	1000	64 ms	95 ms	$\times$0.67
	2000	254 ms	351 ms	$\times$0.72
	4000	998 ms	1218 ms	$\times$0.82
	8000	4037 ms	4523 ms	$\times$0.9
`Mandelbrot`	2000	44 ms	30 ms	$\times$1.4
	4000	168 ms	123 ms	$\times$1.37
	8000	662 ms	491 ms	$\times$1.34
`N-Body`	1000	2.6 ms	0.6 ms	$\times$4.3
	10000	51 ms	39 ms	$\times$1.3
	100000	740 ms	3883 ms	$\times$0.19
`Smoothlife`	128×128	6.8 ms	145 ms	$\times$0.04
	256×256	14 ms	470 ms	$\times$0.03
	512×512	32 ms	1559 ms	$\times$0.02
	1024×1024	122 ms	7718 ms	$\times$0.001

Table 7.12: Comparison of benchmarks from **Accelerate** against **Our** implementation

We believe the slow-down for both Crystal and Tunnel is due to more

efficient code-generation from Accelerate using LLVM. However, we have not been able to confirm this. Our Mandelbrot implementation speed-up is due to our ability to load-balance the work, while Accelerate uses a static approach, with no load-balancing. For N-Body, we see better performance for smaller datasets, but fails to keep up for the largest dataset. Here the performance gain of a modern compiler along with `-ffast-math` are the main reasons. For example, enabling `-ffast-math` cuts the run-time of our implementation down to 1225 ms, which is an 3.5× speed-up.

For Smoothlife, Accelerate heavily outperforms our implementation. The reason is that Smoothlife is mainly an Fast Fourier Transformation (FFT) computation, where Accelerate delegates the computation to the heavily optimized FFTW library, while we use our own FFT implementation. Here our implementation simply gets outmatched.

7.3.3 Rodinia

Similarly to FinPar are the publicly available Rodinia benchmarks for targetting multicore CPUs written in C++ and uses OpenMP to obtain it's parallelism. The Rodinia programs where difficult to benchmark as they are structured weirdly, making it hard to measure the performance in the same way as Futhark, which performs ten consecutive runs. We only provide times from one run here for Rodinia. As such, only when there is a large difference can the results be used to indicate which implementation performs better. The results are shown in Table 7.13.

Program	**Dataset**	**Rodinia**	**Our**	**Speed-up**
`LUD`	64	0.4 ms	0.4 ms	×1.0
	256	1.7 ms	4 ms	×0.43
	512	3.7 ms	15 ms	×0.25
	2048	60 ms	285 ms	×0.21
`K-Means`	8/204800	440 ms	230 ms	×1.92
	5/494019	1607 ms	407 ms	×3.95
`Backprop`	medium	80 ms	66 ms	×1.21

Table 7.13: Comparison of benchmarks from **Rodinia** against **Our** implementation.

For larger matrices, Rodinia's LUD is much faster for several reasons. It uses a tiled matrix multiplication approach along with vectorized operations to gain more speed-up, while we use neither of those currently. Furthermore, our compiler has no cache awareness, which makes our naive matrix multiplication implementation slow.
For K-Means, the histogram computation is sequential in the Rodinia implementation. This causes our implementation to be faster, with a better speed-up for the larger dataset, as there is more parallel work.

For Backprop, our implementation is faster, but the difference is not significant enough to conclude that our implementation is better. Neither implementation uses vectorized operations to speed-up computations. As discussed in the previous section, can such programs leverage vector instructions.

7.3.4 Summary

In this section, we benchmark our work against a variety of implementations from other benchmark suites. We show that we can match or show better performance for some of the benchmarks without the need for manual granularity control. For example, for Mandelbrot, we saw that our approach to irregular parallelism can obtain better performance than the static scheduling approach in Accelerate. For OptionPricing, it's the other way around, where we cannot effectively load-balance the work due to poor code-generation. Our LUD implementation also generated much less efficient code than the hand-written implementations. In summary, if we are to consistently match other implementations' performance, future work should focus on generating more efficient C code.

