Konrad Ludwig Moritz Rudolph

SEQ**AN**×N— Generic Parallelisation of a Sequence Analysis Library

Master Thesis

Generic Parallelisation of a Sequence Analysis Library

Konrad Ludwig Moritz Rudolph

15th March 2011

Supervisor:
Prof. Dr. Knut Reinert

Second advisor:
Dr. Tim Conrad

We have implemented a generic framework that allows running existing sequential algorithms in parallel across large data sets. We have demonstrated the versatility of this framework by parallelising the SeqAn finders that perform an efficient pattern matching, and other sequence analysis algorithms.

In dieser Arbeit wurde ein Framework entwickelt, welches es erlaubt, einen bestehenden sequenziellen Algorithmus parallel auf großen Datenmengen laufenzulassen. Die Anwendbarkeit wurde durch die erfolgreiche Parallelisierung der SeqAn-Finder vorgeführt, welche eine effiziente Mustersuche auf Zeichenketten implementiert. Auch weitere Algorithmen zur Sequenzanalyse wurden erfolgreich parallelisiert.

Au cours de ce travail on a développé un framework permettant d'exécuter des algorithmes séquentiels existants en parallèle sur de grandes quantités de données. On a montré son utilité en mettant en parallèle les algorithmes SeqAn « finder » qui permettent une recherche de correspondance efficace des patterns de texte dans une chaîne de charactères. On a de plus mis en parallèle d'autres algorithmes d'analyse de séquences biologiques.

Contents

Introduction

> Moore's Law is dead.*
>
> *(Gordon Moore)*

Moore's Law has, for nigh half a century, reliably predicted the growth in efficiency of processors: Moore's Law states that the number of transistors that can be placed on a given surface area doubles every two years [Intel Corporation, 2005]. As a consequence, the number of transistors – and consequently, the computing power – of processors has grown exponentially until recently. However, this growth can no longer be sustained due to a combination of several factors. The most important cause are quantum mechanical effects which raise the electrical resistance of the transistors and thus cause heat dissipation problems which result in energy loss [Feynman, 1985; Tanenbaum, 1990].

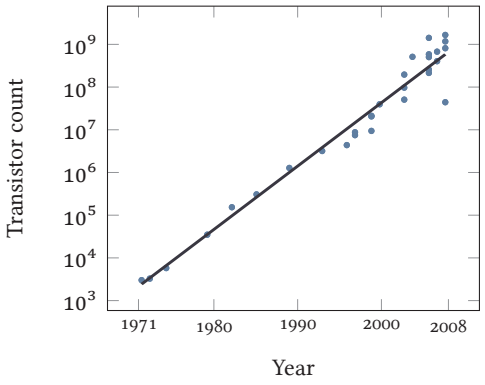

Figure 1: Moore's Law illustrated by the number of transistors of typical processors for each era. Note that the y axis is logarithmic.†

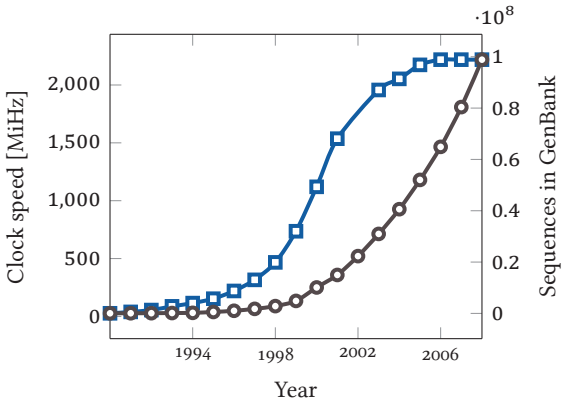

Figure 2: Comparison of clock speed and growth of bioinformatical data. Shown are the growth of the mean clock speed of Intel processors (—□—) and the number of sequenced genomes (—○—).‡

On the other hand, we're dealing with ever increasing amounts of data that our programs have to process. Figure 2 illustrates this using the example of the number of se-

*Dubash [2005]

†en.wikipedia.org/wiki/Moore's_law, retrieved on 2011–01–30

‡www.intel.com, retrieved on 2011–01–29; www.ncbi.nlm.nih.gov/genbank/genbankstats.html, retrieved on 2011–01–29

quenced base pairs of DNA: Thanks to steady improvementes of machines and chemistry, culminating in the advent of so-called *next generation sequencing* techniques, we are observing an exponential growth in the available sequence data that has to be processed. Just a decade ago, the first complete sequence of the human genome was published [Venter *& al.*, 2001]. Not even ten years later, this number has increased by three orders of magnitude with the 1000 Genomes Project [1000 Genomes Project Consortium, 2010].

So our programs don't actually get *faster* on newer machines, they merely don't get slower: In bioinformatics we are truly in the Red Queen's country, and "it takes all the running [we] can do, to keep in the same place".

This means that while the amount of raw data continues to grow exponentially, and processing logic continues to grow more complex, computers do no longer get faster – not only is performance gain no longer exponential, it has in fact all but stopped: "The free lunch is over", in the words of Herb Sutter. This refers to the fact that until now software developers had a "free lunch": existing software would run faster on next-generation machines, simply due to the fact that these machines were faster than the previous generation. No modifcation of the software was necessary. But approximately in 2003, this development has come to a halt (figure 2): Processors bought in 2010 have (approximately) the same clock speed as processors bought in 2004.

But as we can see in figure 1, the increase in the transistor count in computers has not (yet) slowed significantly; it just no longer goes into the making of a single processing unit (CPU). Instead, the number of processing units (CPU) per computer has recently started to increase from just one to several, and, in the near future, many.

i) Parallel programs

Unfortunately, conventional applications cannot benefit from this increase in the number of CPUs, in the same way that a car wouldn't drive faster if we built in a second engine. In order to get a performance increase from a second CPU a program has to be designed such that it can distribute its work across more than one processing unit.

Ideally, a program would spread its work evenly across *all* existing processing units, thus reducing its overall running time proportionally to the number of available CPUs. If we can achieve that, we have re-enabled the free lunch: our software will, once again, run faster on the coming generations of hardware that have more CPUs than the current generation. Unfortunately, spreading a program's work evenly across multiple CPUs is not a trivial task, and even in situations where it is easy to do, it is still a lot of work to re-write the many existing implementations.

When performance plays a role, the next big task is therefore to harness the computational power of these multiple CPUs.

ii) Outline

In the first part of this thesis, we will look at the technical background of parallelisation and which problems arise from it. We will see how a parallel program must be designed and how this process may be structured so that it can be supported by a parallelisation framework. Special focus will be on problems that are, due to their structure, "easy" to parallelise, exhibiting so-called *embarrassingly parallel* structures.

The second part will focus on the design of the framework and which goals we wanted to fulfil. We will define a public interface for the framework and we will discuss some of the issues of its implementation.

Finally, in the third part, we will analyse the performance of the framework using several algorithms that we have parallelised in the course of this thesis.

Part I

Background And Related Work

What is parallel programming? Why do we need it? How can algorithms be implemented in parallel?

I Parallel Programming

The following exposition is based in parts on the brilliant introduction by Barney [2010]. Other sources are Foster [1995]; Kumar *& al.* [1994]; Sanders [2010] and Pankratius *& al.* [2009].

To understand parallel programming, it is useful to first have a look at the alternative – namely, sequential programming. Here, a *program* is a sequence of commands that are executed by a processor*. Any given command potentially relies on the result of a previous command (for example a user input, or simply subsequent steps in a calculation such as $x = c \times (a + b)$). This mandates that the sequence of commands is executed *sequentially*, and every command waits for its predecessor to conclude.

Formally, let a program $C = c_0 \dots c_{n-1}$ be a sequence of executed commands. This bounds the running time of C in $\Theta(n)$ (note that n is not the number of *statements* of a program, since each statement can be executed multiple times, as in a loop, or not at all, as in a non-executed branch of a conditional statement). Let us consider a simple program where each command depends on its predecessor.

Precondition: $A = a_0 \dots a_{n-1}$ is an array of length $n = |A|$ consisting of numbers

```
1 for i ∈ [1 ... n[ do
2     if a_{i-1} > a_i then
3         a_i ← a_{i-1} × 2
```

Algorithm I.1: A simple sequential program

Clearly, in order to compute the element located at position i we must first know the value of the element at position $i - 1$ in the array. This imposes a strict ordering on the

*this already uses a marked simplification by excluding strict evaluation of functional programs and interpreted expression trees

order of execution of the statements.

However, it is immediately obvious that not all commands really depend on their predecessor and thus don't logically need to await their conclusion before being executed. As a simple example, consider a program that transforms every character in a string into its upper-case variant[†]:

Precondition: S is a string

```
1 for s ∈ S do
2     s ← ToUpper(s)
```

Algorithm I.2: Transform a string to upper-case

For this algorithm, the sequence of executed instructions loosely corresponds with the characters of the string and in fact its running time is bounded by $\Theta(|S|)$. But it's obvious that the instructions are not dependent on each other: the character at position i doesn't have to wait until the character at position $i - 1$ has been transformed. In fact, since there are *no* dependencies between the statements whatsoever, this is a perfect example of an *embarrassingly parallel* program.

Conventionally, a program is executed one statement at a time. The *running time* of a program is therefore directly proportional to the number of commands that were executed: $T = \Theta(|C|)$ (if we operate under the usual assumption that the execution time of individual commands is bounded in $\mathcal{O}(1)$).

However, it is sometimes desirable to break up this sequential flow of control.

1.1 Different types of parallelism

1.1.1 Aims of non-sequential execution

There are several fundamentally different aims to parallelisation [Sutter, 2007]:

- To augment the responsiveness of an interactive program (e. g. when reading data from a network socket, or when accepting user input, all while executing computations in the background);
- to implement distributed systems, for example banking applications which need to show a consistent state of the system at all time and thus require synchronisation;
- and to increase the throughput of a computationally heavy algorithm.

[†] ignoring peculiarities like the fact that capital-"ß" is usually written with two letters

We are only concerned with one of these aims, namely with improving the running time and thus the throughput of a program by executing independent statements in parallel. More specifically, we aim to make the program use all available processing resources and re-enable the "free lunch".

Different architectures are amenable to different solutions for parallel programming. In order to define our target architecture, we will first give a brief overview over different architectures and accentuate the differences between them.

1.1.2 Flynn's taxonomy

As we have seen, a program execution consists of an ordered list of *instructions*, operating on *input data*, which is thus transformed into *output data*. One commonly-used way of classifying different kinds of parallel programming is as follows.

We distinguish between programs that execute a single or multiple instructions at the same time, and, on another dimension, between programs that process a single point of data or multiple points of data simultaneously.

These different kinds of programs are summarised in figure I.1. The four different paradigms shown there are implemented on different architectures (except for MISD which arguably has no relevant real-life equivalent). We will therefore have a look at different architectures and match them to the kind of parallelism that they support.

	Single instruction	Multiple instructions
Single data	SISD	MISD
Multiple data	SIMD	MIMD

Figure I.1: Flynn's taxonomy

1.2 Architectures

1.2.1 Uniprocessor

This is the simplest architecture and corresponds closely to the classical von Neumann architecture. There is a single processor that can execute one instruction at a time on a single point of data – it therefore supports the SISD model and doesn't allow truly parallel execution. In particular, the von Neumann model of computation also assumes that there is a *single* channel from and to the memory for both the control unit and the arithmetic logic unit (ALU), and while the von Neumann model is superseded in practice by slightly different architectures, the single data channel (the "*memory bus*") has prevailed.

This architecture is important here primarily because it defines our *classical* paradigm. The uniprocessor is the initial point from which our efforts for parallel programming start off: Conventional programming implicitly *assumes* a von Neumann architecture and we need to be careful when breaking this assumption in parallel programming.

1.2.2 Shared memory, PRAM

The parallel random access memory model (PRAM) is a generalisation of the uniprocessor across multiple processors that access a common memory module (*shared memory*). Historically, this has been a *symmetric* multiprocessing (SMP) architecture where all processors access the shared memory via a common bus. In this schema, all processors can access all the memory equally fast and can thus arbitrarily move memory between processors.

But even before multiprocessing the bandwidth of the memory bus was a bottleneck in computations since CPUs could process memory much faster than they could read it. This bottleneck is aggravated in the SMP architecture since now many processors need to share the same bus and while the computation power has increased P-fold, the memory bandwidth has stayed the same.

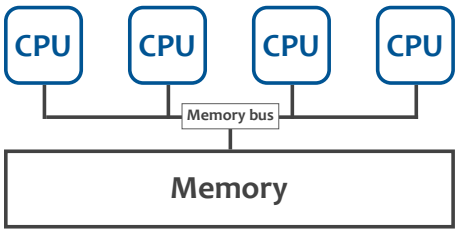

Figure I.2: SMP architecture: Four processors are connected to a common memory module.

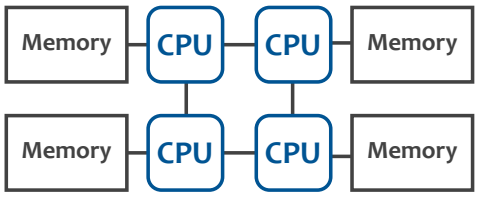

Figure I.3: NUMA architecture: Four processors, each with their own memory module and interconnections.

That's why there has been a shift away from SMP to NUMA[‡], which allows *non-uniform memory access* by giving each processor a dedicated access to its local memory. This makes access to local memory much faster but can in turn lead to slowdowns if a processor needs to access nonlocal memory of another processor. Furthermore, this leads to a complication due to the use of *caches*:

Since access to the main memory is so slow, modern processors use a small additional memory that is built directly onto the chip. Its purpose is to hide the main memory latency by storing recently accessed values directly on chip, making access to nearby addresses much faster. For single processors, it ends here.

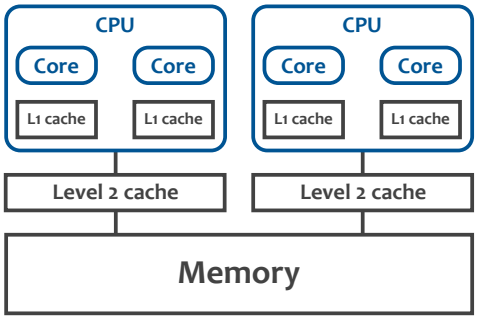

Figure I.4: Level 1 and level 2 caches. A possible configuration of two processors with two cores each, where cores have their own level 1 cache but share a common level 2 cache.

However, with multiple processors there's the problem that each processor has its own non-shared cache and when it writes a value into the cache, this value isn't directly committed to main memory. That way, even when memory accesses across different processors are properly serialised (i. e. there is only ever one write operation at a time), processors will have inconsistent views of the memory. In practice, this is mitigated by an additional piece of hardware which maintains cache coherence. Such an architecture is called *cache coherent* NUMA, CC-NUMA.

Since maintaining cache coherence is expensive, parallel programs can suffer a performance loss if different processors often access the same memory.

Shared memory multiprocessors usually run an operating system that tries to schedule computation resources fairly to all running processes. As a consequence, no single program has exclusive access to all processors. In practice, we can approximate this by saying that a process on average has access to P_A processors where $P_A < P$ the number of processors. This must be considered in the runtime analysis of programs that assume an exclusive access to the processors.

[‡] lse.sourceforge.net/numa/faq/, retrieved on 2011–03–05; www.intel.com, retrieved on 2011-03-05

1.2.3 Distributed memory

In contrast to a shared memory architecture, there exist systems with distributed memory. On the first glance, this may seem equivalent to a NUMA architecture. However, in NUMA the memory is fundamentally still one unit, even if different processing units can access different parts of it at different performance. In a distributed memory system, memory and processors are physically distinct components that need to communicate via explicit channels.

A simple example of this are computers that are connected with each other via the Internet or a local area network. Each individual node in such a network consists of an autonomous system that can perform computations locally. Often, such a system is itself a PRAM architecture. Since individual computers cannot access each others' memory, and sending data to and fro between machines is extremely expensive, we would like to prevent them altogether. This requires extra care on the part of the algorithm designer.

Both shared memory and distributed memory execute instructions in parallel on different data and are thus instances of the MIMD paradigm.

1.2.4 SIMD

In many ways, SISD and MIMD programming are similar. With SIMD, this changes. Like in shared memory architecture, SIMD architectures have multiple processing units that (usually) access shared memory. However, all processing units are controlled by a single *global control unit* which issues one instruction at a time to all processing units.

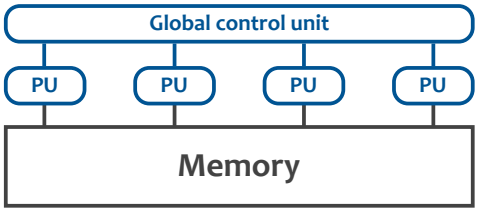

Figure I.5: SIMD architecture. Multiple processing units (PUs) being controlled by a global control unit.

That way, all processors, while running in parallel, are tightly synchronised and always execute the same instruction at the same time. The only difference between the individual processing units is that they execute their instruction on different data. This makes it possible to apply a uniform operation to a lot of data in parallel. Conversely, it makes it very difficult to perform computations that vary even a little in parallel. For example, code that contains `if` conditionals may perform perfectly well on SIMD architectures *as long as* all processing units evaluate it to the same value (i. e. all evaluate either to `true` or to `false`). However, if different processing units need to take different code paths, these

code paths need to be evaluated serially while the processing units in the other code path wait their turn.

This makes SIMD programming especially suitable for applications where a lot of identical computations have to be performed on a lot of data. The prime example of this is image processing, which is why SIMD programming is today mainly implemented in graphics cards processors.

1.3 Limit of parallelism

A sequence of executed commands $C = c_0 \dots c_{n-1}$ has parts that are executed in parallel and invariably also parts that are executed sequentially because they cannot be parallelised. Ideally, we want to keep these parts as small as possible to achieve the maximal speed-up. Let π be the runtime of the parallel section of a program and let σ be the runtime of the sequential part. Then

$$T_p = \sigma + \frac{\pi}{p} \tag{I.1}$$

is the running time of the program on p processors.

$T_1 = \sigma + \pi$ is a special case since it corresponds to the sequential execution of the program: It is proportional to the total number of executed instructions $|C|$ which is also called the *work* of a program. T_∞ corresponds to the length of the longest path of subsequent instructions in the execution, which in turn corresponds to the sum of the length of the sequential parts (the assumption is that with infinitely many processors, the parallel part of a program takes no time at all). This is called the *critical path length*.

The *speed-up* S_p of a program is consequently determined by dividing the work of the program by its path length, that is, by

$$S_p = \frac{T_1}{T_p} \tag{I.2}$$

The goal of parallelisation is to have this speed-up as high as possible. A value ≤ 1 means that there is *no* speed-up. We strive for a speed-up proportional to the number of available processors, i. e. $S_p \approx p$.

⁂

There is an upper bound of how much any program can benefit from parallelisation. This limit is formalised in an argument posited by Amdahl [1967], hence called *Amdahl's law*.

Theorem 1 (Amdahl's law). *The speed-up S_∞ of any given program is limited by the portion of the program that cannot be parallelised. Assuming we can parallelise a fraction $\varphi =$*

$\pi/(\pi + \sigma)$ of a program's control flow. Then S_∞ is limited as follows, assuming an infinite number of processors:

$$S_\infty = \frac{1}{1 - \varphi} \tag{I.3}$$

That is, the speed-up through parallelism is inversely proportional to the serial fraction of the program's execution. In practice we only have p processors, further limiting the speed-up that can be achieved realistically:

$$S_p = \frac{1}{\frac{\varphi}{p} + (1 - \varphi)} \tag{I.4}$$

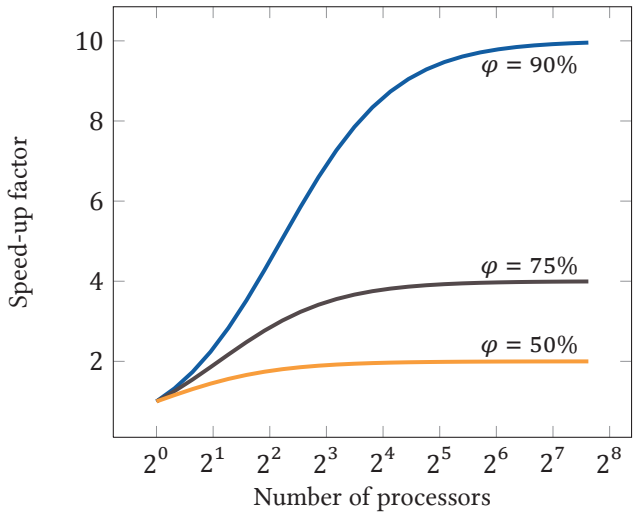

Figure I.6: Amdahl's law illustrated. Speed-up of different programs with varying degree of parallelism, given in percent.

