THE UNIVERSITY OF
WARWICK

AUTHOR: **Peter Krusche** DEGREE: **Ph.D.**

TITLE: **Parallel String Alignments: Algorithms and Applications**

DATE OF DEPOSIT: .....

I agree that this thesis shall be available in accordance with the regulations governing the University of Warwick theses.
I agree that the summary of this thesis may be submitted for publication.
I **agree** that the thesis may be photocopied (single copies for study purposes only).
Theses with no restriction on photocopying will also be made available to the British Library for microfilming. The British Library may supply copies to individuals or libraries. subject to a statement from them that the copy is supplied for non-publishing purposes. All copies supplied by the British Library will carry the following statement:

> "Attention is drawn to the fact that the copyright of this thesis rests with its author. This copy of the thesis has been supplied on the condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the author's written consent."

AUTHOR'S SIGNATURE: .....

USER'S DECLARATION

1. I undertake not to quote or make use of any information from this thesis without making acknowledgement to the author.
2. I further undertake to allow no-one else to use this thesis while it is in my care.

DATE SIGNATURE ADDRESS

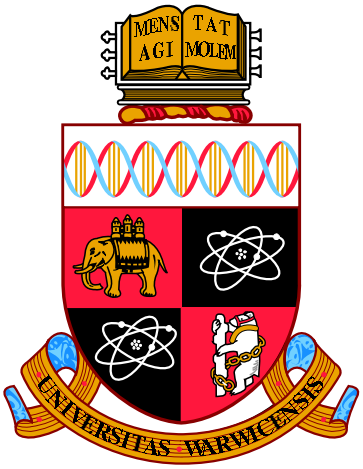
.....

.....

.....

.....

.....



Parallel String Alignments: Algorithms and Applications

by

Peter Krusche

Thesis

Submitted to The University of Warwick

for the degree of

Doctor of Philosophy

Department of Computer Science

May 2010

THE UNIVERSITY OF WARWICK

Contents

List of Tables

List of Figures

Acknowledgments

First and foremost I would like to thank my supervisor, Dr. Alexander Tiskin, who has been a constant source of inspiration, insight, new ideas, and encouragement for me throughout my time of working on this thesis. His patience and attention to detail when reading and commenting on drafts of papers, abstracts and this thesis have been invaluable. I have learned a lot from our discussions about the science of strings and seaweeds, parallel computation, and many other things. This thesis would not have been possible without him.

I am indebted to my office mates and colleagues at the Department of Computer Science not only for providing a stimulating environment for research, but also for entertainment, coffee breaks, and discussions. I am especially grateful to Haris Aziz, John Fearnley, and Michal Rutkowski for their company through a series of different offices, to Harald Räcke, Marcin Jurdziński, and Artur Czumaj for giving me the opportunity to teach seminars on algorithms and for coming up with exciting problem sheets for these, as well as to Oded Lachish for entertaining and helpful discussions. My studies were supported financially by a PhD studentship from the Department of Computer Science, as well as by travel funding from the Algorithms and Complexity Research Group at the University of Warwick.

I would also like to thank Sascha Ott and the people from the Warwick Systems Biology Centre for explaining their work and helping to adapt the algorithms from this thesis to their applications. Furthermore, I would like to thank everyone at the Centre for Scientific Computing, where most of the computational experiments of this thesis were conducted.

My time at Warwick University would probably have been shorter, but also much less enjoyable without the company of my flatmates, friends, fellow volleyball players, and too many others to mention. Special thanks goes out to Ezra Baydur (for the funk), Karishma Balani (MSD), Jan Becker (who was helpful in all sorts of situations), Gernot Herbst (who should work more), Héloïse Imbault (for procrastinating for me, saving me time), Bruno Maia (for the science of cycling), Mike Pierides (all kinds of doctors can be fun), Kristin Schulz (for being a perfect housemate), Chris Sohrmann (the master of everything gazebo), Aleksa Starovic (for lifting heavy things, twice), and Saskia Toennesmann (for the cupcakes!).

Finally, I would like to thank my family for their support and at least trying to understand what they call the "real-world implications" of my work, and Sharae for her patience and understanding, as well as for setting a good example for me.

Declarations

I hereby declare that this thesis represents my own work and to the best of my knowledge it contains no materials previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any other degree at The University of Warwick or any other educational institution, except where the acknowledgment is made in the thesis. Any contribution made to the research by others, with whom I have worked at The University of Warwick or elsewhere, is explicitly acknowledged in the thesis.

Parts of this thesis have been submitted for publication.

An initial version of Chapter 2 was accepted as an abstract for a talk at the Theory & Many-Cores 2009 Workshop (see `http://www.umiacs.umd.edu/conferences/tmc2009/`).

Material from Chapter 4 has been published as:

P. Krusche and A. Tiskin. *Efficient parallel string comparison.* In Proceedings of ParCo, vol. 38 of NIC Series, John von Neumann Institute for Computing, pp. 193–200, 2007.

P. Krusche and A. Tiskin. *Longest increasing subsequences in scalable time and memory.* In proceedings of PPAM 2009, Wroclaw, to appear. 2009

P. Krusche and A. Tiskin. *New algorithms for efficient parallel string comparison.* In proceedings of SPAA 2010, to appear. 2010

Material from Chapter 5 has been published as:

P. Krusche and A. Tiskin. *String comparison by transposition networks.* In London Algorithmics 2008: Theory and Practice, vol. 11 of Texts in Algorithmics, College Publications, 2009.

Material from Chapter 6 has been published as:

P. Krusche and A. Tiskin. *Computing Alignment Plots Efficiently.* In Proceedings of ParCo 2009. To appear. 2009

The program code from this thesis has been used to implement the EARS webservice: `http://wsbc.warwick.ac.uk/ears/main.php`, and in the corresponding paper:

E. Picot, A. Tiskin, P. Brown, P. Krusche, I. Carré, and Sascha Ott. *Evolutionary analysis of regulatory sequences (EARS) in plants*, The Plant Journal, vol. 64:1, pp. 165–176, 2010.

Abstract

String comparison is a fundamental component in many of today's applications of computing, including bio-informatics, signal processing, databases, internet search and many others. A specific type of practical string comparison problem is computing the longest common subsequence (LCS) of two input strings. The LCS problem is equivalent to computing edit distances and strongly connected to computing string alignments, which are of great importance for applications in computational biology.

In this thesis, we show new approaches to using parallel computation in string comparison, which allow us to understand algorithms for computing the LCS on the word-RAM, the PRAM and the BSP models in a unified fashion. This approach is based on recent results on algorithms for semi-local string comparison. Semi-local string comparison is a straightforward extension of the standard LCS problem, in which we ask to compute the LCS for one string and all substrings of another string. We use a revised approach to analyzing BSP algorithms which includes the analysis of communication and I/O cost, as well as of the local memory requirements. We define asymptotic scalability in memory and communication. Scalable communication captures the input/output cost of partitioning a problem into subproblems, and achieving scalable memory allows to obtain small subproblems which can fit into processor caches. In this thesis, we show how to achieve scalable memory and communication for computing longest common subsequences, as well as for computing longest increasing subsequences.

Furthermore, we present new algorithms for semi-local string comparison that are parameterized by the similarity of the input strings, and we discuss aspects of their practical implementation. Alongside with theoretical results, this thesis also comprises an algorithm engineering project, which has the goal of allowing practical applications to make use of the techniques of semi-local string comparison. For a particular problem of local string comparison in evolutionary biology, our methods have improved upon the fastest existing methods. We have obtained a speedup by a factor of more than 14 over the fastest existing implementation, running on a single processor, and achieved the potential to scale to hundreds of processors. This has greatly increased the feasible size of the input sequences that can be compared using our method, potentially allowing loss-free local comparison of entire genomes.

Abbreviations

BSP : Bulk-synchronous parallel(-ism)

BSMP : Bulk-synchronous message passing

CPU : Central processing unit

GPU : Graphics processing unit

DRMA : Direct remote memory access

Flop : Floating point operation

LCS : Longest common subsequence

LIS : Longest increasing subsequence

LLCS : Length of the longest common subsequence

MPI : Message-passing interface

PUB : Paderborn University BSP-Library

PRAM : Parallel random access machine

RAM : Random access machine

SIMD : Single instruction, multiple data

SMP : Symmetric multiprocessing

SPMD : Single program multiple data

Chapter 1

Introduction

String comparison is a fundamental component in many of today's applications of computing, including bio-informatics, signal processing, databases, internet search and many others. A specific type of practical string comparison problem is computing the longest common subsequence (LCS) of two input strings. The LCS problem is equivalent to computing edit distances and strongly connected to computing string alignments, which are of great importance for applications in computational biology. String alignments give a natural way to compare genetic sequences. Levenshtein [1966] was first to formalize edit-distance problems, Needleman and Wunsch [1970] gave first algorithms for string alignment, and Wagner and Fischer [1974] gave the first efficient dynamic programming algorithm for the LCS problem. Gusfield [1997] gives an introduction to string alignment algorithms and biological applications.

In general, longest common subsequences are not difficult to compute, standard algorithms compute the LCS of two strings of length n in time $O(n^2)$. The first algorithm running in quadratic time and space was shown by Wagner and Fischer [1974], and soon after this, a linear space algorithm was given by Hirschberg [1975]. Especially when comparing large sequences, it is important to be able to perform LCS computations efficiently and in short time. Therefore, faster methods for computing the LCS and related measures of string similarity have been researched extensively since these first algorithms were shown.

For the general LCS problem, a lower bound of $\Omega(n^2)$ for the decision tree model of computation was shown by Aho et al. [1976]. Approaches for obtaining practical speedup over the standard dynamic programming algorithms have therefore fallen into two categories. Hunt and Szymanski [1977] have shown the first algorithm that is parameterized by the similarity of the input strings, allowing running times of $o(n^2)$ for highly similar input strings, but $\Omega(n^2)$ in the worst case. Apostolico and Guerra [1987] gave an improved algorithm which and takes time $O(n^2)$ in the worst case, and runs faster for highly similar input strings.

The other approach to faster LCS computation has been to consider more practical computational models, including different types of parallel computation. Masek and Paterson [1980] showed that an asymptotic running time of $O(n^2/\log n)$ can be achieved by using small block precomputation (Arlazarov et al. [1970] applied this method first in an algorithm for boolean matrix multiplication). Their approach is based on precomputing the updates to the dynamic programming table for blocks of size $\log n \times \log n$, using the fact that the number of types of such blocks is $o(2^{\log^2 n})$. This method is related to machine word parallelism, in which operations are carried out in parallel on multiple values encoded in a fixed-size machine word, using standard arithmetic operations. Recently, the word-RAM model has established itself as an approach to model such word-level parallelism. Lower bounds for the RAM model do not necessarily apply to the word-RAM, e.g. sorting can be done in $o(n \log n)$ time (see Hagerup [1998] for further references and discussion). Although different from the RAM model of computation (Aho et al. [1974]), the word-RAM is a practical model for computation since it captures a common feature in almost all modern processors: the ability to carry out complex operations on a single machine word in a constant number of machine cycles. The word-RAM approach is also related to traditional vector parallelism, in which single machine instructions are allowed to perform the same operation on multiple items of data at the same time. First models for this type of parallelism were proposed e.g. by Pratt et al. [1974] and studied in detail by Blelloch [1990]. In string comparison algorithms, this type of parallelism has been exploited e.g. for obtaining bit-parallel

algorithms (see Allison and Dix [1986]; Crochemore et al. [2001]; Hyyrö [2004]). Another example are the subsequence matching algorithms by Boasson et al. [2001], who use a customized word-RAM-style model named the MPRAM. Another subquadratic sequence alignment algorithm was shown by Crochemore et al. [2002], who use Lempel-Ziv compression (see Lempel and Ziv [1976]) to speed up the computation parameterized by the entropy of the input strings.

Another practical approach to obtain faster LCS algorithms is by using parallel computation on a fixed or arbitrary number of processors. Classical theoretical models for parallel computation are the PRAM model (Fortune and Wyllie [1978]; Goldschlager [1982]), and the BSP model (Valiant [1990]). Apostolico et al. [1990] showed a PRAM algorithm for LCS computation, and McColl [1995] showed a simple method to obtain a BSP algorithm for dynamic programming which is directly applicable to LCS computation.

In this thesis, we show new approaches to parallel string comparison, which allow us to understand algorithms for computing the LCS on the word-RAM, the PRAM, and the BSP model in a unified fashion. This approach is based on new algorithms for semi-local string comparison by Tiskin [2010a]. Semi-local string comparison is a straightforward extension of the standard LCS problem, in which we ask to compute the LCS for one string and all substrings of another string. Although this problem has a wide range of other algorithmic applications (Tiskin [2006, 2010a]), we will restrict our attention to its applications in string comparison. In this work, we show parallel algorithms for semi-local string comparison, discuss aspects of their practical implementation, and show how to apply these new methods to a problem in computational biology. Alongside with theoretical results, this thesis also comprises an algorithm engineering project, which has the goal of enabling practical applications to make use of the techniques of semi-local string comparison. We will show one particular application in evolutionary biology, for which our methods have improved upon the fastest existing methods by a factor of over 14, running on a single processor. Furthermore, our implementation has the potential of using parallel computation on hundreds of processors. This has greatly

increased the feasible size of the input sequences which can be compared using our method, potentially allowing loss-free local comparison of entire genomes.

This thesis is structured as follows. In Chapter 2, we show a simple theoretical model for parallelism which forms the basis of our algorithmic analysis, and we discuss the practicality of this model for algorithm implementation. Our model aims to capture the main features of modern parallel computer systems, which includes vector or word-parallelism and data locality, as well as all aspects of an algorithm's scalability. In particular, we analyse the scalability of computation, communication and memory separately. This allows us to study algorithms using a reasonably simple theoretical model, and still ensures that these algorithms can be implemented efficiently. After this, we introduce the basic concepts of semi-local string comparison in Chapter 3. In Chapter 4, we discuss new scalable parallel algorithms for semi-local string comparison. The main new feature of these algorithms is their scalability in communication, and lower synchronisation cost compared to existing algorithms. The core ingredient of these algorithms are new parallel methods for distance multiplication of a certain subclass of Monge matrices. We discuss the practicality of these new algorithms for two applications in string comparison: parallel LCS, and longest increasing subsequence (LIS) computation. Furthermore, these new methods could be applied as a plugin for other algorithmic applications as well. After this, we discuss parameterized algorithms for semi-local string comparison in Chapter 5. We show how traditional parameterized algorithms for the LCS problem can be derived using semi-local string comparison, and we show how algorithms for semi-local string comparison can be parameterized, allowing speedup for very similar or very dissimilar input strings. Finally, in Chapter 6, we show a practical application of our semi-local string comparison algorithms: we implement a very sensitive method for local comparison of two strings. This method has been successfully applied in a biological application by Ott et al. [2009]. Our implementation is now used to provide a web service for evolutionary sequence comparison (Picot et al. [2010b]). We conclude in Chapter 7 by summarizing the results from this thesis and highlighting some directions of future work. The appendix contains a more detailed

description of the algorithm engineering work in this thesis. Appendix A describes a highly optimized parallel vector library for integer vector computations, which has been used to implement and speed up various string algorithms. Appendix B describes our approach to portable coarse-grained parallel programming. Appendix C contains a description of the programs for biological sequence comparison.

Chapter 2

Engineering Parallel Algorithms

In this chapter, we introduce various existing models of parallel computation, and describe the model we will use to study parallel algorithms in this thesis. We adapt the standard bulk-synchronous parallel (BSP) model of computation to include elements of word-parallel computation, as well as from external memory algorithms. Based on this model, we define the concepts of work-optimality, scalable communication and scalable memory, and explain how these concepts relate to practical scalability on modern parallel computers.

2.1 Models of parallel computation

String comparison algorithms can benefit naturally from using parallel computation since their practical applications typically require large input strings to be processed. Furthermore, recent developments in the design of computers include both the potential and the need for parallelism in algorithms: multi-core CPU's, SIMD instruction sets, and highly-parallel graphics processors are commonly available in almost every desktop computer. In new processor designs, the traditional increase in processor clock speed has largely been replaced by an increase of the number of parallel execution units which are available to the programmer. Exploiting parallelism is becoming the only way to truly benefit from these developments and gain

performance by using new processor architectures. In algorithm design, this poses two questions: how well the traditional models of parallel programming are suited to new parallel computers, and which problems require new parallel algorithms. In this chapter, we relate the main classical models of parallel computation to recent developments in parallel computers and highlight the aspects in parallel algorithms which we look at in this work. We then give a brief overview of modern tools for implementing such algorithms. For a more detailed technical discussion, the reader may refer to Appendices A and B, as well as the related program source codes.

A multitude of theoretical models and paradigms for parallel computation have been proposed to close the gap between algorithmic results and real parallel computers. Most common are variants of the PRAM model (see e.g. Fortune and Wyllie [1978]; JáJá [1992]). While the PRAM model allows studying the complexity of a problem in an idealised parallel environment, it does not capture many practical aspects of parallel programming, such as the fact that communication between processors can be more expensive than sequential computation. Despite this limitation, the model is useful for studying the degree of parallelism that can be achieved for a given problem. Moreover, on a small scale, it has been shown possible to simulate PRAM-type computations in practice (Wen and Vishkin [2008]; Paul et al. [2002]).

To account for communication cost and data locality in algorithm design, models such as BSP (Valiant [1990]) and a number of shared memory parallel models (Ramachandran [1997]; Tiskin [1998]) have been proposed. The key idea of most of these models is to separate communication from computation and account for the costs separately. Depending on the specific design of a parallel machine, communication cost can then be modelled accurately using a purpose-designed model (Juurlink and Wijshoff [1996]; Blelloch et al. [1997]), since the linear approximation for communication performance given by the standard BSP model does not give accurate performance predictions when comparing to practical results in many cases (Hill et al. [1998]; Krusche [2005]). However, we would like to point out that a good model for algorithm design does not necessarily have to come equipped with an accurate method of predicting performance on any parallel machine. Our approach

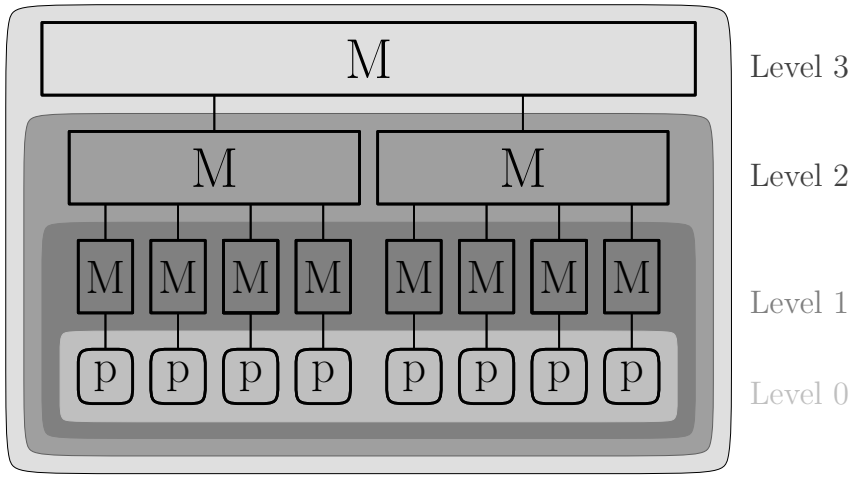


Figure 2.1: A processor-memory hierarchy that can be modelled by Multi-BSP

here is to capture the cost of communication separately, allowing us to predict performance using a more elaborate model if necessary, but mainly focusing our study on how the amount of communication can be decreased asymptotically.

More recent models have described hierarchical parallel computers (de la Torre and Kruskal [1996]), and the concept of network-obliviousness (Bilardi et al. [2007]), which is similar to the cache-oblivious sequential algorithms introduced by Frigo et al. [1999]. This type of model focuses on the question of how well a problem can be decomposed recursively and run on a parallel computer with a hierarchy of caches or buffers. For recent multi-core architectures, similar work has focused on modelling cache performance and deriving generic ways to obtain cache efficient algorithms (Blelloch et al. [2008]) on multi-core processors. Valiant [2008] recently defined an extension of BSP named Multi-BSP to serve as a computational model of multi-core and SMP processors with a cache/memory hierarchy. This last contribution is particularly interesting for the theoretical study of parallel algorithms, since it includes a detailed discussion of optimality and scalability both for work and communication. Furthermore, the model describes both fine-grained PRAM-style parallelism, and coarse-grained parallelism on a hierarchical parallel machine (see Figure 2.1 for an example of a processor-memory hierarchy that can be modelled by the Multi-BSP model).

Another traditional approach to modelling data-parallel computations is vector parallelism. Efficient implementation of vector-parallel algorithms using multiple cores is usually straightforward. Moreover, many modern multi-core processors are capable of executing SIMD vector instructions independently on each core. Specifying vector parallelism explicitly is advantageous because it provides a simple mechanism for speeding up the "bottom level" of a parallel computation, and allows one to make use of automatic vectorisation features in compilers more easily. A simple approach to modelling vector parallelism is extending the standard RAM model (Aho et al. [1974]) by adding bit shift and bitwise Boolean instructions, and allowing one to store arbitrarily large numbers in memory cells (Pratt et al. [1974]; Boasson et al. [2001]). Other approaches for the theoretical modelling of vector parallelism which also account for the length of the individual vectors are studied extensively by Blelloch [1990].

2.2 The BSP model and its variants

The BSP model introduced by Valiant [1990] describes a parallel computer with three parameters (p, g, l). The performance of the communication network is characterized by a linear approximation, using parameters g and l. Parameter g, the *communication gap*, describes how fast data can be transmitted continuously by the network (in relation to the computation speed of the individual processors) after a data transfer has started. The *communication latency* l represents the overhead that is necessary for starting up communication. A BSP computation is divided into *supersteps*, each consisting of SPMD-style local computations and a communication phase. At the end of each superstep, the processes are synchronized using a barrier-style synchronization. Consider a computation consisting of S supersteps. For each specific superstep s with $1 \leq s \leq S$ and each processor q with $1 \leq q \leq p$, let $h^{in}_{s,q}$ be the maximum number of data units received and $h^{out}_{s,q}$ the maximum number of data units sent in the communication phase on processor q. Further, let w_s be the maximum number of operations in the local computation phase. The whole

computation has separate *computation cost*

$$W = \sum_{s=1}^{S} w_s \tag{2.1}$$

and *communication cost*

$$H = \sum_{s=1}^{S} h_s \text{ with } h_s = \max_{1 \leq q \leq p}(h_{s,q}^{in} + h_{s,q}^{out}). \tag{2.2}$$

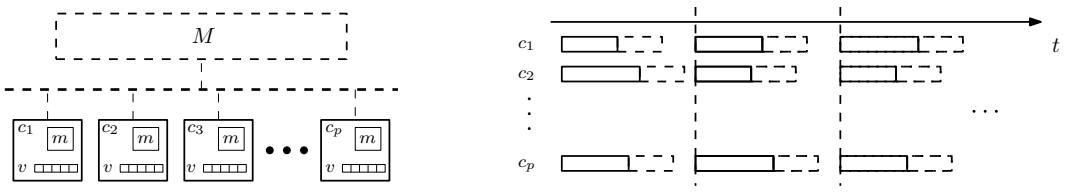
The total running time is given by the sum

$$T = \sum_{s=1}^{S} T_s = W + g \cdot H + l \cdot S. \tag{2.3}$$

Some variations of the BSP model consider a more detailed model of the communication for the individual supersteps (see e.g. Juurlink and Wijshoff [1996]; Blelloch et al. [2008]), which allows one to obtain more accurate runtime predictions. However, at the same time, the models become more specific to the parallel machine the algorithms are run on. The BSPRAM model (Tiskin [1998]) accounts for input/output costs by including an external memory which can be accessed through the communication network. Multi-BSP (Valiant [2008]) is the most recent variant of BSP. In Multi-BSP, a hierarchy of nodes compute and communicate in BSP style. Multi-BSP is an approach to model the time for splitting and communicating input data until it reaches the leaf nodes of the hierarchy (where the actual computation is performed) and then combining and outputting the results. Intuitively, a Multi-BSP algorithm is efficient if this time does not dominate the running time of the computation that is carried out.

2.3 Our model

Based on the models discussed in the previous sections, we study parallel algorithms using a BSP-like approach, combined with a simple model for vector-parallelism based on the MPRAM (Boasson et al. [2001]). This approach is slightly different



(a) BSP computer with external memory and vector units

(b) BSP-style code execution

Figure 2.2: Extended BSPRAM model

from previous work in two aspects. Firstly, we do not attempt to model performance directly. Our model initially does not consider constants to describe the speed of the communication network, latencies or CPU clock frequencies. Our approach is to analyze computation and communication cost separately, and to propose three simple algorithmic properties which ensure asymptotic scalability on hierarchical systems. Furthermore, we attempt to separate "simple" parallelism from algorithmic parallelism which requires more detailed analysis. Finally, our model includes an input/output complexity analysis, i.e. we account for the fact that data may not be distributed to all processors before the computation. We achieve this using a modified version of the BSPRAM (see Tiskin [1998]), which we extend by allowing the use of vector parallelism on each single processor. We will also refer to our model as a *BSP computer with external memory*. We specify criteria for algorithms which allow their efficient simulation on more complex hierarchical models like Multi-BSP (see Valiant [2008]).

We assume that the computation is carried out on a parallel computer with p identical processors, each of which can execute vector instructions. The parallel computation proceeds in supersteps like in the BSP model, i.e. each processor can only access a local subset of data within a single superstep, and data is exchanged between processors at the superstep boundary. In each superstep, we look at the local computation as the maximum running time per processor, and communication as the maximum amount of data that is read or written by a single processor (see Figure 2.2).

The vector capabilities of each processor are modelled independently. We assume that each processor can perform element-wise arithmetic operations and element-wise comparison on vectors of v elements in a single step, where $v \geq 1$ is a constant integer. When considering integer vector computations, we specify the number of bits necessary to store a vector element. For example, when using instructions from the Intel MMX instruction set, see Intel Corporation [1999a], v can be chosen from 2, 4, or 8, restricting the vector elements to 32, 16, or 8 bits respectively. We also note that bit parallelism (see e.g. Powell et al. [2000]; Crochemore et al. [2001]; Hyyrö et al. [2004]) can be modelled by restricting the bit length of each vector element to 1.

To characterize the costs of a computation, we consider the sequential running time $\mathcal{W}(n)$ and the required memory $\mathcal{M}(n)$ as functions of the problem size n. Further, let $\mathcal{I}(n)$ be the maximum of the input and the output size of the problem. E.g. for standard (non-Strassen [1969]) $n \times n$ matrix multiplication, we have problem size n, running time $\mathcal{W}(n) = O(n^3)$, $\mathcal{I}(n) = O(n^2)$, and further $\mathcal{M}(n) = O(n^2)$. For a given problem of size n, it is reasonable to take a sequential algorithm running in time $\mathcal{W}(n)$ as our reference algorithm if a lower bound of $\Omega(\mathcal{W}(n))$ on the running time exists. However, lower bounds, if known, are usually tied to a specific theoretical model of computation which might not correspond completely to the model used for specifying the parallel algorithm. Therefore, we can also consider work optimality in relation to the best known (or most practical), but not necessarily optimal algorithm. We study the following desirable properties in parallel algorithms.

Definition 2.3.1 (Work-optimality)**.** The overall computation time $W(n, p)$ on p processors is the sum of the maximum local computation time over all supersteps. We say that an algorithm is *asymptotically work-optimal* if $W(n, p) = O(\frac{\mathcal{W}(n)}{p})$.

Example 2.3.2. A parallel matrix multiplication algorithm with running time $W(n, p) = O(\frac{n^3}{p})$ is work optimal w.r.t. the standard $O(n^3)$ sequential method. However, it is not work-optimal in an absolute sense, since subcubic algorithms exist (for matrices over a ring, see Strassen [1969]; Coppersmith and Winograd [1990]).

Definition 2.3.3 (Scalable communication)**.** The overall communication cost $H(n,p)$ is the sum over the communication in all supersteps. An algorithm achieves *asymptotically scalable communication* if

$$H(n,p) = O\left(\frac{\mathcal{I}(n)}{p^c}\right), \qquad \text{where } c > 0 \text{ is a constant.} \tag{2.4}$$

In the context of multi-core CPUs, scalable communication can be a simple model for sharing memory bus bandwidth.

Definition 2.3.4 (Scalable memory)**.** The memory cost $M(n,p)$ is the maximum amount of local storage required by a processor across all supersteps. Assume the sequential reference algorithm requires space $\mathcal{M}(n)$. An algorithm achieves asymptotically scalable memory if

$$M(n,p) = O\left(\frac{\mathcal{M}(n)}{p^c}\right), \qquad \text{where } c > 0 \text{ is a constant.} \tag{2.5}$$

Achieving scalable memory is important for algorithms running on multi-core CPUs: it allows one to choose subproblem sizes to fit subproblems into different levels of the CPU cache by recursive partitioning. In practice, programming libraries such as the Intel Threading Building Blocks (TBB) library include functionality to help the implementation of such a partitioning automatically if the algorithm allows it.[1]

Definition 2.3.5 (Synchronization efficiency)**.** An algorithm is synchronization-efficient if the total number of supersteps $S(p)$ is not a function of the problem size n.

Finally, each parallel algorithm can have *slackness conditions*, which are requirements on the relation between n and p to achieve the scalability criteria

[1] TBB partitions problems into subproblems based on ranges of numbers. Given a range of values for which a computation has to be carried out, TBB can be instructed to execute independent computations for subranges in parallel, and to map these to the correct processor cores in order to allow the best re-use of cached information. This is discussed e.g. in the TBB reference manual (Intel Corporation [2010a]) in Sections 3.1 to 3.3 (Splittable Concept, Range Concept, Partitioners).

shown above. A simple example for such a slackness condition is given by BSP parallel prefix computation (see Bisseling [2004]).

Example 2.3.6. Given n values x_1, x_2, ..., x_n and an associative operator $\oplus$, we would like to compute the values x_1, $x_1 \oplus x_2$, $x_1 \oplus x_2 \oplus x_3$, ..., $\bigoplus_{i=1,2,...,n} x_i$. There is a BSP algorithm for this problem which uses $W(n,p) = O(\frac{n}{p})$, $H(n,p) = O(p)$, $M(n,p) = O(\frac{n}{p})$ and $S = O(1)$ if $n \geq p^2$. The slackness condition of requiring $n \geq p^2$ is necessary for the algorithm to be work-optimal. Without requiring $n \geq p^2$, the communication time can become the dominant part of the computation. For example, if we had $p = n$, we get $H(n,p) = O(n)$, i.e. the time for communication would be asymptotically the same as the running time of the sequential algorithm.