Part IV

Final remarks

Chapter 8

Future work and conclusion

8.1 Limitations and future work

This thesis's focus was to improve common code patterns in Futhark and improve such programs' performance, mainly through the run-time system. We did not emphasize our work on optimizing the generated C code and used much of the code already in place from the sequential back-end. This makes for a more distinct evaluation of the new run-time system and shows its efficiency compared to the sequential back-end. The performance gains of the benchmarks compared to the sequential back-end are promising and show that the run-time system can efficiently schedule the parallel work in most cases. However, if we are to compete with established benchmark implementations, future work should put a focus on generating more efficient code with a focus on memory-related issues, such as having better cache-awareness and hoisting more memory allocations. Another key optimization is to enable vectorization through Advanced Vector Extensions (AVX), present on almost all modern CPUs. Such optimization is natural to use in a data-parallel language like Futhark. Finally, we did not thoroughly investigate all performance issues with our implementation of `reduce_by_index` as it requires considerable effort to optimize properly, which can be a project in itself. However, optimizing the implementation to avoid false sharing is a good starting point.

For our run-time system, we saw, for large amounts of work, it is capable of handling regular and nested parallelism by switching between work-sharing and work-stealing. However, we also saw that when the nested parallelism only contains a small amount of work, the overhead of waking up threads can be overwhelming since our implementation does not take into account how much nested work there is. Future work should involve investigating how these cases can be handled better.

For irregular parallelism, we saw mixed results, which would need to addresses in future work. Our approach of using a dynamic chunk size, which amortizes the cost of subtask creation showed varying results depending on the task. Programs which had C estimates close to κ, ended up spending

as much time creating subtasks as working. Additionally, for such estimates, if the workload was not heavily irregular, the benefits or load-balancing becomes less, which resulted in some slow-downs. However, for heavily irregular programs, we saw that our algorithm could provide significant speed-up compared to a static scheduling approach. Future work might include approaches to distinguish between slightly and heavily irregular program, and only use our algorithm for the later programs.

Our scheduler assumes that parallel computations can be described using a linear cost function. While we did not observe a benchmark where this became a problem, future programs might suffer from our assumption. Future work might look into using static analysis to determine the cost function e.g., with methods from [Jost u. a., 2010].

Finally, we only presented an analysis of a subset of the benchmarks in our suite. While our selected benchmark covers a wide variety of programs, we did not have time to analyse all of the benchmarks. Future work should include analysing more benchmarks, including programs that are not in the suite. This will give us more empirical data to provide us with ideas on where to focus the future work and address potential unknown performance issues.

8.2 Conclusion

This thesis presented a design and implementation for a new back-end for the data-parallel language Futhark, targeting multicore CPUs. Our work included extending the Futhark compiler with a new code-generator for generating parallel C code as well as a new run-time system. We show how our implementation provides a level of abstraction between the compiler and the run-time system, where the latter is merely a library and only exposes an API to the compiler. Such an abstraction was the primary design goal for this thesis so that changes can be performed to either in the future, without the need to change the other.

We show how implicit parallelism can provide granularity control without the need for manual tuning from the programmer. Our implementation is based on an oracle-guided scheduling approach, which uses online algorithms to infer the actual work of computations. By considering the cost of parallelization, our run-time system is then able only to parallelize computations whose work can amortize the cost. We show how our algorithms can be used to perform automatic granularity control for regular (nested) parallelism and irregular parallelism using parallel for-loops.

We evaluate our work using a series of programs from the Futhark benchmark suite. Our empirical results show that our new back-end is capable of providing significant speed-up for a large number of Futhark programs compared to the sequential back-end, where we, on average, can achieve 15.6× speed-up on the largest datasets. However, we also saw that more work needs to be done for some. For smaller datasets, we can sometimes observe a slow-down compared to the sequential back-end. Furthermore, our online

algorithm for irregular workload does not work well when the estimated C is approximately the same as the parallelization overhead, which resulted in poor performance.

Finally, we benchmarked our back-end against hand-optimized implementations. Here we saw that our implementation, in some cases, can provide a similar or better performance without the need for manual tuning. However, there are also cases where our implementation come up short, mainly due to inefficient code generation, compared to hand-optimized benchmark programs. When such inefficiencies have been reduced, we will be able to compete more consistently with such benchmarks and provide an alternative for writing implicit parallel programs without the need for manual granularity control.

While this thesis took a significant step towards a new multicore back-end for Futhark, it still also shows that more work can be done to obtain better performance. Future work will undoubtedly improve upon the work presented in this thesis and bring improvements to both the compiler and run-time system to achieve better performance, which is the ultimate goal of Futhark.