However, in bioinformatics, this upper bound is of little concern for the time being: As we have seen, the problem size (i. e. the size of the input data that has to be processed) is becoming ever larger due to the increasing availability of biological data. If the actual data processing is the part of the program which can be parallelised, that fraction of the program's overall runtime will likewise increase, thus improving the achievable speed-up further. This intuitive argument has been formalised in Gustafson's law [Gustafson, 1988]:

Theorem 2 (Gustafson's law). *The speed-up of a program gets closer to being proportional to the number of processors the smaller its sequential fraction is. That is,*

$$S_p = p - \chi \cdot (p - 1) \tag{I.5}$$

where $\chi = \sigma/(\pi + \sigma)$ is the sequential fraction of a program. Assuming that the bulk of the data processing is amenable to parallelisation, χgets smaller the larger the input data is.

Designing a Parallel Program

II

> Too many cooks spoil the broth.
>
> *(Anonymous)*

When implementing an algorithm for execution in parallel, we need to accomodate this in the algorithm's design. In particular, since different CPUs work on different parts of the data, we must make it explicit which part of the input is processed by which resource.

Foster [1995] describes a general way of reasoning about the design of a parallel program. He calls the necessary steps in this process PCAM: partitioning, communication, agglomeration and mapping.

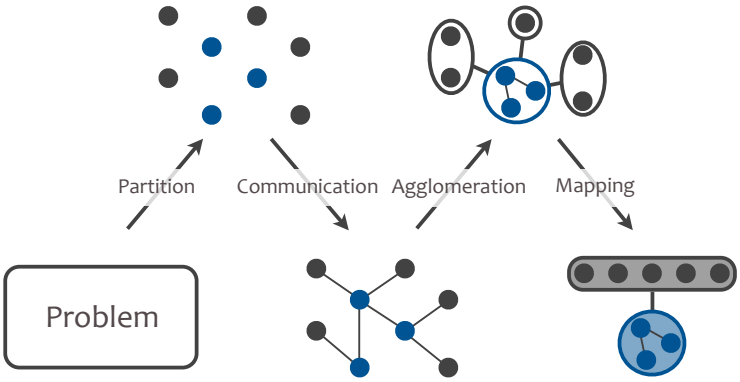

Figure II.1: PCAM illustrated (adapted from Foster [1995])

In the first step we *partition* the problem into tasks that are as small as possible. This decomposition helps us to find parts of the algorithm that can be run in parallel. We classify the decomposition into two kinds: *functional decomposition* and *domain decomposition*.

In functional decomposition, tasks are identified by the function that they perform, and different tasks will perform different functionalities. Basically, we will end up with different tasks where each task describes one (or several) steps of the original algorithm that can (hopefully) be executed in parallel. This is also called *task parallelism*.

In domain decomposition, the focus is on the data instead of the algorithm. The decomposed tasks will thus all have the same functionality, but performed on a different part of the input data. This is also called *data parallelism*, since the same algorithm will be executed in parallel on different data.

In practice, most algorithms can be decomposed either way, and it often makes sense to apply both decompositions. Consider a program which reads a file containing a DNA sequence, filters out all non-DNA content (such as whitespace, annotations and comments), transforms the DNA to upper-case (see algorithm I.2) and writes it back to a file. This program can be functionally decomposed into the following parts: the reading of the input, the filtering, the mapping to upper-case and the writing of the file.

On the other hand, it could additionally be decomposed to benefit from data parallelism, by applying the mapping to upper-case on the letters in parallel.

In the next step, we analyse the *communication* between the tasks that result from our partition. "Commmunication" is here synonymous with "*dependencies*", that is we want to know which requirements any given task has. Ultimately, these requirements determine which other tasks have to be processed before a given task can be processed and thus they guide our parallelisation.

In our example, there would be a dependency between all functional tasks since they rely on the data from the previous step. Additionally, we can note that there are *no* dependencies between our domain decomposed tasks: all letters can indeed be mapped in parallel. The absence of communication is also an important information that this step seeks to establish.

After having decomposed the problem into small tasks in the first step, we now seek to *agglomerate* tasks with identical requirements and group blocks of data to benefit from cache locality. This reduces the overall number of tasks which in turn reduces the overhead of managing all the tasks – both mentally when implementing the parallel algorithm and on the computer at runtime.

In the case of the transformation of DNA to upper-case, this could mean that instead of transforming each letter individually, we divide the input data into contiguous, non-overlapping blocks and transform each of those blocks in parallel. This has the advantage of reducing the overall need of data transfers which is particularly important on distributed architectures but also in the simple NUMA model, where different CPUs could each load one data block and process it efficiently.

The last step in PCAM is *mapping*: We map the tasks obtained in the previous step to our hardware. In distributed systems, this can be an intellectual challenge since we strive to localise communication. However, for the PRAM target architecture this step is done

automatically by the operating system scheduler which maps individual tasks (and the threads that execute them) to processors on the fly.

2.1 Data dependencies and data decomposition

Data can often be decomposed in structured ways that, while dependent on the algorithm, can be categorised into a few broad groups. We use this classification to investigate automated ways in which existing algorithms may be parallelised.

2.1.1 Independent data

Figure II.2: No dependencies allow for trivial parallelisation by processing each element of the input data on any of the processors.

In the simplest case, there are *no* dependencies between the elements in the input. If that is the case we call a program *embarrassingly parallel* since the lack of communication makes parallelisation trivial: execution can happen in any order. Figure II.2 illustrates this by using different colours for the different CPU cores that process a data element; in this case, we're using only two cores but the principle is general. It is important to note that the processor allocation is arbitrary; it doesn't need to follow a strictly alternating pattern as in the figure.

A common class of algorithms that follow this schema is the *map* operation that transforms a container by applying a uniform operation to each element.

```
1 procedure Map(X)
2     for x ∈ X in parallel do
3         Process x
```

Algorithm II.1: The general *map* schema adapted to parallel execution.

2.1.2 Reduction

Figure II.3: Reduction algorithms introduce a dependency between the data elements that can be resolved by systematic, structured distribution of the computation.

Two other common classes of algorithms are *reductions* (also known as *accumulation* or *folding*) and *scans* (also known as *prefix sums*). In both cases, an intermediate result depends on previous intermediate results. They can be parallelised in a similar manner, if the operation that is applied happens to be commutative (and with a bit more effort even if it isn't). Blelloch [1993] has studied this class of dependency extensively and describes a simple schema for its efficient parallelisation.

Despite the fact that such algorithms *do* have a relevance in bioinformatics (e. g. in the creation of q-gram indices, see e.g. Rudolph [2008]), they were – for the time being – omitted from the framework and shall therefore not be considered further here. Instead, this thesis focused on another common kind of non-trivial data dependency.

2.1.3 Divide & conquer

Divide & conquer (D&C) algorithms solve a problem by *dividing* it into several smaller parts which are solved recursively, and *merging* them back together. Tasks are divided until they have become small enough to be trivially solvable (*base case*). Consider the case of sorting algorithms, which are often D&C algorithms: An array of length less than two are always sorted because they are either empty or consist of a single element. A trivial base case for sorting algorithms is therefore attained as soon as the individual problem parts contain less than two elements.

Figure II.4: D&C algorithms introduce another kind of structured, exploitable dependency.

The handling of the base case (and indeed determining whether a base case is reached), dividing and merging are algorithm dependent. However, at every point in the algorithm work is only done on a locally known part of the data, and this data is non-overlapping on the same recursion depth. This means that D&C is highly amenable to parallelisation because non-overlapping parts of the input data can be processed in parallel.

In the simplest case, every "layer" of the recursion (i. e. every invocation with the same recursion depth) is processed at the same time, in parallel. Different layers are processed sequentially. This would be easiest to implement but we shall see later that this simplistic parallelisation can be improved.

```
1 procedure DivideAndConquer(X)
2     if X is a base case then
3         Process base case X
4         return
5     Divide X into [Y_0 ... Y_n[
6     for Y ∈ [Y_0 ... Y_n[ in parallel do
7         DivideAndConquer(Y)
8     Merge [Y_0 ... Y_n[ back into X
```

Algorithm II.2: The general D&C schema naively adapted to parallel execution.

2.1.4 Other dependencies

Unfortunately, not all algorithms fall into one of the above categories – many algorithms exhibit complex dependencies that require a manual case-by-case treatment. We call these *arbitrarily dependent* and they are not handled by our framework.

2.2 Parallel patterns

Other approaches than PCAM exist for the design of parallel programs. We had initially pursued such an approach based on a *pattern language* to describe parallelisation.

The idea of using a special language to describe common patterns of problems and their solutions was initially put forward by Alexander *& al.* [1977] for use in architecture. The approach was adapted to software design by Gamma *& al.* [1995] to describe common patterns in object oriented design.

In a pattern language, a commonly occurring pattern is described using a semi-formal language by detailing *how* and *why* it can be applied, what problem it solves and how it relates to other patterns (i. e. other commonly encountered problems). It also often includes a concrete example that is discussed in detail. Patterns are grouped according to common characteristics. The intent is to make patterns *searchable* so that they can serve as a handy reference, and making their names pervasive to create a common vocabulary.

The latter has been particularly successful with the original design patterns of Gamma *& al.*. The names of most of the patterns described there have entered common speech in object-oriented design and thus facilitate communication.

Massingill *& al.* [2000] have developed a set of such patterns to guide the design of parallel programs. In this framework, the parallel patterns were of limited usefulness because we are only dealing with a handful of well-structured dependencies. For arbitrarily dependent algorithms on the other hand, the patterns provide a powerful tool set.

2.3 Scheduling / Load balancing

A *schedule*[*] is responsible for distributing the task threads on different processor cores. An important goal is to keep all cores busy all the time: As soon as a core has finished its work on a task, it becomes idle (effectively wasting computation resources); the goal of a scheduler is to prevent idle time by assigning a new task to be executed on the idle core.

One can think of a scheduler as the chef in a busy kitchen: her most important task isn't the cooking itself (although she may also lend a hand with the actual process of cooking), nor to provide the recipes. Her task is to supervise the other cooks and make sure that everybody is busy – which is exactly what the scheduler does.

Several different strategies exist to achieve this. In particular, we distinguish between *static* and *dynamic* load balancing. In the first case, the execution schedule is decided upon and fixed before the parallel execution begins. In the latter case, the schedule can be adjusted dynamically during the parallel execution to take new information about the load distribution into account. This adds more overhead at runtime, but promises a more evenly balanced schedule that can react to unevenly distributed work. Figures II.5 and II.6 illustrate static and dynamic load-balancing on independent data in an array.

[*] There is no universally agreed-on terminology in the literature. To avoid confusion, I will use the terms *schedule* as in the OpenMP specification [OpenMP Architecture Review Board, 2008], and synonymously with *scheduling* and *load balancing*

Figure II.5: Static schedule. Tasks are distributed at the start of the computation. Different colours indicate different threads. Dark shading indicates work that is already done. Light shading indicates work that has not yet been processed. The numbers indicate at which time step t a task has been completed.

Figure II.6: Simple dynamic schedule. Tasks are requested by threads as they go. Hollow tasks have not yet been processed. In this example, the grey thread executes its tasks much faster than the blue thread.

There exist more sophisticated dynamic schedules. One particularly interesting schedule was proposed by Blumofe & Leiserson [1999] and is called *work-stealing*.

Figure II.7 illustrates an example of work-stealing where the tasks have been evenly distributed to all threads at the beginning of the computation. This isn't always necessary – it is also conceivable that only one thread starts off with work and all other threads start by stealing work as it becomes available from the first thread. This will be useful in cases where the data decomposition is dynamic, i. e. D&C dependencies.

Figure II.7: Work-stealing schedule. Initially, all tasks are evenly distributed. Different colours indicate different threads. As soon as a thread is out of work it steals tasks from another thread. In this example, the blue thread has executed all its tasks and steals from the grey thread.

Arora & *al.* [1998] have adopted it to the PRAM architecture and proved that it schedules tasks optimally: Given P processors of which P_A are on average available to the computation, a computation which uses the work-stealing schedule completes in expected time $\mathcal{O}(T_1/P_A + T_\infty/P_a)$ – that is, in time proportional to length of the serial portion of the program, plus length of the parallel execution on P_A processors. In particular, Blumofe & Leiserson [1999] have shown that work-stealing is superior in runtime to another class of schedulers known as *work-sharing* (which are implemented in some OpenMP constructs). We have therefore chosen to implement work-stealing in this thesis in addition to the schedules provided by the underlying threading framework OpenMP.

OpenMP

III

OpenMP [by the OpenMP Architecture Review Board, 2008] is an API standard for a compiler extension to C and Fortran to enable a simple operating system independent parallelisation on shared memory multiprocessors. Our main focus lies in its support for so-called *fork/join parallelism*.

3.1 Fork/join parallelism

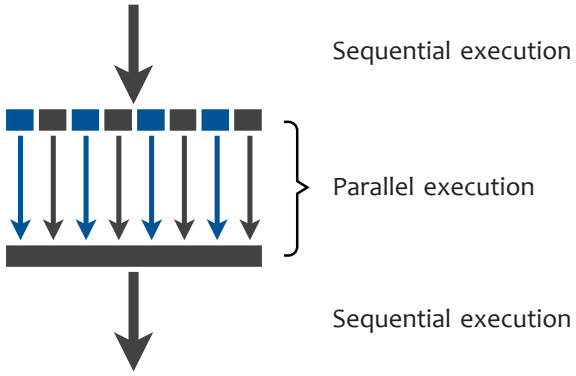

Figure III.1: Fork/join parallelism control flow. Arrows indicate the execution of a sequence of statements. Different colours symbolise execution in different threads but more than two threads are in fact possible.

The default state in fork/join parallelism is the sequential execution in a single thread until the execution flow reaches a directive that tells the thread to fork into a *team* of several child threads, of which the previously single thread is now one (the *master*).

Now the execution resumes in parallel with each thread working on a piece of work that it was assigned (OpenMP has different mechanisms for assigning work to a thread, depending on the chosen schedule). After a thread is done with its work it reaches a *barrier* which causes it to wait until all other threads of its team have completed their work.

When all threads of a team have reached the barrier, all except for the master thread are destroyed and the master thread resumes sequential execution of the program flow.

directives: commands that are executed by the preprocessor of a compiler and usually transform the code

The runtime is advised to spawn thread teams through special instructions in the code. OpenMP is grafted on to C++ through the use of `#pragma` *directives*. For instance, to make OpenMP parallelise a simple `for` loop with no data dependencies, a single command is sufficient:

```
1 #pragma omp parallel for
2 for (int i = 0; i < a.size(); ++i)
3     a[i] *= 2;
```

Listing III.1: A parallel `for` loop in OpenMP

OpenMP takes care of managing the thread teams and scheduling the tasks. It even supports several different schedules which include simple static and dynamic schedules and a so-called *guided* schedule which behaves much like a dynamic schedule but in addition dynamically adapts the block size of a task. This has the advantage of having a low scheduling overhead at the beginning (where large blocks are used) and a fine-grained control at the end, when most of the work is done.

Furthermore, the completely automatic management of the threads implies that OpenMP implementations can realise a more sophisticated handing of threads: spinning up a thread is relatively *costly* on modern operation systems. Creating new threads for each parallel region in a fork/join program would therefore imply a massive overhead of each section. Modern OpenMP implementations mitigate this by managing a *thread pool*.

A thread pool manages idle threads such that they only need to be created once and can be re-used subsequently as they are needed. This means that while the first parallel region will still have a runtime overhead for creating the threads, subsequent parallel regions don't have this overhead, in the best case. The overhead of creating threads, and hence the savings associated with the use of a thread pool, can be enormous. For example, under Windows, "[a]ssuming an x64, single core, dual processor the threadpool takes about 16ms to startup."[*]

Clearly, OpenMP is a clean and simple solution to parallelise existing algorithms that exhibit no data dependencies. However, other data dependencies are supported much less completely. For instance, support for reductions is only rudimentary and support for D&C is non-existent. Furthermore, direct usage of OpenMP means that we need to make changes to existing code, even though these changes are sometimes straightforward.

[*] msdn.microsoft.com, retrieved on 2011–02–07.

At the time of writing, the current version is OpenMP 3.0 which greatly improves the degree of control one has over the thread management. Furthermore, it allows parallel tasks to be specified which maps closely to the abstract description of a functional problem decomposition. Unfortunately, the Microsoft C++ compiler doesn't yet support version of OpenMP, and nor do older versions of other C++ compilers that we nevertheless want to support. This made OpenMP 3 unsuitable for our purposes although it would have greatly facilitated some tasks.

IV Existing Frameworks

The following sections discuss several existing frameworks and APIs geared at parallel programming. This chapter firstly tries to identify potential candidates to solve the same kinds of problems that OpenMP and our framework does. It also takes a step back and looks at some fundamentally different approaches of parallel programming that target different hardware and use different paradigms; these are notably *stream programming* and *distributed computing*.

4.1 Low-level libraries

On the most basic level, support for multithreading, and thus concurrent programming, is provided by the operating system libraries. The most widespread library for this is the **POSIX threads library** (*pthreads*) [Open Group, 2004] that works on most operating systems except Windows.

Since platform dependence is a serious problem and since low-level APIs are always written with a C interface, there exist C++ wrappers for such libraries to make them platform independent and offer a cleaner, easier to use interface. The default implementation is without a doubt **Boost.Thread** [Williams, 2008].

The upcoming C++ standard C++0x will also include multithreading support both in its core language and in the standard library [C++ Standards Committee, 2010]. Once this part of the standard is widely implemented, there won't be any need to use other libraries, and indeed it would be beneficial not to do so since using the language facilities will mean optimal portability. But at the time of writing this isn't widely adopted by C++ compilers yet. Furthermore, if backwards compatibility with older C++ compilers is a requirement, it cannot be relied on either.

4.2 High-level libraries

There are several high-level approaches to multithreading that are complementary to the framework described in this thesis. A similar approach to what we have tried here is pursued by the GCC **parallel mode extension** that is based on the MCSTL project [Singler *& al.*, 2007].[*] Rather than offering an own interface, this library works mostly "behind the curtains" by parallelising some algorithms from the C++ standard library. Apart from parallel implementations for specific algorithms (such as `std::sort`) it also parallelises general-purpose algorithms that can be used as building blocks of other algorithms. If an existing implementation makes good use of the C++ library algorithms, its parallelisation is a mere matter of switching on the compiler's extension.

The Microsoft C++ compiler offers a similar concept with their **Parallel Patterns Library**[†] by offering three basic functionalities: generic algorithms similar to the MCSTL, support for task parallelism and thread-safe adaptations of generic standard containers. The task parallelism functionality is essentially a replacement for OpenMP that uses library features instead of language extensions.

A similar approach is also taken by Intel's **Threading Building Blocks** library[‡] [Willhalm *&* Popovici, 2008].

All these libraries have in common that they start off from some general purpose algorithms (such as the ones found in the C++ standard library `<algorithm>` header) and offer appropriate support structures.

4.3 SIMD Programming

CUDA is an API developed by Nvidia [2010] geared at highly parallel GPU programming using a modification of the SIMD paradigm (Nvidia calls this SIMT for "single instruction, multiple *threads*"). Although this API targets the graphics card hardware, it isn't exclusively or even primarily geared at graphics programming. Rather, it aims to make the graphics card hardware accessible to general purpose applications. This principle is called GPGPU, short for "*general purpose GPU programming*".

While CUDA is a proprietary library that currently only works with Nvidia's GPUs, **OpenCL** is an open standard initially developed by the Khronos Group [2010] consortium for cross-platform SIMD programming, notably also targeting GPUs. Both libraries offer mostly low-level support for SIMD on specialised hardware. For CUDA in particular there exist high-level wrappers (e.g. **Thrust** [Hoberock *&* Bell, 2010]) that approximate the functionality of the other high-level C++ threading libraries.