In contrast to more complex models, our approach to designing parallel algorithms seems simplistic: no modelling of a cache/memory hierarchy is present. However, applying extensions for BSP to include more detailed modelling of memory access or communication is still possible in our model. Furthermore, computations which achieve scalable memory and scalable communication can be broken down recursively, allowing one to study their performance in more detail using a multi-level cache hierarchy model. On a high level, memory and communication scalability of an algorithm indicate the level of a cache hierarchy the algorithm will be able to utilize efficiently, and therefore estimate the effective communication parameters for the BSP model. Moreover, our model includes a separate measure for the "flat" parallelism contained in an algorithm in the form of bottom-level vector parallelism. We chose vector parallelism as a bottom level for our model since this is well suited to string algorithms. In other applications however, it might be more interesting to "plug-in" a v-processor PRAM at each bottom level node, as in a 2-level Multi-BSP computer, or to use a more complex model altogether. While this does not affect the validity of results for scalable memory and communication, more efficient parallel algorithms for future processor architectures may be modelled this way. For example, a "PRAM on a chip" prototype architecture has been proposed by Wen and Vishkin [2008]. For current architectures however, we will consider algorithms using the simpler model described earlier.

2.4 Implementing parallel algorithms

So far in this chapter, we have concentrated on theoretical modelling of parallel algorithms, neglecting the important question of how well this type of modelling corresponds with current approaches to parallel programming and hardware implementation. We restrict our attention to implementing parallel algorithms using the C++ programming language (Stroustrup [1987]). The choice of this language is based on the good performance of C++ code (see Fulgham [2010] for a performance comparison between different programming languages), and the availability of many algorithmic libraries.

We consider two types of parallel computer in this thesis. First, we look at clusters consisting of multiple multi-core processors, since this is the most common kind of parallel computer which is available at reasonable cost. The most common way of programming these systems is to use MPI (Snir et al. [1995]) in combination with a multi-threading library like OpenMP (Quinn [2003]). This approach offers good control over the exact computation and communication pattern that is implemented. However, it also requires a lot of effort to obtain working code – parallelism on processors and processor cores needs to be specified more or less explicitly by the programmer. Furthermore, MPI allows implementing the same communication patterns in multiple different ways, therefore, code optimized for one parallel machine might require additional effort to be equally efficient on a different system. To address this issue, various "higher-level" parallel programming libraries have been proposed. For programming multi-core processors and SMP systems, the STAPL library (Rauchwerger et al. [2001]) provides multi-threaded versions of C++ standard algorithms for sorting, searching, and list or vector operations. Another approach is taken by Intel's Threading Building Blocks (TBB) library (Intel Corporation [2009]). In TBB, the programmer specifies parallelism by a mostly task-parallel programming interface; communication is implemented by shared memory access. Both these libraries include functionality to retain the simple superstep structure of algorithms and adapt to different numbers of processors that are physically available. STAPL achieves this by providing parallel versions of the C++ Standard Template

Library functions to work on large datasets. In our model, each application of such a parallel function is equivalent to executing a sequence of supersteps. Similarly, TBB includes functions for executing parallel loops and parallel prefix operations in a single superstep. Another approach to high-level multi-core programming is taken by Cilk++ (Leiserson [2009]), which introduces additional constructs into the C++ language to implement recursive work splitting. A Cilk program starts out running in a single thread until it recursively creates new tasks. These tasks are scheduled to run in parallel threads, potentially using all the processors that are physically available. TBB implements similar functionality on a library level (as opposed to compiler level for Cilk++).

In Appendix B, we show how to use an MPI-based implementation of the BSPlib standard (Hill et al. [1998]; Suijlen and Bisseling [2010]) together with the TBB library to create portable C++ code which can run on cluster systems as well as multi-core desktop computers. For vector programming, we use a custom implementation of a vector programming library similar to Intel's Performance Primitives (IPP, see Intel Corporation [2010b]) or AMD's Framewave (The Framewave Group [2009]). This library is described in Appendix A. Appendix C describes our implementation of various algorithms from this thesis, based both on BSP and vector programming. Apart from being useful for a practical application, this algorithm engineering project demonstrates that our approach to algorithm design is practical, and can benefit from recent advances in parallel library and compiler design.

Chapter 3

An Introduction to Semi-local String Comparison

In this chapter, we will give an introduction to semi-local string comparison and its applications. In semi-local string comparison, we compare one string to all substrings of another string using an edit distance, or using an alignment score. Semi-local string comparison is a standard technique for computing string alignments in parallel, as well as for various types of incremental string comparison. This chapter will present the theoretical foundations for understanding the new algorithms presented in the following chapters.

3.1 Overview

In this chapter, we will give an introduction to semi-local string comparison and its applications. In semi-local string comparison, we compare one string to all substrings of another string using an edit distance (Levenshtein [1966]; Wagner and Fischer [1974]), or using an alignment score (Gusfield [1997]). The result of semi-local string comparison is a *highest-score matrix* which contains a score for each string-substring pair. Such highest-score matrices have monotonicity properties which can be exploited to obtain efficient algorithms for semi-local string comparison.

We first give a few definitions on integers, permutations, and we define Monge matrices (see Burkard et al. [1996]), which form the basis of the techniques and algorithms shown in this thesis. We also give basic definitions for string comparison, and show how Monge matrices are related to computing string edit distances. A large part of this chapter is taken up by the description of the "seaweed algorithm" by Tiskin [2008a], which is the basic dynamic programming algorithm we use for computing highest-score matrices. The notation used in this thesis is mostly consistent with the work by Tiskin [2010a].

3.2 Integers, matrices and permutations

We denote the interval of integers $\{i, i+1, \ldots, j\}$ by $[i : j]$. We further denote the interval of odd half-integers $\{i+\frac{1}{2}, i+\frac{3}{2}, \ldots, j-\frac{1}{2}\}$ by $\langle i : j\rangle$. Odd half-integers give a convenient notation for looking at exactly one point between two adjacent integers. We mark odd half-integer variables by a '$\hat{}$' symbol. We can also consider infinite intervals like $\langle -\infty : n\rangle$, or $[-\infty : \infty]$.

We can index matrices by Cartesian products of integer or odd-half integer intervals. When indexing a matrix M by odd half-integer values $\hat{\imath}$ and $\hat{\jmath}$, we define that

$$M(\hat{\imath}, \hat{\jmath}) = M(i, j) \text{ with } i = \hat{\imath} - 1/2 \text{ and } j = \hat{\jmath} - 1/2. \tag{3.1}$$

Therefore, if a matrix has integer indices $[0 : m-1] \times [0 : n-1]$, it has odd half-integer indices $\langle 0 : m\rangle \times \langle 0 : n\rangle$.[1]

We look at *permutations* (see Sagan [2010] for an introduction to permutations and related algorithms) and their corresponding *permutation matrices*. We will define a permutation π of size n as a sequence of integers $\pi = (\pi(1), \ldots, \pi(n))$, where $\pi(1), \ldots, \pi(n) \in [1 : n]$, and $\pi(i) \neq \pi(j)$ if $i \neq j$. The permutation matrix P_π

[1] This translation between indices is only necessary for implementing algorithms on matrices with half-integer indices. We will use the same indexing consistently for the same types of matrices in the algorithm descriptions.

corresponding to π has elements $P_\pi(i,j)$ with

$$P_\pi(i,j) = \begin{cases} 1 & \text{if } \pi(i) = j \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

The set of index pairs $\{(i,j) \mid P_\pi(i,j) = 1\}$ will be referred to as the nonzeros of P_π. The composition of two permutations $\pi_3 = \pi_1 \circ \pi_2$ is computed as $\pi_3(j) = \pi_2(\pi_1(j))$. Composition of two permutations is equivalent to multiplication of their corresponding permutation matrices, i.e. we have $\pi_3 = \pi_1 \circ \pi_2 \Leftrightarrow P_{\pi_3} = P_{\pi_2} P_{\pi_1}$. The *identity permutation* $\mathfrak{i}_n$ of size n is defined as $\mathfrak{i}_n(j) = j$ for $j \in [1:n]$. The permutation matrix corresponding to $\mathfrak{i}_n$ is an $n \times n$ identity matrix. Every n-permutation π has an *inverse permutation* $\bar{\pi}$ with $\pi(\bar{\pi}(j)) = j$ and thus $\pi \circ \bar{\pi} = \mathfrak{i}_n$. For each permutation π, we can study the set of its *inversions*, which is defined as

$$I(\pi) = \{(\pi(i), \pi(j)) \mid i < j \text{ and } \pi(i) > \pi(j)\}. \tag{3.3}$$

Informally, the set of inversions lists all pairs of elements of the permutation which are out of the sorted order. The set of inversions for a permutation can be computed using a variation of Mergesort. Faster algorithms are discussed e.g. by Chan and Pătraşcu [2010].

3.3 Monge matrices

We will work with implicit representations of certain types of matrices, which store the differences between matrix elements rather than the elements themselves. We define *distribution matrices* and *density matrices* as follows.

Definition 3.3.1. The elements of the *distribution matrix* D^Σ of a matrix D with indices from $\langle 0:m\rangle \times \langle 0:n\rangle$ are defined as

$$D^\Sigma(i,j) = \sum D(\hat{\imath},\hat{\jmath}) \text{ with } (\hat{\imath},\hat{\jmath}) \in \langle i:m\rangle \times \langle 0:j\rangle, \tag{3.4}$$

$$D = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \qquad D^{\Sigma} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 0 & 1 & 2 & 2 & 3 & 4 \\ 0 & 0 & 1 & 2 & 2 & 3 & 3 \\ 0 & 0 & 0 & 1 & 1 & 2 & 2 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 3.1: Example: density and distribution matrices

where $(i, j) \in [0 : m] \times [0 : n]$.

Definition 3.3.2. The elements of the *density matrix* $D^{\square}$ of a matrix D with indices (i, j) from $[0 : m] \times [0 : n]$ are defined as

$$\begin{aligned} D^{\square}(\hat{\imath}, \hat{\jmath}) \quad = \quad & D\left(\hat{\imath} + \frac{1}{2}, \hat{\jmath} - \frac{1}{2}\right) - D\left(\hat{\imath} - \frac{1}{2}, \hat{\jmath} - \frac{1}{2}\right) - \\ & D\left(\hat{\imath} + \frac{1}{2}, \hat{\jmath} + \frac{1}{2}\right) + D\left(\hat{\imath} - \frac{1}{2}, \hat{\jmath} + \frac{1}{2}\right), \end{aligned} \tag{3.5}$$

having $(\hat{\imath}, \hat{\jmath}) \in \langle 0 : m \rangle \times \langle 0 : n \rangle$.

Definition 3.3.3. A matrix A is called simple, if $(A^{\square})^{\Sigma} = A$.

A specific type of matrices that are useful for our applications are *Monge matrices* (see Burkard et al. [1996] for a survey on Monge matrices and their applications).

Definition 3.3.4. A matrix A is called a Monge matrix, if

$$A(i, j) + A(i', j') \leq A(i, j') + A(i', j) \;\; \textit{for all } i \leq i', j \leq j'.$$

We notice that a matrix A is Monge if and only if its density matrix $A^{\square}$ is nonnegative (as discussed e.g. by Tiskin [2010a]). Therefore, distribution matrices of permutation matrices are Monge.

Definition 3.3.5. The distribution matrix P^{Σ} of a permutation matrix P is called a *simple unit-Monge matrix*.

Elements of a simple unit-Monge matrix can be stored and queried efficiently using an implicit representation (see Tiskin [2008a]). When looking at the nonzeros of P as points in the plane, querying distribution matrix elements reduces to *orthogonal range counting*, which is a common problem in computational geometry. Therefore, efficient data structures for orthogonal range counting can be used to store P^{Σ} space-efficiently using the nonzeros of P. The easiest way to implement this is to use a range tree (see Bentley [1980]) to achieve storage space $O(n \log n)$, construction time $O(n \log n)$, and query time $O(\log^2 n)$ (or $O(\log n)$ using storage space that is higher by a constant factor, see de Berg et al. [2008]). More efficient data structures exist for range searching in computation models different from the standard RAM model (see Aho et al. [1974]). Some examples of such data structures are given by Alstrup et al. [2000], JáJá et al. [2004], and Chan and Pătraşcu [2010]. If we do not require random access to all matrix elements, the following observation can be used to implement incremental queries to implicit simple unit-Monge matrices.

Theorem 3.3.6 (Tiskin [2010a], p. 11, Theorem 2)**.** *Let P be a permutation matrix. If we are given a value $P^{\Sigma}(i, j)$ for fixed i and j, and the locations of the nonzeros in P, we can compute all four values $P^{\Sigma}(i \pm 1, j \pm 1)$ (where they exist) in time $O(s)$ where s is the time needed to retrieve the location of the nonzero in a specific row or column.*

An important problem for Monge matrices is their multiplication in the $(\min, +)$ semiring. We will define this problem here and show how it can be applied to efficient semi-local string comparison.

We consider the product $M_C = M_A \odot M_B$ of two $N \times N$ matrices with

$$M_C(i, k) = \min_j (M_A(i, j) + M_B(j, k)) \quad \textit{where} \quad i, j, k \in [1 : N]. \tag{3.6}$$

This product can be treated like a generic matrix product, and we can obtain a simple $O(n^3)$ solution for it. Slightly subcubic algorithms for generic matrix products have been shown by Chan [2008]. Furthermore, Duan and Pettie [2009] discuss how

to compute certain specific matrix products faster for real-valued matrices, including our $(\min, +)$-products, which they refer to as *dominance-products*.

Lemma 3.3.7. *The* $(\min, +)$ *product of two unit-Monge matrices is also a unit-Monge matrix.*

Proof. See [Tiskin, 2010a, Chapter 2, Theorem 3]. □

If M_A and M_B are Monge matrices of size $n \times n$, we can achieve running time $O(n^2)$ using the efficient algorithm for computing row-maxima in Monge matrices by Aggarwal et al. [1987]. This algorithm, and other methods of matrix distance multiplication are discussed in [Tiskin, 2010a, Chapter 2]. For simple unit-Monge matrices, we can improve further on $O(n^2)$ by using their density matrices as an implicit representation. One such algorithm was given by Tiskin [2008a], it takes $O(n)$ nonzeros of two density permutation matrices P_A and P_B as its input, and computes the implicit product P_C such that $P_C^\Sigma = P_A^\Sigma \odot P_B^\Sigma$ in time $O(n^{1.5})$. This result was improved to $O(n \log n)$ running time by Tiskin [2010b]. It is an open problem whether it is possible to multiply implicit simple unit-Monge matrices faster, Landau [2006] conjectured that this can be done in linear time. In this thesis, we will describe two algorithms for simple unit-Monge matrix multiplication. We will reproduce the algorithms from [Tiskin, 2008a], as well as from [Tiskin, 2010b] in Chapter 4, and obtain parallel algorithms based on them.

3.4 Longest common subsequences

Let $x = x_1 x_2 \dots x_m$ and $y = y_1 y_2 \dots y_n$ be two strings over an alphabet Σ of size σ. We distinguish between contiguous *substrings* of a string x, which can be obtained by removing zero or more characters from the beginning and/or the end of x, and *subsequences*, which can be obtained by deleting zero or more characters in any position. The *longest common subsequence* (LCS) of two strings is the longest string that is a subsequence of both input strings; its length (the LLCS) is a measure for the similarity of the two strings. Substrings of length w are called *w-windows*. For given

x, y and w, the length of the LCS of two w-windows $x_i \dots x_{i+w-1}$ and $y_j \dots y_{j+w-1}$ will be denoted as $WLCS(i, j)$. We will denote the *reversal* $x_m x_{m-1} \dots x_1$ of string x as $\bar{x}$.

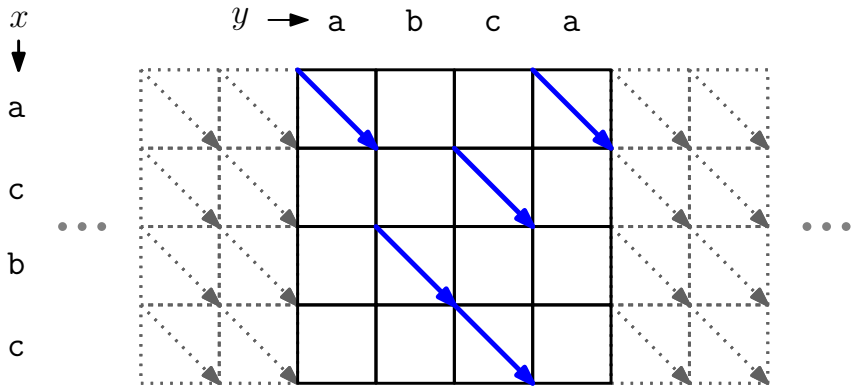


Figure 3.2: Example: extended alignment dag

Definition 3.4.1. Let the *alignment dag (directed acyclic graph)* $G_{x,y}$ for two strings x and y be defined by a set of vertices $v_{s,t}$ with $s \in [0 : m]$ and $t \in [0 : n]$, and edges as follows. We have horizontal and vertical edges $v_{s,t-1} \to v_{s,t}$ and $v_{s-1,t} \to v_{s,t}$ of score 0. Further, we introduce diagonal edges $v_{s-1,t-1} \to v_{s,t}$ of score 1, which are present only if $x_s = y_t$.

Longest common subsequences of x and y correspond to highest-scoring paths in this graph from $v_{0,0}$ to $v_{m,n}$. We will denote by $*y*$ the string y, padded at the left and right with infinitely many special *wildcard characters*, each of which matches all other characters in the alphabet. We call the infinite alignment dag $G_{x,*y*}$ the *extended alignment dag* for x and y (see Figure 3.2).

When drawing the (extended) alignment dag in the plane, its horizontal and vertical edges partition the plane into square cells each of which, depending on the input strings, may contain a diagonal edge of score 1 or not.

Definition 3.4.2. For every pair of characters x_s and y_t, we define a corresponding *cell* $(s - \frac{1}{2}, t - \frac{1}{2})$. Cells corresponding to a matching pair of characters are called *match cells*, and cells corresponding to mismatching characters are called *mismatch cells*.

3.5 Semi-local string comparison and the seaweed algorithm

An interesting extension to computing longest common subsequences is semi-local string comparison. In this problem, we are interested in computing longest common subsequence lengths for one string and all substrings of the other string. In the alignment dag representation, this problem corresponds to computing all highest-scoring paths starting and ending at the graph boundary. Schmidt [1998] proposed an algorithm for computing all such highest-scoring paths in grid dags. This algorithm was applied to string-substring longest common subsequence (LCS) computation by Alves et al. [2008], who gave an $O(n^2)$ algorithm for semi-local comparison of two strings of length n. Tiskin [2008a] developed further understanding of the algorithm and its data structures, obtaining a subquadratic time algorithm for semi-local string comparison including string-substring and prefix-suffix LCS computation. Semi-local string comparison is useful as an intermediate step towards fully-local string comparison, in which all pairs of substrings of the input strings are compared. A straightforward application is computing the LCS efficiently in a sliding window. A simpler variant of this problem was studied by Boasson et al. [2001], they give an algorithm to count the number of fixed size windows of a long text string which contain a short pattern string as a subsequence. Semi-local string comparison is also a useful tool for obtaining efficient parallel algorithms for LCS computation (see Apostolico et al. [1990]; Alves et al. [2006]; Krusche and Tiskin [2007], as well as Chapter 4 of this thesis). A summary of other algorithmic applications is given by Tiskin [2008b].

We now define the semi-local string comparison problem. Solutions to the semi-local LCS problem are given by a *highest-score matrix*.

Definition 3.5.1. In a *highest-score matrix* $A_{x,*y*}$, each entry $A_{x,*y*}(i,j)$ is defined as the length of the highest-scoring path in $G_{x,*y*}$ from $v_{0,i-1}$ to $v_{m,j}$.

Each entry $A_{x,*y*}(i,j)$ with $0 < i \leq j < n$ gives the LLCS of x and substring $y_i \dots y_j$. In a similar way, we can obtain the LLCS of all *prefixes* $x_1 x_2 \dots x_i$ and

Condition	LCS
$0 < i \leq j < n$	*String-substring:* $A_{x,*y*}(i,j) = LLCS(x, y_i \dots y_j)$
$-m < i \leq 0$ *and* $0 < j < n$	*Suffix-prefix:* $A_{x,*y*}(i,j) = LLCS(x_{1-i} \dots x_m, y_1 \dots y_j) - i$
$0 \leq i < n$ *and* $n \leq j < m+n$	*Prefix-suffix:* $A_{x,*y*}(i,j) = LLCS(x_1 \dots x_{m+n-j}, y_{i+1} \dots y_n) + m + n - j$
$-m < i \leq 0$ *and* $n \leq j < m+n$	*Substring-string:* $A_{x,*y*}(i,j) = LLCS(x_{1-i} \dots x_{m+n-j}, y) - i + m + n - j$
otherwise:	$A_{x,*y*}(i,j) = \min(m, n, j-i)$

Table 3.1: LCS lengths given by a highest-score matrix

y, as well as all *suffixes* $y_j \dots y_n$ and x, or also the LLCS of all suffixes $x_i \dots x_m$ and all prefixes $y_1 \dots y_j$ from $A_{x,*y*}$ (see Table 3.1). Since the values of $A_{x,*y*}(i,j)$ for different i and j are strongly correlated, it is possible to derive an implicit, space-efficient representation of matrix $A_{x,*y*}(i,j)$.

Theorem 3.5.2. *Consider a highest-score matrix A. There exists a permutation matrix P_A, for which*

$$A(i,j) = j - i - P_A^{\Sigma}(i,j).$$

P_A^{Σ} is simple unit-Monge (see Definition 3.3.5).

Proof. See [Tiskin, 2010a, Chapter 3, Theorem 9]. □

Corollary 3.5.3. *A highest-score matrix A can be represented implicitly using only $O(m+n)$ space.*

Proof. Note that infinitely many nonzeros exist in the implicit highest-score matrix for any extended alignment dag. However, due to the structure of the extended alignment dag, only a *core* of $m+n$ nonzeros need to be stored. Each of the remaining *off-core* nonzeros can be computed in constant time. The relations between paths through the core of the alignment dag and LCS lengths of substrings of the two input strings are given by Table 3.1. □

Definition 3.5.4. Given the sequence $\{(\hat{\jmath}_0, \hat{\jmath}_k) \;:\; k \in [1 : m]\}$ where $(\hat{\jmath}_0, \hat{\jmath}_k)$ is a nonzero in the implicit highest-score matrix of $x_1 \dots x_k$ and y, a *seaweed* is obtained by connecting the sequence of points $\{(0, \hat{\jmath}_0), (1, \hat{\jmath}_1), (2, \hat{\jmath}_2), \dots, (m-1, \hat{\jmath}_{m-1}), (m, \hat{\jmath}_m)\}$.

The horizontal start and end coordinates $\hat{\jmath}_0$ and $\hat{\jmath}_m$ of each seaweed specify the location of a nonzero in $P_{A_{x,y}}$.

Implicit highest-score matrices can be obtained using the *seaweed algorithm*, which uses dynamic programming on all prefixes of the input strings. This method is graphically illustrated by tracing the seaweeds defined above. We draw seaweeds as curves through the alignment dag cells, starting at an odd half-integer position $\hat{\imath}$ on the top of the alignment dag, and ending at position $\hat{\jmath}$ on the bottom of the alignment dag. Therefore, the curves start between two adjacent vertices $v_{0,\hat{\imath}-\frac{1}{2}}$ and $v_{0,\hat{\imath}+\frac{1}{2}}$, and end between $v_{m,\hat{\jmath}-\frac{1}{2}}$ and $v_{m,\hat{\jmath}+\frac{1}{2}}$. The paths of the seaweeds are computed using the following set of rules.

- Two seaweeds enter every cell in the extended alignment dag, one at the left and one at the top.
- The seaweeds proceed through the cell either downwards or rightwards.
- In the cell, the directions of these seaweeds are interchanged either if there is a match, or if the same pair of seaweeds have already crossed.
- Otherwise, their directions remain unchanged and the seaweeds cross.

Algorithm 1 shows the complete method using pseudocode. The figure on the right hand side illustrates the seaweed behaviour when passing through a cell.

The procedure for querying LCS lengths from implicit highest-score matrices can be implemented by using queries as described in Section 3.3 to compute P_A^Σ from P_A. A graphical interpretation based on seaweeds is shown in Figure 3.3. The LLCS of x and a substring $y_i \dots y_j$ is determined by the number of seaweeds which start and end within the area of the alignment dag induced by x and $y_i \dots y_j$. Following

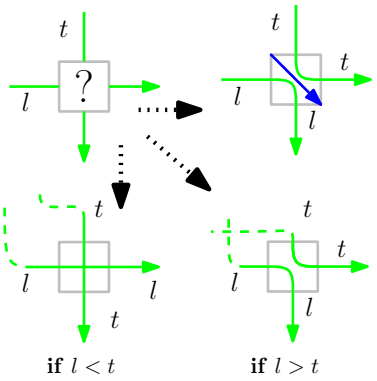
Algorithm 1 The Seaweed Algorithm

input: Strings x and y
output: The implicit highest-score matrix $P_{A_{x,y}}$

set $J[\hat{\imath}] = \hat{\imath}$ for $i \in \langle 0, m+n \rangle$
for $r = 1, 2, \ldots, m$
  $l \leftarrow -r + \frac{1}{2}$
  for $c = 1, 2, \ldots, n$
    $t \leftarrow J[c - \frac{1}{2}]$
    if $x_r = y_c$ or $l > t$
      Swap l and t
    $J[c - \frac{1}{2}] \leftarrow t$
  $J[m + n + r - \frac{1}{2}] \leftarrow t$

$$P_A(\hat{\imath}, \hat{\jmath}) = \begin{cases} 1 & \text{if} \quad \hat{\imath} = J[\hat{\jmath}] \\ 0 & \text{otherwise.} \end{cases}$$

return $P_{A_{x,y}}$



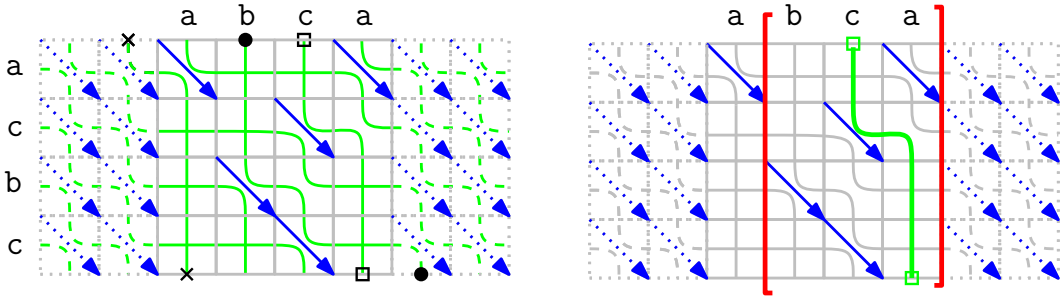
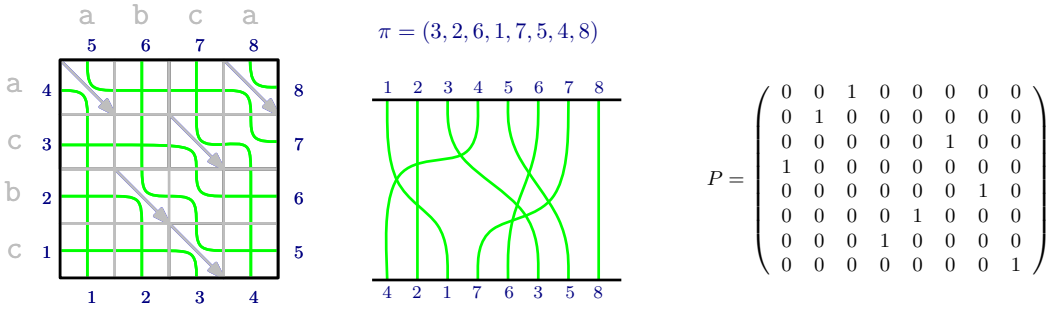


Figure 3.3: Querying LCS scores by seaweeds

the formula given in Theorem 3.5.3, we have to subtract the number of such top-to-bottom seaweeds from the length of the substring $y_i \ldots y_j$ to obtain the LLCS:

$$\begin{aligned} \text{LLCS}(x, y_i \ldots y_j) \quad = \quad & j - i - [\text{number of seaweeds with both their} \\ & \text{horizontal start and end coordinates in } \langle i : j \rangle]. \end{aligned}$$



(a) Seaweeds (b) Seaweed braid and permutation (c) Implicit highest-score matrix

Figure 3.4: Seaweeds, seaweed braid, seaweed permutation and implicit highest-score matrix

3.6 Seaweeds as permutations

Consider an alignment dag for strings of lengths m and n. The seaweed start and end points can be represented by a permutation of size $m + n$.

Definition 3.6.1. Let P_A be the implicit highest-score matrix corresponding to strings x and y of lengths m and n. The *seaweed permutation* π_A for a highest-score matrix A is the $(m+n)$-sized permutation $\pi_A[\hat{\jmath} + \frac{1}{2}] = \hat{k} + \frac{1}{2}$, with $P_A(\hat{k} - m, \hat{\jmath}) = 1$, $\hat{k}, \hat{\jmath} \in \langle 0 : m + n \rangle$.

When drawing the seaweeds without the constraints of the cells in the alignment dag and "straightening" them a little, we obtain *seaweed braids*, which were formalized algebraically by Tiskin [2010b]. The seaweed braids provide a useful intermediate step between storing the full set of seaweed crossings or double crossings, and only storing their permutation: the braid still includes the information on the order in which the seaweeds have crossed. Consider the seaweeds corresponding to a run of the seaweed algorithm on strings x and y. We can obtain the seaweed braid corresponding to these seaweeds by looking at the seaweeds as "rubber strings". We remove the constraints by the alignment dag grid, and straighten the seaweeds a little, preserving their crossings. We are now left with the seaweed braid, which contains the information on the order in which seaweeds cross. This order can be modified to some extent without violating the seaweed properties according to the

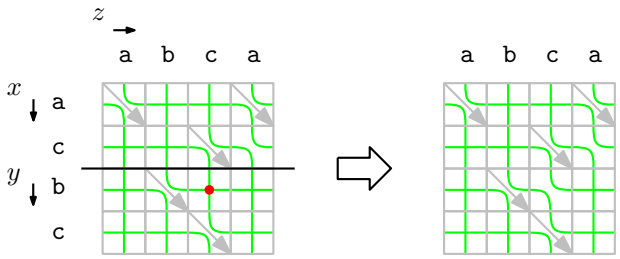


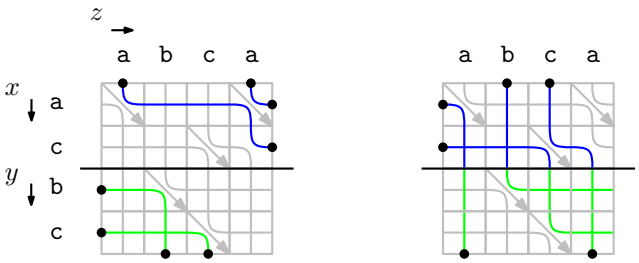
Figure 3.5: Highest-score matrix composition by seaweeds

algebraic rules given by the formal definition of the *seaweed monoid* (see [Tiskin, 2010a, Section 2.2]). We will not use the algebraic approach to seaweeds in this thesis except when drawing figures to illustrate seaweed behaviour. An example for seaweeds, the corresponding braid, permutation and implicit highest-score matrix is shown in Figure 3.4.