Bibliography

[Acar u. a. 2018] Acar, Umut ; Charguéraud, Arthur ; Guatto, Adrien ; Rainey, Mike ; Sieczkowski, Filip: Heartbeat scheduling: provable efficiency for nested parallelism, 06 2018, S. 769–782

[Acar u. a. 2019] Acar, Umut A. ; Aksenov, Vitaly ; Charguéraud, Arthur ; Rainey, Mike: Provably and Practically Efficient Granularity Control. In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA : Association for Computing Machinery, 2019 (PPoPP '19), S. 214–228. – URL https://doi.org/10.1145/3293883.3295725. – ISBN 9781450362252

[Andreetta u. a. 2016] Andreetta, Christian ; Bégot, Vivien ; Berthold, Jost ; Elsman, Martin ; Henglein, Fritz ; Henriksen, Troels ; Nordfang, Maj-Britt ; Oancea, Cosmin E.: FinPar: A Parallel Financial Benchmark. In: *ACM Trans. Archit. Code Optim.* 13 (2016), Juni, Nr. 2. – URL https://doi.org/10.1145/2898354. – ISSN 1544-3566

[Bird 1989] Bird, R. S.: Algebraic identities for program calculation. 32 (1989), April, Nr. 2, S. 122–126. – URL http://www3.oup.co.uk/computer_journal/hdb/Volume_32/Issue_02/tiff/122.tif;http://www3.oup.co.uk/computer_journal/hdb/Volume_32/Issue_02/tiff/123.tif;http://www3.oup.co.uk/computer_journal/hdb/Volume_32/Issue_02/tiff/124.tif;http://www3.oup.co.uk/computer_journal/hdb/Volume_32/Issue_02/tiff/125.tif;http://www3.oup.co.uk/computer_journal/hdb/Volume_32/Issue_02/tiff/126.tif. – ISSN 0010-4620 (print), 1460-2067 (electronic)

[Blelloch u. a. 1993] Blelloch, Guy E. ; Hardwick, Jonathan C. ; Chatterjee, Siddhartha ; Sipelstein, Jay ; Zagha, Marco: Implementation of a Portable Nested Data-Parallel Language. In: *SIGPLAN Not.* 28 (1993), Juli, Nr. 7, S. 102–111. – URL https://doi.org/10.1145/173284.155343. – ISSN 0362-1340

[Blumofe u. a. 1995] Blumofe, Robert D. ; Joerg, Christopher F. ; Kuszmaul, Bradley C. ; Leiserson, Charles E. ; Randall, Keith H. ; Zhou, Yuli: Cilk: An Efficient Multithreaded Runtime System. In:

SIGPLAN Not. 30 (1995), August, Nr. 8, S. 207–216. – URL https://doi.org/10.1145/209937.209958. – ISSN 0362-1340

[Bull u. a. 2012] Bull, J. M. ; Reid, Fiona ; McDonnell, Nicola: A Microbenchmark Suite for OpenMP Tasks. In: Chapman, Barbara M. (Hrsg.) ; Massaioli, Federico (Hrsg.) ; Müller, Matthias S. (Hrsg.) ; Rorro, Marco (Hrsg.): *OpenMP in a Heterogeneous World*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012, S. 271–274. – ISBN 978-3-642-30961-8

[Chase und Lev 2005] Chase, David ; Lev, Yossi: Dynamic circular work-stealing deque, 01 2005, S. 21–28

[Che u. a. 2009] Che, Shuai ; Boyer, Michael ; Meng, Jiayuan ; Tarjan, David ; Sheaffer, Jeremy W. ; Lee, Sang-Ha ; Skadron, Kevin: Rodinia: A Benchmark Suite for Heterogeneous Computing. In: *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*. USA : IEEE Computer Society, 2009 (IISWC '09), S. 44–54. – URL https://doi.org/10.1109/IISWC.2009.5306797. – ISBN 9781424451562

[Conway 1963] Conway, Melvin E.: A Multiprocessor System Design. In: *Proceedings of the November 12-14, 1963, Fall Joint Computer Conference*. New York, NY, USA : Association for Computing Machinery, 1963 (AFIPS '63 (Fall)), S. 139–146. – URL https://doi.org/10.1145/1463822.1463838. – ISBN 9781450378833

[Halstead 1985] Halstead, Robert H.: MULTILISP: A Language for Concurrent Symbolic Computation. In: *ACM Trans. Program. Lang. Syst.* 7 (1985), Oktober, Nr. 4, S. 501–538. – URL https://doi.org/10.1145/4472.4478. – ISSN 0164-0925