Modern general purpose CPUs also offer limited support for SIMD under the trade name **SSE**. SSE instructions allow performing several predefined uniform operations on short

[*] gcc.gnu.org/onlinedocs/libstdc++/manual/parallel_mode.html, retrieved on 2011–02–12
[†] msdn.microsoft.com/en-us/library/dd492418.aspx, retrieved on 2011–02–12
[‡] threadingbuildingblocks.org, retrieved 2011–02–12

vectors of integer or floating-point numbers in parallel. To take advantage of the instructions, the programmer either has to program directly in assembly language or use compiler extensions (so-called "*intrinsics*"). On the other hand, modern C++ compilers have the ability of using SSE instructions automatically where it makes sense. This has become part of the default optimisations performed by any good compiler. Some modern compilers go even further and try to optimise certain loops to use SSE instructions instead. This feature is known as *auto-vectorisation*.

4.4 Distributed computing

The **message passing interface (MPI)** [by the Participants of the MPI Forum, 2009] describes and standardises a communication protocol for running software on a distributed memory system. There exist libraries (so-called *bindings*) for several languages, including C++. In contrast to the above-mentioned alternatives, MPI was designed to work across several physical machines; it can thus be used to execute programs in parallel on whole clusters instead of just single machines.

MapReduce [by Dean & Ghemawat, 2004] is an implementation for parallelisation over clusters provided by Google that uses a strict interface based on the subsequent application of two functions to a data set, namely the functional *map* operation, followed by a *reduction*. Due to the strict communication interface that it imposes on the algorithm the program can be parallelised extremely efficiently and the implementation has several mechanisms that ensure fault tolerance (e.g. when individual nodes in the cluster cease to respond). There exist several other implementations of the MapReduce programming model, most notably **Hadoop** [Apache Hadoop project members, 2010].

⁂

There are two factors that drove us to use OpenMP rather than any of the above solutions. Firstly, OpenMP has some tangible advantages over low-level threading libraries: it works on more platforms and hides much of the complexity of manual thread management. Even though pthreads are portable in theory this falls down in practice as soon as we want to support the Windows ecosystem. The only viable practical solution would have been to encapsulate system-specific APIs in a low-level layer or to use an existing layer, such as Boost.Thread. This would avoid the platform dependency issue but still only provides a low-level API.

Secondly, we could have used a high-level library. The only viable (because platform independent) option here would be the Threading Building Blocks library. Still, this is a *good* solution and the option to use this library could be revisited in the future. On the other hand, there were some reasons not to use it here, too. The level of abstraction offered by the TBB actually isn't well-suited as the basis of our framework; it's too high-level. Rather than using it as a basis, it could serve as a *substitute* of our framework. So

why implement a new framework at all, instead of relying on an existing one?

While there are similarities and overlaps between TBB and the framework we are designing, there are also differences. TBB and related libraries focus on algorithm *building blocks*, small routines that can be reused in the implementation of algorithms. Our goal was different: we wanted to provide an *automated* way of parallelising existing code *without* rewriting it. Using TBB, this is only possible as long as the original algorithm has already been written with concurrency in mind, or at least relying heavily on algorithmic functions as building blocks which could be parallelised.

Algorithms to Parallelise

V

The framework developed in the course of this thesis is geared towards sequence analysis and we focused especially on a set of algorithms that are of particular interest in the context of SeqAn, and attractive targets for parallelisation.

The following algorithms were parallelised in the course of the thesis. They were used to test the usability of the parallelisation library interface as well as the correctness of the task scheduling algorithms. They also served as the basis for the benchmarks and to showcase the functionality of the framework.

The criteria for choosing these algorithms were thus their usefulness in the context of a bioinformatics library and their amenability to parallelisation.

5.1 String matching

The first group of algorithms that we will look at concern *string matching* in biological sequences. This corresponds to the `find` module in SeqAn which is a rewarding area for parallelisation because it is a central part of many sequence analysis algorithms, corresponds to a large portion of the total runtime, and exhibits few data dependencies, thus being easy to parallelise.

Definition 1 (Naming conventions). In the following, sets or sequences are abbreviated with upper-case letters (such as A). Their elements are enumerated using the lower-case equivalent, affixed with an index denoting their position in the sequence. The first element of a sequence shall always be addressed by the index 0 (i. e. $A = a_0 a_1 \dots a_{|A|-1}$). In particular, we will use T to denote an arbitrary (comparably large) *text* and P to denote a (comparably small) *pattern* that shall be searched in the text. Both sequences use elements from the finite alphabet denoted by Σ, and their respective lengths will be denoted by $n = |T|$ and $m = |P|$. $t_{i,j}$ shall denote a substring (infix) of the string T starting at position i and stopping at position j and is a short-cut for $t_i \dots t_j$.

Definition 2 (Exact string matching). Let $T = t_0 \dots t_{n-1}$ be a string of length n and $P = p_0 \dots p_{m-1}$ be a string of length m over a finite alphabet Σ.

Exact string matching is the problem of finding some or all positions i such that $t_{i,i+m-1} = p$ holds.

Sometimes it is desirable to also find inexact matches, that is matches that allow for a few errors. To allow this, we can define an error threshold and use *approximate matching*.

Definition 3 (Approxiate string matching). Let $T = t_0 \dots t_{n-1}$ be a string of length n and $P = p_0 \dots p_{m-1}$ be a string of length m over a finite alphabet Σ.

An *approximate match* is a match between P and $t_{i,i+l}$ (for some l) that contains at most k errors. Depending on our usage, there exist different definitions of what an error is. In bioinformatics, the following types of errors are practical:

- A mismatch between characters t_i and p_j – for example, consider $T =$ "`post`" and $P =$ "`past`" which has a mismatch at positions $(1, 1)$.
- An insertion, where a character in the pattern p_i does not correspond to any character in the text – $P' =$ "`posit`" has an insertion of the letter "`i`" at position 3 relative to T.
- A deletion, where a character is removed in the pattern – $P'' =$ "`pot`" has a deletion of the letter "`s`" at position 2 in T.

Other types of errors exist (e. g. inversions) but are not treated by most common string matching algorithms. Matches that use the above errors are often characterised as having *edit distance* k, also called the *Levenshtein distance* after Левенштейн (Levenshtein) [1965].

An alternative way of defining approximate string matching is to search for matches with an ϵ distance.

Definition 4 (Finding ϵ matches). An *ϵ match* is a match between two strings A and B with an *error rate* of ϵ and a minimum length n_0. That is, there are at most $k = \epsilon \cdot \min\{|A|, |B|\}$ errors and $n_0 \leq \min\{|A|, |B|\}$ (otherwise we would allow spurious matches with the empty string).

There are several different approaches to solving the string matching problem that can be classified into different groups according to several desirable properties. In this thesis, we have focused on a group of algorithms selected as noteworthy by Navarro & Raffinot [2002] and more specifically on those that are also already implemented in SeqAn.[*] Section A.1 highlights some of them.

[*] `http://trac.mi.fu-berlin.de/seqan/wiki/Tutorial/PatternMatching`

5.2 Sorting

sorting: rearranging the elements in a collection according to some ordering criterion

A typical example of divide & conquer (D&C) algorithms is *sorting*, which also plays a big role in bioinformatics[†]. The implementation of two sorting algorithms was used to demonstrate the versatility of the D&C data specification. First of all, we implemented *quicksort* [Hoare, 1962] as an efficient general-purpose sorting algorithm that, with some care, performs well on most input data and that runs *in-place* (i. e. does not require more than constant extra memory), which is an important property especially with very large data sets. Quicksort usually performs in $\mathcal{O}(n \log n)$ expected time and generally outperforms other sorting algorithms despite its worst-case running time of $\mathcal{O}(n^2)$.

Despite quicksort's superior performance in most cases, its potential worst-case running time may still be problematic. For this reason, modern C++ standard library implementations use a slight variation known as *introsort* [Musser, 1997]. Introsort starts out as quicksort but keeps track of the recursion depth. If it recurses too deeply, it switches its algorithm to heapsort which has a guaranteed worst-case running time of $\mathcal{O}(n \log n)$. This performs almost as well as quicksort and has the worst-case running time of $\mathcal{O}(n \log n)$.

Secondly, we implemented *merge sort* (described e.g. in Cormen & *al.* [2005]). Merge sort has the advantage of being *stable*, i. e. the order of two elements a and b that compare equal for the sake of the sorting will not swap their respective position. This property may be important when sorting according to one criterion and subsequently by another. Unfortunately, merge sort with a worst-case runtime of $\mathcal{O}(n \log n)$ requires $\Theta(n)$ extra memory which may be too much for very large data sets, where parallel processing makes the most sense. There exist in-place variants of merge sort which have running time $\mathcal{O}(n \log^2 n)$ but these were not implemented here since we were only after a proof of concept anyway.

[†] for example in the creation of index data structures which are also used in string search

C++ VI

The framework we develop is part of the sequence analysis library SeqAn which is written in C++. Consequently, that is the language that was used for the implementation of this thesis. Since the design of the language C++ has a rather direct bearing on the design of our framework, it might nevertheless be helpful to revisit the reasons for choosing C++ over other languages in the first place.

Originally, C++ was developed as a replacement of the C programming language to support object orientation more readily (hence the original name "C with classes"). However, several changes, in particular the introduction of templates and a standard template library [Stepanov & Lee, 1995] have shifted the focus away from object orientation and towards other programming paradigms, hence the currently popular description of C++ as a "general purpose programming language with a bias towards system programming" [Stroustrup, 2004].

In this context, the properties of *strong typing*, an *extensible type system* and *template programming* are our primary interest. Combined, they allow the design of highly reusable, configurable, extensible algorithms that provide a very high level interface without a runtime *abstraction penalty*.

abstraction penalty: a performance drop due to the use of abstraction in code

6.1 Templates

Besides ordinary class and function declarations, C++ also allows the declaration of *templates* for classes and functions. Such templates are filled in with the appropriate "shape" (usually a type) when they are instantiated. Consider the following example of a conventional class declaration:

```
struct pair {
    int first;
    int second
};
```

Listing VI.1: A simple class holding two data members

This is a class (`struct` and `class` are largely equivalent in C++) holding a pair of values. We can refactor this into a template that can hold arbitrary types:

```
template <typename T>
struct pair {
    T first;
    T second
};
```

Listing VI.2: A simple class template

Here, `pair` acts as a "stencil" where each usage of `T` can be filled in appropriately. That way, we can declare different usages:

```
pair<int> int_pair = { 1, 2 };
pair<float> float_pair = { 1.0f, 2.4f };
```

Listing VI.3: Two different instantiations of the class template

C++ allows the usage of multiple template arguments. The same works very similarly for functions.

This provides a very straightforward mechanism for generalisation: we can now define container types for arbitrary types, and general algorithms that work on these containers. One such algorithm is the `sort` function which sorts a linear container according to some criterion. To illustrate what's so special about `sort`, we will offset it from its close kin, the C function `qsort`.

Like `sort`, `qsort` sorts a linear container. Its declaration looks something like this[*]:

[*] according to BSD's `man 3 qsort`

```
void qsort(
  void *base, size_t nel, size_t width,
  int (*compar)(const void *, const void *));
```

Listing VI.4: The `qsort` declaration

`base` is a pointer (of arbitrary type) that points to the first element in a contiguous block of memory of elements to be sorted. `nel` is the number of elements to be sorted. `width` is the size, in bytes, of one element (note that each element must have the same size). Finally, `compar` is a *pointer to a function* which *compares* two elements (given by their pointers) according to some arbitrary relative ordering that satisfies a *strict weak ordering*.

Definition 5 (Strict weak ordering). A strict weak ordering is a relation $\prec$ that satisfies the following conditions for all elements a, b and c:

irreflexivity $a \prec a$ is always false.
asymmetry $a \prec b \implies \neg(b \prec a)$
transitivity $a \prec b \wedge b \prec c \implies a \prec c$
transitivity of equivalence $a \doteq b \wedge b \doteq c \implies a \doteq c$
where $a \doteq b \iff \neg(a \prec b) \wedge \neg(b \prec a)$

– In practice, most intuitive ordering relations fulfil this definition.

`qsort` is remarkable because it works on arbitrary types, and can use any arbitrary ordering criterion. For example, we could sort integer numbers by their value modulus 60 (e. g. sorting times by their minute value only):

```
int cmp_mod_60(const void* a, const void* b) {
    return ((*(const int*) a) % 60) - ((*(const int*) b) % 60);
}

int times[] = { 153, 134, 49, 250, 232, 50, 348, 292 };
qsort(times, sizeof times / sizeof times[0], sizeof times[0], &cmp_mod_60);
```

Listing VI.5: Invoking `qsort` with an arbitrary comparer in C

Despite its generality, this method has several disadvantages:

1. It only works on contiguous arrays, not on other container data structures
2. It requires casting in the comparer
3. It uses a function pointer, which incurs a runtime overhead compared to calling a normal function
4. Its interface is somewhat difficult to use

Point 3 in particular is worth noting since we are conceivably calling the comparer function *very* often in a sort routine. Furthermore, using an indirection (via the function pointer) here prevents the compiler from optimising this call away (*inlining*), and replacing the code for a function call with the code of the function's body.

The C++ function `sort` solves all these disadvantages rather elegantly. Not only does it work for arbitrary element types, it also works for arbitrary container types and due to the use of templates it requires no casting in the comparer. But arguably most importantly of all, the comparer is *also* a template argument, which allows the compiler to take its code and put it directly into the sorting loop.

```
1 struct cmp_mod_60 {
2     bool operator ()(int a, int b) const { return (a % 60) < (b % 60); }
3 };

5 int times[] = { 153, 134, 49, 250, 232, 50, 348, 292 };
6 sort(&times, &times + sizeof times / sizeof times[0], cmp_mod_60());
```

Listing VI.6: Invoking `sort` with an arbitrary comparer in C++

We have exchanged the function `cmp_mod_60` for a *functor* here. A functor is just a small object that behaves like a function – in C++ parlance, this means that it declares an `operator ()` which makes an object of this type callable as if it were a function. Due to the use of templates, the C++ compiler will generate special code for this call to `sort` which has the type of the container (`int*`) and of the comparer hard-wired and thus it is able to optimise aggressively and reduce the comparer function call down to very few comparison instructions.

The speed-up that this yields varies greatly depending on usage but it can be phenomenal. For example, Meyers [2001] reports a speed-up of 670% when sorting a vector of a million `doubles`.

Other programming languages use other means of implementing a generic sorting function. But it universally holds that either of the following conditions is true:

- The code decides at run-time which comparer function to call, just like C's `qsort`. For example, most object-oriented libraries provide an interface that defines how equal types can be compared, and let classes implement that interface. This uses so-called *subtype polymorphism* via virtual functions; since virtual functions work similar to function pointers, they have the same run-time characteristics.
- Separate code is written for each type that can be sorted. For example, the Java runtime function `java.util.Arrays.sort` is redefined for each of the fundamental data types.[†] Since the implementations are essentially identical (with few exceptions), this results in large quantities of duplicate code that is hard to maintain.

[†] At least in Java Runtime Environment versions 1.2–6

- Code is generated separately for each set of parameter types, as done in C++ via templates.

This seems to indicate that reusable object oriented code fundamentally incurs a runtime overhead that cannot be elided completely. Since performance is of utmost importance for a sequence analysis library that deals with huge data sets, C++ is a worthwhile candidate.

6.2 Iterators

An avid reader may have noticed that our invocation of `sort` in listing VI.6 still used pointers to delimit the start and the end of the range to be sorted, and yet we have claimed that this works for arbitrary containers. C++ abstracts over the concept of pointers by introducing the more general concept of `iterators`, of which pointers are but one instance. In reality, `sort` works for arbitrary iterators.

Definition 6 (Iterators). Formally, let $C = c_1, c_2 \ldots c_{n-1}$ be a collection of elements. Notice that this collection need not be (but may be) well-ordered. We only introduce the ascending positions here to have a way of addressing individual elements. An *iterator* over C is an entity that represents any element at position i, or the special "one-past-end" position n. That is, a valid iterator either refers to an element of the collection or is one-past-end.

In practice, pointers belong to a sub-category of iterators called *random access iterators*. They refer to elements in memory via their respective memory addresses. The one-past-end pointer of a given array is simply a pointer that points to the address directly behind the last element. Furthermore, a random access iterator also knows how many elements lie between itself and another random access iterator of the same container (pointer arithmetic lets us simply subtract two pointers to get that value), and conversely, can jump to other elements by adding an offset (hence the name).

Other classes of iterators are more restricted, for example the *forward iterator* which only knows how to get to next element in the order of the container, but not to any other element. A typical example of such an iterator is a reference to a list node in a linked list: to get to the next element it is sufficient to follow the link to that element.

⁂

Two iterators of the same container delimit a *range*. This, finally, explains the usefulness of iterators: we can model *any* arbitrary range in any container (even "infinite" containers such as sequences), by the half-open interval $[a, b[$ generated by two iterators a and b. A *full* range of a container C is defined simply by $[c_1, c_n[$. In listing VI.6 we have passed the array `times` to the `sort` function via the range $[\texttt{times}, \texttt{times} + n[$ with

$n =$ `sizeof times / sizeof times[0]`. The same can be done using any other container type in C++. The container types of the standard library define member functions `begin` and `end` that return the start and end iterator, respectively. Using this, we can sort a `std::vector`, which, like the C array used previously, defines a contiguous storage of elements with random access.

```
1 std::vector<int> elements;
2 // Fill elements ⋯
3 sort(elements.begin(), elements.end());
```

Listing VI.7: Calling `sort` on C++ container classes

(This uses the default ordering for elements, which is defined by the functor `std::less` that simply implements a natural ordering of the elements.)

This is merely a demonstration of a general guiding principle that pervades the design of algorithm libraries in C++. SeqAn has appropriated and extended this concept.

SeqAn

VII

SeqAn is a sequence analysis framework developed in C++ by Döring *& al.* [2008]. It uses a specially developed techique dubbed *template subclassing* instead of conventional *subtype polymorphism* to achieve a very high level of abstraction and polymorphism while still outperforming object-oriented frameworks considerably on common tasks.

The following exposition merely touches on those aspects that are of special importance for this thesis. For a more general overview see Gogol-Döring & Reinert [2009] or visit the website at www.seqan.de.

7.1 Template subclassing

Conventional subclassing works by inheritance: one class is defined as *inheriting* from another class, and thus takes on its public interface (with few exceptions) and also its implementation. On the other hand, it can *extend* the inherited interface and *refine* the inherited implementation.

In SeqAn, the same is achieved by adding a template argument called `TSpec` to a base class. In order to sublass the base class, the programmer defines a new *type tag* to identify the subclass and creates a (partial) *specialisation* of the base class.

Definition 7 (Type tag). A type tag is an empty class. Its sole purpose is to distinguish one (class or function) template instantiation from another. This makes use of C++' strong typing. In SeqAn, type tags by convention extend a common base template, `Tag<>`.

As an example, let us define a base class `Base` and extend it:

```
1 template <typename TSpec = void>
2 struct Base { };

4 // Declare a type tag for the subclass
5 struct TagDerived_;
```

```
6  typedef Tag<TagDerived_> Derived;

8  // Define the subclass
9  template <>
10 struct Base<Derived> { };
```

Listing VII.1: A simple class hierarchy with a base class and a subclass.

Neither the class `Base⟨⟩` and its subclass `Base⟨Derived⟩` do anything particularly interesting but we have now defined a class hierarchy and can implement functions for them.

```
1 template <typename TSpec>
2 inline void echo(Base<TSpec> const& me) {
3     std::cout << "Hello from Base" << std::endl;
4 }
```

Listing VII.2: A method implemented by `Base⟨⟩`

This defines a method `echo` that takes an instance of `Base` and prints a message. This method can be used with *any* object of type `Base`, in particular also with a subclass object:

```
1 Base<> b;
2 Base<Derived> d;
3 echo(b);
4 echo(d);
```

Listing VII.3: Calling `echo`

This will print the same message twice. But now we can *override* this method for instances of the subclass, just as we could override virtual methods in object-oriented programming:

```
1 template <>
2 inline void echo(Base<Derived> const& me) {
3     std::cout << "Hello from Derived" << std::endl;
4 }
```

Listing VII.4: Overriding the base class method

If we now execute listing VII.3, a different message will be printed for the object `d`, thanks to C++' rules for overload resolution. This also ensures that, unlike in conventional subtype polymorphism which relies on *late binding*, the call dispatch happens at compile time and thus has no overhead at run-time.

late binding: resolution of a type or call at run-time

7.2 Containers

Due to its significance in sequence analysis, the central data structure in SeqAn is a special container, the *string*, which is just a sequence of uniform data. Most algorithms in SeqAn are defined in terms of strings. SeqAn allows parametrising the container to any type, so that strings can be used to store any kind of data, not only the obvious text characters. Thus, strings are, at the bare minimum, just linear, contiguous storage space for data of a uniform type. That way, they completely replace other contiguous containers commonly employed, such as arrays or vectors.