3.7 Highest-score matrix composition

In this section, we consider the composition of highest-score matrices for three strings x, y, and z as follows. Given the highest-score matrices $A_{x,z}$ and $A_{y,z}$, we would like to compute $A_{xy,z}$ efficiently using the implicit representation of highest-score matrices. We will now show how to reduce this problem to $(\min, +)$-multiplication of unit-Monge matrices. The $(\min, +)$ product of two implicit $n \times n$ unit-Monge matrices can be computed in time $O(n \log n)$ using the divide-and-conquer algorithm by Tiskin [2010b].

We start by considering runs of the seaweed algorithm on x and z, on y and z, as well as xy and z. Figure 3.5 shows an example where $x = \texttt{ac}$, $y = \texttt{bc}$, and $z = \texttt{abca}$. When concatenating the seaweeds for x and z with the seaweeds from y and z, we get a subset of seaweeds which now double-cross in the bottom half of the alignment dag. In Figure 3.5, the one double-crossing of this kind is marked using a red dot. We first notice that not all pairs of seaweeds can double-cross.



(a) Trivial seaweeds (b) Non-trivial seaweeds

Figure 3.6: Trivial and nontrivial seaweeds in highest-score matrix composition

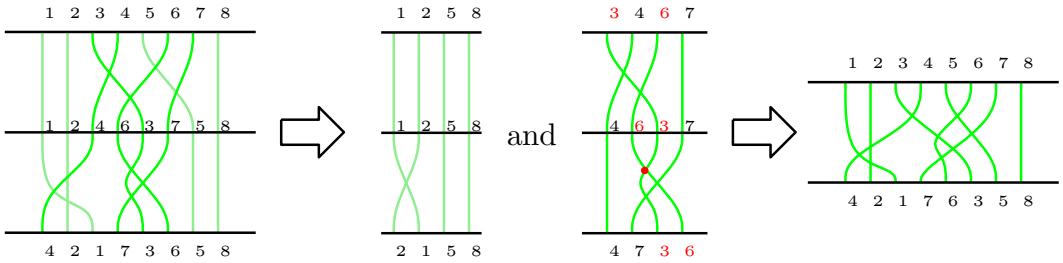


Figure 3.7: Highest-score matrix composition by seaweed braids

In particular, we can retain all seaweeds starting before $-|x|$ at the top, and all seaweeds that end between $|z|$ and $|z| + |x|$ – these will be called *trivial* since they cannot double cross. All remaining seaweeds can have double crossings which we need to remove (see Figure 3.6). Removing the double-crossings in the non-trivial part of the seaweeds is equivalent to computing an implicit $(\min, +)$-product for two $|z| \times |z|$ matrices. We can transform the problem to seaweed braids (see Figure 3.7).

Chapter 4

Parallel String Comparison

In this chapter, we show parallel algorithms for computing longest common subsequences for two input strings, as well as longest increasing subsequences for one input string. Our new algorithms achieve scalability not only in work, but also in communication and memory. Furthermore, we show the first parallel algorithms for computing longest increasing subsequences which achieve general scalablility.

4.1 Background

In the standard BSP model, the LCS of two strings of length n can be computed in $O(\frac{n^2}{p})$ computation time using p processors, $O(n)$ communication and $O(p)$ supersteps using the standard dynamic programming algorithm by Wagner and Fischer [1974] combined with the method for parallel grid dag dynamic programming by McColl [1995] (see also Alves et al. [2003a]; Garcia and Semé [2006]). Semi-local string comparison can be performed in the BSP model at no extra asymptotic cost by combining the grid dag method with the seaweed algorithm. Alternative parallel algorithms by Tiskin [2005, 2008a] compute LCS lengths using a sub-quadratic algorithm for $(\min, +)$ multiplication of implicit unit-Monge matrices. This way, parallel computation of the implicit highest-score matrix for two given strings can be performed in local computation $O(\frac{n^2}{p})$, communication $H(n, p) = O(n \log p)$ and

$S = O(\log p)$ supersteps. Table 4.1 shows a summary of different approaches to parallel LCS computation. In this chapter, we show the first BSP algorithms for semi-local string comparison that achieve scalable communication as well as scalable memory.

We will show two new algorithms for LCS computation which run using work-optimal local computation of $W(n,p) = O(\frac{n^2}{p})$. The first algorithm achieves communication $H(n,p) = O(\frac{n \log p}{\sqrt{p}})$ and uses $S = O(\log p)$ supersteps. The key idea of the algorithm is to carry out the highest-score matrix multiplication procedure in parallel and in a constant number of supersteps. When allowing $\log p$ supersteps for the $(\min, +)$ multiplication algorithm, we can reduce its communication requirements and obtain a semi-local string comparison algorithm running with $H(n,p) = O(\frac{n}{\sqrt{p}})$ and uses $S = O(\log p)$ supersteps.

Our parallel algorithms are based on new, efficient methods for parallel $(\min, +)$ multiplication of implicit simple unit-Monge matrices. Another application of these parallel algorithms is computing the longest increasing subsequence (LIS) of characters in an input string. The LIS problem is a classical problem in theoretical computer science and mathematics, and has been studied extensively (see e.g. Knuth [1973]; Bespamyatnikh and Segal [2000]; Aldous and Diaconis [1999]; Schensted [1961]). We are given a sequence of n numbers, and we are asked to find the longest increasing subsequence. The LIS problem is closely related to patience sorting. In the patience sorting problem, we are looking for a cover of a given sequence of n integers which consists of a minimal number of decreasing subsequences (see Mallows [1973]). The number of decreasing subsequences of such a minimal cover is equal to the length of the longest increasing subsequence. For sequences of length n, the fastest sequential algorithm for patience sorting/longest increasing subsequence computation runs in time $O(n \log n)$ in the comparison-based model, which only allows less-than or equal comparison on the sequence elements. Further, it is possible to achieve $O(n \log \log n)$ in the integer arithmetic model[1]. We will work in the comparison-based model.

[1]Furthermore, Crochemore and Porat [2008] have shown a parameterized algorithm which runs in time $O(n \log \log k)$, where k is the length of the longest increasing subsequence.

Table 4.1: Parallel algorithms for LCS/Levenshtein distance computation of two strings with length n on p processors. We show parallel work $W(n,p)$, communication $H(n,p)$ and the number of supersteps S.

$W(n,p)$	$H(n,p)$	S	*Description*
$O\left(\frac{n^2}{p}\right)$	$O(n)$	$O(p)$	These bounds are obtained for global LCS computation using the parallel algorithm for grid dag computation by McColl [1995] in combination with the standard dynamic programming method for the LCS problem by Wagner and Fischer [1974]
$O\left(\frac{n^2}{p}\right)$	$O(n)$	$O(p)$	In a similar fashion as shown above, we can obtain a work-optimal algorithm for semi-local string comparison using McColl [1995]'s method and the dynamic programming algorithms by Alves et al. [2006], or the seaweed algorithm shown in Chapter 3.
$O\left(\frac{n^2 \log n}{p}\right)$	$O\left(\frac{n^2 \log p}{p}\right)$	$O(\log p)$	Alves et al. [2002] proposed this parallel algorithm for semi-local string comparison. This algorithm is based on distance multiplication of highest-score matrices in their explicit $O(n^2)$ space representation.
$O\left(\frac{n^2}{p}\right)$	$O(pn \log p)$	$O(\log p)$	This algorithm by Alves et al. [2003b] improves the bounds from the (Alves et al. [2002]) paper for the special case of semi-local string comparison in which only string-substring distances need to be computed.
$O\left(\frac{n^2}{p}\right)$	$O(n \log p)$	$O(\log p)$	Using sequential multiplication of implicit highest-score matrices, the previous algorithms can be improved to these bounds (see Tiskin [2005]).
$O\left(\frac{n^2}{p}\right)$	$O\left(\frac{n \log p}{\sqrt{p}}\right)$	$O(\log p)$	This was the first algorithm for (semi-local) LCS computation to achieve scalable communication (see Krusche and Tiskin [2007]) by using the parallel highest-score matrix multiplication shown in Section 4.2.
$O\left(\frac{n^2}{p}\right)$	$O\left(\frac{n}{\sqrt{p}}\right)$	$O(\log^2 p)$	Using the new parallel algorithm for highest-score matrix multiplication shown in Section 4.3 (see also Krusche and Tiskin [2010]), we can improve scalability for communication and memory.

A number of parallel algorithms have been proposed for the LIS problem. However, none of these algorithms achieve general work-optimality in relation to the fastest sequential algorithms, or have rather restrictive slackness conditions. We would first like to point out that the problem is trivially in the complexity class $\mathcal{NC}$, as it is an instance of the shortest path problem in a grid dag, which is in $\mathcal{NC}$ by reduction to $(\min, +)$ matrix multiplication (see McColl [1993]). However, the parallel algorithm resulting from this reduction is far from work-optimal.

On the EREW PRAM model with p processors, Nakashima and Fujiwara [2006] showed that the problem can be solved using time $O(\frac{n \log n}{p})$, but only if $p < n/m^2$ where m is the number of decreasing subsequences in a solution of the equivalent patience sorting problem. The value of m is equal to the length of the longest increasing subsequence, and this algorithm becomes asymptotically sequential once $m > \sqrt{n}$. Any sequence of n numbers must have either a monotonically increasing or a monotonically decreasing subsequence of minimum length $\sqrt{n}$ (see Erdős and Szekeres [1935]; Hammersley [1972]). Therefore, for any sequence of numbers, the condition $p < n/m^2$ definitely inhibits parallelism either for running the algorithm on the sequence itself, or on its reversal.

Semé [2006] gave a BSP-style algorithm which uses time $O(n \log(n/p))$. Since $1 \leq p \leq n$, we have $O(n \log(\frac{n}{p})) = O(n \log n)$ which is asymptotically the same as the sequential algorithm. Another parametrized algorithm by Semé and Youlou [2007] uses the LARPBS model (see Pan [1998]) and allows solving the problem in time $O(k)$ on n processors if k is the length of the longest increasing subsequence. A generic approach is to reduce the problem to computing the LCS of two strings of length n. However, this is clearly not work-optimal as it gives time $O(n^2/p)$. The LIS problem is a special case of computing the longest common subsequence (LCS) of two permutation strings, which can be solved in time $O(n \log n)$ (see Hunt and Szymanski [1977]) in the comparison-based model. While multiple work-optimal and scalable algorithms for the general LCS problem exist, it remains open to obtain a similar result for the LIS problem.

Tiskin studied the more general problem of semi-local comparison of permutation strings and showed a sequential algorithm with running time $O(n \log^2 n)$, as well as various applications (see Tiskin [2006, 2008b]). The result shown here is a parallel algorithm for semi-local comparison of permutation strings, which as a special case also solves the patience sorting/longest increasing subsequence problems. We obtain time $W(n,p) = O(\frac{n \log^2 n}{p})$ for comparing two arbitrary permutations of size n semi-locally on p processors. This is a work-optimal algorithm for semi-local comparison of permutation strings. However, it is not work-optimal for patience sorting, as the fastest sequential algorithm for this problem runs in $O(n \log n) = o(n \log^2 n)$. Nevertheless, there are cases when using our algorithm is advantageous, in particular when the problem size is too large for the sequence to fit into the memory (or cache) of only one processor, or when we need to use many processors, possibly with restricted communication bandwidth. In these cases, our algorithm is superior to previous approaches by achieving scalable communication cost of $O(\frac{n}{p} \log p)$ (this allows processors to share the communication workload). Furthermore, we show that our algorithm achieves scalable memory cost of $O(\frac{n}{p})$ on each processor. Therefore, to our knowledge, our solution for the longest increasing subsequence problem is asymptotically the best parallel algorithm which is actually scalable.

All our parallel algorithms are based on computing highest-score matrices for independent parts of the input, and then using highest-score matrix composition to obtain the final result. The predominant part of the work in highest-score matrix composition is taken up by the $(\min, +)$ multiplication of two implicit unit-Monge matrices. We will give a parallel algorithm for computing such $(\min, +)$ products, which can be used as an algorithmic plugin for solving various string comparison problems in parallel. The idea of using Monge matrices and distance multiplication for obtaining parallel algorithms that solve the LCS problem has been applied to LCS computation on the PRAM e.g. by Apostolico et al. [1990]. Since highest-score matrix multiplication is the dominant part of the work for highest-score matrix composition, this immediately allows us to obtain work-optimal algorithms for parallel

semi-local string comparison if the sequential multiplication algorithms run in $o(n^2)$. We will now show such algorithms and how they can be parallelized work-optimally, with scalable communication, and with scalable memory.

Recall that in highest-score matrix multiplication, we compute the product $M_C = M_A \odot M_B$ of two $n \times n$ matrices with

$$M_C(i,k) = \min_j(M_A(i,j) + M_B(j,k)), \quad \text{where} \quad i,j,k \in [1:n].$$

Our algorithms work on the implicit representation of highest-score matrices, i.e. we multiply the non-trivial parts of two highest-score matrices (see Section 3.7). We therefore need to compute the nonzeros of P_C, such that

$$P_C^{\Sigma} = P_A^{\Sigma} \odot P_B^{\Sigma}, \tag{4.1}$$

given the nonzeros of two $n \times n$ permutation matrices P_A and P_B as an input. We assume throughout the chapter, that (without loss of generality) n is a power of 2 for simplicity of presentation.

The remainder of this chapter is structured as follows. We show in Section 4.2 how $(\min, +)$ multiplication of simple unit-Monge matrices can be parallelized in a constant number of supersteps in subquadratic time. This algorithm gives the best theoretical number of supersteps, but is mostly shown here for historical reasons. It was superseded by the algorithm shown in Section 4.3, which is superior in every aspect except for requiring an asymptotically larger number of supersteps to preserve reasonable slackness conditions. In Section 4.4, we show how our algorithms can be applied to obtain parallel algorithms for LCS computation with scalable communication. Finally, we discuss the suitability of our algorithms for parallel LIS computation in Section 4.5.

4.2 Parallel unit-Monge matrix multiplication in $O(1)$ supersteps

We will now show a parallel algorithm for $(\min, +)$ multiplication of unit-Monge matrices which runs in $W(n,p) = O(\frac{n^{1.5}}{p})$, $H(n,p) = M(n,p) = \frac{n}{p^{0.5}}$, and $S = O(1)$. This algorithm is based on a quadtree partitioning of the output matrix in order to locate its nonzeros, and was the first to achieve scalable communication for parallel LCS computation.

We will first show an algorithm to compute such an implicit matrix product sequentially in time $O(n^{1.5})$, which we then parallelize to run in $W(n,p) = O(\frac{n^{1.5}}{p})$. The sequential algorithm was shown first by Tiskin [2008a], and was the first algorithm to achieve sub-quadratic running time for $(\min, +)$ multiplication of implicit simple unit-Monge matrices. This matrix multiplication method uses a divide-and-conquer approach that recursively partitions the output matrix P_C into smaller blocks in order to locate its nonzeros. We compute the number of nonzeros contained in each block. We can stop recursing in two cases: either if the block does not contain any nonzeros, or if it is of size 1×1 and thus specifies the location of a nonzero.

Definition 4.2.1. The rectangular submatrix of a permutation matrix P corresponding to a set of indices $\langle i_0 - h : i_0\rangle \times \langle k_0 : k_0 + w\rangle$ is called a *P-block*. We denote the number of nonzeros contained in such a block as follows:

$$P(\langle i_0 - h : i_0\rangle \times \langle k_0 : k_0 + w\rangle) = \sum_{(\hat{\imath},\hat{\jmath}) \in \langle i_0 - h : i_0\rangle \times \langle k_0 : k_0 + w\rangle} P(\hat{\imath}, \hat{\jmath}). \tag{4.2}$$

This generalizes Definition 3.3.1, which corresponds to the case when $i_0 = n$ and $k_0 = 0$. We also note:

$$\begin{aligned} P(\langle i_0 - h : i_0\rangle \times \langle k_0 : k_0 + w\rangle) \;=\; & P^{\Sigma}(i_0 - h, k_0 + w) \\ & - \; P^{\Sigma}(i_0, k_0 + w) \\ & - \; P^{\Sigma}(i_0 - h, k_0) \\ & + \; P^{\Sigma}(i_0, k_0) \end{aligned} \tag{4.3}$$

Consider the square P_C-block with index set $\langle i_0 - h : i_0\rangle \times \langle k_0 : k_0 + h\rangle$. To compute the number of nonzeros contained within this block using Equation (4.3), we need the values $P_C^{\Sigma}(i_0, k_0)$, $P_C^{\Sigma}(i_0 - h, k_0)$, $P_C^{\Sigma}(i_0, k_0 + h)$, and $P_C^{\Sigma}(i_0 - h, k_0 + h)$ of the distribution matrix at the four corners of the block. Each of these values can be computed using Equation (3.6). Furthermore, the differences of the P_C^{Σ} values in $\langle i_0 - h : i_0\rangle \times \langle k_0 : k_0 + h\rangle$ from $P_C^{\Sigma}(i_0, k_0)$ are determined only by sets of *relevant* nonzeros in P_A and P_B.

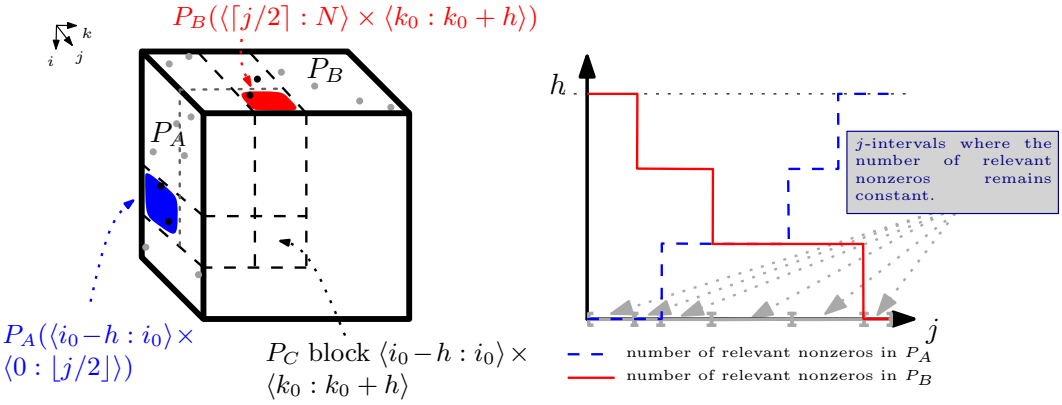


Figure 4.1: Matrix product cube representation and relevant nonzeros

Definition 4.2.2. The relevant nonzeros for a P_C-block corresponding to indices in $S = \langle i_0 - h : i_0\rangle \times \langle k_0 : k_0 + h\rangle$ are located in the P_A-block with indices $\langle i_0 - h : i_0\rangle \times \langle 0 : n\rangle$, and in the P_B-block with indices $\langle 0 : n\rangle \times \langle k_0 : k_0 + h\rangle$ (see also Figure 4.1, left side).

As the block size decreases, fewer nonzeros are relevant for a block. We split the strips of relevant nonzeros at a position $j/2$ with $j \in [1:2n]$, and count nonzeros up to $\lfloor j/2 \rfloor$ in P_A, and from $\lceil j/2 \rceil$ in P_B. Therefore, we are interested in the values $P_A(\langle i_0 - h : i_0 \rangle \times \langle 0 : \lfloor j/2 \rfloor \rangle)$ and $P_B(\langle \lceil j/2 \rceil : n \rangle \times \langle k_0 : k_0 + h \rangle)$. This is illustrated on the left side of Figure 4.1. The figure shows the matrix product cube representation where each point in the cube corresponds to an elementary $(\min, +)$ product. The left side of the cube corresponds to matrix P_A, the top to matrix P_B and the front to matrix P_C. The nonzeros we count are contained within the areas highlighted in grey on the left side of Figure 4.1. Since P_A and P_B are permutation matrices, a maximum of $2h$ relevant nonzeros exist for each $h \times h$ block in P_C. Also, for any P_C-block, the number of relevant nonzeros in P_A up to j is nondecreasing when j increases, and the number of relevant nonzeros in P_B after j is nonincreasing. In the right-hand side Figure 4.1, the dashed line shows an example for the number of relevant nonzeros in P_A, and the solid line shows an example for the number of relevant nonzeros in P_B. We can also see in this figure that *contiguous j-intervals* exist within which the number of relevant nonzeros in P_A and P_B remains the same. We can use this fact to encode $P_A(\langle i_0 - h : i_0 \rangle \times \langle 0 : \lfloor j/2 \rfloor \rangle)$ and $P_B(\langle \lceil j/2 \rceil : n \rangle \times \langle k_0 : k_0 + h \rangle)$ using only $O(h)$ space, and assign a unique index $d \in [-h : h]$ to each j-interval.

Definition 4.2.3. Consider a P_C-block corresponding to indices in $S = \langle i_0 - h : i_0 \rangle \times \langle k_0 : k_0 + h \rangle$. Let

$$\begin{aligned} \Delta_A^S(d) &= \text{any } P_A(\langle i_0 - h : i_0 \rangle \times \langle 0 : \lfloor j/2 \rfloor \rangle) \quad \text{and} \\ \Delta_B^S(d) &= \text{any } P_B(\langle \lceil j/2 \rceil : n \rangle \times \langle k_0 : k_0 + h \rangle) \text{ for any } j \end{aligned} \tag{4.4}$$

from the interval where

$$P_A(\langle i_0 - h : i_0 \rangle \times \langle 0 : \lfloor j/2 \rfloor \rangle) - P_B(\langle \lceil j/2 \rceil : n \rangle \times \langle k_0 : k_0 + h \rangle) = d. \tag{4.5}$$

The predicate "any" states that we can take the value for any index j in the given interval (as they are all equal). We can compute these sequences from the relevant nonzeros for a P_C-block by an $O(h)$ scan.

For each P_C-block, we are also interested in the sequence of minima

$$M^S(d) = \min\,(P_A^\Sigma(i_0, j) + P_B^\Sigma(j, k_0)) \text{ with } j \text{ as above.} \tag{4.6}$$

The predicate "min" must be taken over all values of j corresponding to d, since the values $P_A^\Sigma(i_0, j) + P_B^\Sigma(j, k_0)$ can differ within a j-interval.

We can now obtain the values of P_C^Σ at the four corners of the current P_C-block in time $O(h)$ by taking the minimum over all values $d \in [-h : h]$:

$$\begin{aligned}
P_C^\Sigma(i_0, k_0) &= \min_{d\in[-h:h]} M^S(d), \\
P_C^\Sigma(i_0 - h, k_0) &= \min_{d\in[-h:h]} (\Delta_A^S(d) + M^S(d)), \\
P_C^\Sigma(i_0, k_0 + h) &= \min_{d\in[-h:h]} (\Delta_B^S(d) + M^S(d)), \\
P_C^\Sigma(i_0 - h, k_0 + h) &= \min_{d\in[-h:h]} (\Delta_A^S(d) + \Delta_B^S(d) + M^S(d)).
\end{aligned} \tag{4.7}$$

From these values, the number of nonzeros in the current block can be obtained using Equation (4.3). If this number is zero, the recursion can terminate at the current P_C-block. Otherwise, the algorithm proceeds to recursively partition the block into four subblocks of size $h/2$ in order to locate the nonzeros. We partition the P_C-block into subblocks with the following four sets of indices:

$$\begin{aligned}
S_{\blacksquare\square} &= \langle i_0 - h/2 : i_0\rangle \times \langle k_0 : k_0 + h/2\rangle \\
S_{\square\blacksquare} &= \langle i_0 - h/2 : i_0\rangle \times \langle k_0 + h/2 : k_0 + h\rangle \\
S_{\blacksquare\blacksquare} &= \langle i_0 - h : i_0 - h/2\rangle \times \langle k_0 : k_0 + h/2\rangle \\
S_{\square\square} &= \langle i_0 - h : i_0 - h/2\rangle \times \langle k_0 + h/2 : k_0 + h\rangle.
\end{aligned} \tag{4.8}$$

In order to partition the block, it is necessary to determine the sequence M^S and the relevant nonzeros for all P_C-subblocks. Splitting the sets of relevant nonzeros

can be done trivially in $O(h)$. To establish the sequence $M^{S'}$ for each P_C-subblock, we count the number of nonzeros that can contribute to the minima in each of the subblocks. Consider the P_C-block with indices in $S = \langle i_0 - h : i_0 \rangle \times \langle k_0 : k_0 + h \rangle$. The nonzeros in P_A and P_B which can affect the minima of any of the four subblocks are those relevant for the lower left subblock $S_{\blacksquare}$, as the minima are always taken at the lower left corner of a P_C-block.

$$\begin{aligned} \bar{\Delta}_A^S(d) &= \text{any } P_A(\langle i_0 - h/2 : i_0 \rangle \times \langle 0 : \lfloor j/2 \rfloor \rangle) \quad \text{and} \\ \bar{\Delta}_B^S(d) &= \text{any } P_B(\langle \lceil j/2 \rceil : n \rangle \times \langle k_0 : k_0 + h/2 \rangle) \quad \text{with } j \text{ as in (4.5).} \end{aligned} \tag{4.9}$$

It is now possible to compute the sequence $M^{S'}$ for each of the four P_C-subblocks by taking the minimum over the minima for the current block, adding the numbers of nonzeros $\bar{\Delta}_A^S$ and $\bar{\Delta}_B^S$ where they could affect the minima at the lower left corner of the respective subblock. Let $d' \in [-h/2 : h/2]$, and let $d \in \{d \mid \bar{\Delta}_A^S(d) - \bar{\Delta}_B^S(d) = d'\}$ when computing the following predicates.

$$\begin{aligned} M^{S_{\blacksquare}}(d') &= \min\ M^S(d), \\ M^{S_{\blacksquare\square}}(d') &= \min\ M^S(d) + \bar{\Delta}_B^S(d), \\ M^{S_{\square\blacksquare}}(d') &= \min\ M^S(d) + \bar{\Delta}_A^S(d), \\ M^{S_{\blacksquare\blacksquare}}(d') &= \min\ M^S(d) + \bar{\Delta}_A^S(d) + \bar{\Delta}_B^S(d). \end{aligned} \tag{4.10}$$

This is equivalent to computing the minima over the j-intervals for the P_C-subblocks.

Theorem 4.2.4. *Given their implicit representation, two permutation distribution matrices of size $n \times n$ can be multiplied over the $(\min, +)$ semiring in time $O(n^{1.5})$.*

Proof. By analysis of the quadtree recursion resulting from the partitioning shown above. Each recursive call in the quadtree recursion shown above can have four children up to level $\log_4 n$ since the output is a permutation matrix. Up to level $\log_4 n = \frac{1}{2} \log_2 n$, the block size will decrease by a factor of two at each level. Therefore, the work for these levels is dominated by level $\log_4 n$, which requires time $O(n \cdot n/2^{1/2 \log_2 n}) = O(n^{1.5})$. In the levels above $\log_4 n$, maximally n blocks

can still contain nonzeros, therefore, the work for these levels is dominated by level $\log_4 n$ as well. Overall, we get running time $O(n^{1.5})$ (see also Tiskin [2008a]). □

We will now show how to obtain a parallel version of this highest-score matrix multiplication algorithm. We assume w.l.o.g. that $\sqrt{p}$ is an integer, and that every processor has an unique identifier q with $0 \leq q < p$. Further assume that all parameters match, such that e.g. n/p is integer. The initial distribution of the nonzeros of the input matrices is assumed to be even among all processors, so that every processor holds n/p nonzeros of P_A and P_B.

Definition 4.2.5. We define a two-dimensional grid of processors, where each processor q corresponds to exactly one pair $(q_x, q_y) \in [1 : \sqrt{p}] \times [0 : \sqrt{p} - 1]$. We also assign a different P_C-block with index set

$$S_q = \left\langle (q_x - 1) \cdot \frac{n}{\sqrt{p}} : q_x \cdot \frac{n}{\sqrt{p}} \right\rangle \times \left\langle q_y \cdot \frac{n}{\sqrt{p}} : (q_y + 1) \cdot \frac{n}{\sqrt{p}} \right\rangle \tag{4.11}$$

to each processor q.

Note first that the recursive divide-and-conquer computation from the previous section has at most p independent problems at level $\frac{1}{2} \log_2 p$. In the parallel version of the algorithm, we start the recursion directly at this level and compute the sequences $\Delta_A^{S_q}$, $\Delta_B^{S_q}$ and M^{S_q} from scratch. To compute the values of sequence M^{S_q} for every processor q, we must compute the elementary $(\min, +)$ products

$$D_A^{\Sigma}\left(q_x \cdot \frac{n}{\sqrt{p}}, j\right) + D_B^{\Sigma}\left(j, q_y \cdot \frac{n}{\sqrt{p}}\right) \quad \text{with } j \in [0 : n]. \tag{4.12}$$

Theorem 4.2.6. *Given the distributed implicit highest score matrices P_A and P_B, all $p \cdot n$ elementary $(\min, +)$ products from Equation (4.12) can be computed on a BSP computer in $W = O(n/\sqrt{p})$, $H = O(p + n/p)[= O(n/p)$ if $n > p^2]$, and $S = O(1)$.*

Proof. As we only have the implicit form of the highest score matrices distributed among the processors, computing values $D_A^{\Sigma}(q_x \cdot n/\sqrt{p}, j)$ and $D_B^{\Sigma}(j, q_y \cdot n/\sqrt{p})$ requires counting nonzeros in P_A and P_B.

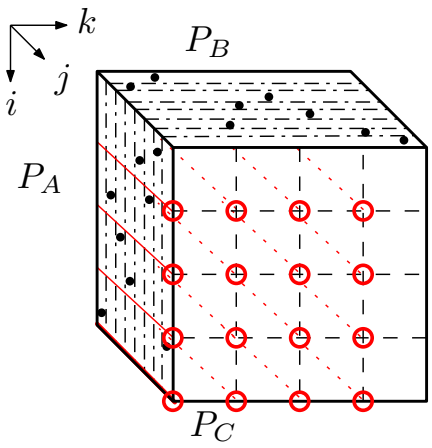


Figure 4.2: Computing block-minima in P_C by parallel prefix

First we redistribute the nonzeros to strips of width n/p by sending each nonzero $(\hat{\imath}, \hat{\jmath})$ in P_A and each nonzero $(\hat{\jmath}, \hat{k})$ in P_B to processor $\lfloor(\hat{\jmath} - \frac{1}{2}) \cdot p/n\rfloor$. This is possible in one superstep using communication $O(n/p)$.