[Hellfritzsch 2018] Hellfritzsch, Sune: Efficient Histogram Computation on GPGPUs, URL https://futhark-lang.org/student-projects/hellfritzsch-msc-thesis.pdf, 2018

[Henriksen u. a. 2014] Henriksen, Troels ; Elsman, Martin ; Oancea, Cosmin: Size Slicing: a hybrid approach to size inference in Futhark, 09 2014. – ISBN 978-1-4503-3040-4

[Henriksen u. a. 2016] Henriksen, Troels ; Larsen, Ken F. ; Oancea, Cosmin E.: Design and GPGPU Performance of Futhark's Redomap Construct. In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. New York, NY, USA : ACM, 2016 (ARRAY 2016), S. 17–24. – URL http://doi.acm.org/10.1145/2935323.2935326. – ISBN 978-1-4503-4384-8

[Henriksen u. a. 2017] Henriksen, Troels ; Serup, Niels G. W. ; Elsman, Martin ; Henglein, Fritz ; Oancea, Cosmin E.: Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates. In: *SIGPLAN Not.* 52 (2017), Juni, Nr. 6, S. 556–571. – URL http://doi.acm.org/10.1145/3140587.3062354. – ISSN 0362-1340

[Henriksen u. a. 2019] Henriksen, Troels ; Thorøe, Frederik ; Elsman, Martin ; Oancea, Cosmin: Incremental Flattening for Nested Data Parallelism. In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA : Association for Computing Machinery, 2019 (PPoPP '19), S. 53–67. – URL https://doi.org/10.1145/3293883.3295707. – ISBN 9781450362252

[Intel 2011] Intel: Intel Threading Building Blocks, https://software.intel.com/content/www/us/en/develop/tools/threading-building-blocks.html. (2011). – URL https://github.com/oneapi-src/oneTBB

[Iwasaki und Taura 2016] Iwasaki, S. ; Taura, K.: A static cut-off for task parallel programs. In: *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2016, S. 139–150

[Johnsson 1985] Johnsson, Thomas: Lambda Lifting: Transforming Programs to Recursive Equations. In: *Proc. of a Conference on Functional Programming Languages and Computer Architecture*. Berlin, Heidelberg : Springer-Verlag, 1985, S. 190–203. – ISBN 3387159754

[Jost u. a. 2010] Jost, Steffen ; Hammond, Kevin ; Loidl, Hans-Wolfgang ; Hofmann, Martin: Static Determination of Quantitative Resource Usage for Higher-Order Programs. In: *SIGPLAN Not.* 45 (2010), Januar, Nr. 1, S. 223–236. – URL https://doi.org/10.1145/1707801.1706327. – ISSN 0362-1340

[Larsen und Henriksen 2017] Larsen, Rasmus W. ; Henriksen, Troels: Strategies for Regular Segmented Reductions on GPU. In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*. New York, NY, USA : ACM, 2017 (FHPC 2017), S. 42–52. – URL http://doi.acm.org/10.1145/3122948.3122952. – ISBN 978-1-4503-5181-2

[Lê u. a. 2013] Lê, Nhat M. ; Pop, Antoniu ; Cohen, Albert ; Zappa Nardelli, Francesco: Correct and Efficient Work-Stealing for Weak Memory Models. In: *PPoPP '13 - Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. Shenzhen, China, Februar 2013, S. 69–80. – URL https://hal.inria.fr/hal-00802885

[Lea 2000] Lea, Doug: A Java Fork/Join Framework. In: *Proceedings of the ACM 2000 Conference on Java Grande*. New York, NY, USA : Association for Computing Machinery, 2000 (JAVA '00), S. 36–43. – URL https://doi.org/10.1145/337449.337465. – ISBN 1581132883

[Neumann 1945] Neumann, John v.: First Draft of a Report on the EDVAC. 1945. – Forschungsbericht

[OpenMP 2008] OpenMP: Architecture Review Board Application Program Interface. https://www.openmp.org/. (2008)

[Thoman u. a. 2013] Thoman, Peter ; Jordan, Herbert ; Fahringer, Thomas: Adaptive Granularity Control in Task Parallel Programs Using Multiversioning. In: Wolf, Felix (Hrsg.) ; Mohr, Bernd (Hrsg.) ; Mey, Dieter an (Hrsg.): *Euro-Par 2013 Parallel Processing*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013, S. 164–177. – ISBN 978-3-642-40047-6