Strings in SeqAn can be parametrised in two ways: by their *type* of data that they store, and by their *specialisation*, as explained above. Thus, the conventional string in SeqAn is written as `String<T, TSpec>`. The default specialisation is the `Alloc` string which behaves much like a C++ `std::vector`.

However, the concept of a string can be generalised even more to a *container*. For example, there are containers which provide *views* into strings, such as *segments*, that abstract over infixes in strings.

When implementing algorithms for SeqAn, it is important to keep the interface as general as possible. In particular, since strings aren't the only containers, this must be taken into account. Like the `std::sort` function introduced in chapter VI, it is often sufficient to define the algorithm in terms of a range. This works since the contract of containers in SeqAn mandates the existence of appropriate iterators. However, sometimes having just the iterators isn't enough, and further information about the type is required. This is solved in SeqAn (and in other modern C++ libraries) via *metafunctions*.

7.3 Metafunctions

For example, assume that we want to write a function that reverses an arbitrary container. Implemented for a well-known range, it might look as follows:

```
void reverse(CharString& str) {
    unsigned const len = length(str);
    for (unsigned i = 0; i < len / 2; ++i) {
        char tmp = str[i];
        str[i] = str[len - 1 - i];
        str[len - 1 - i] = tmp;
    }
}
```

Listing VII.5: The function `reverse` implemented for `CharStrings`

But what if we don't want to use a string over characters? How can we declare the `tmp` variable for an arbitrary type? If we just had to care about strings then it would be sufficient to make element type a template:

```
template <typename T> void reverse(String<T>& str) ⋯
```

Listing VII.6: A first attempt to make `reverse` generic

But since we want to work with arbitrary containers, this no longer works. Enter metafunctions. A metafunction declares a type *as a function of* another type. In this case, we need the `Value` metafunction which, given a container, tells us the element type of the container.

```
template <typename C>
void reverse(C& str) {
    unsigned const len = length(str);
    for (unsigned i = 0; i < len / 2; ++i) {
        typename Value<C>::Type tmp = str[i];
        str[i] = str[len - 1 - i];
        str[len - 1 - i] = tmp;
    }
}
```

Listing VII.7: A better generic version of `reverse`

Here, `Value⟨C⟩::Type` yields the correct type for each container.

7.4 Multiple dispatch

We have shown how template subclassing realises dynamic method call dispatch at compile time by relying on class template specialisation and overloading. This means that SeqAn can be used very much like an object-oriented framework, but without incurring

the runtime overhead. But there is one noteworthy difference between conventional subtype polymorphism and the mechanism provided via overloading: Conventional subtype polymorphism only works on the calling object's type*. Consider the following method call (in a conventional object oriented language, e.g. C++ or Java):

```
1 a.method(b);
```

Listing VII.8: Method call in object-oriented notation in a C++-like language.

Here, the actual target `method` is influenced by the dynamic type of `a` but *never* by the dynamic type of `b`: A method call is dispatched depending only on one type (disregarding overloading, which doesn't work with dynamic types, only with static types). By contrast, overloading and template specialisation work on *any* number of arguments. This means that SeqAn's template subclassing allows dynamic dispatch depending on multiple dynamic types. In OOP parlance, this is called *multiple dispatch* and is very hard to realise.

However, it is an integral part of SeqAn. For example, multiple dispatch is necessary to determine dynamically which `find` method to call in SeqAn's *Finder* module: Finders implement the string matching algorithms briefly discussed in chapter V. To perform a string matching, the method `find` is called with two arguments: a *finder* and a *pattern*, both of which are refined via template subclassing. Those two arguments control different aspects of the pattern matching algorithm. In order for SeqAn to call the correct `find` method depending on finder and pattern, it relies on multiple dispatch.

As we will see, the parallelisation framework also relies on multiple dispatch. Other designs would have been possible as well but since the mechanism already exists in SeqAn it seemed natural to adapt the interface to convention.

By designing our own framework within SeqAn, we must ensure two things: First of all, that the framework is as general as possible and works well with all SeqAn types which it aims to support. We achieve this by programming against interfaces, rather than implementations, and by using the methods and metafunctions defined by the container concept.

Secondly, that types in the framework adhere to these same concepts: Our framework contains container-like classes – we need to make sure that those fully implement the SeqAn sequence interface to ensure that they can be reused by other SeqAn modules.

*With few notable exceptions, such as Common Lisp [ANSI X3J13 committee, 2004]

Part II

Methods & Implementation

A description of the interface and the implementation of the framework and a walk-through example illustrating its usage.

VIII Goals & Scope

8.1 Design goals

The time allotted for a master thesis isn't nearly enough to implement, from scratch, a complete framework of the complexity envisaged here. One of the design goals from the start was therefore to find just the right balance between immediate usefulness and extensibility. We picked out two particular scenarios which we judged to be of great interest in a sequence analysis library and implemented these. Furthermore, the framework was designed in such a way as to accommodate other typical scenarios for parallelisation.

The scenarios that were implemented cover two domains:

1. Independent execution of a calculation on one-dimensional data sets (i. e. strings), and
2. the D&C paradigm.

Both concepts are very general and of direct applicability to sequence analysis; in particular, most string search and sequence comparison algorithms can be performed in parallel by dividing the input data into independent blocks, and performing the computation on these blocks. On the other hand, several algorithms commonly used in sequence analysis rely on D&C strategies. Among these are quicksort, Hirshberg's (and related) algorithms for approximate sequence comparison and linear-time suffix array construction.

The modularity of the framework design guarantees that the framework can be extended and improved in two directions:

1. *Outwards* extension by adding new features such as schedulers and data dependency schemas, and
2. *Downwards* substitutions by improving specific modules.

As an example of the former, consider *scan* or *reduction* data dependencies (see section 2.1) which were omitted from the current framework, but which can be added easily.

As for the latter, consider the work-stealing schedule for independent data whose implementation in this thesis is based on the implementation in the GCC parallel extension[*] [Singler & Konsik, 2008]: several improvements upon the original algorithm exist (see chapter XIV) and could be implemented here.

⁂

The design of the framework is specifically targeted at the large quantities of uniform data usually found in sequence analysis. Since most data exists in the form of sequences anyway, offering support for a **data decomposition** seems obvious. Although the framework does not offer direct support for task parallelism, there exists a direct mapping so that modelling task parallelism isn't difficult.[†]

The framework should make it easy to adapt existing algorithms to parallel execution; in particular, it shouldn't be necessary to modify the actual algorithm at all, if possible. That means that the framework needs to provide an infrastructure that *delegates* work to the existing algorithm. The main work of the framework is to distribute the data to the concurrent threads so that each thread can process its local data. That means that in many cases the original algorithm implementation needs not be modified at all; it will merely be called with different arguments (namely, a small portion of the whole input data) and there will be a wrapper call to invoke the algorithm in parallel, and gather the results.

However, this isn't always possible. Even in the simplest case, a D&C algorithm needs some modification to scale across multiple processors because D&C algorithms are not generally implemented by making the divide and merge steps explicit. In these cases, the framework still strives to make adapting the algorithm as effortless as possible. It does so by offering a standard interface for each type of data decomposition that corresponds to the general outline of these algorithms (see section 2.1).

8.2 Scope

Our framework targets the **SMP architecture**: this is a promising target since it exploits the parallelism that is already available in today's machines, without the need to buy separate, specialised hardware or expensive cluster architectures. Furthermore, this is an area where we can expect enormous growth in the near future.

We have decided to target common dependencies between **one-dimensional data**. We are well aware of the relevance of multi-dimensional data decomposition (e. g. in parallel

[*] the implementation can be found in the GCC C++ standard library header file `parallel/workstealing.h`

[†] reduction to data parallelism works by creating an array of the tasks, although this simplistic approach may be prohibitive for a large number of tasks; a lazy data structure must then be employed

matrix multiplication) but we have decided that catering to more general decompositions would make the API significantly more complicated and there is actually not a lot of common ground between these decompositions that could be exploited in a systematic manner. Furthermore, one-dimensional data has a special relevance in the context of the SeqAn library which deals almost exclusively with sequence data.

We also did not implement a collection of parallel algorithm building blocks, as other parallel programming libraries do. Our main focus was to make existing algorithms run in parallel *without* changing them where possible (independent data) so parallel data structures have a limited usefulness. However, this is a decision that could be revisited in the future to make parallel implementations of *new* algorithms easier.

Realisation

IX

> “Cooking is like Asynchronous Programming without any synchronisation primitives.
>
> (@chrisoldwood)

The implementation was done in two stages after the initial design: first, an incomplete prototype was developed and tested by implementing an interface for the finder algorithms. Based on the experience from that implementation, the API of the protoype was refined and missing parts were implemented. The final API design is described in detail in the next chapter.

The initial prototype only used a naive parallelisation approach since the goal was to keep it simple, and only the interface was of actual importance. The main properties we looked for in the prototype were ease of use, and the ability to *easily* and *efficiently* adapt existing algorithms, i. e. without incurring an overhead in either writing the code nor its execution.

9.1 Blocked decomposition

In our classification of algorithms the strongest guarantee we can make is that the algorithm has *no* data dependencies between individual tasks. We have furthermore noted that pattern searching in a string is independent: Each position in a string can be tested independently whether it matches with a pattern.

Figure IX.1: Naive pattern matching. Patterns are matched position by position against a reference string (above) simultaneously by two different threads (blue and grey).

But this is not how efficient pattern searching algorithms work; this naive approach in the worst case has to compare every position in the reference string with every position in the pattern. Advanced pattern searching algorithms strive to prevent just that. There exist two general strategies to do this.

One strategy is to avoid looking at every single position in the reference sequence. A preprocessing collects information about the pattern that makes it possible to *skip* some portions of the reference. Horspool's algorithm is an example of this.

A second alternative is to maintain a *state* in the pattern matcher that allows to make multiple comparisons at the same time, either in the reference or in the pattern (or both). The bit-parallel pattern matchers are an example of this. In both cases, the strict independent data model breaks down because different threads need to share information about either the position or the state.

Figure IX.2: Skipping pattern matching. After matching at the first position, the algorithm can skip several positions.

The problem can be solved by coalescing adjacent positions into blocks, and have each thread process one block. In fact, this makes sense even for truly independent data to increase locality of reference and corresponds to the agglomeration step in PCAM. By having such blocks, we can run the pattern matching in parallel in different blocks and within these blocks the algorithms can skip positions or share state. However, this is not enough: If blocks are non-overlapping then two adjacent blocks have a common border. Any hit of the pattern that would span this border will never be found since neither of the blocks checks for such a hit because the underlying pattern matching algorithm only checks for hits that lie *completely within* the range that they are given.

This must be solved by making adjacent blocks overlapping so that every possible position is checked. The size of the overlap varies and is determined (in the case of pattern matching) by the length of the pattern and the number of allowed errors: If we allow insertions and deletions then the ending position of a pattern may be shifted by as many as k positions, where k is the number of allowed errors.

We can generalise this overlapping of blocks by noting that some algorithms, when working on a given range within a container, may require a *history* of previous positions. The framework accommodates this by allowing users to override a function `getHistory` for the blocked data decomposition that determines by how many positions each but the first block is extended backwards. However, this means that two threads may access the same (overlapping) memory at the same time. It is therefore *forbidden* to modify this memory and *it is undefined* what happens if one thread modifies the overlapping memory.

9.2 Work-stealing for independent data

locks: abstract data type to forbid concurrent execution of a piece of code

critical sections: code which yields undefined behaviour when executed concurrently

For an independent data decomposition, the work-stealing is rather straightforward. We have implemented two variants of it, one using *locks* to secure *critical sections*, and one lock-free implementation.

Both the locked and lock-free implementation of the work-stealing algorithm for independent data follow the same general outline which is summarised in algorithm IX.1.

The call to `yield` in line 11 allows other threads to progress. This is important to prevent thread starvation when the operating system schedules threads in an adaptive manner and with more threads than can be executed in parallel, as noted by Arora *& al.* [1998].

The work per thread is managed in a queue (implemented as a simple array with indices pointing to the first and last element). In the lock-free implementation, work is removed from the queue by incrementing or decrementing these pointers atomically. However, an atomic increment is actually not enough here since we also need to retrieve the current value (so that the thread knows which element it has removed from the queue). This doesn't work with the `atomic` pragma of OpenMP: this only allows modifying a value, not reading it at the same moment.

The operation we need here is know as *fetch and add* (a single-threaded implementation

```
1  Split(work) into small chunks (of fixed size)
2  Distribute tasks evenly to threads T

3  for t ∈ T in parallel do
4      Mark t as busy
5      while at least one thread is busy do
6          while t still has unfinished work do
7              Retrieve next chunk of work task
8              Process(task)

9          Mark t as idle

10         while nothing was stolen do
11             Yield()
12             Pick random thread v
13             Try to steal work from v
14             if all threads are idle then breakfrom parallel section

15         Mark t as busy

16 Merge(processed tasks) into result
```

Algorithm IX.1: Work-stealing schedule for independent data decomposition. The client needs to provide the methods `Process`, `Split` and `Merge`.

is shown shown in listing IX.1). It can be implemented via locks but most processors implement this instruction directly. Unfortunately, there is no compiler independent way to access this instruction so we had to rely on compiler intrinsics.

```
1 template <typename T> T fetchAndAdd(T& value, T increment) {
2     T old = value;
3     value += increment;
4     return old;
5 }
```

Listing IX.1: Single-threaded implementation of the fetch-and-add operation

The benchmarks (chapter XII) indicate that in practice both the locking and non-locking work-stealing variants for independent data performed on par – but the variant using locks was much easier to implement because fewer possibilities for race conditions exist.

Race conditions constitute one of the main difficulties to implementing multi-threaded

code. We had the advantage that at every point in the framework the dependencies between threads are very explicit. This reduces the opportunity for such errors.

Nevertheless, several hard to track race conditions found their way into the lock-free work-stealing code. Because they trigger only very rarely, unit tests did not reliably flag them and the debugging process necessary to unearth them took several days. This proved the folk wisdom that lock-free programming is a lot harder than programming with locks. We therefore decided to only implement a locking version for the D&C work-stealing scheduler.

9.3 Divide & conquer

Näher & Schmitt [2009] have noted that the class of D&C algorithms can be further subdivided into sub-categories which invite different solutions. While there exists a general parallel implementation for all D&C algorithms, this is not the most efficient for some cases. In particular, algorithms which have a trivial[*] "merge" step (compare algorithm II.2) can be parallelised more easily than those which have a non-trivial "merge" step.

Ideally, this difference would have no bearing on the public interface; but letting C++ figure out whether or not an algorithm implements the `recursiveMerge` method (which performs the merging) isn't reliably possible.[†] As a consequence, there are now two classes for D&C algorithms: `DivideAndConquer⟨⟩` and its specialisation `TrivialMerge`.

The D&C framework works by dividing a task into sub-tasks unless that task is already a leaf. If the task is a leaf, then it is processed. Tasks that are processed are then merged back together into their parent task. It is necessary that every task remember its parent to percolate partial solutions in the correct order to the top (unless no merging takes place).

Furthermore, in order to merge back into a task X, all its children have to be processed. In a recursive implementation this happens automatically – but in the work-stealing code, this has to be made explicit. We solved this by having each thread manage *two* task lists: the first is a queue of tasks that have not yet been started; the second is a stack of tasks that have been divided and await merging. Other threads can only steal from the queue, not from the stack. The implementation is summarised in algorithm IX.2.

For `TrivialMerge`, the algorithm is simpler because we don't have to manage a stack of started jobs. In particular, this means that the code from line 13 to line 17 as well as the other related checks are missing. This renders the data structure that represents a single task vastly more simple: It does no longer need to maintain references to its children and the number of busy children. This in turn means that the trivial merge algorithm can be implemented relying solely on thread-local stack space.

[*] meaning, they perform no operation

[†] a detailed discussion can be found at stackoverflow.com/q/4805807, retrieved on 2011–01–26.

The general-purpose algorithm on the other hand needs to maintain *pointers* to tasks which were in our case realised via raw pointers to freestore memory. Such a use of manual memory management and raw pointers is usually not a good idea but SeqAn lacks a cheap smart pointer mechanism that could be used here. Implementations such as `std::tr1::shared_ptr` or `boost::shared_ptr` [Dimov, 2002] were not used because the code should work on older compilers and SeqAn generally strives to minimise the reliance on external libraries.

```
1  Create root task and assign it to thread t_0
2  Mark t_0 as busy

3  for t ∈ T in parallel do
4      while at least one thread is busy  or t has started tasks do
5          while t can successfully dequeue task from its task queue do
6              if IsLeaf(task) then
7                  Process(task)
8                  Mark task as done
9              else
10                 Divide(task) into [child_0 ... child_n[
11                 Push task onto started tasks
12                 Enqueue child tasks to task queue in decreasing order of size

13         while t has started tasks do
14             Inspect the top task on stack
15             if task has busy children then break
16             Merge(children) into task
17             Pop task from stack

18         Mark t as idle

19         if at least one thread is busy then
20             while nothing was stolen do
21                 Yield()
22                 Pick random thread v
23                 Try to steal work from v
24                 if all threads are idle then break
25             if nothing was stolen then continue
26             Mark t as busy
```

Algorithm IX.2: Work-stealing schedule for D&C data decomposition. The client needs to provide the methods `Divide`, `Merge`, `IsLeaf` and `Process`.

9.4 Thread-local storage

The `Algorithm` specification needs to store the thread-local algorithm state whose type is specified via a template argument. Now the question arises: what to do when *no* state needs to be stored? We cannot simply use `void` as the state type since it is illegal to declare member of type `void`. One possibility would be a partial specialisation for a `void` state but we opted for a slightly different solution, providing a custom class called `EmptyStorage` which is just an empty class.

But a member of an empty class type *still* requires storage due to the object identity rules in C++. Instead of specialising the `Algorithm` class for an empty state, we employ a trick to ensure that the state member will only consume space if it is non-empty: We *inherit* from the state template type. That way, we profit from *empty base class optimisation* which ensures that an empty base class of any class will not consume any memory [Vandevoorde & Josuttis, 2003].

Interface

X

In the simple case of an algorithm without data dependencies, there are only the following operations that need to be implemented for each algorithm:

1. Dividing the work into independent tasks,
2. Perform one independent task,
3. Merge the results back together.

In practice, at least the first step is often a no-operation since dividing the work is trivial.

For D&C, this is slightly more complex; although the operations are similar, the dividing and merging needs to be performed repeatedly (namely, before and after each recursive solution of a subproblem). Additionally, the parallel framework needs to know whether it has reached an atomic subproblem, i. e. a *leaf* in the computation tree [Näher & Schmitt, 2009].

Users of the framework need to be able to specify which algorithm should be run – this entails implementing the already mentioned functions. Furthermore, users must be able to invoke a parallel computation, and specify how many threads to use, which load balancing strategy should be used, and which data to run the algorithm on. All this has to be captured in the public interface of the framework and should be as easy as possible without restricting extensibility.

The following object model uses an UML like format. UML itself could not be used since UML cannot model multiple dispatch and tag-based specialisation.

The object model in figure X.1 shows the classes and their relationships. Now follow the methods implemented by these classes. Methods signatures are given as base classes only, without all template arguments.