Every processor holds all $P_A(\hat{\imath}, \hat{\jmath})$ and all $P_B(\hat{\jmath}, \hat{k})$ for $\hat{\jmath} \in \langle q \cdot \frac{n}{\sqrt{p}} : (q+1) \cdot \frac{n}{\sqrt{p}}\rangle$. Since D_A^Σ and D_B^Σ are obtained from P_A and P_B by the sum given in Equation (3.4), we can compute the values $D_A^\Sigma(q_x \cdot n/\sqrt{p}, j)$ and $D_B^\Sigma(j, q_y \cdot n/\sqrt{p})$ in blocks of n/p on every processor by using a parallel prefix – respectively parallel suffix – operation (see McColl [1993]) over index j. We have $\sqrt{p}$ instances of parallel prefix (respectively parallel suffix), one for each value of q_x (respectively q_y). Therefore, the total cost of the parallel prefix and suffix computation is $W(n,p) = O(n/p \cdot \sqrt{p}) = O(n/\sqrt{p})$; the communication cost is negligible as long as $n/p \geq p \;\Rightarrow n \geq p^2$. The parallel prefix and suffix operations can be carried out in $S = O(1)$ supersteps by computing intermediate (local) prefix results on every processor, performing an all-to-all exchange of these values, and then locally combining on every processor the local results with the corresponding intermediate values. After the prefix and suffix computations, every processor holds n/p elementary $(\min, +)$ products. □

Now that the elementary $(\min, +)$ products have been computed, we redistribute the data, so that each processor q has the data it needs to continue processing P_C-block S_q. To be able to continue the recursive procedure at this level, every pro-

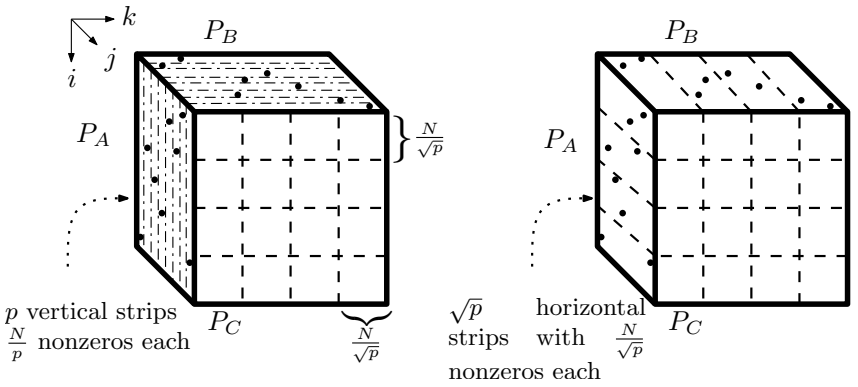


Figure 4.3: Cube representation of $(\min, +)$ matrix product

cessor must have the $O(n/\sqrt{p})$ values of sequence M^{S_q}, and the sets of relevant nonzeros in P_A and P_B which allow us to compute $\Delta_A^{S_q}$ and $\Delta_B^{S_q}$.

Theorem 4.2.7. *If every processor holds n/p elementary $(\min, +)$ products, as well as at most n/p nonzeros in P_A and the same number of nonzeros in P_B, we can redistribute data such that each processor q holds the sequence M^{S_q} and the relevant nonzeros for block S_q in one superstep using communication $H(n,p) = O(n/\sqrt{p})$.*

Proof. Each nonzero in P_A (and in P_B respectively) increases the overall number of nonempty j-intervals by one for each of the $\sqrt{p}$ P_C-blocks where this nonzero is relevant; a nonzero does not affect the number of j-intervals for any other P_C-blocks. Each j-interval is assigned one value in the corresponding sequence M^{S_q}. Therefore, the total number of M^{S_q}-values per processor before redistribution is at most $n/p \cdot \sqrt{p} + p = n/\sqrt{p} + p$. The total number of M^{S_q}-values per processor after redistribution is $n/\sqrt{p}$, therefore the communication is perfectly balanced, apart from the maximum of p values that can arise due to processor boundaries "splitting" a j-interval. Since every processor holds n/p nonzeros before redistribution, and every nonzero is relevant for $\sqrt{p}$ P_C-blocks, redistributing the relevant nonzeros can also be done in $O(n/\sqrt{p})$ communication (see Figure 4.3, right). After this, every processor has all the data that are necessary to perform the sequential procedure from the previous section on its P_C-block. □

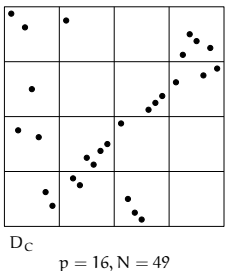


Figure 4.4: Nonzeros in P_C

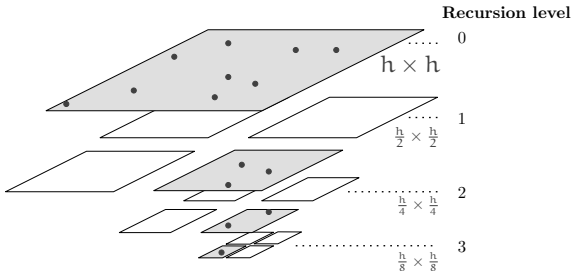


Figure 4.5: Partitioning a P_C-block

A first result for the BSP complexity of parallel highest-score matrix multiplication follows immediately.

Corollary 4.2.8. *The resulting parallel highest-score matrix multiplication procedure has BSP cost*

$$W(n,p) = O\left(\left(\frac{n}{\sqrt{p}}\right)^{1.5}\right) = O\left(\frac{n^{1.5}}{p^{0.75}}\right),\ H(n,p) = O\left(\frac{n}{\sqrt{p}}\right),\ \textit{and}\ S = O(1). \tag{4.13}$$

The implicit highest-score matrix multiplication procedure we have developed so far is not work-optimal: in the denominator of the running time we have $p^{0.75}$ instead of p, which is required for work-optimality. This is a problem when applying this procedure to parallel permutation string comparison, as the highest-score matrix multiplication part of the work can then become dominant.

The parallel multiplication algorithm works by locating nonzeros in the resulting implicit highest-score matrix recursively. For work distribution, the current algorithm partitions the matrix into a grid of $\sqrt{p} \times \sqrt{p}$ blocks (see Figure 4.4) in its precomputation step. Because not every block needs to contain the same number of nonzeros, a load-imbalance can occur and prevent work-optimality. In the worst case, only $\sqrt{p}$ blocks contain nonzeros, and therefore only $\sqrt{p}$ blocks of size $n/\sqrt{p}$ are processed in parallel. The sequential complexity for processing a $h \times h$ block is $O(h^{1.5})$. By setting $h = n/\sqrt{p}$, we obtain a bound for the computation time of $O(n^{1.5}/p^{0.75})$.

We will now show a load-balancing strategy for work-optimal highest-score matrix multiplication with scalable communication. To achieve this, we first prove

a small extension to the theorems by Tiskin [2008a]. A useful tool is parameterizing the time for locating nonzeros by the number of nonzeros we are looking for within a P_C-block. Each processor should locate a set of nonzeros contained within a different set of leaves of the partitioning quadtree (see Figure 4.5 for an illustration). One way to implement this is to pick subsets of nonzeros based on the order in which the sequential algorithm discovers the nonzeros. When partitioning, we then know for each subblock how many nonzeros the sequential algorithm would have discovered in this block, and therefore also know the index of the first and last nonzero in each block. This allows us to only partition those blocks which contain nonzeros in the subset we are looking for. We prove the following theorem.

Lemma 4.2.9. *Consider a P_C-block of size $h \times h$ in which we would like to locate k nonzeros. Assume we are given sequence M^S for this block and the relevant nonzeros in P_A and P_B. On a single processor, the nonzeros in the P_C-block can be located in $O(h\sqrt{k})$ time.*

Proof. When partitioning the P_C-block using the quadtree scheme shown in Section 3.3, the number of subblocks can increase geometrically only up to level $\frac{1}{2}\log k$. At this level, only k blocks can contain nonzeros from the subset we are looking for. Therefore, k independent subproblems exist, each of which requires time $O(h/\sqrt{k})$ in order to be partitioned further. Since P_C is a permutation matrix, the size of the k subproblems must decrease geometrically from this level on. Therefore, the running time is dominated by level $\frac{1}{2}\log k$, and the cost in the worst case is bounded by $O(h/\sqrt{k} \cdot k) = O(h\sqrt{k})$. □

The reason why we have not achieved work-optimality so far is that only one processor might need to locate all $\frac{n}{\sqrt{p}}$ nonzeros in a block. After the pre-processing phase in which we obtain the relevant nonzeros and minima M^S in each of the p P_C-blocks, we introduce another step to balance the work of locating nonzeros.

Lemma 4.2.10. *Highest-score matrix multiplication of two $n \times n$ implicit highest-score matrices on p processors has BSP cost*

$$W(n,p) = O\left(\frac{n^{1.5}}{p}\right),\ H(n,p) = O\left(\frac{n}{\sqrt{p}}\right),\ \textit{and}\ S = O(1). \tag{4.14}$$

Proof. Each processor holds data of size $O(\frac{n}{\sqrt{p}})$ to process a P_C-block of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$. If this block contains more than $\frac{n}{p}$ nonzeros, it cannot be processed work-optimally by a single processor.

Therefore, we partition all blocks into zero or more *complete* groups with exactly $\frac{n}{p}$ nonzeros, and possibly one *incomplete* group which contains less than $\frac{n}{p}$ nonzeros. If the block contains at most $\frac{n}{p}$ nonzeros, then we have a single group in this block. We broadcast the data for the block to enough processors such that each processor has to locate exactly one complete group of nonzeros. This is possible with BSP cost $H(n,p) = O(\frac{n}{\sqrt{p}})$, and $S = O(1)$ using a two-phase broadcast as shown e.g. by Bisseling [2004] (pp. 66–67). Each processor then only locates the $\frac{n}{p}$ nonzeros in its respective group, and possibly the nonzeros in one incomplete group if one existed for its block. The running time for locating $\frac{n}{p}$ nonzeros in a block of size $\frac{n}{\sqrt{p}}$ is $O(\frac{n^{1.5}}{p})$ due to Lemma 4.2.9. As each of our P_C-blocks can only contain maximally $\frac{n}{\sqrt{p}}$ nonzeros, the maximum number of processors we broadcast to is $\sqrt{p}$. Moreover, the maximum number of complete groups is p, since there are only n nonzeros in total. Furthermore, maximally one incomplete group per processor can exist, which limits the number of such groups to p. The maximum total number of separate groups of nonzeros is therefore $2p$. Each of these groups can be processed in time $O(\frac{n^{1.5}}{p})$ on a single processor, which gives the claimed work-optimal bound. □

4.3 Parallel unit-Monge matrix multiplication in $O(\log p)$ supersteps

In the previous section, we have shown a parallel algorithm for simple unit-Monge matrix multiplication running in $W(n,p) = O(\frac{n^{1.5}}{p})$, $H(n,p) = M(n,p) = (\frac{n}{p^{0.5}})$, and $S = O(1)$. In this section, we give a new parallel algorithm for simple unit-

Monge matrix multiplication, based on a newer and faster sequential multiplication method (see Tiskin [2010a,b]). We will show an algorithm that runs in $W(n,p) = O(\frac{n \log n}{p})$, $H(n,p) = O(\frac{n \log p}{p})$, $M(n,p) = O(\frac{n}{p})$, and $S = O(\log p)$.

We start by describing the sequential algorithm that is the basis for our parallel algorithm. This algorithm works by partitioning the input permutation matrices into two half-sized parts each. Assume without loss of generality that n is a power of 2. We define $P_{A,lo}$ as the $\frac{n}{2} \times \frac{n}{2}$ sized permutation matrix which is induced by the nonzeros in $P_A(\langle 0 : n\rangle, \langle 0 : \frac{n}{2}\rangle)$. Analogously, we obtain $P_{A,hi}$ from $P_A(\langle 0 : n\rangle, \langle \frac{n}{2} : n\rangle)$, $P_{B,lo}$ from $P_B(\langle 0 : \frac{n}{2}\rangle, \langle 0 : n\rangle)$, and $P_{B,hi}$ from $P_B(\langle \frac{n}{2} : n\rangle, \langle 0 : n\rangle)$. We then have

$$P'^{\Sigma}_{C,lo} = P^{\Sigma}_{A,lo} \odot P^{\Sigma}_{B,lo} \quad \text{and} \quad P'^{\Sigma}_{C,hi} = P^{\Sigma}_{A,hi} \odot P^{\Sigma}_{B,hi}, \tag{4.15}$$

Matrices $P'_{C,lo}$ and $P'_{C,hi}$ can be obtained by recursively calling our multiplication algorithm for matrices of size $\frac{n}{2} \times \frac{n}{2}$ to compute the result of Equation (4.15). We can preserve the indices of the rows and columns in P_A and P_B which were deleted obtaining $P_{A,lo}$, $P_{A,hi}$, $P_{B,lo}$, and $P_{B,hi}$, and use them to convert the resulting matrices $P'_{C,lo}$ and $P'_{C,hi}$ to matrices $P_{C,lo}$ and $P_{C,hi}$ of size $n \times n$ by adding rows or columns of zeros. Notice that since P_A and P_B are permutation matrices, the sum $P_{C,lo} + P_{C,hi}$ of the resulting matrices will form a permutation matrix as well. We get (see Tiskin [2010b] for details)

$$P^{\Sigma}_{C}(i,k) = \min(P^{\Sigma}_{C,lo}(i,k) + P^{\Sigma}_{C,hi}(0,k),\ P^{\Sigma}_{C,hi}(i,k) + P^{\Sigma}_{C,lo}(i,n)\ ). \tag{4.16}$$

The full procedure for this divide step of the computation is shown in Algorithm 2. The function ImplicitMult carries out the highest-score matrix multiplication recursively on the two subproblems and will be defined in Algorithm 4.

It now remains to merge the partial results $P_{C,lo}$ and $P_{C,hi}$ to obtain P_C according to (4.16). By analysing the difference

$$\delta(i,k) \quad = \quad (P^{\Sigma}_{C,lo}(i,k) + P^{\Sigma}_{C,hi}(0,k)) - (P^{\Sigma}_{C,hi}(i,k) + P^{\Sigma}_{C,lo}(i,n)), \tag{4.17}$$

Algorithm 2 Recursive simple unit-Monge matrix multiplication, divide step

procedure ImplicitMult_Split_Recurse(P_A, P_B)
input: Two implicit unit-Monge matrices P_A and P_B of sizes $n \times n$
output: A pair of matrices $(P_{C,lo}, P_{C,hi})$ of sizes $n \times n$

if $n = 1$ return $((1), (1))$

{ 1. Compute the index transformation to obtain the half-sized matrices $P_{A,lo}$, $P_{A,hi}$, $P_{B,lo}$, $P_{B,hi}$ }

{ Split P_A into $P_{A,lo}$ and $P_{A,hi}$}
$i_{lo} = \frac{1}{2}$
$i_{hi} = \frac{1}{2}$
for $\hat{\imath} = \frac{1}{2}$ to $n - \frac{1}{2}$
  let $\hat{\jmath}$ s.t. $P_A(\hat{\imath}, \hat{\jmath}) = 1$
  if $\hat{\jmath} > \frac{n}{2}$
    $I_{hi}(i_{hi}) = \hat{\imath}$
    $i_{hi} \leftarrow i_{hi} + 1$
  else
    $I_{lo}(i_{lo}) = \hat{\imath}$
    $i_{lo} \leftarrow i_{lo} + 1$
for all $(\hat{\imath}, \hat{\jmath}) \in \langle 0 : n)^2$ with $P_A(\hat{\imath}, \hat{\jmath}) = 1$
  if $\hat{\jmath} > \frac{n}{2}$
    let $\hat{\imath}'$ s.t. $I_{hi}(\hat{\imath}') = \hat{\imath}$
    $P_{A,hi}(\hat{\imath}', \hat{\jmath} - \frac{n}{2}) = 1$
  else
    let $\hat{\imath}'$ s.t. $I_{lo}(\hat{\imath}') = \hat{\imath}$
    $P_{A,lo}(\hat{\imath}', \hat{\jmath}) = 1$

{ Split P_B into $P_{B,lo}$ and $P_{B,hi}$}
$k_{lo} = 1$
$k_{hi} = 1$
for $\hat{k} = \frac{1}{2}$ to $n - \frac{1}{2}$
  let $\hat{\jmath}$ s.t. $P_B(\hat{\jmath}, \hat{k}) = 1$
  if $\hat{\jmath} > \frac{n}{2}$
    $K_{hi}(k_{hi}) = \hat{k}$
    $k_{hi} \leftarrow k_{hi} + 1$
  else
    $K_{lo}(k_{lo}) = \hat{k}$
    $k_{lo} \leftarrow k_{lo} + 1$
for all $(\hat{\jmath}, \hat{k}) \in \langle 0 : n)^2$ with $P_B(\hat{\jmath}, \hat{k}) = 1$
  if $\hat{\jmath} > \frac{n}{2}$
    let $\hat{k}'$ s.t. $K_{hi}(\hat{k}') = \hat{k}$
    $P_{B,hi}(\hat{\jmath} - \frac{n}{2}, \hat{k}) = 1$
  else
    let $\hat{k}'$ s.t. $K_{lo}(\hat{k}') = \hat{k}$
    $P_{B,lo}(\hat{\jmath}, \hat{k}) = 1$

{ 2. Recursive calls }

{ Compute $P_{C,lo}$ }
$P'_{C,lo} = \text{ImplicitMult}(P_{A,lo}, P_{B,lo})$
for all $(\hat{\imath}, \hat{k}) \in \langle 0 : \frac{n}{2})^2$
  with $P'_{C,lo}(\hat{\imath}, \hat{k}) = 1$
  $P_{C,lo}(I_{lo}(\hat{\imath}), K_{lo}(\hat{k})) = 1$

{ Compute $P_{C,hi}$ }
$P'_{C,hi} = \text{ImplicitMult}(P_{A,hi}, P_{B,hi})$
for all $(\hat{\imath}, \hat{k}) \in \langle 0 : \frac{n}{2})^2$
  with $P'_{C,hi}(\hat{\imath}, \hat{k}) = 1$
  $P_{C,hi}(I_{hi}(\hat{\imath}), K_{hi}(\hat{k})) = 1$

---

return $(P_{C,lo}, P_{C,hi})$

---

we get

$$\delta(i,k) \quad = \quad \sum_{\hat{\imath}\in\langle 0:i\rangle,\hat{k}\in\langle 0:k\rangle} P_{C,hi}(\hat{\imath},\hat{k}) - \sum_{\hat{\imath}\in\langle i:n\rangle,\hat{k}\in\langle k:n\rangle} P_{C,lo}(\hat{\imath},\hat{k}). \qquad (4.18)$$

Based on the sign of δ, we can determine the nonzeros of P_C as follows:

1. If $\delta(\hat{\imath}+\frac{1}{2},\hat{k}+\frac{1}{2}) \leq 0$, we have $P_C(\hat{\imath},\hat{k}) = P_{C,lo}(\hat{\imath},\hat{k})$.
2. If $\delta(\hat{\imath}-\frac{1}{2},\hat{k}-\frac{1}{2}) \geq 0$, we have $P_C(\hat{\imath},\hat{k}) = P_{C,hi}(\hat{\imath},\hat{k})$.
3. If $\delta(\hat{\imath}+\frac{1}{2},\hat{k}+\frac{1}{2}) > 0$ and $\delta(\hat{\imath}-\frac{1}{2},\hat{k}-\frac{1}{2}) < 0$, we have $P_C(\hat{\imath},\hat{k}) = 1$.

Consider two coordinates (i,k) on the two-dimensional plane. We assign different colours to areas in the plane based on the sign of $\delta(i,k)$ as follows. Let $\texttt{Colour}(i,k) = \texttt{red}$ if $\delta(i,k) < 0$, $\texttt{Colour}(i,k) = \texttt{green}$ if $\delta(i,k) = 0$, and $\texttt{Colour}(i,k) = \texttt{blue}$ if $\delta(i,k) > 0$. Also, let $\texttt{Colour}(i,k) = \texttt{red}$ if $i < 0$ or $k < 0$, and $\texttt{Colour}(i,k) = \texttt{blue}$ if $i > n$ or $k > n$. Furthermore, we can define a set of colours for each half-integer point on the plane as the set of the colours of all four adjacent integer pairs. We have $\texttt{ColourSet}(\hat{\imath},\hat{k}) = \{\texttt{Colour}(\hat{\imath}\pm\frac{1}{2},\hat{k}\pm\frac{1}{2})\}$. Using this colouring, we can determine the nonzeros of P_C by using Algorithm 3 to separate its areas.

Since δ is monotonic in both its parameters, we can find the nonzeros in P_C by tracing the upper or lower boundary of the set $\delta^{-1}(\{0\})$, which corresponds to the green area in Figure 4.6. This can be done in linear time by computing values of δ incrementally along a path on the upper or lower boundary of $\delta^{-1}(\{0\})$ (see also Tiskin [2010b]). Algorithm 3 shows how to trace the upper boundary of the area where $\delta(i,k) = 0$. The colours correspond to the three cases shown above: in the red area, we have $P_C(\hat{\imath},\hat{k}) = P_{C,lo}(\hat{\imath},\hat{k})$, and in the blue area, $P_C(\hat{\imath},\hat{k}) = P_{C,hi}(\hat{\imath},\hat{k})$. In the green area, we find nonzeros that correspond to case 3 from above. Since we only advance the values $\hat{\imath}$ and $\hat{k}$ in steps of one, we can use Theorem 3.3.6 to compute all required values of the difference function δ in constant time starting with $\delta(n,0) = 0$. When we have isolated a nonzero that corresponds to case 3, we store its location in list L. Array T contains for each column $\hat{k}$ the respective row $\hat{\imath}$ of the top boundary of the green area.

Algorithm 3 Tracing the top boundary of $\delta^{-1}(\{0\})$

```
procedure Trace_Top (P_{C,lo}, P_{C,hi})
input: A pair of n × n matrices (P_{C,lo}, P_{C,hi})
output: Array T containing the top boundary,
        and a list L of nonzeros from case 3

î = n − 1/2
k̂ = 1/2
while î > 0 and k̂ < n
  if ColourSet(î, k̂) = { red, green, blue }
   { We have discovered a nonzero in P_C . }
    L ← L ∪ {(î, k̂)}
   { we move up, see case (a) in Figure 4.6 }
    î ← î − 1
  else if ColourSet(î, k̂) = { red, green }
   { if the half-integer point above is monochromatic red, we have reached
     the top boundary. }
     if ColourSet(î − 1, k̂) = { red }
      { store top boundary for column k in T[k̂] }
       T[k̂] = î
       { top boundary, go right, see case (b) in Figure 4.6 }
       k̂ ← k̂ + 1
     else { otherwise, we are tracing the left boundary and need to move up.
       See case (c) in Figure 4.6 }
       î ← î − 1
return (T, L)
```

Once we have separated the green area from the red and the blue areas, we can locate the nonzeros in P_C according to the three conditions shown above. The full sequential method as described in Tiskin [2010b] is shown in Algorithm 4. The running time of this algorithm is $O(n \log n)$, since in each recursive step, we take linear time to partition into two half-sized subproblems using Algorithm 2, and we also take linear time to merge the results of these subproblems using Algorithms 3 and 4.

For the parallel version of this algorithm, we assume that the nonzeros of the input matrices are initially distributed arbitrarily, but in equal fractions across p processors. Obtaining a parallel version of Algorithm 2 is straightforward. We execute each step on p processors in parallel, each processor can work independently on the set of nonzeros it holds for splitting the input matrices. In the recursive call,

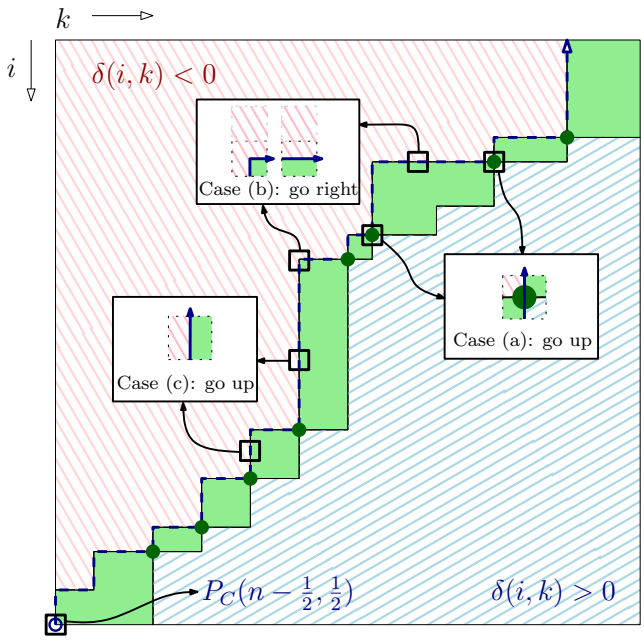


Figure 4.6: Illustration of Algorithm 3

Algorithm 4 Recursive simple unit-Monge Matrix Multiplication

```
procedure ImplicitMult( P_A, P_B )
input: Implicit n × n unit-Monge matrices P_A, P_B
output: An n × n matrix P_C, with P_C^Σ = P_A^Σ ⊙ P_B^Σ

(P_{C,lo}, P_{C,hi}) ← ImplicitMult_Split_Recurse(P_A, P_B)
(T, L) ← Trace_Top (P_{C,lo}, P_{C,hi})

for k̂ ∈ ⟨0 : n⟩
  { Nonzeros from case 3 were stored in list L.
    When we do not have such a nonzero, we use the nonzeros from P_{C,lo} and
    P_{C,hi} according to cases 1 and 2. }
  if there is any î with (î, k̂) ∈ L
      P_C(î, k̂) = 1
  else
  { distinguish cases 1 and 2 using the top boundary of the green area }
   if there is any î with P_{C,lo}(î, k̂) = 1 and î ≤ T[k̂]
      { nonzeros from case 1 are copied from P_{C,lo} }
      P_C(î, k̂) = 1
   else if there is any î with P_{C,hi}(î, k̂) = 1 and î ≥ T[k̂]
      { nonzeros from case 2 are copied from P_{C,hi} }
      P_C(î, k̂) = 1
return P_C
```

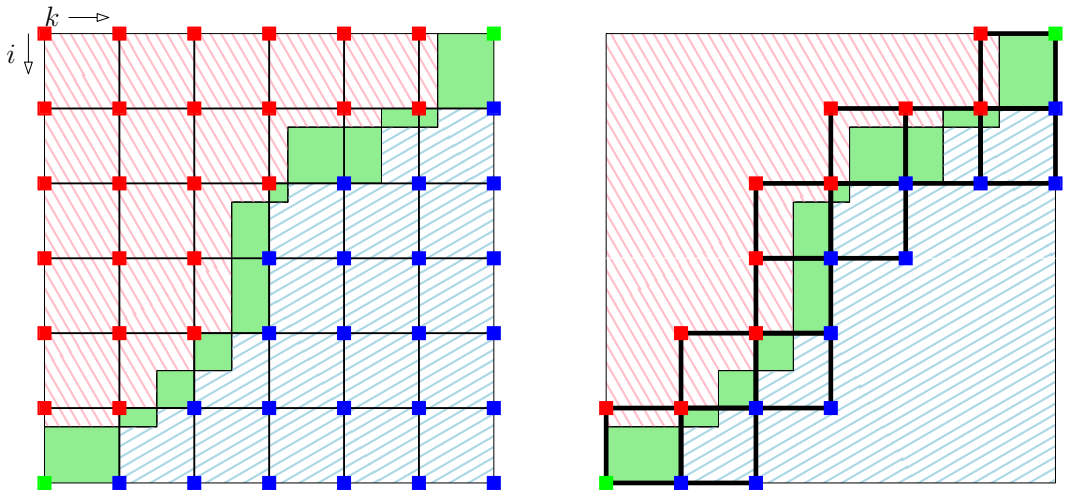


Figure 4.7: Partitioning into a grid of $p \times p$ blocks

we partition the processors into two subsets of size $\frac{p}{2}$, to execute the recursive calls in parallel. After $\log p$ such recursive steps, each processor works on an independent subproblem of multiplying two implicit matrices with $\frac{n}{p}$ nonzeros each. Each processor can solve this subproblem in time $O(\frac{n}{p} \log \frac{n}{p}) = O(\frac{n \log n}{p})$. It remains to merge the resulting $P_{C,lo}$ and $P_{C,hi}$ matrices. We now describe a parallel version of Algorithm 3.

In order to achieve scalable communication, we partition the output matrix P_C into a grid of blocks sized $\frac{n}{p} \times \frac{n}{p}$ (assuming w.l.o.g. that n is a multiple of p). For each such block, we find the colour of its four corners in order to determine the types of the nonzeros contained in it. We compute $\texttt{Colour}(r \cdot \frac{n}{p}, s \cdot \frac{n}{p})$ for all pairs $(r, s) \in [0 : p] \times [0 : p]$. This results in p^2 values that give the value of `Colour` at all intersections of the $p \times p$ grid (see Figure 4.7, left). We call a block for which all the corners have the same colour *monochromatic*. According to these values, we can determine where the nonzeros in each block come from:

- If the block is monochromatic red, all nonzeros within the block are taken from $P_{C,lo}$ since the value of δ is negative throughout the block and therefore case 1 from above applies to all values of P_C inside this block.

- If the block is monochromatic blue, all nonzeros within the block are taken from $P_{C,hi}$ since the value of δ is positive throughout the block and therefore case 2 from above applies to all values of P_C inside this block.
- If the block is monochromatic green, the block cannot contain any nonzeros since it does not contain areas in which δ becomes positive or negative. Therefore, none of the three cases shown above can apply to any value of P_C contained in this block.
- If the block is non-monochromatic, it intersects the boundary of $\delta^{-1}(\{0\})$ (the green area), and we need to trace this boundary through the block to locate nonzeros corresponding to case 3. Furthermore, knowing the intersection of the boundary with the block, we can distinguish the nonzeros from cases 1 and 2 within the block.

Observation 4.3.1. We can have at most $O(p)$ blocks that are non-monochromatic.