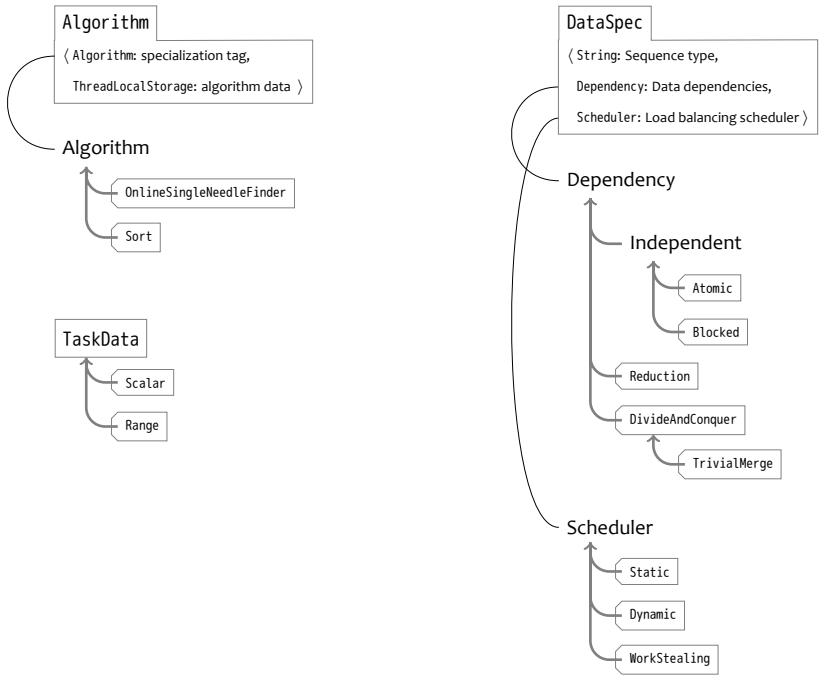

Figure X.1: The final draft of the framework object model. Base classes are set in rectangles. The available specialisations of some of their template arguments are shown.

10.1 parallelFor

To execute an algorithm in parallel, users need to invoke `parallelFor`. There exist several overloads of the method:

```
1 parallelFor(Algorithm&, DataSpec&)
2 parallelFor(Algorithm&, DataSpec&, unsigned int)
3 parallelFor(Algorithm[], DataSpec&)
```

Listing X.1: `parallelFor` client interface

If a third argument is given, it indicates the number of threads that should be used. If the argument is omitted, the framework uses as many threads as there are CPU cores. Alternatively, the user may pass an array of `Algorithm` instances. In this case, it is assumed that there is one `Algorithm` instance for every thread and the size of the array indicates the number of threads to be used. This only works for the independent data decomposition.

10.2 DataSpec

```
1 DataSpec(String)
2 host(DataSpec): String&
3 begin(DataSpec): Iterator
4 end(DataSpec): Iterator
5 length(DataSpec): Size
```

Listing X.2: Methods of the DataSpec class

Specialisations of the DataSpec class may need to override some or all of these methods.

10.3 Algorithm

The Algorithm class itself only defines constructors and a single method to return the thread-local state.

```
1 Algorithm()
2 Algorithm(State)
3 state(Algorithm): ThreadLocalStorage
```

Listing X.3: Methods of the Algorithm class

Users who want to run algorithms in parallel need to specialise the Algorithm class by overriding the following methods. Signatures which end in { } have a default implementation and don't need to be overridden.

```
1 process(Algorithm&, TaskData&)
2 preprocess(Algorithm&, TaskData&) { }
3 postprocess(Algorithm&, TaskData&) { }

5 split(Algorithm[], Algorithm)
6 merge(Algorithm&, Algorithm[]) { }
```

Listing X.4: Methods of the Algorithm class for Independent⟨⟩ data

Unlike the process method, the preprocess and postprocess are not run on each task. They initialise / tear down the thread-local state for a consecutive execution of tasks. This means that if thread 1 executes three *consecutive* tasks then preprocess is invoked for the first task, process is invoked for each task and postprocess is invoked for the last task.

```
getHistory(Algorithm, TaskData): Size { return 0; }
```

Listing X.5: Methods of the `Algorithm` class for `Independent⟨Blocked⟩` data

For D&C algorithms, different methods have to be implemented.

```
recursiveSplit(Algorithm[], Algorithm, TaskData[], TaskData)
recursiveMerge(Algorithm&, Algorithm[], TaskData&, TaskData[]) { }
isLeaf(Algorithm, TaskData): bool
process(Algorithm&, TaskData&)
```

Listing X.6: Methods of the `Algorithm` class for `DivideAndConquer⟨⟩` data

Recursive splitting and merging is performed on tasks, which consist of task data and the algorithm state.

10.4 TaskData

The `TaskData` classes are provided by the framework. Depending on whether an independent atomic decomposition is used, tasks consist of *scalar* task data or *range* task data.

```
host(TaskData): String&
begin(TaskData): Iterator
end(TaskData): Iterator
length(TaskData): Size
```

Listing X.7: `TaskData` methods

A ranged task is constructed as follows:

```
TaskData(String&, Iterator, Iterator)
```

Listing X.8: `TaskData` constructor

For scalar task data, the `length` is always `1` and the `begin` and `end` iterators return a range that contains just that scalar value. It has one additional method and a constructor:

```
TaskData(Iterator)
value(TaskData): T&
```

Listing X.9: `TaskData⟨Scalar⟩` methods

The `value` method returns a reference of the scalar value represented by the task.

Using the Framework

XI

This chapter will illustrate the usage of the framework with an example problem. We will look at two aspects: writing an interface that allows the execution of an existing algorithm in parallel, and invoking the parallel algorithm via `parallelFor`.

At the highest level, `parallelFor` needs to know two things:

1. Which algorithm to perform, and
2. which input data to perform it on.

Before giving a detailed explanation, let us first consider an example problem whose parallelisation will guide the explanation of the parallel framework.

11.1 An example problem

We will consider a very simple example that we want to parallelise – counting the base frequencies in a DNA string:

Definition 8 (Counting base frequencies). Let T be a DNA string. We want to know, for each base, how often it occurs in the string relative to the other bases; that is, its frequency.

Sequentially, this could be written as shown in the following listing:

```
1 template <typename TString, typename TChar>
2 inline void countBaseFrequencies(
3     TString const& str,
4     Map<Pair<TChar, double> >& frequencies)
5 {
6     typedef typename Iterator<TString const>::Type TIterator;
7     unsigned counts[ValueSize<TChar>::VALUE] = { 0 };
```

```
9       for (TIterator i = begin(str); i != end(str); ++i)
10          ++counts[ordValue(*i)];
11
12      for (unsigned c = 0; c < ValueSize<TChar>::VALUE; ++c)
13          insert(frequencies, static_cast<TChar>(c),
14                 double(counts[c]) / length(str));
15  }
```

Listing XI.1: `countBaseFrequencies` function

This algorithm has two phases: counting and normalising. The aim of this example is to parallelise the counting step.[*]

11.2 The algorithm

Every parallel algorithm needs an own specialisation of the `Algorithm` class. This class encodes information that is specific to that algorithm:

First of all, the actual algorithm needs to be identified, and implemented. Since the purpose of this framework is to reuse existing code rather than having to rewrite it, most implementations of the `Algorithm` will only be a thin wrapper that accept arguments and call existing algorithm implementations. Thus, an `Algorithm` specifies *which* algorithm to use, and invokes it at the right time.

Every algorithm needs a unique name:

```
1  template <typename TChar>
2  struct CountBaseFrequencies;
```

Listing XI.2: The algorithm specialisation tag

To perform the algorithm, an `Algorithm` implementation merely needs to override a specific function, `process`, that is invoked by `parallelFor` on some part of the input data:

[*] Parallelising the normalisation would also be possible, but not particularly interesting, since the number of operations here is limited by the alphabet size, which is comparably small in general and for DNA in particular. Furthermore, parallelising access to a dictionary data structure has its own share of problems which will not be discussed here.

```
template <typename TChar, typename TTaskSpec>
inline void process(
    Algorithm<CountBaseFrequencies<TChar> >& algorithm,
    TaskData<TTaskSpec>& data);
```

Listing XI.3: Declaration of the `process` method

`process` is invoked separately for each thread and works on an independent piece of input data. `TaskData` is described later; for now it's sufficient to know that `data` refers to some part of the input data that corresponds to a single task, i. e. that can be processed by a single thread at a time.

Now we need some place to store the intermediate results of the calculation (the `counts` variable) in listing XI.1. Every `Algorithm` instance is thread-local; that is, there is exactly one thread that owns, reads from and writes to a given algorithm instance. Each `Algorithm` has an associated thread-local storage. By default, this thread-local storage is empty (`EmptyStorage`, in fact). The users can write their own thread-local storage for an algorithm.

```
template <typename TChar>
struct CbfStorage {
    unsigned counts[ValueSize<TChar>::VALUE];
};
```

Listing XI.4: The storage class for the `CountBaseFrequencies` algorithm

But we need to tell the `Algorithm` class that this storage is to be associated with our algorithm. We do this by specialising the `DefaultThreadLocalStorage` metafunction:

```
template <typename TChar>
struct DefaultThreadLocalStorage<CountBaseFrequencies<TChar> > {
    typedef CbfStorage<TChar> Type;
};
```

Listing XI.5: Assigning the default storage

Now the `process` function can be implemented:

```
template <typename TChar, typename TTaskSpec>
inline void process(
    Algorithm<CountBaseFrequencies<TChar> >& algorithm,
    TaskData<TTaskSpec>& data)
{
    typedef typename Iterator<TaskData<TTaskSpec> const>::Iterator
```

```
7         TIterator;

9     CbfStorage& store = state(algorithm);

11    for (TIterator i = begin(data); i != end(data); ++i)
12        ++store.counts[ordValue(*i)];
13 }
```

Listing XI.6: `process` **method** for the base frequency count.

The above code is sufficient to specify the algorithms' description fully. It can now be executed in parallel by instantiating a data specification of the input data, and calling `parallelFor`.

11.3 The data specification

Everything related to the input data is fed into `parallelFor` via an object of type `DataSpec`, short for *data specification*. First and foremost, it stores the input data in the form of a generic container (see section 7.2). Notably, this also allows specifying only *segments* of an existing container as the input data.

Secondly, the data specification contains information about the kind of data dependencies that the algorithm has. Since our example has no data dependencies, this is `Independent⟨⟩`.

Thirdly, a data specification states which load balancing schedule should be used for the parallel execution. Since the work in the example algorithm is evenly distributed over the complete data range, it makes sense to choose a static schedule since that has a reduced run-time overhead compared to more sophisticated schedules.

Putting it all together, we end up with a declaration that looks as follows:

```
1 typedef DataSpec<DnaString, Independent<>, StaticSchedule> TCbfDataSpec;
```

Listing XI.7: Declaration of a data specification for the input

We can do a bit better by specifying that our `process` method works on a *range* instead of on individual data elements. This allows the paralleliser to coalesce adjacent blocks of data, rendering the computation slightly more efficient. This is achieved by using a specialisation of the `Independent` data decomposition that works on ranges:

```
1 typedef DataSpec<DnaString, Independent<Range>, StaticSchedule> TCbfDataSpec;
```

Listing XI.8: Declaration of the data specification using ranged tasks

To recap, the data specification encodes the following information:

- The element and container type of the input data,
- the data decomposition that the algorithm presupposes, and
- the load balancing schedule to be used for the parallel execution.

11.4 The task data

The `process` method has two arguments: the thread-local algorithm description, and an object of type `TaskData`. A *task* corresponds to a part of the input data that is processed by one thread. Formally, it is an *infix* of the input data. At any given time, busy threads are working on disjunct tasks; that is, it is guaranteed that no two threads access the same task data at the same time. As long as a `process` method restricts its access to the task data and the thread-local algorithm state, it is therefore *safe* from cross-thread contention. Race conditions cannot occur, and no synchronisation or locking has to be employed.

11.4.1 Range tasks

A task is either *scalar* or a *range*.

Of course, in a *truly* independent data decomposition (like in our example) the computation can be performed on each element separately. However, this is not always the case. Consider string matching: in order to determine whether a pattern matches at a given position in a text, it is not enough to examine that position. Instead, we need to examine several consecutive positions.

11.4.2 Scalar tasks

A scalar task comprises a single, independent data element of the input data. It is represented by an instance of the class `TaskData⟨Scalar⟨TString⟩⟩`, where `TString` is the type of the underlying container. To access the data associated with a scalar task, the `value` function is invoked. We could rewrite the `process` function from above using a scalar task:

```
template <typename TChar, typename TString>
inline void process(
    Algorithm<CountBaseFrequencies<TChar> >& algorithm,
    TaskData<Scalar<TString> >& data)
{
    CbfStorage& store = state(algorithm);
    ++store.counts[ordValue(value(data))];
}
```

Listing XI.9: Implementation of the process function

Of course, we now also need to change the `DataSpec` definition back since we're no longer dealing with range tasks. `Independent⟨Atomic⟩` corresponds with `TaskData⟨Scalar⟩` (and furthermore, `Atomic` is actually the default so we omitted it above). In addition to the `value` function, scalar tasks also implement the `Range` interface, and act as a range of length 1. As a consequence, we don't actually have to rewrite the `process` method when switching from a ranged to an atomic decomposition, we just need to keep the method signature general enough so that it fits either type, as we have done in listing XI.6.

11.5 Divide & conquer

So far, we have implemented an algorithm that works on independent data. We will now very briefly turn to D&C algorithms. In fact, the general work flow is very much identical. The only difference is that other methods have to be implemented.

To illustrate the usage, we will implement a very simple quicksort version. To keep the implementation simple it will only work on values of type `int` and the default comparison. This can be trivially generalised. The interested reader is referred to the actual implementation that was used for the benchmarks.

Once again, we will start by declaring our algorithm. Quicksort is in-place and requires no additional thread-local storage so we do not need to specialise the `ThreadLocalStorage`.

```
struct TagQuicksort_;
typedef Tag<TagQuicksort_> Quicksort;
```

Listing XI.10: The quicksort algorithm tag

The first thing any D&C algorithm needs is the `isLeaf` function which determines whether a leaf task has been reached. We use a simple size threshold to decide whether a leaf task has been reached.

```
template <typename TTaskData>
inline bool isLeaf(Algorithm<Quicksort> const&, TaskData<TTaskData> const& task) {
    return length(task) < 1000;
}
```

Listing XI.11: Have we reached a leaf task?

And once we have reached a leaf task we need to process it. We do this by invoking the standard library `sort` routine.

```
template <typename TTaskData>
inline void process(Algorithm<Quicksort>&, TaskData<TTaskData>& task) {
    sort(begin(task), end(task));
}
```

Listing XI.12: Process a leaf task

Now there needs just one thing to do: the divide step which partitions the data into two parts where all elements in the first half a less than a pivot element and all elements of the second half are larger than or equal to the pivot. Again, we use standard library functions for the task. To keep matters simple, we choose the first element as the pivot.

`recursiveSplit` must fill two arrays: the child algorithm specifications and the child tasks. The child algorithm specifications aren't interesting in this case since the algorithm has no thread-local state. Nevertheless, the `childTasks` array must be resized to the correct size since this tells the parallel framework how many divisions are needed. The child tasks are created based on the two partitions (the pivot element is excluded – it already is at the correct position).

```
template <typename T, typename TComp, typename TTaskData>
inline void
recursiveSplit(
    String<Algorithm<Quicksort> >& children,
    Algorithm<Quicksort> const& algorithm,
    String<TaskData<TTaskData> >& childTasks,
    TaskData<TTaskData> const& task
) {
    typedef Task<TTaskData> TTask;
    typedef typename Iterator<TTask>::Type TIterator;

    resize(children, 2);
    TIterator split = partition(begin(task) + 1, end(task),
                                bind2nd(less<int>(), *begin(task)));
    iter_swap(begin(task), split);
```

```
16      appendValue(childTasks, TTask(host(task), begin(task), split));
17      appendValue(childTasks, TTask(host(task), split + 1, end(task)));
18  }
```

Listing XI.13: The divide step

Calling the algorithm is also simple. We only need to pay attention to one bit – telling the data specification that our algorithm has a trivial merge step. This is important because the framework unfortunately cannot figure this out automatically.

```
1  String<int> array;
2  // Fill array ···
3  DataSpec<String<int>, DivideAndConquer<TrivialMerge>, WorkStealing<> > spec(array);
4  parallelFor(Algorithm<Quicksort>(), spec);
```

Listing XI.14: An example call of the parallel quicksort algorithm

Part III

Results & Discussion

Performance Analysis

XII

12.1 Benchmark design

We expect to observe a linear speed-up with the number of processors. The benchmarks will therefore examine how the framework behaves with different algorithms when run with different numbers of threads. We also want to know how large the overhead of using the framework is; we do this by comparing the running times with a non-parallelised version of the algorithms, which we will call the *baseline*.

For the benchmark of the independent data decomposition we have implemented the string matching algorithms showcased in section A.1. We have compared these implementations to the equivalent implementations already available in SeqAn. The re-implementation was done in order to see whether the current finder interface in SeqAn incurred any overhead.

For the benchmark of the D&C data decomposition we have implemented the sorting algorithms shown in section A.2. We have tested two use-cases: shuffled data and sorted data – all the while bearing in mind that this wasn't a benchmark evaluating the respective qualities of sorting algorithms and their implementations, merely a good effort to evaluate a parallel sorting implementation of representative data.

Furthermore, we want to compare how the different load-balancing strategies perform. In particular, we need to analyse whether work-stealing keeps all processors busy even if work is distributed unevenly to begin with. To measure this, we have generated a list of tasks of different sizes (so that their processing takes a different amount of time) and have shuffled this list. We have then replicated this task list so that we ended up with one copy for each thread. Now, a copy was made of this list of tasks and sorted by task size.

We now end up with two arrays of tasks, one sorted and one unsorted. The arrays are now processed in parallel using the static and the work-stealing schedule.

Figure XII.1: Task arrays used in the work-stealing benchmark. Different shades denote differently large tasks. In this case, we are using four threads. The above array is the shuffled variant and below is the sorted variant.

In the case of the shuffled array, we expect that all schedules take about the same time since the static schedule divides the whole array into fixed-size chunks which correspond exactly to one another; every thread has the same amount of work to do. This allows us therefore to measure the runtime overhead of the work-stealing schedule compared to the static schedule.

For the sorted array, one thread in the static schedule will have substantially less work to do than another thread. We expect therefore that processing takes longer. But when using the work-stealing schedule this imbalance should be mitigates – the processing should take the same time as with the unsorted array. The ratio between these run-times gives us the efficiency of the work-stealing schedule.

Finally, we need to remember that some of our work-stealing implementations do not stall unnecessarily due to one thread having to wait for another. This was tested by taking times before and after the acquisition of locks and comparing the time spent waiting with the overall progress of the program.

12.2 Results & discussion

The benchmarks were performed on a Linux machine with GCC 4.3.2. The specifications of the hardware are shown in table XII.1. All programs were compiled with high optimisations enabled (`-O3`) and with debug symbols disabled. No further specific optimisations (such as `-funroll-loops`) were enabled.

Hyper-threading is Intel's name for hardware threads which is a support for concurrent threads built into the CPU hardware. Hardware threads cannot execute concurrently but the CPU reserves to separate instruction pipelines for two threads. Thus, if one thread is busy waiting for main memory, the other thread can resume executing instructions from its instruction pipeline. This masks memory latencies.

In our benchmarks, we compared the scalability of different algorithms. We also compared different parallelisation schedules. Finally, we compared the performance of our frameworks against the performance of other frameworks. In particular, were interested

[*] Table adapted from Martin Riese's master thesis. Used with permission

Part	Configuration
Processors	2 Intel Xeon X5550 with hyper-threading
Cores	2 per processor
Hardware threads	2 per core
Clock speed	2.66 GiHz
Cache level 1	32 KiB per core (data & instructions each)
level 2	256 KiB per core
level 3	8 MiB shared
Main memory	72 GiB

Table XII.1: Configuration of the machine used for the benchmarks*

in the overhead incurred by our framework. To establish this, we benchmarked against a "raw" OpenMP implementation of some algorithms that does not rely on our framework.

Furthermore, we have measured the performance of using the GCC parallel extensions instead of our framework. The GCC parallel extensions acted as a stand-in for other libraries such as the TBB which use a similar principle and thus we expect their performance to differ only slightly. Furthermore, examining the implementation of the GCC parallel extensions did not reveal any obvious abstraction overhead so we can reasonably expect it to perform competitively.

We do not need to worry about the scalability with increasing input size, therefore we have only performed benchmarks with one input size to determine the speed-up factor depending on the number of threads.

The implementations of the various load-balancing schedules contain a number of hard-coded tuning parameters. The values for these parameters were adapted from the code in the GCC parallel extensions [Singler & Konsik, 2008].