Proof. The upper boundary of $\delta^{-1}(\{0\})$ intersects at most $2p$ blocks in our grid, the same is true for the lower boundary. Therefore, the maximum number of non-monochromatic blocks is $4p$ (see also Figure 4.7, right). □

Lemma 4.3.2. *If P_C is of size $n \times n$ and $n > p^3$, we can compute* `Colour`$(r \cdot \frac{n}{p}, s \cdot \frac{n}{p})$ *for all pairs $(r, s) \in [0:p] \times [0:p]$ using $W(n,p) = O(\frac{n}{p})$, $H(n,p) = O(\frac{n}{p})$, $S = O(1)$ and $M(n,p) = O(\frac{n}{p})$.*

Proof. The value of $\delta(i,j)$ can be split into the two sums

$$\begin{aligned} \delta_{lo}(i,k) &= \textstyle\sum_{\hat{\imath} \in \langle i:n \rangle, \hat{k} \in \langle k:n \rangle} P_{C,lo}(\hat{\imath}, \hat{k}), \text{ and} \\ \delta_{hi}(i,k) &= \textstyle\sum_{\hat{\imath} \in \langle 0:i \rangle, \hat{k} \in \langle 0:k \rangle} P_{C,hi}(\hat{\imath}, \hat{k}). \end{aligned} \tag{4.19}$$

We can compute the values δ_{lo} and δ_{hi} on a $p \times p$ grid using parallel prefix in BSP time $W(n,p) = O(\frac{n}{p})$ and communication $O(p^2)$ in a constant number of supersteps. This can be implemented by distributing the nonzeros of $P_{C,lo}$ and $P_{C,hi}$ in "strips" of size $\frac{n}{p} \times n$. Processor q, $q \in [1:p]$, receives all nonzeros $(\hat{\imath}, \hat{k})$ with $(q-1) \cdot \frac{n}{p} < \hat{\imath} \leq q \cdot \frac{n}{p}$ in

a single superstep using communication $O(\frac{n}{p})$. Then, each processor q can compute p values

$$\begin{aligned}
\delta'_{q,lo}(r) &= \textstyle\sum_{\hat{\imath},\hat{k}} P_{C,lo}(\hat{\imath},\hat{k}), \text{ and} \\
\delta'_{q,hi}(r) &= \textstyle\sum_{\hat{\imath},\hat{k}} P_{C,hi}(\hat{\imath},\hat{k}), \text{ with } r \in [1:p], \\
&\qquad \hat{\imath} \in \langle (q-1)\cdot\tfrac{n}{p} : q\cdot\tfrac{n}{p}\rangle, \text{ and} \\
&\qquad \hat{k} \in \langle (r-1)\cdot\tfrac{n}{p} : r\cdot\tfrac{n}{p}\rangle. \qquad (4.20)
\end{aligned}$$

We have p^2 values each for $\delta'_{q,lo}$ and $\delta'_{q,hi}$, one value for each grid intersection, which we broadcast to all processors. After the broadcast, each processor can evaluate $\texttt{Colour}(r\cdot\frac{n}{p}, s\cdot\frac{n}{p})$ for all grid points $(r,s) \in [0:p]\times[0:p]$ in time $O(p^2)$. If $n > p^3$, we have $O(\frac{n}{p}+p^2) = O(\frac{n}{p})$, therefore, we get the claimed bounds on computation and communication. □

Lemma 4.3.3. *Given the nonzeros of two $n\times n$ permutation matrices P_A and P_B, distributed equally across $p < \sqrt[3]{n}$ processors, we can compute the nonzeros of a matrix P_C with $P_C^\Sigma = P_A^\Sigma \odot P_B^\Sigma$ using $W(n,p) = O(\frac{n\log n}{p})$, $H(n,p) = O(\frac{n}{p}\log p)$, $M(n,p) = O(\frac{n}{p})$, and $S = O(\log p)$.*

Proof. We have $W(n,p) = O(\frac{n\log n}{p})$ due to the last recursive step in the parallel version of Algorithm 2. Afterwards, we combine the resulting $P_{C,lo}$ and $P_{C,hi}$ matrices in $\log p$ supersteps until we have obtained a distributed version of P_C. At level l of this merging tree, we have $r = p/2^l$ processors merging matrices of size $\frac{n}{2^l}\times\frac{n}{2^l}$. This can be done in a constant number of supersteps using $W(n,p) = O(\frac{n}{p})$, $M(n,p) = H(n,p) = O(\frac{n}{p})$ due to Lemma 4.3.2. We have $\log p$ such levels, which gives the claimed bounds. □

This new parallel algorithm is work-optimal w.r.t. the sequential method shown in Tiskin [2010b], and has scalable communication within a log-factor of the optimum. Our algorithm has cubic slackness, requiring $n > p^3$ since otherwise, the work for computing the values of `Colour` in the proof of Lemma 4.3.2 becomes dominant. However, this poses no realistic restriction on the problem sizes even on hundreds of

thousands of processors – the expected problem size to justify the use of a parallel system of that scale will easily be large enough. The only criterion by which the algorithm from Section 4.2 is superior to our new method is the number of required supersteps: in Section 4.2, only a constant number of supersteps is required. Our new algorithm could be adapted to run in a constant number of supersteps, however, this would introduce an exponential slackness condition which would render the method impractical. The algorithm from this section is also much simpler than the previous algorithm shown in Section 4.2, and is very likely to be practical. In the next sections, we will study two applications of this algorithm.

4.4 Parallel LCS computation

Semi-local string comparison is useful for obtaining efficient parallel algorithms for LCS computation (see Apostolico et al. [1990]; Alves et al. [2006]; Krusche and Tiskin [2007]). The sequential highest-score matrix multiplication procedure shown by Tiskin [2010a] can be used to derive parallel algorithms that solve the semi-local LCS problem by partitioning the alignment dag into p strips. The problem is solved independently on one processor for each strip using dynamic programming to compute the implicit highest-score matrices using the seaweed algorithm. After this, we merge the resulting highest-score matrices in a binary tree of height $\log p$. This procedure requires data of size $O(n)$ to be sent by every processor in every level of the tree, and the sequential merging requires time $O(n^{1.5})$ for computing the resulting highest-score matrix. The problem with this simple approach is that it does not achieve scalable communication, since $O(n)$ items of data must be transferred.

We now show how to achieve scalable communication using the algorithm from Section 4.3.

Theorem 4.4.1. *Given two strings x and y of length n which are distributed in equal fractions between p processors, we can compute a distributed version of their implicit highest-score matrix $A_{x,y}$ using $W(n,p) = O(\frac{n^2}{p})$, $H(n,p) = O(\frac{n}{\sqrt{p}} + \frac{n \log^2 p}{p}) = O(\frac{n}{\sqrt{p}})$, $M(n,p) = O(\frac{n}{\sqrt{p}})$, and $S = O(\log^2 p)$.*

Proof. As the first step, each processor compares a single pair of substrings from $x_{(q-1)\cdot\frac{n}{\sqrt{p}}} \dots x_{q\cdot\frac{n}{\sqrt{p}}}$ and $y_{(r-1)\cdot\frac{n}{\sqrt{p}}} \dots y_{r\cdot\frac{n}{\sqrt{p}}}$ with $q, r \in [1 : \sqrt{p}]$. This requires computation work $O(\frac{n^2}{p})$. After that, our unit-Monge matrix multiplication algorithm is used at every level of a quadtree-like merging step to compute distributed implicit highest-score matrices for longer pairs of substrings. At the bottom level, the matrices are merged sequentially, requiring time $O(\frac{n}{\sqrt{p}} \log \frac{n}{\sqrt{p}}) = O(\frac{n \log n}{\sqrt{p}})$. At higher levels of the quadtree, blocks are merged in parallel. In particular at level $\log r$, $1 \leq r \leq p$, the block size is $\frac{n}{\sqrt{r}}$, and each merge is performed by a group of p/r processors using computation time $O(\frac{\frac{n}{\sqrt{r}} \log \frac{n}{\sqrt{r}}}{p/r})$. The sum of the running times over all $\log p$ levels of the merging phase is equal to $O(\frac{n \log n}{\sqrt{p}})$. Therefore, the overall running time $W(n,p) = O(\frac{n^2}{p} + \frac{n \log n}{\sqrt{p}} \cdot \log p) = O(\frac{n^2}{p})$. The communication and memory requirements are dominated by each processor having to read and store the data for the two substrings of length $\frac{n}{\sqrt{p}}$ it needs to compare, which gives $H(n,p) = M(n,p) = O(\frac{n}{\sqrt{p}})$. Overall, we need $S = O(\log^2 p)$ supersteps since each level $\log r$ in the merging tree requires $\log p/r$ supersteps to execute. □

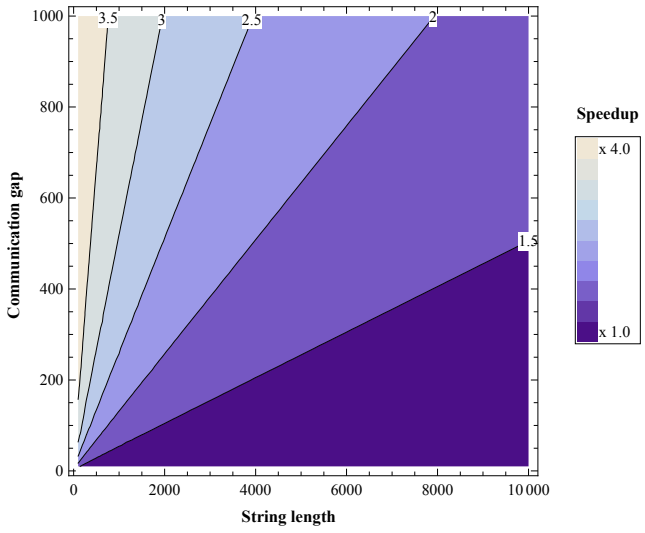


Figure 4.8: Potential speedup for LCS computation through scalable communication, $p = 16$

The communication cost of this new algorithm is dominated by the cost required to read the input strings at the boundary of the $\sqrt{p} \times \sqrt{p}$ grid blocks.

It seems that this might be a natural lower bound, which prohibits improving the communication performance any further. However, a proof of such a lower bound does not seem trivial.

Performance improvements over the standard grid dag method can be expected on parallel machines with a relatively low-bandwidth communication network where the time for communication can become the dominant part of the computation. In practice, communication cycles can take be between 100 and 1000 times longer than computation cycles (see Krusche and Tiskin [2006] for an experimental study). In this case, using our new algorithm could give a speedup factor between 1 and $\sqrt{p}$ over using the standard grid dag approach. We can model the speedup over the grid dag approach as follows. Let g be the communication gap from the standard BSP model, i.e. g specifies how long communication takes compared to computation. We have the running time for LCS computation using the grid dag method as

$$T_{dag}(n,p) = \frac{n^2}{p} + g \cdot n, \tag{4.21}$$

and the running time using our new algorithm

$$T_{new}(n,p) = \frac{n^2}{p} + \frac{n \log n}{\sqrt{p}} + g \cdot \frac{n}{\sqrt{p}}. \tag{4.22}$$

The speedup is estimated as

$$S(n) = \frac{T_{dag}(n,p)}{T_{new}(n,p)}. \tag{4.23}$$

Figure 4.8 shows the potential speedup over the grid dag method when using our new parallel algorithm on 16 processors. The maximal theoretical speedup that can be obtained from using our method on such a system is 4. The communication gap values are chosen between 100 and 1000, estimating that communicating a single integer would take between 100 and 1000 times longer than evaluating a cell in the seaweed algorithm. Such values would be realistic e.g. for reading the sequence data from main memory in an SMP system (which is usually around 100 times slower

than reading from L1 cache or a register, see Fog [2010]), or when communicating the data using a network, which is much slower (see Skillicorn et al. [1997]; Krusche [2005]; Krusche and Tiskin [2006] for performance measurements). We see from our figure that speedup of up to the theoretical maximum of $\sqrt{p}$ is possible within this estimation, and that exploiting scalable communication can in fact be used to improve the performance for LCS computation with reasonable problem sizes.

4.5 Parallel permutation string comparison

Another application of our algorithm is parallel computation of longest increasing subsequences. Using our new algorithm for multiplication of simple unit-Monge matrices, we can get within a $\log n$-factor of work-optimality for this problem. We use iterated application of highest-score matrix composition to solve the problem of semi-local *permutation string comparison*. In permutation string comparison, we have $|x| = |y| = |\Sigma| = n$, and x and y each contain exactly one occurrence of every character. The LIS problem for a string x can be solved by observing that any LCS of x and the sequence of all characters from Σ in ascending order is a LIS of x. We can also consider *n-permutation substrings* which are defined as arbitrary substrings of a permutation string of length n. We say that two permutation substrings are character-disjoint if they consist of disjoint sets of characters.

We observe that the alignment dag for two permutation strings of length n contains exactly n match cells. Therefore, we can show that we can read the input and generate the set of match cells in the alignment dag using perfectly scalable memory and communication. This is possible because our input strings are permutation strings. Each character in the alphabet is responsible for exactly one match cell. Therefore, we can read and distribute the input more efficiently than in the general LCS case.

Lemma 4.5.1. *The set of match cells for two permutation strings of length n can be obtained using time $W(n,p) = O(\frac{n \log n}{p})$, as well as scalable communication and memory $H(n,p) = M(n,p) = O(\frac{n}{p})$.*

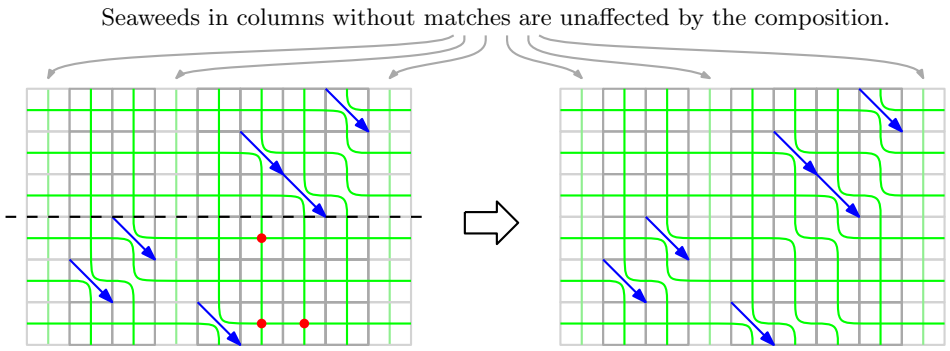


Figure 4.9: Highest-score matrix composition for permutation strings

Proof. We would like to show that we only need to store a fraction of n/p of both input strings on each processor. This is obviously the case for input string x, which is partitioned into disjoint substrings of length n/p. However, we must still show that it is not necessary for each processor to store the entire other input string y. Initially, each processor $q \in [1:p]$ holds substrings $x_{(q-1)\cdot\frac{n}{p}+1} \ldots x_{q\cdot\frac{n}{p}}$ and $y_{(q-1)\cdot\frac{n}{p}+1} \ldots y_{q\cdot\frac{n}{p}}$. In order to determine the position of each character from $x_{(q-1)\cdot\frac{n}{p}+1} \ldots x_{q\cdot\frac{n}{p}}$ in y, we need to find the sorting permutation of y, after which each processor can retrieve the positions of all characters in its fraction of x. In the comparison-based model, this is possible in BSP cost $W(n,p) = O(\frac{n \log n}{p})$, $H = O(\frac{n}{p})$ and $S = O(1)$ supersteps e.g. using parallel sorting by regular sampling (see Shi and Schaeffer [1992]; Tiskin [1998]). After sorting, we can redistribute the data using communication $O(n/p)$ such that each processor holds all matches for matching its substring of x against y. At no point in this initial sorting and redistribution procedure do we need to store more than $O(\frac{n}{p})$ elements of data on any processor. □

In our permutation string comparison algorithm, we partition the alignment dag into strips of height $\frac{n}{p}$. When using highest-score matrix composition to combine two such strips, we observe that we only need to multiply simple unit-Monge matrices of size $\frac{n}{p} \times \frac{n}{p}$.

Lemma 4.5.2. *Consider highest-score matrix composition for two character-disjoint n-permutation substrings x and y, both of length m, and a permutation string z of*

length n. We can compute the implicit representation of highest-score matrix $A_{xy,z}$ from the implicit representations of $A_{x,z}$ and $A_{y,z}$ by computing a $(\min, +)$ product of two $2m \times 2m$ highest-score matrices.

Proof. Since x and y are permutation substrings, each column in the alignment dags $G_{x,z}$ and $G_{y,z}$ contains either a single match cell, or no match cells at all. Moreover, since x and y are character-disjoint, the combined alignment dag $G_{xy,z}$ cannot have columns with more than one match. Consider the behaviour of the seaweed algorithm on $G_{xy,z}$. Each column in $G_{x,z}$ that does not contain a match will only contain seaweed crossings. Therefore, all seaweeds which start in a column without a match in $G_{xy,z}$ cannot double-cross any other seaweeds since they start and end in the same column in $G_{x,z}$, $G_{y,z}$ and $G_{xy,z}$. Only $2m$ seaweeds can have non-trivial double-crossings. These double-crossings can be resolved by computing a $(\min, +)$-product for two matrices of size $2m \times 2m$. All other seaweeds are unaffected by the concatenation of the alignment dags. An illustration is shown in Figure 4.9 □

Theorem 4.5.3. *The semi-local LCS problem for permutation strings of length n can be solved on a BSP computer using $W(n,p) = O(\frac{n \log^2 n}{p})$, $H(n,p) = O(\frac{n \log p}{p})$, $S = O(\log^2 p)$ and $M(n,p) = O(\frac{n}{p})$.*

Proof. We partition one of the input strings into substrings of length n/p, and compute the highest-score matrix for each of these substrings compared to the other input string in parallel in time $O(\frac{n \log^2 n}{p})$ (see Tiskin [2010b]), which is the computationally dominant part of this algorithm. This is possible because the inputs are permutation strings, and therefore only $\frac{n}{p}$ character matches exist for each substring. We can therefore apply Lemma 4.5.2 to reduce the computation to an input of size $\frac{n}{p}$. We then start merging the highest-score matrices in parallel using our new parallel highest-score matrix multiplication algorithm. Consider level $l \in [0 : \log p]$ of the merging process, and let $r = 2^l$. We again use the fact that the resulting highest score matrix for comparing a permutation string and a permutation-substring of length $2rn/p$ can be stored in $O(2rn/p)$ space. Furthermore, each merge is performed by

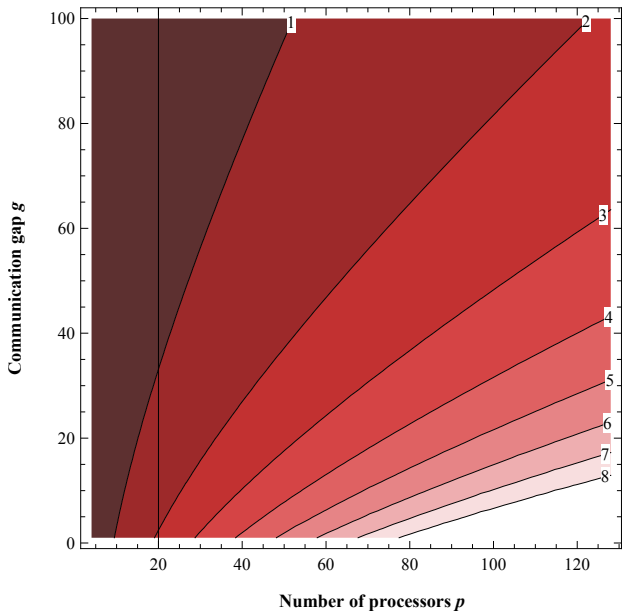


Figure 4.10: Estimated LIS computation speedup if input is known to all processors, input sequence length fixed as $n = 10000$

a group of $2r$ processors. By taking the sum over all values r corresponding to the $\log p$ levels of the merging process, we get computation time

$$O\left(\sum_r \left(\frac{2rn\log(2rn/p)}{p}\right)/(2r)\right) = O\left(\frac{n\log n}{p}\right),$$

and communication cost

$$O\left(\sum_r \frac{2rn}{p}/(2r)\right) = O\left(\frac{n\log p}{p}\right)$$

for the merging phase. This analysis includes the top level of the merging tree where $r = p$. The number of supersteps $S = O(\log^2 p)$, as the merging tree contains $O(\log p)$ levels which require $O(\log p)$ supersteps each. □

We have now shown that our algorithm is scalable in computation, communication and memory for semi-local permutation string comparison. However, the question remains whether we can expect to achieve speedup for LIS computation with reasonable problem sizes. For estimating this speedup, consider the following

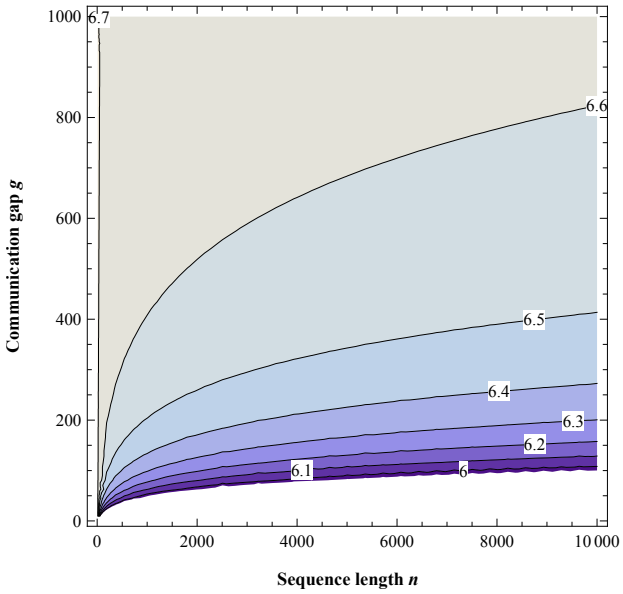


Figure 4.11: Estimated LIS computation speedup for distributed input strings, number of processors fixed as $p = 16$, record size $r = 2$

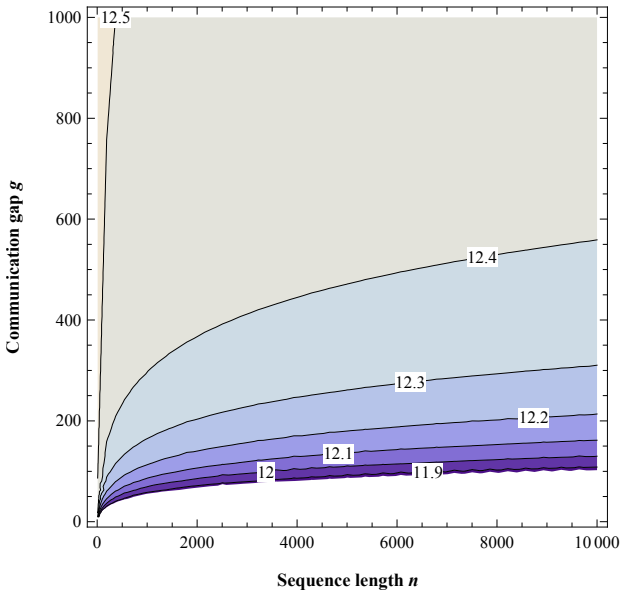


Figure 4.12: Estimated LIS computation speedup for distributed input strings, number of processors fixed as $p = 16$, record size $r = 10$

simple performance model. We compute the LIS of a sequence with n elements using p processors. We assume that we have a *record size* r, which specifies the size of each sequence element. We consider two different input data distributions.

1. In the first scenario, we assume that the entire sequence is known to every processor before starting the computation. We define the numbers of sequence elements which need to be transferred between processors before our computation as $b_{par} = b_{seq} = 0$. In this case, the only way for our parallel algorithm to run faster than the sequential method is by achieving scalability for computation on many processors. Its scalable communication property is not an advantage over the sequential method anymore, since the sequential algorithm can run without any communication in this case.

2. In the second scenario, we assume that each processor initially holds an equal fraction of the sequence. This setup is favourable for our parallel algorithm, since our sequential algorithm will need to first collect the input sequence data to a single processor which will then perform the computation. In practice, this case would occur if the sequence is a result of a previous parallel computation, or if the sequence consists of large data records which cannot all be stored on a single processor. In our model, we have to add $b_{seq} = n$ data elements to the communication cost, since these need to be distributed to all processors first. In the parallel algorithm, we need only to compensate for having a distributed input by adding a term $b_{par} = r\frac{n}{p}$, as each processor only needs to read an $O(\frac{n}{p})$-sized fraction of the input.

Let g be the communication gap from the standard BSP model. The running time for the sequential computation is then given by

$$t_{seq}(n) = n \log n + g b_{seq}. \tag{4.24}$$

We neglect constant factors, as we are only looking at the performance ratio compared to the parallel algorithm. The running time of the parallel computation using

p processors is

$$t_{par(p)}(n,p) = \frac{n \log^2 n}{p} + g \cdot \left(b_{par} + \frac{n}{p} \log p \right). \tag{4.25}$$

We compute the speedup

$$S(p) = \frac{t_{seq}(n)}{t_{par}(n,p)}. \tag{4.26}$$

If the input sequence is known to all processors in advance, we fix the sequence length and see if our algorithm can achieve scalability on different numbers of processors. Figure 4.10 shows estimations for this case, fixing the sequence length to 10000 elements. We see that our algorithm can still achieve speedup over the sequential method, however requiring a comparatively large number of processors. Applications which correspond to this case would therefore not benefit much from using our new algorithm, although at least some speedup can be obtained.

In the second scenario with a distributed input sequence, we look at running the algorithm on sequences of different lengths on 16 processors. This can correspond to a large SMP-style system for small values of g (values between 10 and 100 would be typical), or to a small cluster system, which would have higher values of g. Figure 4.11 shows the estimated speedups for a range of string lengths and communication gap parameters, assuming that we have a record size of $r = 2$. This corresponds to computing the LIS of a sequence of double-precision floating point values: each input sequence element requires 64 bits of space, and we can perform our highest-score matrix computations using 32-bit integers. In Figure 4.12, we evaluate the same scenario for a larger record length of $r = 10$ (this could occur e.g. in database applications). We see that using our parallel algorithm becomes beneficial if the input sequences become longer, and if the communication gap is large. Furthermore, our algorithm can potentially achieve very good speedup when the record size is large. We conclude that in the scenario where we would like to compute an LIS of a sequence that is pre-distributed in equal fractions between processors (either in separate processor caches or main memory in an SMP system,

or on different nodes of a cluster system), our new algorithm is very likely to be practically scalable.

In conclusion, we would like to point out that these speedup estimations only serve the purpose of evaluating the practical potential of a stronger theoretical result. Our main aim is to show a first approach to parallel LIS computation which is scalable, and pointing out that despite the LIS problem being well-studied, no algorithms had been proposed so far which are scalable and can compete with the fastest sequential method. Improving our algorithm to achieve work-optimality would greatly improve its potential for practical application.

Chapter 5

Parameterized Semi-local String Comparison

In this chapter, we present new algorithms for comparing highly similar or highly dissimilar strings. For the standard LCS problem, many such algorithms exist. We show how these existing algorithms can be obtained using semi-local string comparison. Most of these algorithms are limited to global LCS computation, i.e. computing the LCS for the input strings only. We show new parameterized algorithms in this chapter which compute semi-local string alignments as introduced in Chapter 3, and are therefore usable as a plug-in to speed up the parallel algorithms from Chapter 4.

5.1 Background

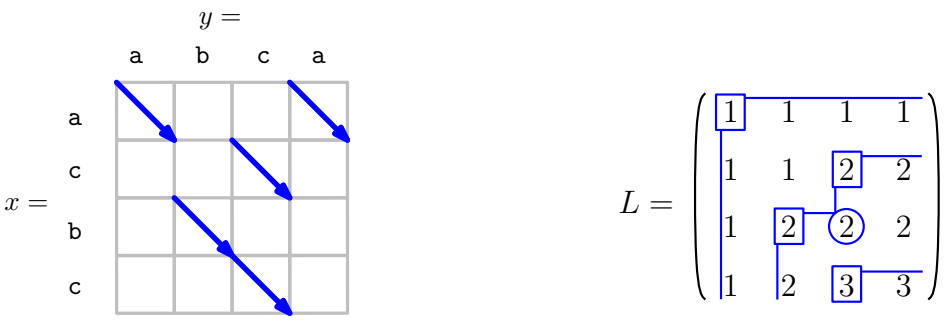
In this chapter, we look at parameterized algorithms for semi-local string comparison, which run faster either for highly similar or highly dissimilar input strings. A multitude of such algorithms have been shown for the LCS problem, the best known example being perhaps the algorithm by Hunt and Szymanski [1977], which has been used e.g. to implement the `diff` command in Unix operating systems (Free Software Foundation (FSF) [2010b]). We will show how traditional parameterized

algorithms for the LCS problem can be understood using a simplified variant of the seaweed algorithm, and also show parameterized algorithms for improving the performance of semi-local string comparison itself.

More efficient special case algorithms for the LCS problem can be obtained when parameterizing either by the number r of match cells in the alignment dag, by the length p of the LCS, or by the edit distance. Previously, high-similarity string comparison has been considered by Hirschberg [1977]; Apostolico and Guerra [1987]; Nakatsu et al. [1982]; Ukkonen [1985]; Myers [1986]; Kumar and Rangan [1987]; Wu et al. [1990]; Rick [1995, 2000]. All these papers give LCS algorithms for highly similar strings, running in time $O(ne)$, where e is either the edit distance between the strings (as shown by Ukkonen [1985]), or a different closely related similarity measure. High-dissimilarity string comparison has been considered by Hirschberg [1977]; Rick [1995, 2000]; the best running time for LCS on highly dissimilar strings is $O(np + n \log n)$. A good survey of parameterized string comparison algorithms is given by Hirschberg [1997].

The basis of parameterized LCS computation for dissimilar strings is to determine the LCS of two strings as a longest *chain* of match cells $(i_1, j_1), (i_2, j_2), \ldots, (i_p, j_p)$ with $i_1 < i_2 < \ldots < i_p$ and $j_1 < j_2 < \ldots < j_p$. We define a partial order on the set of match cells by $(i_1, j_1) \prec (i_2, j_2)$ if and only if $i_1 < i_2$ and $j_1 < j_2$; further, we say that (i_1, j_1) is dominated by (i_2, j_2). Due to Dilworth [1950], the minimum number of antichains (sets of pairwise incomparable elements) necessary to cover a partially ordered set is equal to the length of the longest chain. Therefore, the LCS of two strings can be obtained by computing a minimal antichain decomposition of the set of matches under the $\prec$ ordering. Consider chains ending at a match (i, j). If any longest such chain has length k, then this match is said to have *rank* k. If match (i, j) has rank k and for all other matches (i', j') of rank k either $i' \geq i$ and $j' < j$ or $j' \geq j$ and $i' < i$, then match (i, j) is called (k-)dominant.