12.2.1 Independent data

Figure XII.2 shows some discrepancies between the implementations already found in SeqAn and those written for the benchmarks of this framework. However, the Horspool algorithm showcases that the overhead of the SeqAn API is not substantial. For the naive finder, SeqAn uses the `std::search` from the C++ standard library which turns out to be more efficient than a naive loop.

On the other hand, SeqAn performs substantially worse on the bit-shifting algorithms. The most probable reason for this is the fact that SeqAn supports pattern lengths that exceed the machine word size (64 on 64 bit machines). Our own implementation, being merely a proof of concept, does not do that. In fact, SeqAn tries to mitigate the added costs of supporting large pattern sizes by adding checks for the special case that the pattern length is less than or equal to the machine word size and resorting to a more

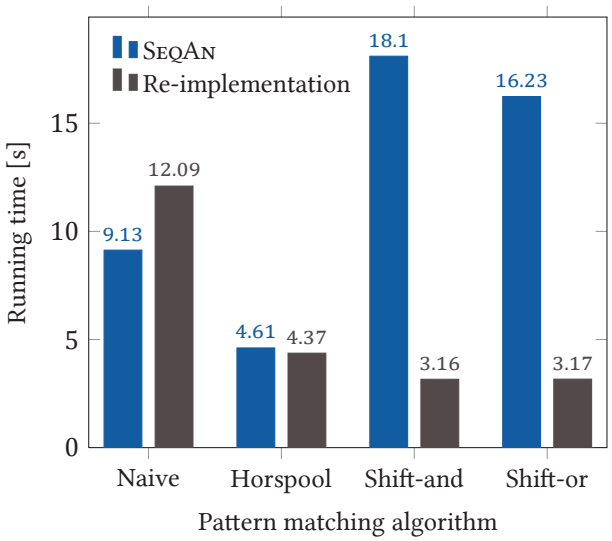

Figure XII.2: Comparison between SeqAn and re-implemented pattern matching algorithms. Algorithms were run sequentially, average was taken over 320 runs. Each run searched for the occurrence of 100 reads of length 36 in chromosome 2L of *Drosophila melanogaster* with a length of 23 011 544 bp.

efficient implementation in this case. However, this check still incurs a small runtime overhead and as we can see above, this turns out to be quite substantial. We did not analyse the reasons for this discrepancy any further.

We will focus the remaining analysis on the re-implemented Horspool finder since it performs almost equally well in our implementation and in SeqAn and its parallelisation scales well across many cores.

Figure XII.3 shows the comparison of different parallel implementations of the Horspool algorithm running on varying numbers of threads. The "sequential" variant represents the baseline and does never use more than one thread. It is worth noting that the version using the GCC parallel extensions does not scale beyond eight threads. The machine only has eight cores but every core is able to execute two interlaced hardware threads.

It is worth nothing that all schedules of our framework perform indistinguishably from the manual OpenMP parallelisation. This means that the framework does not incur a measurable overhead over a manual parallelisation. Furthermore, all three versions perform identical. This is not very surprising since the work is quite evenly distributed over the whole range and thus over all threads. A static schedule thus does not result in a large number of idle threads which could be re-distributed using a dynamic schedule.

Figures XII.4 and XII.5 show the same benchmark; this time, however, we plotted the

(relative and absolute) speed-ups instead of the absolute running times. As expected, the speed-up of the parallel implementations is linear in the number of threads and once again we can see that the GCC parallelisation doesn't scale beyond the number of cores. There is no big difference between the relative and the absolute speed-up. This shows us that the overhead of using the parallel framework can be neglected (otherwise we would expect a larger relative than absolute speed-up).

In figures XII.6 to XII.9 we compare the relative speed-ups of different finder algorithms. Both the naive search and Horspool's algorithm scale linearly over all 16 threads. The bit-shift algorithms do not take advantage of the concurrent hardware threads. This is readily explained by the fact that hardware threads do not actually run simultaneously – they can only be used to hide memory latencies.

12.2.2 Divide & conquer

The benchmarks for the D&C algorithms show some irregularities. Unfortunately, not all of them can be explained – work in this area is not finished.

The results in figures XII.10 and XII.11 show the baseline – the un-parallelised C++ standard library `sort` function – against differently parallelised implementations. Notably, the work-stealing schedule performs as expected, running faster with an increasing number of threads. The static load balancing behaves exactly like the other baseline, the sequential execution using our framework.

The close match of these two curves shows that both must in fact accidentally run the same schedule instead of different ones. In fact, the "sequential" execution *does* use the static load balancing, albeit with a fixed thread count of 1. Clearly, this isn't working due to a not yet discovered bug in the implementation. Since the code also runs faster than the native `sort` implementation it seems likely that it is always running on multiple threads.

More interestingly, the GCC parallel extension version does not scale beyond two threads. This can be explained by the fact that it parallelises the partitioning step. We will need to obtain feedback about this behaviour from the programmers. This was not yet done due to time constraints.

Figures XII.11 and XII.13 show the relative speed-up of the quicksort and merge sort implementations. Note that unlike with the independent data decomposition the speed-up is not linear in the number of threads – instead, it seems to follow a logarithmic growth.

When comparing the parallel quicksort run under a work-stealing schedule with an implementation that additionally parallelises the first partition step (which is otherwise done sequentially) we see that any speed-up gained by the parallel partition can be neglected (figures XII.14 and XII.15). However, this may be simply due to the fact that the general-purpose parallel partitioning algorithm (taken from the GCC parallel extensions) performs worse the than the specialised three-ways partition used otherwise.

Our parallel mergesort implementation doesn't scale beyond four threads. This can be seen in figures XII.13 and XII.15. Our first guess was that the non-trivial merging schedule induces a lot of waiting at locks. We tested this by putting stopwatches around the locking code and plotted the progression of the algorithm.

The result is shown in figures XII.16 and XII.17. These graphs illustrate the *progress* made by the execution, i. e. the time not spent waiting for a lock. Every dot in the graph corresponds to an event, either entering or leaving a critical section. Every diagonal line corresponds to progress. We expect to see *horizontal* lines for prolonged waiting periods. The fact that we don't see horizontal lines shows that no significant time was spent waiting.

It turns out that prolonged waiting for locks was not the cause of the slowdown.

12.2.3 Work-stealing efficiency

Table XII.2 shows the improvement of work-stealing over the static schedule when working on sorted tasks of different size. We note that the highest improvement (of 70%) is reached with eight threads. When comparing the performance on shuffled tasks (so that each thread in the static schedule has the same amount of work), static and work-stealing perform equally well.

Threads	Static	Work-stealing	Improvement
1	1.00	1.00	0.00%
2	1.38	1.99	44.50%
4	2.38	3.85	61.74%
6	3.37	5.50	63.28%
8	4.34	7.40	70.63%
10	5.14	7.47	45.35%
12	5.46	7.71	41.14%
14	6.00	7.98	32.96%
16	6.44	8.28	28.66%

Table XII.2: Efficiency of work-stealing compared to static scheduling for different threads. The columns "static" and "work-stealing" show the speed-up of the static and work-stealing schedules compared to sequential execution. The last column shows the improvement of work-stealing over the static schedule.

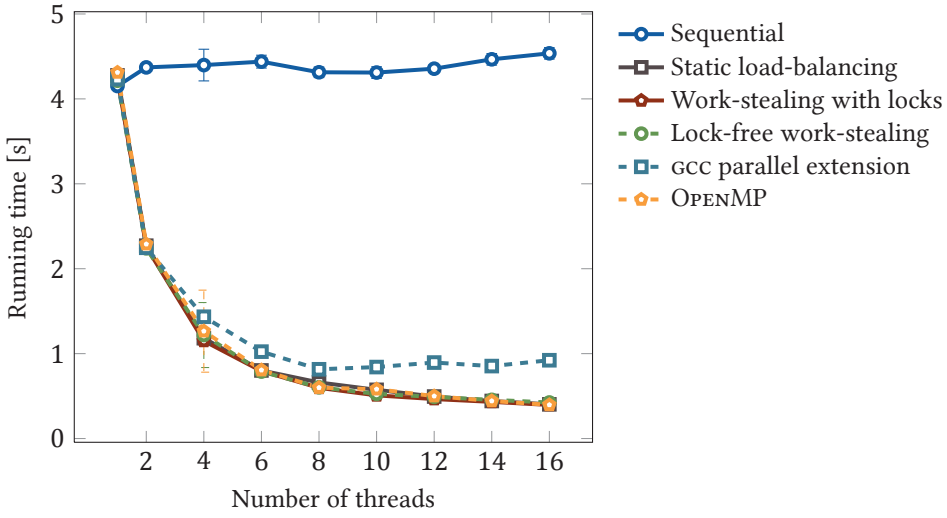

Figure XII.3: Running times of Horspool algorithm. The numbers are the average over 20 runs each. Each run searched for the occurrence of 100 reads of length 36 in chromosome 2L of *Drosophila melanogaster* with a length of 23 011 544 bp.

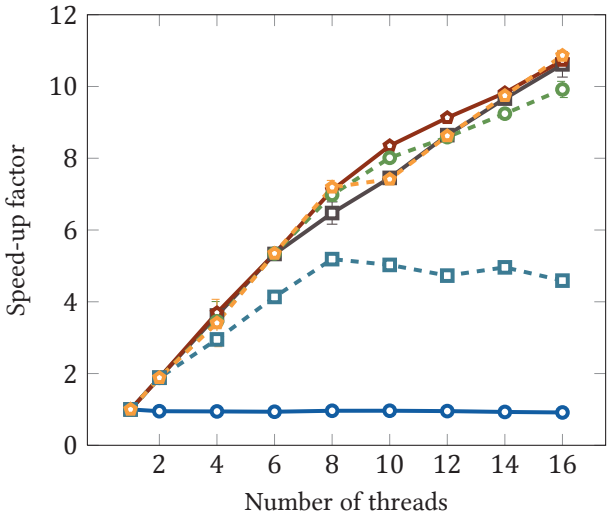

Figure XII.4: Relative speed-up of Horspool finder. The speed-up is calculated as the ratio between the time of a schedule using n threads by the time of the same schedule using one thread. The legend is identical to that in figure XII.3.

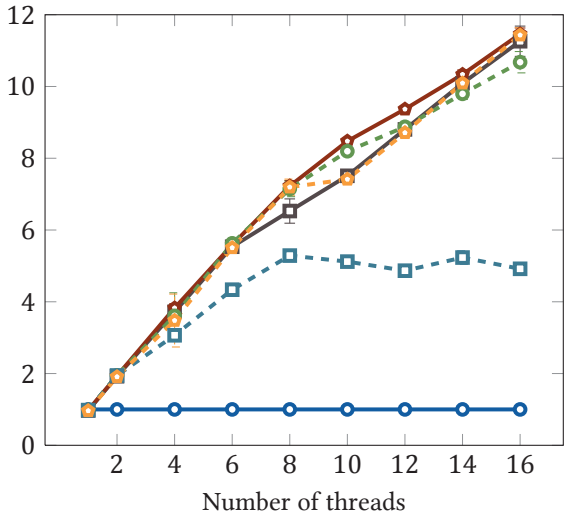

Figure XII.5: Absolute speed-up of Horspool finder. The speed-up is calculated as the ratio between the time of a schedule using n threads by the time of the sequential schedule.

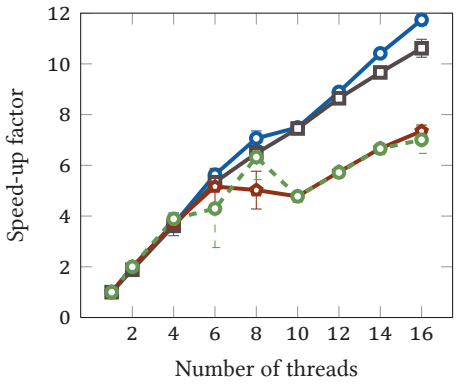

Figure XII.6: Relative speed-ups of all finder algorithms using the static schedule

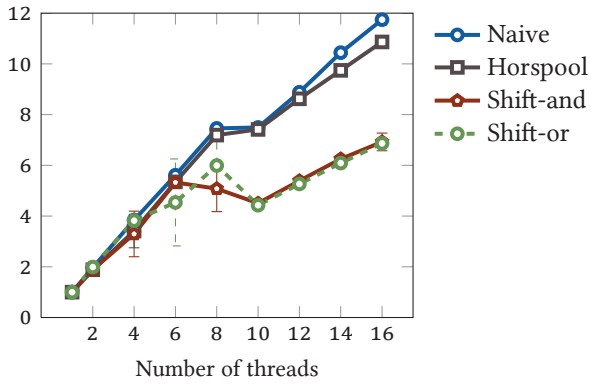

Figure XII.7: Relative speed-ups of all finder algorithms using Manual OpenMP parallelisation

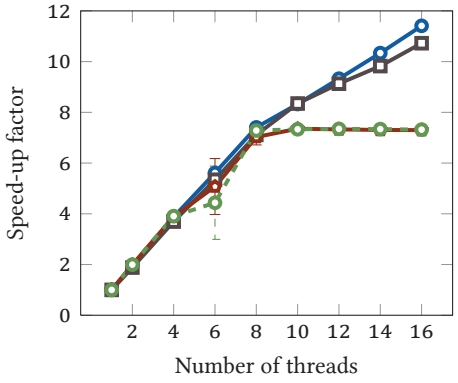

Figure XII.8: Relative speed-ups of all finder algorithms using the work-stealing schedule with locks

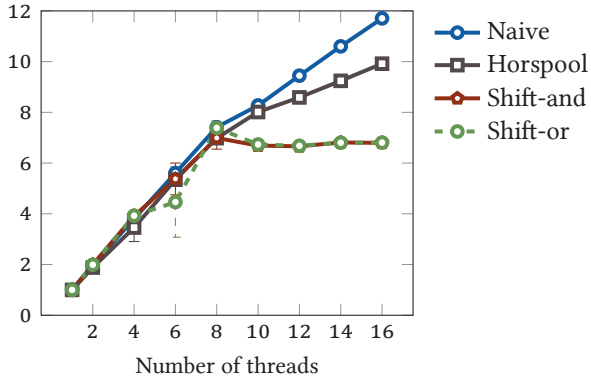

Figure XII.9: Relative speed-ups of all finder algorithms using the lock-free work-stealing schedule

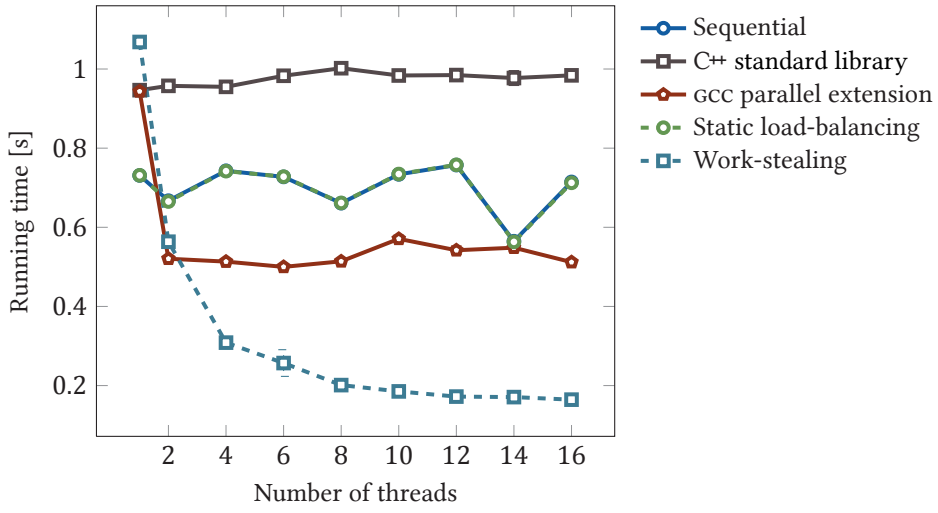

Figure XII.10: All-to-all comparison of `sort` implementations. Shown are the parallelised quicksort implementations against the standard library `sort` method and the GCC parallel extensions `sort` on an array of 10 000 000 shuffled integers. The values are averaged over five runs.

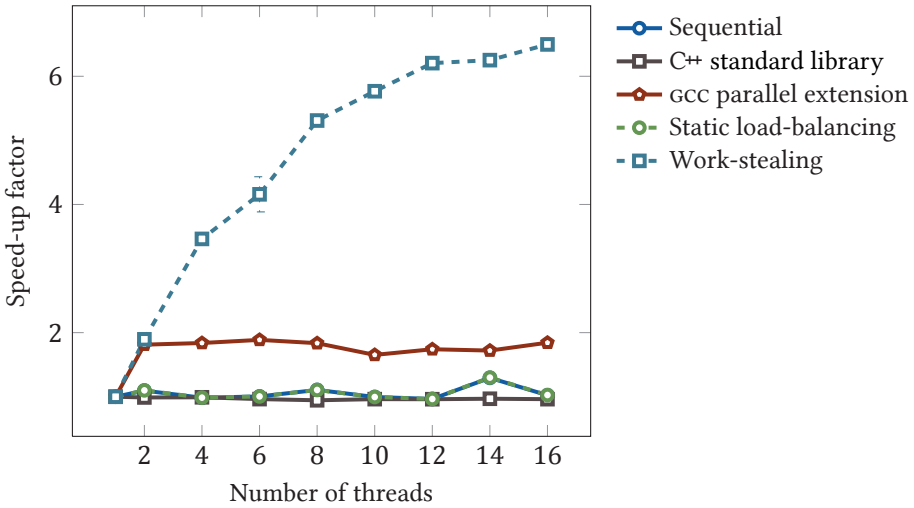

Figure XII.11: Speed-up of `sort` implementations. Parallelised quicksort implementations against the standard library `sort` method and the GCC parallel extensions `sort`.

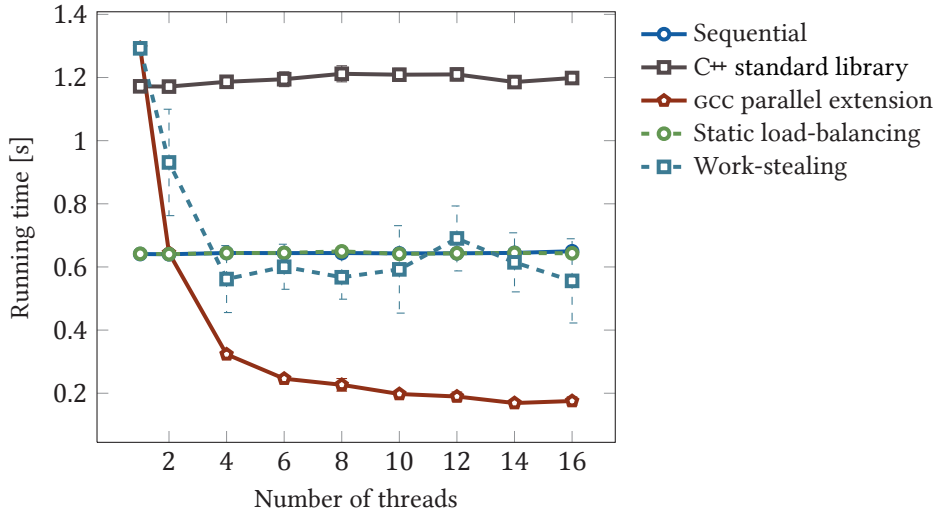

Figure XII.12: All-to-all comparison of `stable_sort` implementations. Parallelised merge sort implementations against the standard library `stable_sort` method and the GCC parallel extensions `stable_sort`.

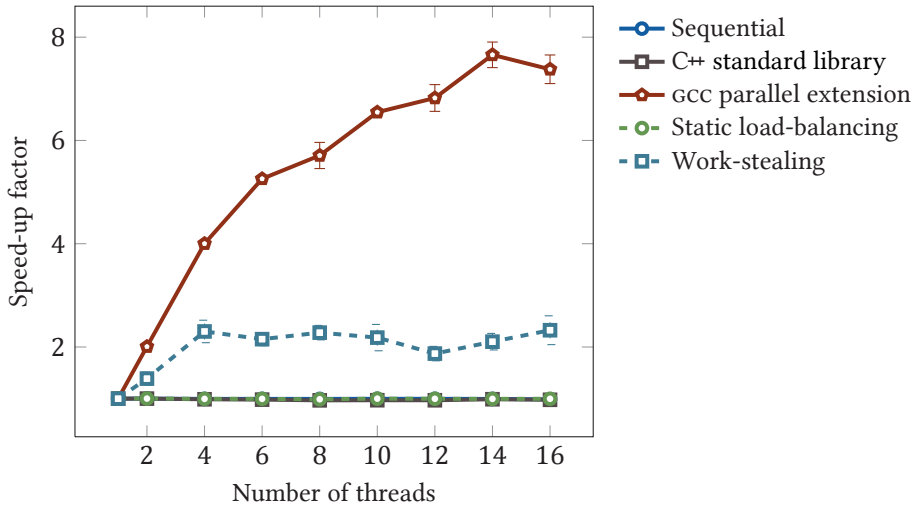

Figure XII.13: Speed-up of `stable_sort` implementations. Parallelised merge sort implementations against the standard library `stable_sort` method and the GCC parallel extensions `stable_sort`.

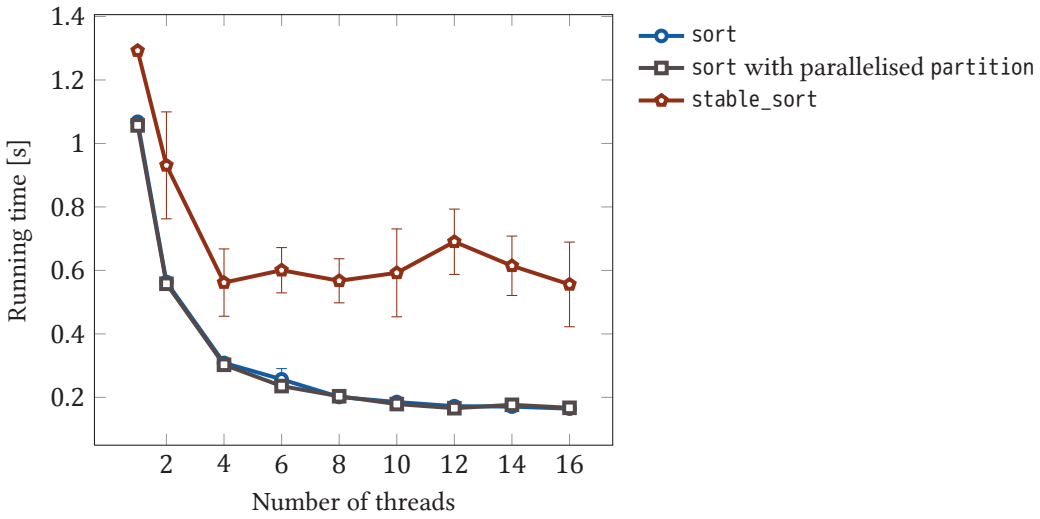

Figure XII.14: Different sorting algorithms. Comparison of quicksort, merge sort and quicksort where the first partitioning step has also been parallelised. All algorithms run under the work-stealing scheduler.

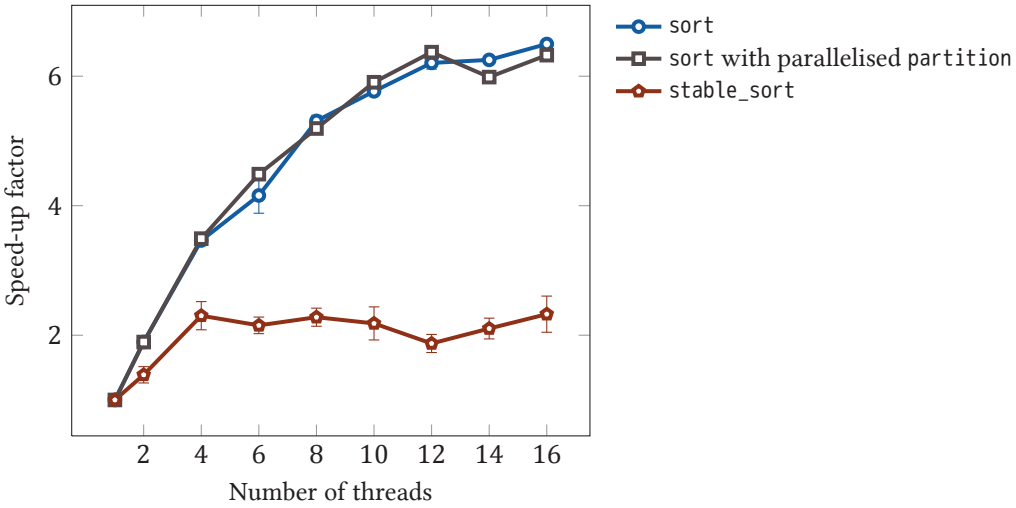

Figure XII.15: Speed-up of comparison of different sorting algorithms. Comparison of quicksort, merge sort and quicksort where the first partitioning step has also been parallelised.

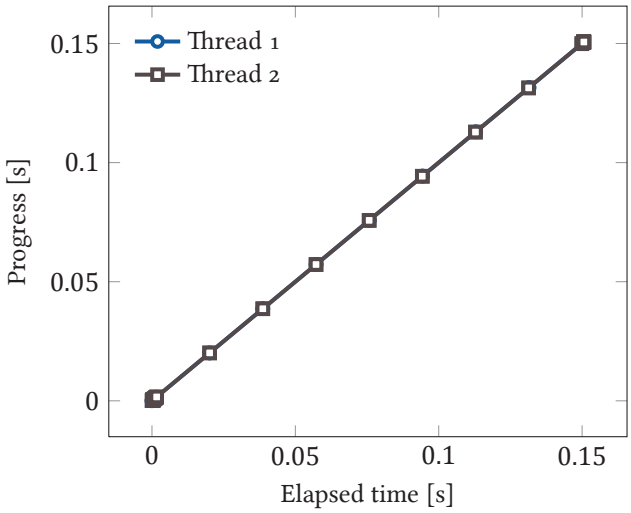

Figure XII.16: Progress of a D&C algorithm with work-stealing running on two threads. The algorithm shown here is a simple sum over an array of 250 000 000 elements, parallelised via D&C.

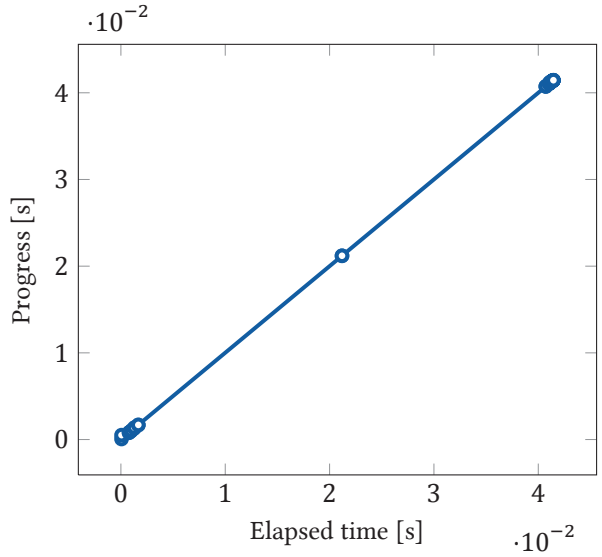

Figure XII.17: Progress of a D&C algorithm with work-stealing running on eight threads. All threads overlap completely, thus we only see a single line instead of eight.

Evaluation

XIII

> “Beware of bugs in the above code; I have only proved it correct, not tried it.
>
> *(Donald E. Knuth)*

13.1 The framework

Implementing several algorithms has shown the usefulness of the framework in parallelising existing algorithms without having to care about all the complex details of parallelisation, provided one follows a few simple guidelines.

On the other hand, the example implementation of quicksort has clearly shown that while processing can be delegated to `std::sort`, the non-trivial divide step still has to be hand-coded. In the case of quicksort this may *seem* simple but it actually isn't since the choice of the pivot for partitioning is crucial and using `std::partition` is entirely inadequate [see Bentley & McIlroy, 1993].

Thus, parallel quicksort requires implementing a good pivot choice (we chose pseudo-median of 9) and a good partitioning function (we chose three-ways partition which corresponds to the Dutch flagsort partition by Dijkstra [1976] but with the improved invariant by Bentley & McIlroy). These algorithms, though well-documented in literature and conceptually very simple (once understood), provide opportunities for many subtle bugs due to off-by-one errors. In fact, their correctness proof is much simpler than their correct implementation. All in all, the parallelisation of quicksort counts over 100 lines of non-trivial code (including correctness assertions) and covers almost all of the quicksort implementation proposed by Bentley & McIlroy.

Still, there are substantial gains from the framework even in this case, since the scheduling ensures an effortless parallelisation, even if implementing the algorithm itself isn't trivial. In fact, the code very closely approximates a sequential implementation; the only difference is a clear separation of the different parts of the algorithm which is required by the parallel framework, but also greatly improves understandability of the algorithm.

⁂

A different problem comes to light when implementing a code that makes use of the thread-local storage to save intermediate results, such as our merge sort implementation

with auxiliary memory. Since we want to prevent redundant copies where ever possible, the logic becomes non-trivial: Since the result of the merge ends up not in the data specification but in the auxiliary memory, we need to move it back to the actual data specification upon completion of the algorithm. To prevent redundant copies, we only want to do this *once*, at the top recursion level. This means that at recursion level $i > 0$ the sorted data after merging resides in the auxiliary memory.

But this is *not* the case for leaf nodes; here, the memory was sorted in-place, in the task data description that maps to the data specification. This in-place sorting at the leaf level is necessary since the whole data specification might actually be *so* small that it falls below the threshold defined in `isLeaf`. As a consequence, neither `recursiveSplit` nor `recursiveMerge` is ever called and the result of the sorting should be available directly in the data specification as per the contract of the sort function.

In sequential implementations this kind of reasoning is simpler since all the data is available at all time. In the parallel implementation, the data is distributed across threads and we explicitly want to *avoid* communication, to the point that it is forbidden outside of the well-defined paths of the D&C schema.

⁂

The framework's architecture was geared towards extensibility. This was particularly good to try out and benchmark different techniques. For everyday usage, some configurations may be completely unnecessary, e. g. there's no good reason ever to use a static load balancing for algorithms that aren't totally predictable, such as D&C.

13.2 Suitability of OpenMP

OpenMP has allowed some very fast prototyping when implementing the framework, thus allowing to try out ideas with throwaway code. Essentially, OpenMP makes caring about threads trivial. This comes at the price of a reduced control over the threads. This is fine in general: we can control threads via the built-in schedules and some tweaking parameters.

For greater control, OpenMP 3 offers offers the `task` construct. This would come in handy in the implementation of our parallel D&C implementation. Since we couldn't rely on OpenMP 3 we had to work around the lack of this feature in prior versions.

All in all however, using OpenMP was certainly much easier than using either raw threads or an existing threading wrapper for C++, mainly due to the automatically managed thread pool. In a next step, it is worth considering whether this automatic control is step for step transferred to a SeqAn specific infrastructure.

This would allow implementing a centralised control for threads that encompass both this framework as well as code written (in SeqAn) outside of this framework. At the moment, this is difficult to impossible because we can only exert as much control as

OpenMP grants us. On the other hand, this could potentially make integration with other C++ libraries harder, which would of course not use the SeqAn threading control mechanism.

13.3 Algorithms

Some constraints apply to the algorithm adaptors we have implemented using the parallel framework. We will discuss these restrictions in the following.

When using the parallel pattern **finder**, long patterns may not be found under some circumstances. Specifically, since the finder uses a blocked decomposition, we are constrained by the size of the blocks. If a pattern is larger than the size of a block, it won't be found even if it occurs in the reference sequence.

It is the caller's responsibility to test this because there is no good way for the parallel find algorithm to perform the check. It *could* test this in the `Process` method but this would mean redundant work which contracts both C++'s philosophy of "you only pay for what you use" and SeqAn's philosophy of avoiding redundant checks. We could have added an additional method to the contract of the `Algorithm` class which does a contract checking before invoking the algorithm. But this would have bloated the interface and it is not clear whether this method has general applicability and how it would best report errors (since SeqAn doesn't use exceptions). We might need to revisit this decision.

Our **unstable sort** implementation uses quicksort. However, as we have seen modern C++ libraries use introsort instead to avoid quicksort's worst-case. We have used quicksort to keep the implementation simple and in fact we have succeeded in avoiding a common pathological worst-case – pre-sorted arrays – by a smart choice of the pivot. Nevertheless, an introsort implementation would be beneficial.

Another constraint of our quicksort implementation is the very first partition step which runs sequentially on a single thread in our implementation. In fact however, there is a well-known parallelisation available for the partition routine and we should take advantage of it. Unfortunately, this currently breaks the scope of this framework since the parallelisation of the partition has a different data dependency.

In our preliminary tests using the parallel partition that is part of the GNU parallel extension we have not been able to observe a substantial speed-up compared to the sequential partition. But the comparison is problematic since the GNU parallel extension doesn't implement the extremely efficient three-ways partition that we have used in the rest of the algorithm. It would be interesting to see how a parallel three-ways partition performs.

The **stable sort** implementation also has an issue since it always uses additional memory. For very large arrays (where parallel processing makes the most sense) this is prohibitive. C++ standard library implementations automatically switch to the in-place variant of merge sort when they fail to allocated enough temporary memory. A solid parallel implementation should do the same, or even rely exclusively on the in-place variant.

Conclusion & Outlook

XIV

14.1 Conclusion

The parallelisation of existing algorithms has showcased the framework's usefulness in two ways:

For a lot of algorithms (those without data dependencies) it is actually possible to run the algorithm in parallel with *no* change to the algorithm and only minimal additional code. Furthermore, the parallelisation scales well over a lot of processors and has a small overhead, so that its application is beneficial even with few processors: it is immediately useful on today's machines.

On the other hand, once an algorithm has data dependencies, we actually need to rewrite the algorithm. Still, we could show that our framework makes this rewrite trivial because all that needs to be done is the distribution of the code into separate methods, which is trivial *if* the code is clearly written. Furthermore, local variables used by the code might need to be transferred to the thread-local storage but this, too, will be trivial in most cases.

In summary, we could show that the framework supports an efficient, effortless parallelisation of existing code. In that sense, it has achieved its goal. Unfortunately, not every part of the algorithm is really working as expected. In particular, the parallelisation for D&C with non-trivial merge remains unfinished because we were unable to determine and eliminate all the causes for its slowdown.

14.2 Improvements to the framework

The framework still contains a crucial design error: the data dependency is modelled as part of the `DataSpec`– we were mislead by the fact that it presumably describes the *data*, which would mean that it rightly belongs with the data specification. In reality, the data dependencies are a property of the *algorithm* and do not describe the data. Thus, they should make the `Dependency` trait part of the `Algorithm` class instead. This would

also simplify the API since there are currently methods whose arguments are `Algorithm` and `TaskData` instances but which logically belong to the `Dependency` trait (e. g. the `isLeaf` method which works on an `Algorithm` but is only present for `DivideAndConquer` dependencies).

14.3 Improvements to work-stealing

The work-stealing schedules on blocked independent data rely on a chunk size, which is the number of independent tasks that is stolen from the work queue. At the moment, this number is fixed in the code but the performance of the work-stealing is very sensitive to processing time of individual tasks. Consequently, the shorter individual tasks are, the bigger should the chunk size be.

An improvement could modify the chunk size dynamically at run-time. Two fundamental ways exist to guide this change.

1. Using a predetermined function that starts off with a (very) large chunk size and decreases in size as the algorithm progresses to allow fine-grained work-stealing in the last phase of the algorithm when most of work has been finished. This would be very similar to the "guided" schedule of OpenMP which also uses an adaptive chunk size.

2. Alternatively, the scheduler could benchmark the first (or even each) iteration and adapt the chunk size of the next iteration according to the benchmark. This would require determining a good ratio between chunk size and execution time of an individual task.

Some research has already been done to improve the work-stealing schedule using runtime adaptation. This work includes, but is not limited to Tzannes *& al.* [2010]; Wang *& al.* [2010]; Agrawal *& al.* [2008].

14.4 Outlook

Despite the fact that there is still some work to do, this is not just a proof of concept. The framework *is* usable and most importantly for the immediate usage, most of the algorithms that have been parallelised during this thesis can be directly used in code and will provide an immediate benefit on multiple processors.

Furthermore, the code is well tested and we expect it to be robust and reliable. A look into the code base will reveal a list of "to do" marks. These should by no means taken to mean that the code cannot be used.

The code can be found in the SeqAn svn repository under the directory `seqan/projects/benchmarks/parallel`. This thesis can also be found online at http://madr.at/msc.

Part IV

Appendix

Supplementary material, algorithm pseudo-codes and bibliography.

Algorithms

A

A.1 Pattern matching

A.1.1 Boyer-Moore-Horspool algorithm

The Boyer-Moore-Horspool algorithm [Horspool, 1980] (simply called "Horspool" by Navarro & Raffinot [2002]) is a simplification of the Boyer-Moore algorithm [Boyer & Moore, 1977]. Like the latter, it is a suffix-based approach – that is, we start comparing increasingly large suffixes of the pattern with a given position of the text. The algorithm is based on the observation that certain character comparisons in a string search are unnecessary, if we have already gained the information that the current character of the text cannot occur in the pattern at certain positions. The pattern is pre-processed to build a table of the first position (from the end of the pattern) at which each character can occur.

A.1.2 Bit-parallel algorithms

The bit-parallel algorithms are a class of algorithms that use efficient bit operations on bit vectors (machine words) to perform multiple calculations in parallel, thus improving the algorithms' overall performance significantly. The following variant is due to Baeza-Yates & Gonnet [1989] and is also known as "shift-or". This is a variation of the closely related "shift-and" variant which we will ignore here since the following is a slight improvement without altering the concept significantly.

Preprocessing
1 **for** $\sigma \in \Sigma$ **do**
2 $d[\sigma] \leftarrow m$
3 **for** $i \in [0 \ldots m-1[$ **do**
4 $d[p_i] \leftarrow m - i - 1$

Matching
5 $pos \leftarrow 0$
6 **while** $pos < n - m + 1$ **do**
7 $j \leftarrow m$
8 **while** $j > 0$ and $p_{j-1} = t_{pos+j-1}$ **do**
9 $j \leftarrow j - 1$
10 **if** $j = 0$ **then**
11 Report hit at position pos
12 $pos \leftarrow pos + d[t_{pos+m-1}]$

Algorithm A.1: Boyer-Moore-Horspool string matching

Preprocessing
1 **for** $\sigma \in \Sigma$ **do**
2 $B[\sigma] \leftarrow -1$
3 **for** $j \in [0 \ldots m[$ **do**
4 $B[p_j] \leftarrow B[p_j]$ & $\sim(1 \ll j)$

Matching
5 $D \leftarrow -1$
6 **for** $pos \in [0 \ldots n[$ **do**
7 $D \leftarrow (D \ll 1) \mid B[t_{pos}]$
8 **if** D & $(1 \ll (m-1)) = 0$ **then**
9 Report hit at position $pos - m + 1$

Algorithm A.2: Shift-or algorithm

A.2 Sorting

A.2.1 Quicksort

The following quicksort algorithm is directly adapted from Bentley & McIlroy [1993]. It uses a pseudo-median of nine (the median of three evenly spaced medians of three) to find a suitable pivot element, and the split-end partitioning algorithm presented in that paper which itself is an improvement of the original three-ways partition, also known as the Dutch flag sort [Dijkstra, 1976]. The partition maintains an interesting non-trivial invariant which Bentley & McIlroy nicely explain.