The set of all dominant matches completely specifies the table of prefix-prefix LCS lengths $L(i, j) = LLCS(x_1 \ldots x_i, y_1 \ldots y_j)$. Let the contours of L be formed by the rows and columns of cells through which the values of L increase



(a) input strings and alignment dag (b) prefix-prefix LCS lengths and contours

Figure 5.1: Parameterized LCS Computation

by one. A cell $(\hat{\imath}, \hat{\jmath})$ belongs to a contour in L if $L(\hat{\imath} + \frac{1}{2}, \hat{\jmath} + \frac{1}{2}) > L(\hat{\imath} - \frac{1}{2}, \hat{\jmath} + \frac{1}{2})$, $L(\hat{\imath} + \frac{1}{2}, \hat{\jmath} + \frac{1}{2}) > L(\hat{\imath} + \frac{1}{2}, \hat{\jmath} - \frac{1}{2})$, or $L(\hat{\imath} + \frac{1}{2}, \hat{\jmath} + \frac{1}{2}) > L(\hat{\imath} - \frac{1}{2}, \hat{\jmath} - \frac{1}{2})$. Figure 5.1 (b) shows an example. All match cells belonging to the same contour form an antichain in a minimal antichain decomposition, and each contour is specified completely by the dominant matches on it[1].

Since parameterized algorithms process the input match-by-match instead of computing the entire prefix-prefix LCS score matrix, it is necessary to pre-process the input strings to obtain lists of match cells. Different approaches exist for this, depending on the assumptions that can be made about the input alphabet. Generally, it is necessary to allow less-than/greater-than comparisons in addition to testing for equality (otherwise, $\Omega(mn)$ was shown to be a lower bound, see Aho et al. [1976]). Based on this assumption, we can obtain a set of match lists which give for every character c in x the positions i where $y_i = c$ in $O(n \log n)$ time. These lists usually allow queries for increasing or decreasing sequences of i-values and are called occurrence lists or match lists accordingly. The lists are obtained by determining the inverse sorting permutation for y (i.e. a permutation that transforms a sequence which contains all characters from y in sorted order into y). For every character c in x, we can find the head of a list of match positions in time $O(\log n)$ by binary search. For small alphabets, it is possible to pre-process the input in

[1]These contours are called forward contours by Rick [1995, 2000].

time $O(n \log \sigma)$ to obtain a similar representation (see Hirschberg [1977]; Apostolico [1997] for discussion). We will denote the result of this preprocessing as follows.

Definition 5.1.1. The functions $\mu_i : \mathbb{N} \to [1:n] \cup \infty$ for $i \in [1:m]$ specify the match positions. We have for all $k \in [1:n-1]$:

- $\mu_i(k) = j, j \neq \infty \Rightarrow x_i = y_j$,
- $\mu_i(k) < \mu_i(k+1)$ or $\mu_i(k) = \mu_i(k+1) = \infty$.

This notation allows storing the match lists using $O(m+n)$ space. Algorithm 5 shows how to obtain these functions for arbitrary ordered alphabets in time $O(n \log n)$. We can obtain these functions for arbitrary ordered alphabets in time $O(n \log n)$ by sorting one of the input strings and then using binary search to create the match lists. For small alphabets of size $\sigma < n$, the sorting permutation can be determined in time $O(n \log \sigma)$ by counting character frequencies for all characters contained in y. After this pre-processing step, we can determine $\mu_i(k)$ in $O(1)$ time using $O(m+n)$ storage.

Algorithm 5 Pre-processing x and y to obtain μ_i

```
input: strings x and y with |x| = m and |y| = n
output: functions μ_i with i ∈ [1 : m]

Obtain sorting permutation Y of y {i.e.  y_{Y[j]} ≤ y_{Y[j+1]} with j ∈ [1 : n]}
for i ∈ [1 : m] do
  find j_i^min = min_j y_{Y[j]} = x_i and j_i^max = max_j y_{Y[j]} = x_i
   { by binary search using Y, time O(log n) }
  μ_i(k) = { Y[j_i^min + k − 1]   if  k ∈ [1 : j_i^max − j_i^min + 1]
           { ∞                    otherwise.
end for
```

In this chapter, we develop a new interpretation of standard and semi-local LCS algorithms, based on a certain class of traditional comparison networks known as transposition networks. This approach allows us to obtain new algorithms for sparse semi-local string comparison and for comparison of highly similar and highly dissimilar strings, as well as semi-local comparison of run-length compressed strings. The remainder of this chapter is structured as follows. We describe the transposition

network method in Section 5.2. We then show new algorithms for sparse semi-local string comparison in Section 5.3, show how to compare run-length compressed strings semi-locally in Section 5.4, and discuss comparing highly similar or highly dissimilar strings in Section 5.5. We finally give a new, transposition-network based algorithm for highest-score matrix composition in Section 3.7.

5.2 The transposition network method

Comparison networks (see e.g. Cormen et al. [2001]) are a traditional method for studying oblivious algorithms for sorting sequences of numbers. A *comparison network* has n inputs and n outputs, which are connected by an arbitrary number of *comparators*. A comparator has two inputs and two outputs. It compares the input values and returns the larger value at a prescribed output, and the smaller value at the other output. We will draw comparison networks as n wires, where pairs of wires may be connected by comparators that operate on the values passing through the wires. Comparators are usually grouped into a sequence of k *stages*, where each wire is connected to at most one comparator in a single stage. A comparison network is called *transposition network* if all comparators only connect adjacent wires.

Transposition networks allow for another interpretation of the seaweed algorithm. As shown in Figure 5.3, every mismatch cell behaves like a comparator on the starting points of the seaweeds that enter the cell from the left and the top. The larger value is returned on the right output, and the smaller value is returned on the bottom output. For a match cell, the input values are not compared but just translated top to right and left to bottom. Therefore, we can define a transposition network for every problem instance as follows.

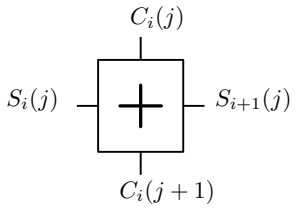
Definition 5.2.1. The network LCSNET(x, y) has $m + n$ diagonal wires. Every mismatch cell $(\hat{\imath}, \hat{\jmath})$ corresponds to a comparator in stage $\hat{\imath} + \hat{\jmath}$ connecting wires $m - \hat{\imath} + \hat{\jmath}$ and $m - \hat{\imath} + \hat{\jmath} + 1$ (see Figure 5.3). Match cells do not contain comparators.

As comparators in the network correspond to cells in the alignment dag, we choose the convention of drawing the network wires top left to bottom right. Values moving through a cell or comparator can therefore move either down or to the right.

The network LCSNET(x, y) realizes the seaweed algorithm. At the beginning, the input values are in inversely sorted order in relation to the direction of the comparators. As these values pass through the network, they trace the paths of the seaweed curves in the alignment dag. Note that the direction of the comparators can be determined arbitrarily, as long as it is opposite to the sorting of the input sequence. Another degree of freedom when defining transposition networks lies in the behaviour of comparators for equal inputs. Even though this does not affect the network output, changing the convention of swapping or not swapping equal values can simplify specification of non-oblivious algorithms for computing the output of LCSNET(x, y).

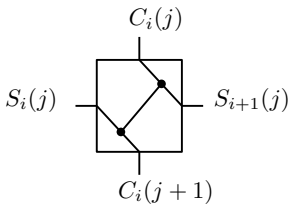
In order to solve the global or semi-local LCS problem for strings x and y using the transposition network method, we have to define appropriate input values for LCSNET(x, y). In order to obtain the full set of nonzeros of the implicit highest-score matrix, the inputs are set to the seaweed starting points: input $\hat{\jmath} + m + \frac{1}{2}$ is initialized with $\hat{\jmath}$, $\hat{\jmath} \in \langle -m : n \rangle$. Let the vector O denote the output of the network. If all comparators return the larger input on the bottom output, and the smaller input on the right output, the pairs $(O(\hat{\jmath}+m+\frac{1}{2}), \hat{\jmath}+m)$ with $\hat{\jmath} \in \langle -m : n \rangle$ correspond to the core nonzeros of the corresponding implicit highest-score matrix. Since there are $O(mn)$ comparators in the transposition network, the resulting algorithm runs in time $O(mn)$.

Using the transposition network method, we can see the connection between semi-local string comparison and existing LCS algorithms is the fact that both approaches compute LCS scores incrementally for prefixes of the input strings. The standard LCS dynamic programming approach computes LCS lengths, and the seaweed algorithm computes implicit highest score matrices for all prefixes of the input strings. When looking at this relationship in more detail, it becomes clear that standard LCS algorithms can be obtained by the transposition network method



Truth table

$S_i(j)$	$C_i(j)$	$S_{i+1}(j)$	$C_{i+1}(j)$
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1



Truth table

$S_i(j)$	$C_i(j)$	$S_{i+1}(j)$	$C_{i+1}(j)$
0	0	0	0
1	0	1	0
0	1	1	0
1	1	1	1

Figure 5.2: Bit-parallelism through addition in 0-1-transposition networks

using input values of only zero or one. A first direct consequence are bit-parallel LCS algorithms as given by Crochemore et al. [2001, 2003], which can be obtained by computing the output of the transposition network cell-column by cell-column using bit-vector boolean operations and bit-vector addition. Figure 5.2 shows that the truth tables for a full adder and conditional binary comparison are almost the same and differ only in one bit value. The mismatching value can be corrected by Boolean manipulation, obtaining the same algorithm as was given by Crochemore et al. [2001]. In the remainder of this chapter we will show further examples where existing algorithms for comparing two strings globally can be derived from transposition networks, and discuss generalizing them to semi-local string comparison.

5.3 Sparse semi-local string comparison

We now consider *sparse string comparison*, i.e. string comparison parameterized by the number of matches r in the alignment dag. Hunt and Szymanski [1977] proposed an algorithm for sparse string comparison that computes the LCS of two input strings in $O((r+n)\log n)$ time. An extreme case of this is the comparison of permutation strings of length n over the alphabet $\Sigma = [1:n]$. In this case, only n match cells exist. Tiskin [2006] gave an $O(n^{1.5})$ algorithm for semi-local comparison

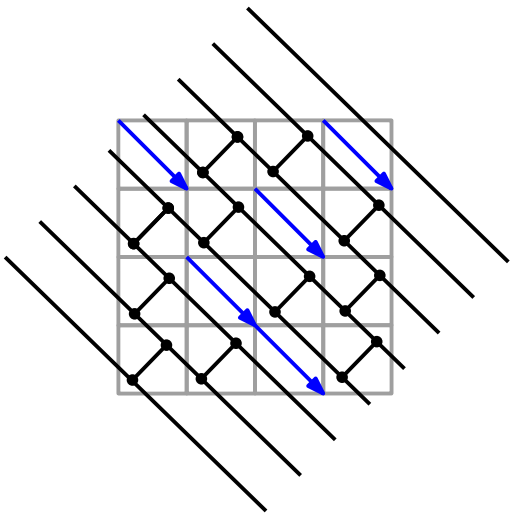


Figure 5.3: Comparison network of an alignment dag

of permutation strings, and improved this result to $O(n \log^2 n)$ later (see Tiskin [2010b]).

Since in sparse string comparison the alignment dag contains few matches, large rectangular areas of the transposition network have full sets of comparators. These areas will be denoted as follows.

Definition 5.3.1. Let network DIAMOND(m, n) be defined as an LCSNET network which corresponds to a problem instance with no matches. It therefore contains a full set of $m \cdot n$ comparators.

We now give a more general sparse semi-local string comparison algorithm parameterized by the number of matches. We will first show a non-oblivious algorithm to compute the output of DIAMOND networks efficiently, and then propose a technique for evaluating a LCSNET network by partitioning it into smaller DIAMOND networks.

Consider an $m' \times n'$ rectangular area in the alignment dag with only mismatch cells, and the corresponding DIAMOND(m', n') network. Such an area occurs whenever two substrings over disjoint character sets are compared. The network consists

of a full set of $m' \times n'$ comparators and $m' + n'$ wires.[2] If the first m' and the following n' wires are initialized with two pre-sorted sequences of numbers, this network works as a merging network (see Munter [1993]). The problem of merging pre-sorted sequences can be solved non-obliviously in time $O(m' + n')$. However, as the inputs to the DIAMOND network are not necessarily pre-sorted, this is not sufficient.

Theorem 5.3.2. *It is possible to compute the outputs of the* DIAMOND *network non-obliviously in time* $O((m' + n') \log(m' + n'))$ *if the inputs are in arbitrary order. Additionally, if the sorting permutation of the inputs is known (but the inputs are still in arbitrary order), the problem can be solved in* $O(m' + n')$ *time, as the factor of* $\log(m' + n')$ *only comes from the initial sorting step.*

Proof. To non-obliviously compute the output of DIAMOND(m', n'), consider the path that the largest input takes through the network. If the largest input enters the network on wire j, all comparators it passes will return it as the larger element, which means that it will reach the leftmost output possible. We then proceed through the remaining inputs in descending order, determining for every input the leftmost output it can reach, considering that some outputs have already been occupied by larger values. Any current value that enters the comparison network on a wire j that is less than m' wires ahead of the first free output will be translated to the first (leftmost) available output. If the current value enters the network more than m' wires to the right of the first available output, it can only pass through m' comparators and will therefore reach output $j - m'$. The free outputs are indicated by a Boolean array K, where occupied outputs are marked with a value of true. Since we proceed through the input values in descending order, this yields the same output as direct evaluation of the transposition network. The entire algorithm is shown in Algorithm 6. □

[2]Note that the opposite case of a rectangular area in the alignment dag which only contains matches is trivially solved in linear time as it corresponds to a transposition network without comparators.

Algorithm 6 Computing the output of DIAMOND(m', n')

```
input: I[1], ..., I[m' + n']
output: O[1], ..., O[m' + n']

let I[L[1]] > I[L[2]] > ... > I[L[m' + n']] { L is the sorting permutation }
for (j ∈ [1 : m' + n']) do K[j] ← false { K contains the non-free outputs }
β ← 1 { β points to the leftmost free output }
for k = 1, 2, ... m' + n' { I[L[k]] is the next largest element }
 if L[k] < β + m' then { Maximum reaches leftmost free output }
  O[β] ← I[L[k]] { Translate input value to output }
  K[β] = true { Mark output as occupied }
  while (K[β]) do β ← β + 1
 else { Maximum goes to leftmost output it can reach }
  O[L[k] − m'] ← I[L[k]] { Translate input value to output }
  K[L[k] − m'] = true { Mark output as occupied }
 end if
end while
```

Using Algorithm 6, we obtain an improved algorithm for sparse semi-local comparison. For simplicity assume that both strings are of length n and (w.l.o.g.) that n is a power of 2.

Theorem 5.3.3. *After pre-processing the input strings for obtaining match lists, the problem of semi-local string comparison can be solved in $O(n\sqrt{r})$ time.*

Proof. We first find the sorting permutations of the input strings. This is possible in time $O((m+n)\log\min(\sigma, \max(m, n)))$, similar to obtaining μ_i in Algorithm 5 on page 70.

After this pre-processing, we partition the alignment dag into blocks using a recursive quadtree scheme. Consider processing such a block of size $w \times w$. Let this block correspond to comparing substrings $x_k \dots x_{k+w-1}$ and $y_l \dots y_{l+w-1}$. As an input for each such block, we have the sorting permutations of the two corresponding substrings, the input values for the transposition network corresponding to the block, and also the sorting permutation for these input values. For each block, we obtain the output values of its transposition network and their sorting permutation as follows.

For a $w \times w$ block, we can count the number of matches in it in time $O(w)$ by linear search in the sorting permutations of the corresponding substrings. Whenever

we find a block that does not contain any matches, we stop partitioning and use Algorithm 6 to compute the outputs of the corresponding comparison network. Otherwise, we continue to partition until we obtain a 1×1 block that only consists of a single match.

A 1×1 leaf block consisting of a single match can be processed trivially in constant time. Due to Theorem 5.3.2, we can compute the outputs for a $w \times w$ mismatch block in $O(w)$ time when the sorting permutation is known for the inputs. The sorting permutation for the root block of the quadtree is known, since the root of the quadtree corresponds to the full alignment dag, and the inputs to its transposition network form a sequence sorted in reverse. For all other blocks, we keep track of the sorting permutation of both its input and output elements. For every output we can trace the input it came from before executing Algorithm 6 and therefore know the permutation that was performed by the transposition network within the block. Knowing this permutation and the sorting permutation of the inputs allows us to establish the sorting permutation of the outputs in time $O(w)$.

To summarize, given the input values and their sorting permutation for every leaf block of the quadtree recursion, we can compute the output values and their sorting permutation in time $O(w)$. All non-leaf blocks are partitioned into four sub-blocks of size $w/2 \times w/2$. The inputs and their sorting permutation are split and used to recursively process the sub-blocks. We can then establish the sorting permutation of the outputs for the entire block in linear time by merging. To compute the outputs of any intermediate block we therefore need time $O(w)$ in addition to the time necessary for recursively processing the sub-blocks.

Consider the top $\log_4 r$ levels of the quadtree. In each subsequent level, the number of blocks increases by at most a factor of four, and the block size decreases by a factor of two. Therefore, this part of the quadtree is dominated by level $\log_4 r$ which contains at most r blocks, each of size $n/\sqrt{r}$. The total work required on this part of the tree is therefore $O(r \cdot n/\sqrt{r}) = O(n\sqrt{r})$.

The remaining levels of the quadtree can each have at most r blocks that still contain matches. The block size in each level still decreases by a factor of two.

Therefore, this part of the quadtree is also dominated by level $\log_4 r$ and requires the same asymptotic amount of work. The overall time for the algorithm is therefore bounded by $\sum_{j=0}^{\log_4 r} O(n/2^j \cdot 4^j) + \sum_{j=\log_4 r+1}^{\log_4 n} O(n/2^j \cdot r) = O(n\sqrt{r})$. The resulting algorithm has running time $O(n\sqrt{r})$, and thus provides a smooth transition between the dense case ($r = n^2$, running time $O(n^2)$) and the permutation case ($r = n$, running time $O(n^{1.5})$). □

5.4 Semi-local LCS computation for run-length compressed strings

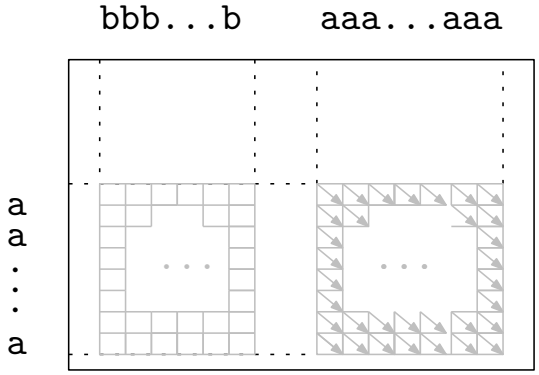


Figure 5.4: Alignment dag for run-length compressed strings

Another straightforward application of Algorithm 6 is comparing run-length compressed strings (see Apostolico et al. [1999]). In this compression method, a run of repeating characters is encoded by a single character together with the number of repetitions. A run-length encoded string $X = X_1X_2X_3 \dots X_{\overline{m}}$ consists of $\overline{m}$ character runs X_j of lengths $|X_j|$. The length of the full string is therefore $m = \sum_{j=1\dots\overline{m}} |X_j|$. When constructing the alignment dag for comparing two run-length compressed strings $X = X_1X_2X_3 \dots X_{\overline{m}}$ and $Y = Y_1Y_2Y_3 \dots Y_{\overline{n}}$, rectangular areas without matches occur when character runs in X and Y mismatch. Analogously, large rectangular areas with containing only match cells occur if the characters do match (see Figure 5.4). Using the comparison network method and Algorithm 6, these rectangular areas can be processed in cost proportional to their perimeter.

Given two input strings with uncompressed lengths m and n, and compressed lengths $\overline{m}$ and $\overline{n}$, this method results in an algorithm for semi-local comparison which has cost $\sum_{i\in[1:\overline{m}],j\in[1:\overline{n}]} O(|X_i| + |Y_j|) = O(\overline{m}n + m\overline{n})$. This is as good as the result by Bunke and Csirik [1993], additionally solving the more general problem of semi-local string comparison of run-length compressed strings.

5.5 High similarity and dissimilarity string comparison

In Section 5.3 we described an efficient algorithm for semi-local string comparison, parameterized by the overall number of matches. We now describe an application of the transposition network method to designing algorithms that are parameterized by the LCS length p of the input strings or their LCS distance $k = n - p$. Such parametrization provides efficient algorithms when the corresponding parameter is low, i.e. when the strings are highly dissimilar or highly similar.

In the paper by Hunt and Szymanski [1977], matches are processed row by row to establish which antichain they belong to. Apostolico and Guerra [1987] improved this algorithm by avoiding the need to consider non-dominant matches (see Section 5.1), and changing the order in which the match cells are processed. This allows us to obtain an algorithm that is parameterized by the length of the LCS. Further, there have been various extensions to this approach, which improve the running time by either using different data structures (see Eppstein et al. [1992]) or narrowing the area in which to search for dominant matches hence giving algorithms which are efficient both when the LCS of the two strings is long or short (see Rick [1995]). In this section, we will show how the transposition network method can be used to match these algorithms for global LCS computation. For semi-local alignment, we achieve a running time of $O(np)$, which is efficient for dissimilar strings.

We will now show the connection between the antichain decomposition of the set of match cells and the transposition network method. Consider an LCSNET network with the following input values: The first m wires (i.e. the inputs on left

hand side of the alignment dag) are initialized with ones, and the following n wires (i.e. the inputs at the top of the alignment dag) are initialized with zeros. On all comparators, smaller values are returned at the bottom output. We will refer to this specific transposition network setup as *$LCSNET(x, y)$ with 0/1 inputs*. Using only zeros and ones as inputs to LCSNET(x, y) corresponds to tracing seaweeds anonymously, only distinguishing between those seaweeds that start at the top and those seaweeds that start at the left. The 0-1 transposition network approach allows us to understand previous results for parameterized LCS computation in terms of transposition networks, and helps to extend some of these to semi-local string comparison.

Corollary 5.5.1. *In* LCSNET(x, y) *with 0/1 inputs as described above, let p be the number of ones reaching output wires below $m + 1$ (i.e. the bottom of the alignment dag). This number is equal to the number of zeros reaching an output wire above m (i.e. the right side of the alignment dag), and $LLCS(x, y) = p$.*

Proof. From Theorem 3.5.3, we know that $LLCS(x, y) = n - d$, where d is the number of seaweeds that start at the top and end at the bottom of the alignment dag. The number of zeros ending up at the bottom is therefore equal to d, and the number of ones ending up at the bottom is equal to $n - d = LLCS(x, y)$. Since the transposition network outputs a permutation of the input, and since we have n input zeros, $n - d$ zeros must end up at the right. □

We will now look at the behaviour of LCSNET(x, y) with 0/1 inputs in more detail. In order to be able to trace paths of individual values, we must specify the behaviour of the comparators for equal input values (note that changing this specification does not change the output of LCSNET(x, y)). Assume that comparators in LCSNET(x, y) swap their input values if these are equal. If the alignment dag contains only mismatch cells and therefore a full set of comparators, all ones move from the left to the right, and all zeros move from the top to the bottom. When introducing a match cell and hence removing a comparator, the zero that enters the match cell at the top is translated to the right, and the value of one entering the

match cell at the left is translated to the bottom. We trace these two values further: as identical values are swapped by convention, both the one (and equally the zero) will not change direction of movement and be passed on vertically (horizontally in case of the zero) through all comparators. We will refer to ones which move downwards and to zeros which move to the right as *stray*. Stray values only change direction again when they either encounter a match cell or another stray value. If two stray values enter the same cell, they leave this cell in the original directions, the one moving rightwards, and the zero moving downwards. This happens independently of whether this cell contains a match: in a match cell, no comparison is performed, the stray zero is returned at the bottom and the stray one is returned at the right. In a mismatch cell, the zero is also returned at the bottom since it is the smaller value. Therefore, two stray values always return to their original direction of movement when meeting in the same cell. Another observation is that any cell which has exactly one stray input value must have equal inputs. If such a cell is a match cell, the stray input value returns to its original direction of movement, and the other input becomes stray. If the cell does not contain a match, the inputs are exchanged by convention, and the stray value remains stray. To summarize, stray values caused by a match cell will start a row (stray zeros) or column (stray ones) of cells which output stray values. This row or column only ends when meeting another column or row of cells which output stray values.

Figure 5.5 shows an example of the LCSNET(x, y) with 0/1 inputs for the problem instance shown in Figure 5.1 on page 69. It seems intuitive from this figure that the stray zeros and ones trace contours in L.

Theorem 5.5.2. *A cell belongs to a contour in the matrix of prefix-prefix LCS lengths L if and only if it has at least one stray value as an input or output.*

Proof. This follows from Corollary 5.5.1 by induction on the number of contours. If L has no contours, no match cells can exist. If there is exactly one contour in L, all match cells must belong to this contour, and the contour splits the set of cells into two parts of mismatch cells. Consider the set of mismatch cells to the top/left of the contour. All cells in this set have zeros as their top input and ones as their left input

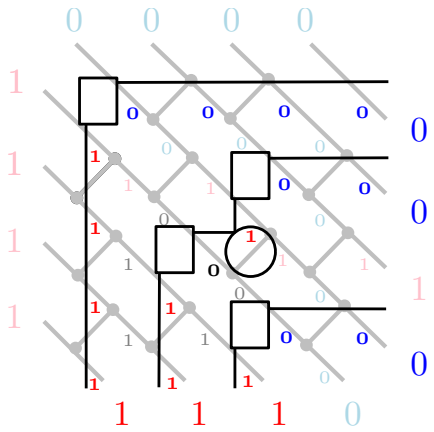


Figure 5.5: LCSNET(x, y) with 0/1 inputs

since these are either the input values to the transposition network, or have been translated through the previous mismatch cells as shown in case (e) of Figure 5.7. All dominant matches on the contour must have a zero as their top input and a one as their left input as well, since they must be at the right and below a case (e) mismatch cell, or equivalently at the top or left of the alignment dag. Dominant match cells output a stray zero on the right and a stray one on the bottom (see case (d) in Figure 5.7). Any cell that has a stray zero as its left input and a zero as its top input must be to the right of a match. As there is only one contour the cell cannot be below another match and therefore L will increase vertically in this cell since the prefix-prefix LCS can be extended by the first match to the left. Symmetrically, this is true for any cell with a stray and a none-stray one as its inputs (see cases (a) and (b) in Figure 5.7). In the only remaining case, two stray values meet in the same cell $(\hat{\imath}, \hat{\jmath})$ (case (c) in Figure 5.7). In this case, the prefix-prefix LCS could either be extended by using the matches above $(\hat{\imath}, \hat{\jmath})$ or by using the matches to the left of $(\hat{\imath}, \hat{\jmath})$, but not by using both since they are incomparable under the $\prec$ ordering (and no path containing one of each of those matches exists in the alignment dag). Now consider the cells immediately to the right or below the contour. These cells cannot be to the right or below dominant matches (otherwise they would belong to the contour). Therefore, these cells must all have non-stray inputs (i.e. a zero at the top input and a one at the left input), since cells on the horizontal contour

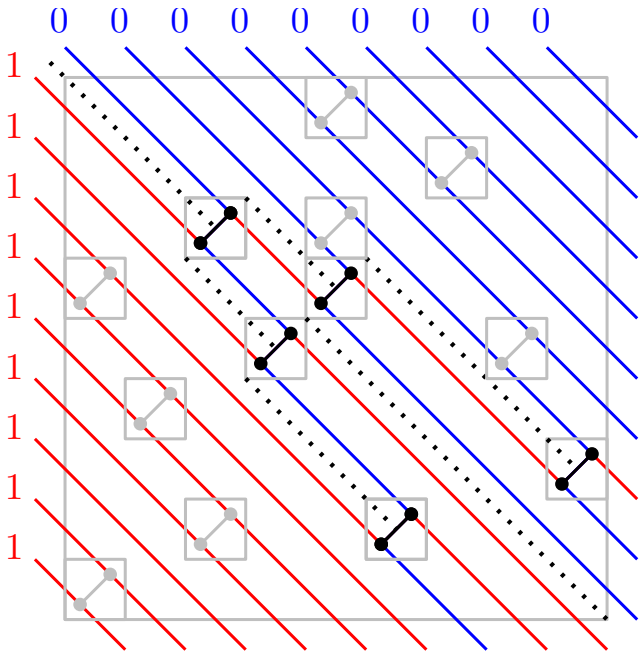


Figure 5.6: Comparing highly similar strings

output zeros on the bottom, cells on the vertical contour output ones at the right, and contour knees output a zero on the bottom and a one at the right output. As all the cells immediately neighbouring the contour to the right or below must be mismatch cells (only one contour exists, therefore all match cells are on it), they all belong to case (e) in Figure 5.7 and in consequence all cells below or to the right of them as well. Therefore, Theorem 5.5.2 is true in the case where only one contour exists. Furthermore, all additional contours must either have case (e) cells on top and to their left, or border directly on another contour. Cell contours output non-stray values on the right/the bottom if they have non-stray inputs. Therefore, Theorem 5.5.2 is also true for more than one contour. □

Algorithm 7 shows how to compute the output of LCSNET(x, y) with 0/1 inputs. Both cases inside the while loop can occur exactly p times in every row, as the list of ones moving downwards can maximally have p entries (one for each antichain). If the operations "find", "insert" and "delete" are implemented using binary search trees, all operations on the set *Ones* would take $O(\log p)$ time, giving

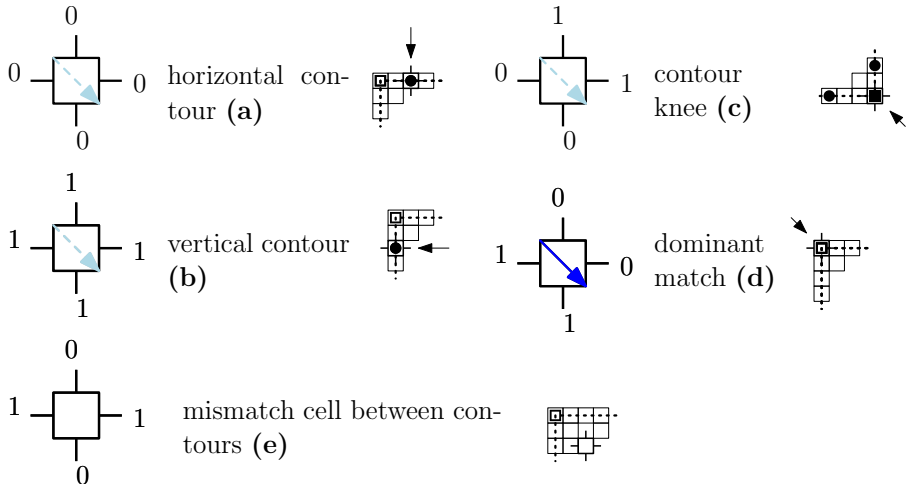


Figure 5.7: Interpreting 0-1 transposition network cells and their inputs as contours.

a running time of $O(mn \log p)$. Note, however, that all operations have increasing key values in every row, and that the number of insertion, deletion and search operations is equal to the number d of dominant match cells. Therefore, finger searching (see Brodal [2005] for an overview of finger searching and its applications) and an argument as by Apostolico and Guerra [1987] can be used to obtain a bound of $O(m \log n + d \log(mn/d))$.