It should be noted that a randomised pivot could also be used here; Bentley & McIlroy decided against it solely on the grounds that "a library sort has no business side-effecting the random number generator." This refers to the standard C random generator which uses a global state. The argument doesn't apply to SeqAn which provides a strong random number generator that encapsulates the state into class instances and thus doesn't suffer from this problem.

```
1  procedure Quicksort(A = [a_begin ... a_end[)
2      if |A| < 2 then return

       All the following variables define indices to elements in A
3      m ← begin + |A| /2
4      s ← |A| /8

       Calculate pseudo-median of 9 as pivot
5      lower ← MedianOf3(A, begin, begin + s, begin + 2s)
6      mid ← MedianOf3(A, m, m − s, m + s)
7      higher ← MedianOf3(A, end − 1 − 2s, end − 1 − s, end − 1)
8      pivot ← MedianOf3(A, lower, mid, higher)
9      less, more ← Partition(A, pivot)
10     Quicksort([a_begin ... a_less[)
11     Quicksort([a_more ... a_end[)
```

Algorithm A.3: Quicksort

A.2.2 Merge sort

We implemented the basic out-of-place two-ways merge sort found e.g. in Knuth [1998]. The only interesting bit about this algorithm was how to prevent the copy in line 6. Due to the fact that each level in the recursion tree spawns its own thread-local states, the storage locations of the source and the target of the merge operation are always distinct.

```
1 function MedianOf3(A, i, j, k)
2     if a_i < a_j then
3         if a_j < a_k then return j
4         else if a_i < a_k then return k
5         else return i
6     else
7         if a_k < a_j then return j
8         else if a_k < a_i then return k
9         else return i
```

Algorithm A.4: Median of three

Only one copy needs to be made, on the topmost level, when the thread-local temporary storage is copied back into the original data location.

```
1  function Partition(X = [x_begin ... x_end[, pivot)
2      Swap(x_begin, x_pivot)
3      a ← begin + 1 , b ← begin + 1
4      c ← end − 1 , d ← end − 1

5      while not done do
6          while b ≤ c and x_b ≤ x_pivot do
7              if x_b = x_pivot then
8                  Swap(x_a, x_b) , a ← a + 1
9              b ← b + 1
10         while c ≥ b and x_c ≥ x_pivot do
11             if x_c = x_pivot then
12                 Swap(x_d, x_c) , d ← d − 1
13             c ← c − 1
14         if b > c then break
15         Swap(x_b, x_c) , b ← b + 1 , c ← c − 1

       Swap elements from both ends to middle
16     s ← Min(a − begin, b − a)
17     l ← begin , h ← b − s
18     while s > 0 do
19         Swap(x_l, x_h)
20         l ← l + 1 , h ← h + 1 , s ← s − 1
21     s ← Min(d − c, end − 1 − d)
22     l ← b , h ← end − s
23     while s > 0 do
24         Swap(x_l, x_h)
25         l ← l + 1 , h ← h + 1 , s ← s − 1
26     return begin + (b − a), end − (d − c)
```

Algorithm A.5: Split-end partition

```
1  procedure MergeSort(A = [a_begin ... a_end[)
2      if |A| < 2 then return
3      m ← begin + |A| /2
4      MergeSort([a_begin ... a_m[)
5      MergeSort([a_m ... a_end[)
6      A ← Merge([a_begin ... a_m[, [a_m ... a_end[)
```

Algorithm A.6: Merge sort

```
1  function MERGE(X = [x_a ... x_b[, Y = [y_c ... y_d[)
2      Z ← [z_0 ... z_{n−1}[ an array of size |Z| = |X| + |Y|
3      i ← a , j ← c , k ← 0

4      while not done do
5          if a_i ≤ b_j then
6              z_k ← a_i , k ← k + 1 , i ← i + 1
7              if i = b then
8                  while j ≠ d do z_k ← y_j , k ← k + 1 , j ← j + 1
9                  break
10         else
11             z_k ← b_j , k ← k + 1 , j ← j + 1
12             if j = d then
13                 while i ≠ b do z_k ← a_i , k ← k + 1 , i ← i + 1
14                 break
15     return Z
```

Algorithm A.7: Two-ways merge

Techniques & Methods

B

B.1 Software engineering

The content of this section doesn't pertain directly to the results of this thesis. It is nevertheless included because we believe that good engineering is an important factor in every software endeavour, and that adhering to the principles listed below has improved the quality (both in terms of performance as well as reliability) of the framework.

We tried to apply software engineering best practices throughout. In particular, this includes:

- Fact checking in the form of assertions in the code.
- Extensive (unit) testing. That way, we ensure that the tests are actually run and test the correct functionality, rather than greenlighting on a whim.
- Code was compiled using the strictest error level provided by the compiler, and warnings were treated as errors.
- Code coverage to ensure that tests cover all code paths. This was particularly important to ensure that the correct template specialisations got used.
- Checks for memory leaks using Valgrind [Valgrind Developers, 2010].
- Profiling to identify performance bottlenecks in the code using Shark [Apple Inc., 2011].
- The SeqAn documentation system was used to provide a detailed documentation of the framework's API
- Manual memory management was eschewed in favour of C++' automatic memory management mechanisms.

Assertions and unit testing as well as code coverage reporting were implemented using SeqAn's testing framework.[*] Where it made sense, tests were written *before* the actual code and a test-driven methodology was used. Combined with assertions, this makes thinking about edge cases of the algorithms very explicit which aids debuggability as well as adding to the documentation: well-written test cases *document* algorithm's edge cases.

Furthermore, the unit tests include large-scale tests on random input data to improve confidence in the implementation by the sheer volume of testing. This has turned out to be useful to discover some subtle race conditions in the parallel code that occur only very rarely, and turn up only a once in a few hundred test runs.

All in all, there are almost 300 unit tests in this project. Many of those tests are completely identical, and merely run on different configurations of algorithm, scheduler and data. To avoid having to write all these tests, the test suite makes heavy use of templates to generate the code necessary for those tests.

Compiling with warnings treated as errors also helped to keep the code clean since most warnings in the GCC compiler actually turn out to be legitimate and the very few that aren't can be circumvented easily.

The framework was developed using the GNU tool chain so GDB [Free Software Foundation, 2010] was the natural choice for the debugger software.

[*] An introduction can be found at trac.mi.fu-berlin.de/seqan/wiki/HowTo/WriteTests.

Bibliography

C

> “Nanos gigantium humeris insidentes
>
> *(Bernardus Carnotensis)*

The 1000 Genomes Project Consortium. *A map of human genome variation from population-scale sequencing.* Nature, 10 2010. **volume 467 (7319)**; pp. 1061–1073.
http://dx.doi.org/10.1038/nature09534

Kunal Agrawal, Charles E. Leiserson, Yuxiong He *&* *al.* *Adaptive work-stealing with parallelism feedback.* ACM Transactions on Computer Systems, Sep 2008. **volume 26**; pp. 7:1–7:32.
http://doi.acm.org/10.1145/1394441.1394443

Christopher Alexander, Sara Ishikawa *&* Murray Silverstein. *A Pattern Language.* Oxford University Press, 1977.

Gene M. Amdahl. *Validity of the single processor approach to achieving large scale computing capabilities.* In *Proceedings of the April 18–20, 1967, spring joint computer conference*, AFIPS ’67 (Spring). ACM, New York, NY, USA, 1967, pp. 483–485.
http://doi.acm.org/10.1145/1465482.1465560

The ANSI X3J13 committee. *Common Lisp: ANSI INCITS 226–1994 (R2004)*, 2004.

Apache Hadoop project members. *Hadoop MapReduce.* Software, Aug 2010. Release 0.21.0.
http://hadoop.apache.org/mapreduce/

Apple Inc. *Shark.* Software, 2011.

Nimar S. Arora, Robert D. Blumofe *&* C. Greg Plaxton. *Thread Scheduling for Multiprogrammed Multiprocessors.* In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures.* Puerto Vallarta, Mexico, 1998.

Ricardo A. Baeza-Yates *&* Gaston H. Gonnet. *A new approach to text searching.* In N. J. Belkin *&* C. J. van Rijsbergen, editors, *Proceedings of the 12th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR ’89. ACM,

New York, USA, 1989, pp. 168–175.
http://doi.acm.org/10.1145/75334.75352

Blaise Barney. *Introduction to Parallel Computing*, Oct 2010. Retrieved on 2010–11–18.
https://computing.llnl.gov/tutorials/parallel_comp/

Jon L. Bentley & M. Douglas McIlroy. *Engineering a sort function.* Software – Practice & Experience, Nov 1993. **volume 23**; pp. 1249–1265. ISSN 0038-0644.
http://portal.acm.org/citation.cfm?id=172704.172710

Guy E. Blelloch. *Synthesis of Parallel Algorithms*, chapter Prefix Sums And Their Applications. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993, 1st edition.

Robert D. Blumofe & Charles E. Leiserson. *Scheduling Multithreaded Computations by Work Stealing.* Journal of the ACM, Sep 1999. **volume 46 (5)**; pp. 720–748.

Robert S. Boyer & J Strother Moore. *A fast string searching algorithm.* Communications of the ACM, Oct 1977. **volume 20**; pp. 762–772. ISSN 0001-0782.
http://doi.acm.org/10.1145/359842.359859

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & *al. Introduction to Algorithms*, chapter 2.3.1. MIT Press, Cambridge, Massachusetts, 2005, 2nd edition, p. 28.

Jeffrey Dean & Sanjay Ghemawat. *MapReduce: simplified data processing on large clusters.* In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, volume 6. USENIX Association, Berkeley, CA, USA, 2004, p. 10.
http://labs.google.com/papers/mapreduce.html

E. W. Dijkstra. *A discipline of programming*, volume 1. Prentice-Hall, Englewood Cliffs, NJ, 1976.

Peter Dimov. *Boost Smart Pointers.* Software, Jan 2002.
http://www.boost.org/libs/smart_ptr

Andreas Döring, David Weese, Tobias Rausch & *al. SeqAn – An efficient, generic C++ library for sequence analysis.* BMC Bioinformatics, 2008. **volume 9**; p. 11.
http://dx.doi.org/10.1186/1471-2105-9-11

Manek Dubash. *Moore's Law is dead, says Gordon Moore.* Techworld News, Apr 2005. Retrieved on 2010–11–21.
http://news.techworld.com/operating-systems/3477/moores-law-is-dead-says-gordon-moore/

Richard Feynman. *The Computing Machines in the Future.* In *Nishina Memorial Lectures*, volume 746 of *Lecture Notes in Physics*. Springer Berlin / Heidelberg, 1985, pp. 99–114.
http://dx.doi.org/10.1007/978-4-431-77056-5_6

Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley, 1995.

The Free Software Foundation. *GDB: The GNU Project Debugger.* Software, 2010.
http://www.gnu.org/software/gdb/

Erich Gamma, Richard Helm, Ralph Johnson *&* *al. Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

Andreas Gogol-Döring *&* Knut Reinert. *Biological Sequence Analysis Using the SeqAn C++ Library.* CRC Press, 2009.

John L. Gustafson. *Reevaluating Amdahl's law.* Communications of the ACM, May 1988. **volume 31**; pp. 532–533.
http://doi.acm.org/10.1145/42411.42415

C. A. R. Hoare. *Quicksort.* The Computer Journal, 1962. **volume 5 (1)**; pp. 10–16.
http://comjnl.oxfordjournals.org/content/5/1/10.abstract

Jared Hoberock *&* Nathan Bell. *Thrust: A Parallel Template Library.* Software, 2010. Version 1.3.0.
http://www.meganewtons.com/

R. Nigel Horspool. *Practical fast searching in strings.* Software - Practice *&* Experience, Jun 1980. **volume 10 (6)**; pp. 501–506.
http://dx.doi.org/10.1002/spe.4380100608

Intel Corporation. *Excerpts from* A Conversation with Gordon Moore: Moore's Law. Video, 2005. Retrieved on 2010–11–16.
http://www.intel.com/pressroom/kits/events/moores_law_40th/

Khronos Group. *OpenCL.* Software, 2010. Release 1.1; retrieved on 2011–01–05.
http://www.khronos.org/opencl/

Donald Ervin Knuth. *The Art of Computer Programming*, volume 3 – Sorting and Searching, chapter 5.2.4. Sorting by merging. Addison-Wesley, 1998, 2nd edition.

Vipin Kumar, Ananth Grama, Anshul Gupta *&* *al. Introduction to parallel computing: design and analysis of algorithms.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994. ISBN 0-8053-3170-0.

Влади́мир И. Левенштейн (Levenshtein). *Двоичные коды с исправлением выпадений, вставок и замещений символов.* In *Доклады Академий Наук СССР*, volume 163. 1965, pp. 845–848.

Berna L. Massingill, Timothy G. Mattson & Beverly A. Sanders. *Patterns for Finding Concurrency for Parallel Application Programs.* In *Proceedings of the Seventh Pattern Languages of Programs Workshop.* 2000.

Scott Meyers. *Effective STL: 50 specific ways to improve your use of the standard template library*, chapter 46. Addison-Wesley, Indianapolis, IN, USA, 2001, 3rd edition.

David R. Musser. *Introspective Sorting and Selection Algorithms.* Software: Practice and Experience, 1997. **volume 27 (8)**; pp. 983–993. ISSN 1097-024X.
http://dx.doi.org/10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-#

Stefan Näher & Daniel Schmitt. *Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, chapter Multi-core Implementations of Geometric Algorithms. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-03455-8, pp. 261–274.
http://portal.acm.org/citation.cfm?id=1611840.1611862

Gonzalo Navarro & Mathieu Raffinot. *Flexible Pattern Matching in Strings – Practical Online Search Algorithms for Texts and Biological Sequences.* Cambridge University Press, Cambridge, United Kingdom, 2002.

Nvidia. *CUDA.* Software, Nov 2010. Release 3.2; retrieved on 2011–01–05.
http://www.nvidia.com/object/cuda_home_new.html

The Open Group. *IEEE 1003.1c-1995: Portable Operating System Interface (POSIX®) – System Application Program Interface (API) Amendment 2: Threads Extension (C Language).* Technical report, IEEE Standards Association, 2004.

OpenMP Architecture Review Board. *OpenMP Application Programming Interface.* Technical report, OpenMP Architecture Review Board, May 2008. Version 3.0.
http://www.openmp.org/

Victor Pankratius, Walter F. Tichy & Frank Otto. *Softwareentwicklung für moderne, parallele Plattformen.* Lecture slides, 2009. Retrieved on 2010–11–18.
http://www.ipd.uka.de/Tichy/teaching.php?id=157

Participants of the MPI Forum. *MPI: A Message-Passing Interface Standard.* Technical report, University of Tennessee, Sep 2009.
http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

Konrad Ludwig Moritz Rudolph. *Implementation of a Read Mapping Tool Based on the Pigeon-hole Principle.* Bachelor thesis, Freie Universität Berlin, Aug 2008.

Carl Sagan. *The Demon-Haunted World.* Random House, 1995.

Peter Sanders. *Parallele Algorithmen.* Lecture slides, Feb 2010. Retrieved on 2010–11–18.
http://algo2.iti.kit.edu/1399.php

Johannes Singler & Benjamin Konsik. *The GNU libstdc++ parallel mode: software engineering considerations.* In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*. ACM, New York, NY, USA, 2008, pp. 15–22.
http://doi.acm.org/10.1145/1370082.1370089

Johannes Singler, Peter Sanders & Felix Putze. *Euro-Par 2007 Parallel Processing*, chapter MCSTL: The Multi-core Standard Template Library. Springer, Berlin/Heidelberg, 2007, pp. 682–694.

The C++ Standards Committee. *ISO/ISC DTR 19769: Working Draft, Standard for Programming Language C++* . Technical report, ISO, Nov 2010.

Alexander Stepanov & Meng Lee. *The Standard Template Library.* Technical report, HP Laboratories, Oct 1995.

Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2004, 3rd edition.

Herb Sutter. *The free lunch is over: A fundamental turn toward concurrency in software.* Dr. Dobb's Journal, Mar 2005. **volume 30 (3)**.
http://www.gotw.ca/publications/concurrency-ddj.htm

Herb Sutter. *The Pillars of Concurrency.* Dr. Dobb's Journal, Jul 2007; p. 2.
http://www.drdobbs.com/high-performance-computing/200001985

Andrew S. Tanenbaum. *Structured Computer Organization.* Prentice-Hall, 1990, 3rd edition.

Alexandros Tzannes, George C. Caragea, Rajeev Barua & *al. Lazy binary-splitting: a run-time adaptive work-stealing scheduler.* In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel computing*. ACM, ACM, 2010, pp. 179–190.

The Valgrind Developers. *Valgrind.* Software, 2010.

David Vandevoorde & Nicolai M. Josuttis. *C++ Templates: The Complete Guide*, chapter 16.2 The Empty Base Class Optimization (EBCO). Addison-Wesley, 2003, 1st edition.

J. C. Venter, M. D. Adams, E. W. Myers & *al. The sequence of the human genome.* Science, Feb 2001. **volume 291 (5507)**; pp. 1304–1351.
http://dx.doi.org/10.1126/science.1058040

Lei Wang, Huimin Cui, Yuelu Duan & *al. An adaptive task creation strategy for work-stealing scheduling.* In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10. ACM, New York, NY, USA, 2010, pp. 266–277.
http://doi.acm.org/10.1145/1772954.1772992

Thomas Willhalm & Nicolae Popovici. *Putting Intel Threading Building Blocks to work.* In *Proceedings of the 1st international workshop on Multicore software engineering*, IWMSE '08. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-031-9, pp. 3–4.
http://doi.acm.org/10.1145/1370082.1370085

Anthony Williams. *Boost.Thread.* Software, 2008.
http://www.boost.org/doc/html/thread.html

Acknowledgements

A lot of people have helped in the development of this thesis. Most importantly, I would like to thank Manuel Holtgrewe who has provided many of the ideas and directions for this thesis and for his help and criticism during the implementation and subsequent evaluation, for fixing SeqAn bugs that have stymied my progression and for proofreading the thesis.

I would also like to thank Martin Riese, Paul Theodor Pyl and René Märker who have provided me with ideas and distraction when I needed it.

Thanks also go to Knut Reinert for giving me the idea of generalising the `getHistory` functionality for the blocked data decomposition, which made the adaptation of the finder algorithms substantially easier and shorter. I also want to thank my advisors, Knut Reinert and Tim Conrad, for agreeing to supervise the thesis.

Finally, I would like to thank Anja Thormann for supporting me, discussing with me, and proofreading, criticising and praising every single aspect of this thesis; and my sister Sophia for being an awesome sous-chef.

Colophon

E

This document was typeset in LaTeX using the XeTeX engine. The main text is set in Linux Libertine. Candara was used for the figure captions and the code was set in WenQuanYi Zen Hei Mono. Formulæ use Cambria Math. The types used for the ampersands are TeX Gyre Pagella and PilGi.

The thesis was written using the MacVim editor. Additional software used to produce figures and statistics includes Apple Keynote, Inkscape and Python.

The animal on the cover is an invisible dragon, *Draco invisible*. It can be found all over the world but its favourite habitat is reportedly Carl Sagan's garage. More information on this peculiar animal can be found in Sagan [1995].

The following LaTeX packages were used to produce this document: nag · scrkbase · scrbase · keyval · scrlfile · tocbasic · typearea · fixltx2e · ifXeTeX · AMSmath · amstext · amsgen · amsbsy · amsopn · xcolor · xltxtra · ifLuaTeX · fontspec · expl3 · xparse · l3names · ε-TeX · l3basics · l3expan · l3tl · l3int · l3quark · l3seq · l3toks · l3prg · l3clist · l3token · l3prop · l3msg · l3io · l3skip · l3box · l3keyval · l3keys · l3precom · l3xref · l3file · l3fp · l3luatex · calc · xkeyval · fontspec-patches · fontenc · xunicode · realscripts · metalogo · graphicx · graphics · trig · unicode-math · l3keys2e · catchfile · infwarerr · ltxcmds · ε-TeXcmds · trimspaces · fix-cm · filehook · filehook-scrlfile · ΠΟΛΥΓΛΩΣΣΙΛ (polyglossia) · etoolbox · makecmds · microtype · ifthen · amsthm · suffix · natbib · scrpage2 · xspace · listings · float · booktabs · todonotes · TikZ · PGF · pgffor · pgfrcs · everyshi · pgfcore · pgfsys · pgfcomp-version-0-65 · pgfcomp-version-1-18 · pgfkeys · marvosym · rotating · multirow · array · algorithm · algpseudocode · algorithmicx · fancyvrb · caption · caption3 · needspace · tocstyle · pgfplots · wrapfig · changepage · mdwlist · titlesec · siunitx · translator · verbatim · url · hyperref · ifpdf · pdfTeXcmds · kvsetkeys · pdfescape · ifVTeX · hycolor · xcolor-patch · letltxmacro · kvoptions · intcalc · bitset · bigintcalc · atbegshi · stringenc · rerunfilecheck · cleveref · color · nameref · refcount · gettitlestring.

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich das Dokument und die Arbeit selbständig angefertigt habe. Alle benutzten Quellen wurden angegeben. Zitate wurden stets als solche kenntlich gemacht.

Berlin, den 15. März 2010 Konrad Rudolph