Consider the problem of comparing two strings that are highly similar. Myers [1986] proposed an algorithm to compare strings in time $O(ne)$, where e is the edit distance between the strings. The idea behind this algorithm is to incrementally extend only the longest paths in the alignment dag until the LCS is found. A similar algorithm can be obtained by using 0-1 transposition networks as follows.

If the two input strings are identical, no comparators exist on the main diagonal of alignment dag cells, i.e. between transposition network wires m and $m + 1$. This means that no ones can get to the right hand side, and no zeros can get to the bottom of the alignment dag. We can look at this as two streams of zeros and ones, and do not need to evaluate comparisons within a single stream of zeros or ones. The only comparators which can possibly swap inputs are the ones between streams. If a comparator occurs between two streams, the inputs will only be swapped if the zero is input from the top, i.e. we can restrict our attention to the

Algorithm 7 Transposition network based antichain decomposition

```
input: Match lists μ_y from preprocessing
output: LLCS p, dominant matches
Ones ← ∅ { ordered list containing positions of ones moving downwards }
for i = 1...m do
  j ← 0
  left ← 1
  while j ≤ n do
    if left = 1 then { one moving left to right }
      {let find return n+1 if no element is found}
      find smallest o in Ones with o > j
      find smallest m in μ_i with o > j
      j ← m
      { we look for a match cell with a zero coming from the top }
      if o > m then
        insert j into Ones
        report dominant match: (i, j)
        left ← 0
      end
    else { zero moving left to right }
      find smallest o in Ones with o > j
      if o exists then { zero re-joins top-to-bottom stream }
        delete o from Ones
        left ← 1
        j ← o
      else
        exit while { zero reaches right hand side }
      end if
    end if
  end while
end for
p ← |Ones|
```

upper boundaries of streams of ones. Figure 5.6 shows an example. The comparators drawn in black are those between streams of zeros and ones which must swap their inputs.

Definition 5.5.3. Let a *1-0 boundary* in stage s of LCSNET(x, y) with 0/1 inputs be defined as any location in this stage where two adjacent wires l and $l+1$ carry values one and zero respectively.

Corollary 5.5.4. *The number of 1-0 boundaries in any stage of the transposition network is smaller than* $k+1 = n-p+1$.

Proof. By induction: Assume $m = n = 1$. The transposition network has two wires which are initialized with a zero and a one. Therefore, the number of 1-0 boundaries must be less or equal than 1. The LCS distance k can be 0 or 1. Increasing m or n by one adds another row or column of comparators to the transposition network. Consider the case of adding a column of comparators (i.e. increasing n by one). Each 1 which is output at the right hand side can only cause one 1-0 boundary. Furthermore, ones do not move downwards in comparisons. Therefore, a new 1-0 boundary can only be created if a value of 1 from the left hand side reaches the right hand side, which means that the number of 1-0 boundaries cannot increase by one in this case without also increasing k by one. However, k cannot increase by more than one, since maximally a single value of 1 reaches the right hand side. Symmetrically, when increasing m by one, we add a row of comparators at the bottom. If we have k zeros at the bottom, each of these zeros can only be part of a single 1-0 boundary. We can only gain a single 0 on the bottom by increasing m by one, in which case also k increases. Therefore $k + 1$ always dominates the number of 1-0 boundaries. □

Using this insight, the LLCS of two strings x and y with $|x| = |y| = n$ can be computed in time $O(nk)$. This is done by tracing the intersections of the 1-0 boundaries with the $2n - 1$ antidiagonals of the alignment dag, as this is the only place where change can occur. By Corollary 5.5.4, we know a bound for the number of 1-0 boundaries. At each intersection of a 1-0 boundary with an antidiagonal, the corresponding characters in x and y must be compared to check whether a comparator exists. This can be done in constant time, and since there are $2n - 1$ antidiagonals we get the claimed running time. Note that this algorithm does not require any pre-processing to obtain match lists.

Corollary 5.5.5. *All dominant matches must be on a 1-0 boundary in the transposition network.*

Proof. This follows immediately from Theorem 5.5.2. □

Corollary 5.5.5 allows us to narrow down the area in which to search for dominant matches, and can be used to extend the Algorithm by Apostolico and Guerra [1987] to achieve running time $O(kp)$, similarly to the one by Rick [1995].

Theorem 5.5.6. *The implicit highest-score matrix for comparing two strings of length n can be computed in time $O(np)$.*

Proof. Using the 0-1 transposition network, we are able to determine for every match cell whether it is dominant or non-dominant, as well as for every mismatch cell whether it is part of a contour. Looking at this in the more general setting of semi-local string comparison where we need to trace all seaweeds individually, we can still see that non-trivial comparisons between seaweeds can only occur when the cell is actually part of a contour. Cells outside the contours are always mismatch cells which compare an input originating at the left hand side of the alignment dag to an input originating at the top of the alignment dag. Therefore all the comparators in these cells can be replaced by swap operations (i.e. they contain seaweed crossings).

Given all dominant matches on a contour and the values on all transposition network wires before they intersect the contour, we can compute the values on all wires of transposition network after the intersection in time which is linear in the length of the contour. As all comparators between contours perform swap operations, we can also compute the permutation of values performed between two contours in time linear in the length of the longer contour.

It is possible to compute the set of all k-dominant matches with $k \in [1 : p]$ in $O(np)$ time. We can use the algorithm by Apostolico and Guerra [1987] for this.[3] Knowing the dominant matches in every antichain, we can trace its complete contour in time linear in its length. No contour can have length l longer than $2n$, and there are exactly $p = \mathrm{LLCS}(x, y)$ contours. Further, we can obtain the inputs and outputs of all cells in a contour of length l in time $O(l)$ with $l \leq 2n$. Therefore, the worst case running time of our algorithm for semi-local string comparison is bounded by $O(np)$. □

[3] A practical algorithm for computing a list of dominant matches has been described by Crochemore et al. [2003]

Chapter 6

Computing Alignment Plots Efficiently

In this chapter, we consider the problem of computing alignment plots, which consists in computing alignment scores for all pairs of windows in two input strings. This problem is relevant in computational biology for comparing genomic sequences. We will show a new, fast implementation of alignment plot computation and discuss how it is applied to finding evolutionally conserved regions in various plant genes. We also compare the running times of our algorithm to other approaches, and show that it is faster than all other comparable loss-free alignment plot methods.

6.1 Background

In this chapter, we consider the problem of computing alignment plots, which consists in computing alignment scores for all pairs of windows in two input strings. These plots are closely related to dot plots, which are a standard method for local comparison of biological sequences introduced by Gibbs and McIntyre [1970] and Maizel and Lenk [1981]. When creating a dot plot, a substring to substring distance is computed for all pairs of fixed-size windows in the input strings. The result can

be visualized by a plot showing a dot for each pair of windows that achieves a distance value below a fixed threshold. Commonly, the Hamming distance (obtained by comparing the characters in both input strings for every position and counting the number of mismatches) is used since it can be computed in linear time. Efficient algorithms for computing such Hamming-filtered dot plots were given e.g. by Maizel and Lenk [1981] and by Krumsiek et al. [2007]. When comparing two strings of length m and n, the optimal running time for computing a Hamming-filtered dot plot is $O(mn)$. However, the computationally most intensive part can be reduced to $O(m \log n)$ using suffix arrays (Krumsiek et al. [2007]).

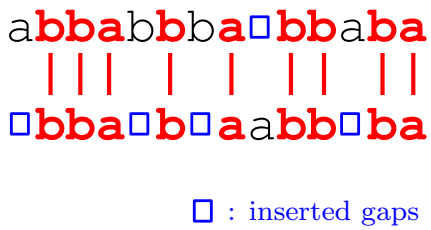


Figure 6.1: String alignment example

The weakness of the dot plot method is that the Hamming distance is only a rather crude measure of string similarity. Using a string edit distance or alignment score (Gusfield [1997]) for dot plot filtering can greatly improve the sensitivity of the method. The LLCS scores computed by the seaweed algorithm are the simplest example for such an alignment score: another interpretation of LCS computation is *string alignment* (see [Gusfield, 1997, p. 209 ff.]). An alignment of strings x and y is obtained by putting a subsequence of x into one-to-one correspondence with a (not necessarily identical) subsequence of y, character by character and respecting the index order (see Figure 6.1). The corresponding pairs of characters, one from x and the other from y, are said to be *aligned*. A character not aligned with a character of another string is said to be aligned with a *gap* in that string. Finding the LCS corresponds to computing a maximum alignment when assigning the scores $w_{=} = 1$ to aligning a matching pair of characters, $w_{\_} = 0$ to inserting a gap, and $w_{\neq} = 0$ to aligning two mismatching characters. More general alignments than the LCS can be

obtained using the Needleman and Wunsch [1970] algorithm, which allows for gap penalties as well as different scores for each individual pair of matching/mismatching characters, forming a *pairwise score matrix*, or *substitution matrix*. The seaweed algorithm can be generalized to score matrices with small rational scores at the price of a constant factor blow-up of the alignment dag (see Tiskin [2008b]). Practically used examples of substitution matrices are given by Henikoff and Henikoff [1992] and Dayhoff et al. [1979]. We will discuss adapting the seaweed method to substitution matrices slightly more complicated than the LLCS case in Section 6.2.

In the context of biological sequence comparison, this idea has been implemented by Ott et al. [2009], where a sequential algorithm with running time $O(mnw^2)$ is used to compute alignment plots with a fixed window length w. Their algorithm works by applying the method of Needleman and Wunsch [1970] to each pair of windows. Furthermore, this algorithm computes bounds on the scores by reusing scores from overlapping window pairs. These bounds are used to skip those pairs scoring below a threshold, and to speed up the dynamic programming computation of the Needleman/Wunsch alignment scores. Using bit-parallel LCS computation as discussed in Chapter 5 for computing the window scores, a constant factor speedup can be obtained. Another algorithm which can be adapted to computing alignment plots is given by Rasmussen et al. [2006]. They give a very fast algorithm for loss-free local LCS computation which uses q-gram filters. Our method is superior to their algorithm when searching for local similarities with low alignment scores, since the q-gram method is only effective when restricting the search to windows with more than 90% local similarity (for a window length of 100, this would mean that we only report window pairs with an LLCS of over 90). While this is sufficient for many applications in computational biology, in our application (Ott et al. [2009]; Picot et al. [2010b]), matches of much lower similarity are sought. A similar argument applies to using tools like BLAST (Altschul et al. [1990]) or more advanced variations of it like PatternHunter (Ma et al. [2002]): these are heuristic methods which work very well for finding local alignments of high similarity, but can be less sensitive when searching for fixed-length local alignments with similarities

below 70% (e.g. Ma et al. [2002] studied the sensitivity of their algorithm for a range of local similarities). Our method does not depend on the local match similarity at all: we can report all matching window pairs regardless of their alignment score. A drawback of using the seaweed method in comparison to BLAST is its restriction to alignment scores with simple gap penalties, whereas BLAST allows one to use generic substitution matrices. Currently, we can overcome this limitation only by increasing the size of the alignment dag (see [Tiskin, 2010a, Section 3.4] and Section 6.2), which slows down our computation. In the absence of a theoretical generalisation, a practical method to use arbitrary alignment scores could consist in using our algorithm to isolate areas which are of interest for applying more sensitive and computationally more intensive comparison algorithms.

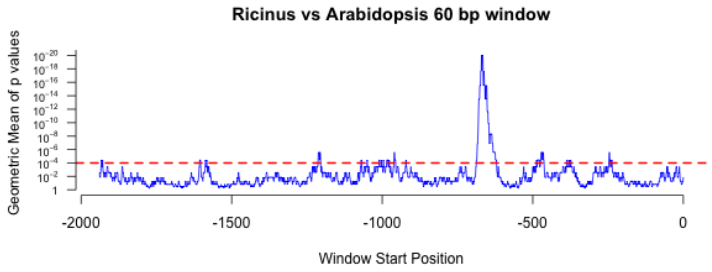


Figure 6.2: Alignment profile produced by the EARS webservice

Our implementation of alignment plots has been integrated into a webservice for evolutionary analysis of regulatory sequences (EARS, see Picot et al. [2010b]), which enables the swift generation of alignment plots for short genetic sequences of up to 2000 characters. The main biological application for such comparisons is the computation of conservation profiles for non-coding regions in DNA promoter fragments (see Picot et al. [2010a], and references therein). Promoter fragments are areas in DNA which will normally not produce proteins, but rather regulate whether a certain gene will be expressed (see Alberts et al. [2007] for an introduction to terminology and biological aspects of this application). The analysis of these promoter fragments is both difficult and of great importance for biologists, as knowledge of the exact locations of regulatory DNA regions is necessary in order to understand

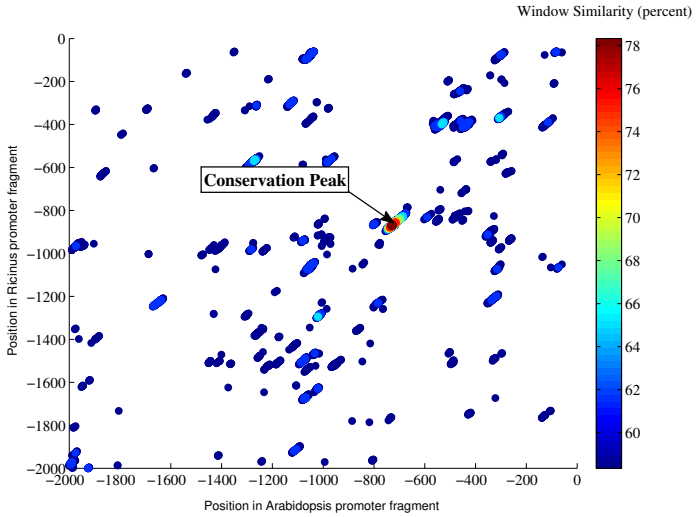


Figure 6.3: Full alignment plot

the complex mechanisms underlying gene expression. Figure 6.2 shows the output of the EARS web tool for comparing the 2000 base-pair promoter sequence of the AT5G61380 gene in Arabidopsis thaliana (see The Arabidopsis Information Resource (TAIR) [2010]) to the promoter sequence of the orthologous gene in Ricinus communis (castor bean). Figure 6.3 shows the full alignment plot (using a similarity threshold, requiring at least 55% sequence similarity). We can clearly see a similarity peak in both figures, which identifies a genomic region that is highly conserved between both species. Picot et al. [2010a] argue that such conservation can be used to identify areas in the DNA that may contain regulatory elements which can then be verified experimentally. Furthermore, Picot et al. [2010a] found the alignment plot method to be more sensitive than e.g. using BLAST or other sequence comparison methods in some cases, establishing alignment plot computation as a useful addition to the set of standard tools for biological sequence analysis.

In this chapter, we will show how to compute alignment plots efficiently in theory and practice using semi-local string comparison. First, we discuss different techniques for improving the practicality of the seaweed method, and show different theoretical approaches to the problem. We then show how to engineer a fast data-parallel algorithm, running in time $O(mn\sqrt{w}/\gamma)$, using vector operations that work

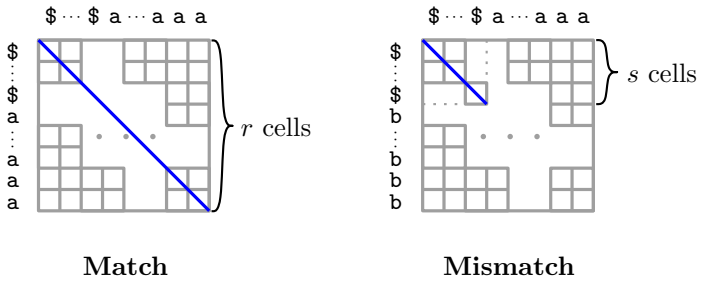


Figure 6.4: Custom alignment dag for more general substitution matrices

on γ values in parallel. We also show experimental results from an implementation of this algorithm which uses MMX/SSE instructions (Intel Corporation) for vector parallelism, and we compare it to an implementation using a graphics processor (GPU) for the same task. We also show a coarse-grained parallel variant of this algorithm which uses MPI (Snir et al. [1995]) for running on the cluster systems at the Centre for Scientific Computing at Warwick (see The University of Warwick [2009]), and on the Warwick Systems Biology Cluster.

6.2 String alignments with pairwise scores

Longest common subsequences are a more accurate measure for string similarity than e.g. the Hamming score. However, in practical applications, more general alignments can be of interest. In this section, we discuss how substitution matrices with small rational weights can be implemented using a modification to the alignment dag, and show how we can implement small gap penalties without affecting the practicality of our algorithm. Tiskin [2010a] has shown how to apply the seaweed algorithm to compute more general alignment scores. To implement this, we normalize the alignment scores as follows (see also Gusfield et al. [1994]; Rice et al. [2000]). Assuming that $w_{=} \neq 0$, we set

$$w'_{=} = 1, \qquad w'_{\neq} = \frac{w_{\neq} - 2w_{\_}}{w_{=} - 2w_{\_}}, \text{ and} \qquad w'_{\_} = 0. \tag{6.1}$$

When computing an alignment score h' using these new weights, we can retrieve the alignment score h for the original weights as

$$h = h' \cdot (w_= - 2w_\_) + (m+n) \cdot w_\_. \tag{6.2}$$

If all weights involved are rational, we use the following procedure to create an alignment dag which allows one to compute the alignment scores using the seaweed algorithm. Let $w_\neq = \frac{s}{r} w_=$ for two positive integers s and r with $s < r$. We create *blown-up input strings* x' and y', where we replace each character α by s '`$`' characters, followed by $r - s$ copies of α (see Figure 6.4). Given LLCS(x', y'), we can compute the alignment score h for x and y as $h = \frac{\text{LLCS}(x',y')}{r}$.

Example 6.2.1. Consider alignments with match score $w_= = 1$, mismatch score $w_\neq = 0$ and gap penalty $w_\_ = -0.5$. To compute these alignments, we modify the input strings by adding a new character '`$`' to the alphabet, which we insert before every character in both input strings such that e.g. `abab` transforms into `$a$b$a$b`. We can obtain an alignment dag consisting of multiple cells representing a single match/mismatch as shown in Figure 6.5. For input strings x and y of length m and n, the alignment score $S(x, y)$ can be retrieved from LLCS of the modified strings x' and y' as

$$S(x, y) = 0.5 \cdot (LLCS(x', y') - m - n), \tag{6.3}$$

as each mismatch contributes exactly 1 to the LCS and is equivalent to insertion of a gap. We expect the running time of our algorithm to increase by a factor of four by this reduction, as both input strings double in size.

Match **Mismatch**

Figure 6.5: Alignment dag blow-up

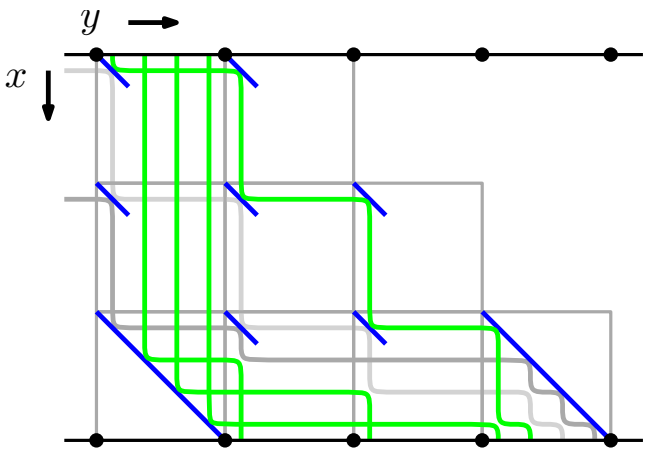


Figure 6.6: Seaweeds with a 4-grid

When querying scores from a highest-score matrix for our blown-up input strings, we are only interested in values with index pairs (i, j) having $i \bmod r = j \bmod r = 0$. We call r the *grid size*. For each pair of characters, we have generated a square of $r \times r$ cells in the alignment dag. Longest paths ending in the middle of such a square do not give a meaningful alignment score; therefore, we do not need to be able to query their lengths exactly.

Definition 6.2.2. Let A be a highest-score matrix. The r-grid approximate highest-score matrix A_r is defined as follows. We have $A_r(i, j) = A(i, j)$ if $i \bmod r = j \bmod r = 0$. Otherwise, we have $A(i, j) - r \leq A_r(i, j) \leq A(i, j) + r$.

Informally, the r-grid approximate highest-score matrix gives exact LCS lengths only for substrings which align to a grid of r characters. For substrings which start or end in between grid points, we allow an error of $\pm r$. Highest-score matrices with an r-grid can be computed using a modified version of the seaweed algorithm. We assign the seaweed start points rounded to the r-grid we defined, which creates r-bundles of seaweeds that have the same starting point on the top. When comparing two seaweeds with the same starting point, we allow them to cross, since the particular relative order of the seaweeds between grid points is not important for querying scores for substrings that align to the grid. Comparisons of seaweeds starting within a single grid interval to all seaweeds which start to the left or to

the right of this interval have the same result. The order in which the seaweeds from a single r-bundle arrive at the bottom does not affect the number of seaweeds dominated by a window which starts at a position $k \bmod r = 0$. Looking at the resulting seaweeds, we get a picture as shown in Figure 6.6.

6.3 Alignment plots

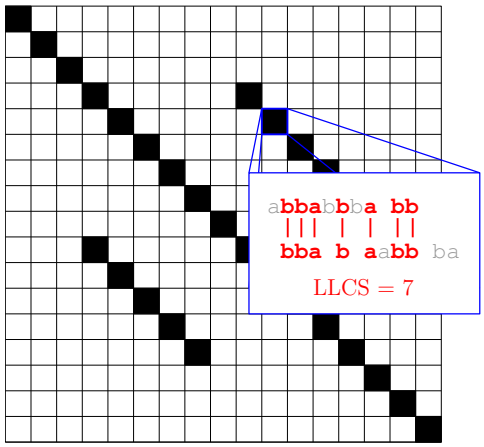


Figure 6.7: Alignment plot illustration

We are interested in computing $WLCS(i, j)$ for all $i \in \{1, 2, \ldots, m - w\}$, $j \in \{1, 2, \ldots, n - w\}$ (see Figure 6.7). A straightforward method to do this would be to compute the LCS independently for each pair of windows using dynamic programming. This method has a worst case running time $O(mnw^2)$. We will show how to improve on this running time by adapting the seaweed method to computing alignment plots. For computing alignment plots, we do not require random access to all elements of $A(i, j)$. We will now show how restrictions on the query pattern allow for a trade-off between storage space and flexibility of queries to $A(i, j)$. We restrict the range of $A(i, j)$-values we require by introducing the window length w as a parameter. For computing alignment plots, we are not interested in longest paths which are longer than w, since we always query scores for substrings of length $\leq w$.

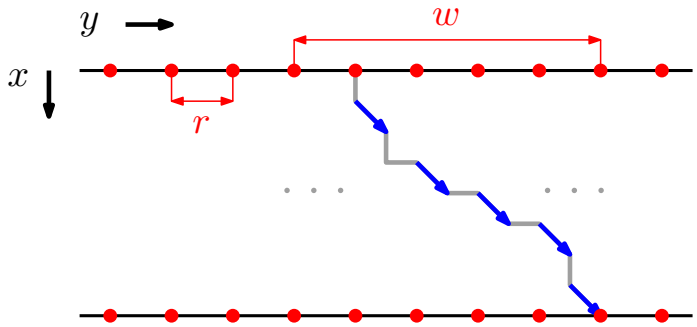


Figure 6.8: A highest-scoring path given by (w, r)-restricted highest-score matrices

Definition 6.3.1. Let A be a highest-score matrix. The w-window restricted highest-score matrix A_w is defined as follows. We have $A_w(i, j) = A(i, j)$ if $j - i \leq w$. Otherwise, we require that $A(i, j) \leq A_w(i, j) \leq A(i, j) + (j - i - w)$.

Informally, the window-restricted highest-score matrix gives the exact LCS lengths for all substrings of length less than or equal w exactly, and might have an additive error of up to the substring length minus the window length otherwise. We can combine this restriction with the r-grid approximate highest-score matrices from Definition 6.2.2

Definition 6.3.2. Let A be a highest-score matrix. We define the (w, r)-restricted highest-score matrix $A_{w,r}$ as the w-window restricted matrix A_w, which is r-grid approximate.

This restriction on the highest-score matrices allows us to reduce the number of nonzeros we need to store, and also to reduce the number of bits required to represent seaweeds.

Proposition 6.3.3. *To represent a w-window restricted highest-score matrix implicitly, we only need to store the nonzeros $(\hat{\imath}, \hat{\jmath})$ of the corresponding implicit highest score matrix, for which $\hat{\jmath} - \hat{\imath} < w$.*

Proof. Straightforward from Theorem 3.5.3 and Definition 6.3.2. □

Definition 6.3.4. The *span* of a seaweed is defined as the horizontal distance it covers in the extended alignment dag. A seaweed corresponding to a nonzero $(\hat{\imath}, \hat{\jmath})$ has span $\hat{\jmath} - \hat{\imath}$.

Proposition 6.3.5. *Consider comparing two strings x and y of lengths m and n. We can represent the start and end coordinates of a single seaweed in the corresponding (w, r)-restricted highest-score matrix using $O(\log(w/r))$ bits.*

Proof. We store the seaweeds in a vector S of size $m+n$, where each vector element stores $\lceil \log(w/r + 1) \rceil$ bits. For each nonzero $(\hat{\imath}, \hat{\jmath})$, we have one vector element $S(\hat{\imath} + 1/2) = \min(2^{w/r+1} - 1, (\hat{\jmath} - \hat{\imath})/r)$. Each vector element stores the distance a seaweed covers. The starting point of each seaweed is equal to its index in the vector.

It is straightforward to see that we only need $O(\log w)$ bits for a vector element: seaweeds in a (w, r)-restricted highest-score matrix become irrelevant once their span is larger than w, since the corresponding nonzeros will not affect any LCS for a substring of length $\leq w$ (see Theorem 3.5.3). In order to reduce the number of bits to $O(\log w/r)$, we use the fact that we only need to answer LCS queries correctly if $i \bmod r = j \bmod r = 0$. In this case, the specific permutation induced by running the seaweed algorithm on the alignment dag is not important for seaweeds starting within $[k : k+r-1]$ with $k \mod r = 0$. We therefore do not need to distinguish the individual r seaweeds in each of these groups: once they reach the bottom, we only need to know their starting position within a window of size r. We can thus divide the distance values by r, which gives the claimed number of required bits. □

Note that an extreme case of computing (w, r)-restricted highest score matrices using the seaweed algorithm gives a bit-parallel algorithm for LCS computation (see Chapter 5). Given two strings x and y, we set $w = \max(|x|, |y|)$ and $r = |x|$, and therefore only require a single bit for storing a seaweed. Informally, we only distinguish between seaweeds starting before and after position 0. This is equivalent to the algorithm from Crochemore et al. [2001]. Furthermore, Proposition 6.3.5 implies that the number of bits used to represent a seaweed is not affected by using rational

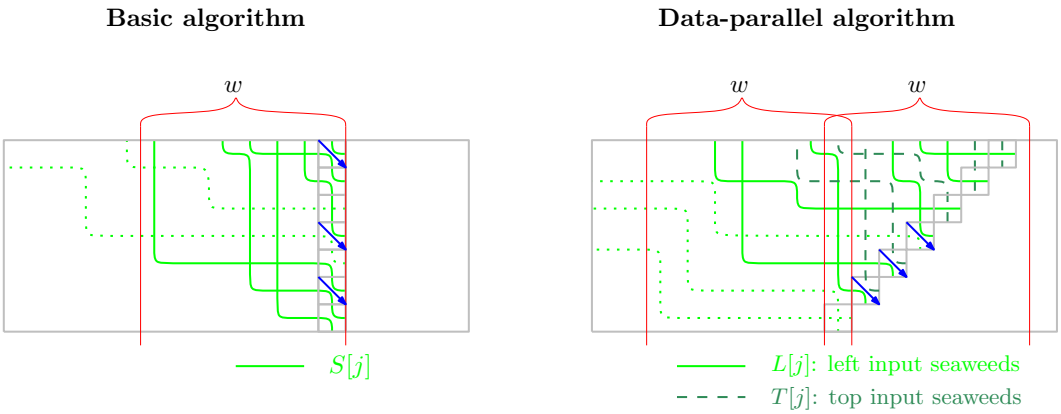


Figure 6.9: Seaweeds in a sliding window

score matrices. When using the scoring scheme introduced in Example 6.2.1, it is sufficient to compute a $(w, 2)$-restricted highest score matrix.

6.4 A data-parallel alignment plot algorithm using only vertical vector operations

The seaweed algorithm shown in the last section can be used to compute the LCS of all pairs of w-windows simultaneously in time $O(wn)$ for two strings of respective lengths w and n (i.e. one of the strings consists of only one w-window). By Theorem 3.5.3, the LCS of x and any w-window $y_i \dots y_{i+w-1}$ is computed as the number of seaweeds starting and ending within the odd half-integer range $\langle i : i+w \rangle$. Intuitively, we can keep track of the LCS length between a sliding w-window in the one input string and the other string by counting the number of seaweeds that reach the bottom of the alignment dag. Therefore, our algorithm can compute the LLCS for all w-windows in a single pass over all columns of cells in the alignment dag.[1] We obtain a first improved algorithm for comparing all pairs of w-windows.

[1] A similar problem was studied by Boasson et al. [2001]. In their paper, the goal is to compute the number of w-windows in y which contain x as a subsequence. Our algorithm is similar to theirs, however, it solves the more general problem of computing the LLCS in all w-windows which do not necessarily contain the complete string x as a subsequence.

Theorem 6.4.1. *Given two strings x and y of lengths m and n, the LLCS for all pairs of w-windows between x and y can be computed in time $O(mnw)$.*

Proof. We apply the seaweed algorithm for computing the implicit w-restricted highest-score matrices for y and all substrings of x that have length w. Each application of the seaweed algorithm therefore runs on a strip of height w and width n of the alignment dag corresponding to $x_i \dots x_{i+w-1}$ and y. In each column j, exactly one new seaweed starts at the top of the alignment dag, and exactly one seaweed ends at the bottom. We track seaweeds ending within $\langle j-w : j \rangle$. We store the start points of these seaweeds in an array $B[k]$, $k \in [1 : w]$. For each column, we need to evaluate w cells in the alignment dag according to Algorithm 1. This is possible in time $O(w)$. The seaweed ending up at the bottom replaces the seaweed in B which is now ending just outside the w-window. We can then count the number d of seaweeds which start within $\langle j-w : j \rangle$ in time $O(w)$. The LLCS of $y_{j-w+1} \dots y_j$ and $x_i \dots x_{i+w-1}$ can then be calculated as $w - d$. In total, we have to process n columns using time $O(w)$ in every strip. In total, $m - w$ strips exist, therefore we obtain running time $O(mnw)$. □

While this direct application of the seaweed method gives an asymptotic improvement on the method of computing the LCS independently for every pair of windows by dynamic programming, it is not necessarily more practical. The dynamic programming method can exploit the fact that we are only interested in windows with an alignment score beyond a given threshold. More importantly, the dynamic programming method allows one to improve performance by introducing a step size h, and only comparing w-windows starting at positions that are multiples of h. We now show that our algorithm can be modified to take advantage of these techniques, and also efficiently parallelised.

A method for obtaining speedup for dynamic programming methods as above is by using vector instructions. Parallelizing the problem on a coarse-grained level can be trivially done by distributing the computation of the strips between multiple processors (see Section 6.7). We are more interested in achieving CPU-level

parallelism in this section. A common method to speed up dynamic programming computation is to use vector instructions to compute multiple entries in the dynamic programming table in parallel.

A practical example for vector parallelism that is applicable here are Intel's MMX instructions (see Intel Corporation [1999a]). MMX provides instructions for integer vector arithmetic and comparison. MMX also includes instructions for performing bit-shifts on multi-byte vectors (`PSLLx`), comparing individual words in parallel (`PCMPEQx` and `PCMPGTx`), and performing bitwise and/or operations (`PAND` and `POR`). The main difference between the theoretical MP-RAM model of vector parallelism and the implementation on a specific machine is that the number of words which can be processed in parallel is limited by the size of the vector that fits into a register on the machine (e.g. 64 bits for MMX, subdivided into bytes, words, or double words). We mainly use integer vector parallelism here, although it would also be possible to implement our algorithm using floating point vector processing (e.g. using SSE, see Intel Corporation [1999b]).

In the following, we will denote vector variables using capital letters: U is a vector, and $U(j)$ an individual element. All operations on vectors are performed element-wise unless specified otherwise. We assume that all elements in the vector are v-bit values. If an element of a vector has all bits set, then this represents the value of $+\infty$, having $\infty \equiv 2^v - 1$.

When carrying out the seaweed algorithm on columns of the alignment dag, the result of every subsequent cell of the column depends on the result from the cell above it. To be able to process multiple cells in parallel, we need to process cells by antidiagonals (see Figure 6.9). We can then use vector operations to implement each step in the the seaweed algorithm, as each cell can be processed only using data computed in the previous step.

We would like to track seaweeds only if they are within the w-window of interest. In order to keep the required value of v as small as possible (and hence allow a high degree of vector parallelism), we identify seaweeds by the distance of their starting points to the current column. This distance can be represented

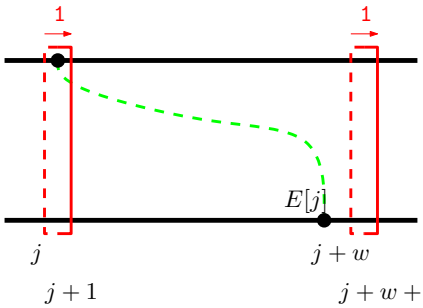
using $\lceil \log_2 2w + 1 \rceil$ bits. We first introduce the operation `saturated_inc` for incrementing all elements in a vector unless they are larger than $2w$. (i.e. operation `saturated_inc`(V) yields $\min(V(k)+1, \infty)$ for each vector element $V(k)$). Another common feature vector operation is mask generation. Operation `match_mask`(V, W) generates a vector which contains the value ∞ at all positions k, where $V(k) = W(k)$ and zero otherwise. We also define a compare/exchange operation $(V', W') \leftarrow$ `sort_elemwise`(V, W), which exchanges $V(k)$ and $W(k)$ only if $V(k) > W(k)$ and returns the result as a pair of vectors (V', W'). Finally, we require an operation to exchange vector elements conditionally. We introduce operation $V' \leftarrow$ `exchange_if`(V, W, M), which returns vector elements $V(k)$ if $M(k) = \infty$, and $W(k)$ otherwise. All these functions can be vectorized efficiently using MMX. The implementation of these functions is shown in Appendix A.

The only part which cannot be vectorized is the maintenance of array B, which counts the seaweeds that have reached the bottom of the alignment dag. However, we only used this counting method for simplicity, we can in fact implement the counting procedures in $O(w)$ space and $O(1)$ time per operation. We look at the problem as a variant of incremental queries to an implicit highest-score matrix (see Observation 3.3.6).

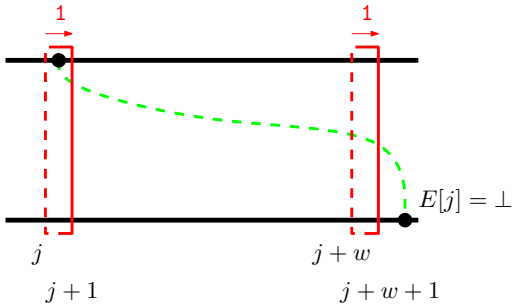
Lemma 6.4.2. *We can keep track of the number of seaweeds contained within a sliding w-window of the alignment dag in $O(w)$ space and $O(1)$ time for every constant-size shift of the window.*

Proof. We need to show that moving the window by one step requires constant time. Consider storing two arrays of size w, S and E. If our window starts at position j, we store the start point of the seaweed ending at $j + \hat{\imath}$ with $\hat{\imath} \in \langle 0 : w \rangle$ in $S((j + \hat{\imath} - \frac{1}{2}) \bmod w)$. Furthermore, for all seaweeds ending within the window, we store the end point of the seaweed starting at $j + \hat{\imath}$ in $E((j + \hat{\imath} - \frac{1}{2}) \bmod w)$. We also maintain a count d of seaweeds which have reached the bottom of the window. Initially, we set the count of seaweeds starting/ending within the window to zero and we initialize $S(i) = E(i) = \perp$ (we use $\perp$ to indicate that the starting/ending position of the corresponding seaweed is outside our window). We can update the

(a) Removing a seaweed at the left

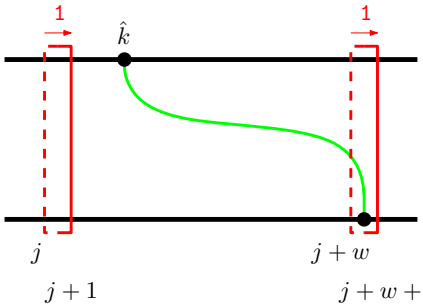


Case 1: Seaweed ended within the window $\Rightarrow$ $d \leftarrow d - 1$

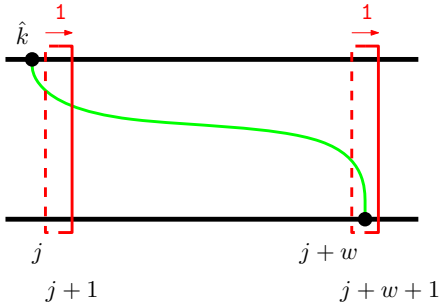


Case 2: Seaweed endpoint unknown (must be outside the window) $\Rightarrow$ d remains the same

(b) Adding a seaweed at the right



Case 1: New seaweed ends within the window $\Rightarrow$ $d \leftarrow d + 1$



Case 2: New seaweed ends outside the window $\Rightarrow$ d remains the same

Figure 6.10: Counting seaweeds in a sliding window

two arrays and the count as follows. When moving the window one step to the right, we must enter the seaweed reaching the bottom at position $j + w + \frac{1}{2}$ into our arrays, and we can remove the seaweed which starts at position $j + \frac{1}{2}$. For the new window, we have $j' = j + 1$. We update the two arrays at position $j \bmod w$ as follows. If $S(j + w - 1 \bmod w) \geq j$, or if $E(j \bmod w) \neq \perp$ (these two conditions are equivalent), we decrement d. We then set $S(j+w-1 \bmod w)$ and $E(j \bmod w)$ to $\perp$. After this, we add the seaweed ending at $j + w + \frac{1}{2}$ by putting its start point $\hat{k}$ into $S(j + w - 1 \bmod w)$, and by writing $j + w + \frac{1}{2}$ into $E(\hat{k} - \frac{1}{2} \bmod w)$ if $\hat{k} > j$. Also, if $\hat{k} > j$, we increment d. This procedure keeps track of the number of seaweeds contained within the window due to the fact that in each step, at most one seaweed

can be added to, and at most one seaweed can be removed from inside the window. Figure 6.10 shows all cases both for adding and removing the seaweeds. We can show symmetric operations for the case when the window is moved to the left. □

The full data-parallel algorithm is shown as Algorithm 8. If the window size

Algorithm 8 Vector-parallel seaweed algorithm

```
input : strings x and y, step size s
output : the LLCS of all pairs of w-windows in x and y.

let P, L, T, C be vectors of size |x|

set P(k) = x_k + 1 for k ∈ [1 : w] { translate pattern into vector }
set E(i) = ⊥ for i ∈ [1 : w]
set d = 0

for i = 1 ... m − w (using step size s)
 set C(k) = 0 { zero matches all characters }
     T(k) = ∞, L(k) = ∞ for k ∈ [1 : w]
     B(k) = ∅
 for j = 1 ... n + w
   L ← saturated_inc(L) ; T ← saturated_inc(T)
   C ← C≪v; C[1] ← y[j] + 1 { get the next character in y }
   let M = match_mask(C, P) { generate match mask }
    (T′, L′) ← sort_elemwise(T, L) { sort distance pairs in mismatch cells }
    T ← exchange_if(T′, L, M) { exchange distances in match cells }
    L ← exchange_if(L′, T, M)
   { can we remove a seaweed that was contained in the window? }
   if E(j − w mod w) ≠ ⊥ then
    d ← d + 1
    E(T(w) + j − w mod w) = ⊥
   { a new seaweed reached the bottom }
   if T(w) < w then
    E(T(w) + j − w mod w) = j
    d ← d + 1
   T ← T≪v; T(1) ← 0 { move top inputs down by one }
   WLCS(i, j) ← w − d
 end for
end for
```

is large, the degree of vector parallelism in the algorithm can be further increased by introducing a horizontal step size h and computing an implicit (w, hr)-restricted highest-score matrix. When using MMX, we need to minimize the number of bits per vector element to allow using MMX instructions with the greatest degree of

vector parallelism. This will still be relevant when using SSE4, but less so on newer vector architectures like Larrabee (Seiler et al. [2009]; Abrash [2009]), which allow for larger-scale vector parallelism on 32-bit integer elements, similar to GPU computing. Apart from the higher degree of vector parallelism, we further save time by computing the implicit (w, hr)-restricted highest-score matrix, since we then only need to increment the vectors L and T in Algorithm 8 every hr steps.

Another feature of the algorithm shown in this section is that it only uses *vertical* vector operations, i.e. operations which are carried out separately on each vector element. Although newer versions of MMX also include horizontal operations which combine adjacent elements in a single vector, using vertical operations is more efficient and allows for the highest degree of vector parallelism on this platform.

6.5 A data-parallel alignment plot algorithm for graphics processors

In the previous section, we described an algorithm for computing alignment plots using vertical vector parallelism. This type of vector parallelism is well suited for implementation using the vector extensions on traditional processors. Graphics processors (GPUs) give a different approach to vector programming, which allows for a more flexible way to define vector operations. We use the ATI/AMD Brook+ compiler (Advanced Micro Devices Inc. [2010]), which has a very similar programming concept to OpenCL[2], the recently agreed standard for GPU programming (Khronos Group [2010]). Operations for each vector element can be defined in a high-level programming language. Brook+ differentiates between horizontal and vertical operations by defining vectors (or matrices) as *data streams*. Normally, such data streams have elements that are processed independently, i.e. each vector operation accesses exactly one stream element at the same time. However, data streams can also be used for random access, in which case they are called *gather/scatter streams*. An operation on such streams is then carried out in parallel by many simple processing

[2]The work in this thesis is based on Brook+ rather than OpenCL since at the time of implementing our algorithms, no implementation of OpenCL was available.

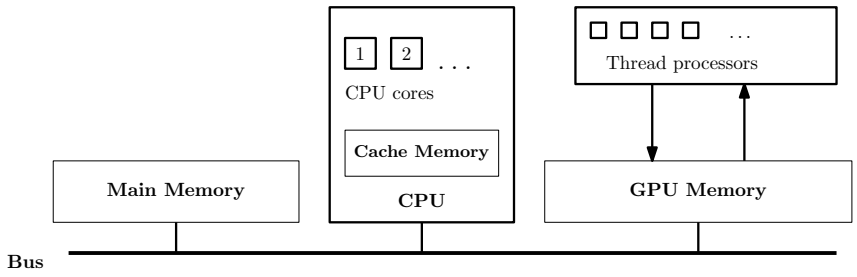


Figure 6.11: CPU and GPU as BSP computers with external memory

units, which allows more flexibility compared to SIMD-style vector programming. These processing units are named differently by the two main manufacturers of GPUs, they are called *thread processors* by ATI/AMD, see [Advanced Micro Devices Inc., 2010], and *stream processors* by NVidia, see [NVIDIA Corporation, 2009].

In this section, we show a vector parallel algorithm developed using the ATI Brook+ compiler. The algorithm is suitable for implementation on different types of graphics processors. The basis of this method is again the seaweed algorithm, however, without adaptation to vertical vector operations (in fact, the algorithm using only vertical operations did not perform well on a GPU in practice). We will deviate in this section from the convention of showing generic pseudocode and instead discuss the actual implementation. The algorithm itself has been discussed in Chapters 3 (page 27) and 5 (Section 5.2, see page 71). In this section, we will show the modifications that are necessary for the algorithm to perform well on a GPU.

We will first introduce the basic concepts of GPU programming, aiming for a generic description on an algorithmic rather than code level. A computer with a GPU can be modelled as two connected BSP computers with external memory. Both a multi-core CPU and the GPU can access the main memory of the system by means of a communication bus. The CPU can initiate data transfers from the main memory of the system into the GPU's dedicated memory. The CPU also controls the execution of code on the GPU in superstep-style. The code for each thread processor on the GPU is specified in form of a *computational kernel*. We

can assume for our purposes that each processor on the GPU will execute the same kernel code on different items of data, which it reads from the GPU memory. In many problems, it is important to balance the number of computations and the number of memory access operations for efficient GPU-based parallelisation. In GPU programming, usually a different terminology from regular parallel computing is applied: we measure the maximum *throughput* of a computational kernel, and look at its main limiting factors. A GPU has a theoretical maximum throughput, which is calculated as the maximum number of elements that can be processed per second if each processor is able to read its input data utilizing the full memory bandwidth. The maximum throughput of an algorithm can be limited by memory access. This can happen if multiple processors read and write the same item of data, or if data access patterns have not been implemented to access data in the most efficient order for caching and prefetching in order to hide the memory access latency. Theoretical models for describing this kind of *coalesced* memory access were studied by Ha et al. [2008] and its practical implications are discussed in many GPU programming code samples, see NVIDIA Corporation [2010]. Notice that thread processors do not have direct access to the system's main memory, or CPU caches. The memory of the GPU is organized in form of vectors or matrices which are called *streams*. A kernel computation specifies which elements of such a vector or matrix need to be read, how they will be combined, and which elements will be written. Execution of such a kernel on a stream is equivalent to executing a single superstep on our coupled BSP computer.

We will show examples in this section that are given in C++ with AMD's Brook+ extensions (Advanced Micro Devices Inc. [2010]; Buck et al. [2004]).[3] In Brook+, the programmer specifies computational kernels which will run on the GPU as functions which take streams as parameters. The main deviation of Brook+ from the C++ standard is that it includes additional functionality for specifying which data resides on the GPU, and how it is accessed. It differentiates two types of streams:

[3]This is very similar to the new OpenCL standard, but allows simpler presentation of our methods.

- The definition of `int p<>` declares a vector, which is accessed in a vertical fashion, i.e. each call to the kernel function works on exactly one element in `p`.

- `[in|out] int o[]` declares a vector, which can be accessed randomly by the kernel. The keywords `in` and `out` specify whether the vector can only be read or also be written by the kernel. In Brook+, these types of streams are named *gather* and *scatter streams*.

A single kernel can work on multiple streams at the same time. Kernel function calls are allocated to physical thread processors automatically by a scheduler. In each call to a kernel function, the location of the current data items can be queried using the `instance()` function. This function returns the indices of the data elements processed by the current call for all streams declared using `<>`. Data can be written to and read from GPU memory using the functions `streamRead()` and `streamWrite()`.

We will now show a simple example for GPU computation. We show how to implement an algorithm for inverting a permutation which is stored as an array of integers.

Example 6.5.1 (Inverting a permutation)**.** We can store a permutation as a 1-dimensional stream on the GPU. The CPU can copy permutation data to the GPU memory first, and then call the following kernel on the stream.

```
// GPU code
kernel void invert_permutation(int p<>, out int o[]) {
    int idx = instance().x;
    o[p] = idx;
}
// [simplified] CPU code
void invert(int n, int * permutation) {
  // this corresponds to memory on the GPU
  int gpu_permutation_in<n>;
  int gpu_permutation_out<n>;
```

```
  // copy data over to GPU
  streamRead(gpu_permutation_in, permutation);
  invert_permutation(
    gpu_permutation_in,
    gpu_permutation_out);
  // copy result back
  streamWrite(gpu_permutation_out, permutation);
}
```

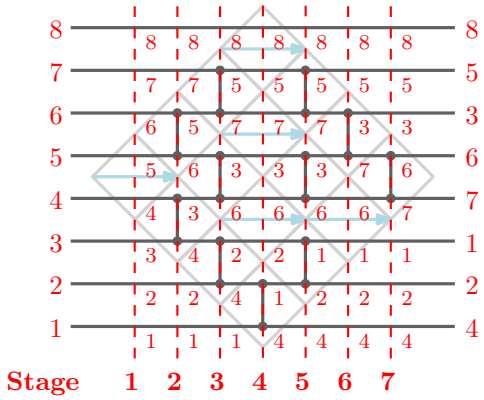


Figure 6.12: Transposition network evaluation by stages

We implement the seaweed algorithm in the same form as shown in Chapter 5 by using a transposition network. We start with a transposition network LCSNET as in Definition 5.2.1 on page 71. For two strings x and y, network LCSNET(x, y) has $|x| + |y|$ wires. Its input for semi-local string comparison is the reverse identity permutation, and from its output we get the nonzeros in the highest-score matrix of x and y. The network LCSNET(x, y) can be decomposed into $|x|+|y|-1$ stages, which have independent comparisons. These stages are equivalent to the antidiagonals of alignment dag cells processed in the last section. However, in the transposition network view, we only work on one vector of values, which is given by the numbers on the wires of the transposition network between stages. Figure 6.12 shows an example for the transposition network approach, where $x =$ `acbc` and $y =$ `abca` (the network has been rotated by 45 degrees for better visibility of the intermediate

results). The red numbers indicate the value on each wire, after every stage. The input of the network is the reverse identity permutation, and the output of the network is the inverse of the seaweed permutation (see Section 3.6). The kernel for evaluating a transposition network is shown in Listing 6.1, the corresponding CPU code is shown in Listing 6.2.

Listing 6.1: Comparison network evaluation GPU kernel

```
kernel void compnet_stage(
```

Variables `x` and `y` point to the input sequences in GPU memory.

```
 char x[], char y[],
```

The following parameters give the intersection of the current stage in the comparison network with the alignment dag. Variable `offset` gives the distance of the first comparator from the first wire, `len` gives the number of alignment dag cells we process. Variables `x_start` and `y_end` give the offsets into the input strings.

```
 int offset, int len,
 int x_start, int y_end,
```

We declare the input and output permutations `p` and `o`. We need random access to `p` since we will want to compare two adjacent values for each cell with a comparator. The output is computed independently for each element.

```
 int p[], out int o <>) {
 int idx = instance().x;
```

We process the output stream vertically, so we check whether there is actually anything to do for this particular cell. This might cause a few kernel calls which do nothing where stages have a small intersection with the alignment dag, but this doesn't seem to hurt performance – the kernel only carries out a few comparisons and copies the data from the input to the output stream. In fact, experiments showed that restricting the range of the output streams on which the kernel is run before calling the kernel code causes more overhead than this approach.

```
 if(idx >= offset && idx < offset+len) {
```

We now have to compute the locations of the characters corresponding to the current alignment dag cell, and determine whether it is a match cell.

```
    int idxd2 = (len - idx+offset - 1) >> 1;
    int match =
      ((int)x[x_start + idxd2] == (int)y[y_end - idxd2]);
    int minmax = ((idx-offset) & 1) == 0 ? 1 : 0;
```

Here, we carry out an individual alignment dag cell operation. If we have a match, no values exchange places, and we can copy the value to the same output wire.

```
      if(match != 0) {
        o = p[idx];
      } else {
```

If the characters do not match, we need to compare the values on two adjacent wires (this is the reason why we needed to declare `p` as a gather stream).

```
        int idx_min = ((idx-offset) & ~1) + offset;
        int idx_max = ((idx-offset) & ~1) + offset + 1;
        if (minmax == 0) {
          o = min(p[idx_min], p[idx_max]);
        } else {
          o = max(p[idx_min], p[idx_max]);
        }
      }
    } else {
      o = p[idx];
    }
}
```

Each individual kernel call processes one alignment dag cell. The kernels are called by the following piece of CPU code.

Listing 6.2: Comparison network evaluation CPU code

```
void GPUSemiLocal(
```

The function gets the input sequences, their lengths and the input seaweed permutation as parameters. The input seaweed permutation can either be the reverse identity permutation if we want to compute the seaweed permutation for `x` and `y`, or any valid inverse seaweed permutation when comparing strings incrementally.

```
const char * x, const char * y,
unsigned int m, unsigned int n,
int * permutation) {
```

We initialize `x_stream` and `y_stream` as copies of `x` and `y` in GPU memory. We also create two streams which will alternate in storing the output data between stages.

```
char x_stream <m>, y_stream<n>;
streamRead(x_stream, x);
streamRead(y_stream, y);
int p_stream<m+n> [2];
streamRead(p_stream[0], permutation)
int _in = 0, _out = 1;
```

The following values give the boundaries of the area in the alignment dag that is relevant for the current stage in the transposition network.

```
int mid = m;  int h = 1;
int x_start = 0; int y_end   = 0;
```

We execute $\mathtt{m}+\mathtt{n}-1$ stages in the network.

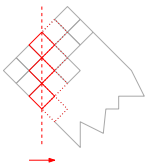
```
for (unsigned int j = 1; j < m+n; ++j) {
  int offset = mid-h;
  int len = 2*h;
```

Here, we schedule the kernel operations on the cells in the alignment dag. Notice that the call to `compnet_stage` is asynchronous and will return immediately. This way, cells will be processed as soon as both their inputs and a thread processor are available. When thinking about our algorithm in the BSP model, all kernel calls between calls of `streamRead` and `streamWrite` correspond to a single superstep that is executed on the GPU and CPU in parallel. Synchronization between GPU and CPU is performed implicitly once we try to copy result data back to the main

memory. Synchronization occurs on the GPU if a compute kernel call cannot be scheduled because it is still waiting either for a previous call to finish or for data to arrive from the main memory.

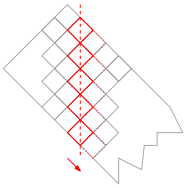
```
compnet_stage(x_stream, y_stream,
        offset, len, x_start, y_end,
        p_stream[_in], p_stream[_out] );
```

We now exchange input and output pointers for the next stage, and compute the new offsets.
In the first triangle of the comparison network, the height of the stage increases, and we move one step forward in string `y`.



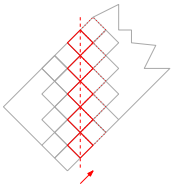
```
swap(_in, _out);
if (j < m && j < n) {
  h+= 1;    y_end+= 1;
```

If `m` > `n`, we get an area in the middle of the network where `h` = `n`. We compare the entire string `y` to a sliding window in `x`.



```
} else if(j < m && j >= n) {
  mid-= 1;    x_start+= 1;
```

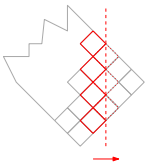
Symmetrically, if `m` < `n`, we get an area in the middle of the network where `h` = `m` and where we compare the entire string `x` to a sliding window in `y`.



```
} else if(j < n && j >= m) {
  mid+= 1;    y_end+= 1;
```

Finally, in the last stages of the comparison network, the length of the stages will decrease, and we move forward in x.

```
    } else {  // j >= m && j >= n
      h-= 1;    x_start+= 1;
    }
  }
```



In the final step of our algorithm, we need to invert the output permutation, since the transposition network computes the inverse seaweed permutation.

```
  swap(_in, _out);
  invert_permutation(*p_stream[_in], *p_stream[_out]);
}
```

The GPU implementation shown above did not perform as well as the algorithm from the previous section. The main issue is that it does not allow for using vector processing on large enough chunks of the data, which causes the overhead for scheduling computations on thread processors to become dominant in the running time. A simple workaround for this problem is to use the GPU to compare one long string y to multiple short strings $x_1, \ldots, x_k$ at the same time. An adapted kernel is shown in Listing 6.3.

Listing 6.3: Comparison network evaluation for multiple strings

```
kernel void compnet_stage_2d (
        char x[][], char y[],
        int offset, int len, int x_start, int y_end,
        int p[][], out int o<>) {
    int idx = instance().y;
    int idy = instance().x;
    if(idx >= offset && idx < offset+len) {
        int idxd2 = (len - idx+offset - 1) >> 1;
        int c_x = (int)x[idy][x_start + idxd2];
        int c_y = (int)y[y_end - idxd2];
        int match = (c_x == c_y) ? 1 : 0;
```

```
        int minmax = ((idx-offset) & 1) == 0 ? 1 : 0;
        if(match == 0) {
            int idx_min = ((idx-offset) & ~1) + offset;
            int idx_max = idx_min + 1;
            int val_l = p[idx_min][idy];
            int val_t = p[idx_max][idy];
            if(minmax == 0) {
                o = min(val_l, val_t);
            } else {
                o = max(val_l, val_t);
            }
        } else {
            o = p[idx][idy];
        }
    } else {
        o = p[idx][idy];    } }
```

The main lessons learned from implementing the seaweed algorithm on a GPU can be summarized as follows. The algorithm using only vertical vector operations turned out to be unsuitable for GPU implementation, since it creates too much overhead when implemented using the Brook+ instructions. Direct implementation of the transposition network method gave better results, in particular when scheduling the computation of multiple strips in the alignment dag in parallel to take advantage of large numbers of thread processors. The main difference between SIMD vector parallelism on a CPU and a GPU is the degree of vector parallelism that is possible, and the higher degree of flexibility in implementing it. The number of data items a CPU can process in parallel is restricted to 8 or 16, depending on the instruction set used. On the other hand, the CPU has a more elaborate setup with data caches and pipelines, which makes it easier to achieve good data throughput. On a GPU, we can easily have hundreds or thousands of (simple) thread processing units, which can execute code in parallel. However, the parallelism offered by these units can only be exploited by careful algorithm implementation, since otherwise data

sharing and memory access bandwidth can become bottlenecks which slow down the computation. Furthermore, smaller computations must be bundled to make best use of the scheduling mechanisms. We will show a summary of experimental results in Section 6.8.

6.6 Reducing redundant computation for small window sizes

Although the vector-parallel algorithms have a reduced dependency on the window size, they still have an asymptotic running time that is worse compared to the best theoretical method, which runs in time $O(mn)$. Tiskin [2008b] suggests a tree approach to avoid recomputing all seaweeds in each strip of height w. Implicit highest-score matrices are computed for strips of height 2^l, $l \in 1 \ldots \log_2 w$, and merged using a sub-quadratic highest-score matrix composition procedure. However, this procedure potentially has a high computational overhead, and requires implementing the complicated highest-score matrix multiplication procedures shown in Chapter 4. Therefore, obtaining a practical advantage from using this method is not likely for small window sizes.

Here, we propose a more practical improvement to the vector-parallel algorithm that reduces amount of overlapping computation. The seaweeds corresponding to multiple w-windows in one input string can be computed by extending a smaller strip of seaweeds at the top and at the bottom (see Figure 6.13). In order to implement such an overlap method, we need to update a highest-score matrix in the two cases when a character is inserted either at the beginning or at the end of one of the input strings. The case of inserting a character at the end is trivial: we append a row of cells in the alignment dag and execute the seaweed algorithm on it. However, in order to compute the seaweeds resulting from appending a row at the top of the alignment dag requires a slight modification of the seaweed algorithm.

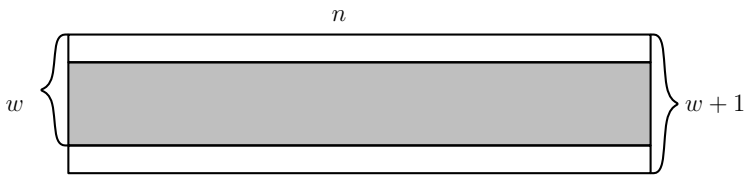


Figure 6.13: Using strip overlap to speed up computation

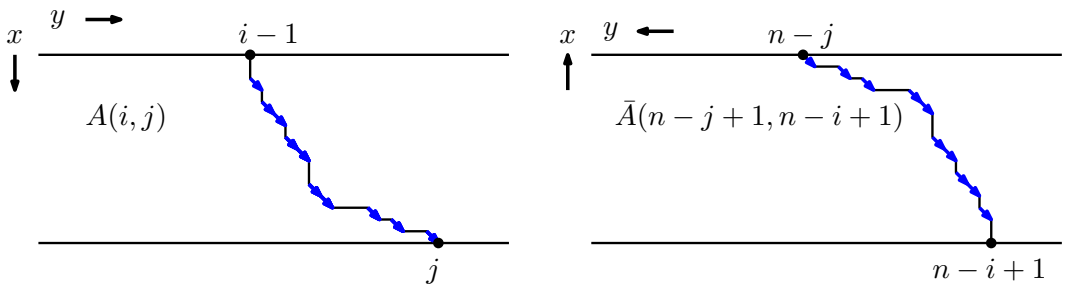
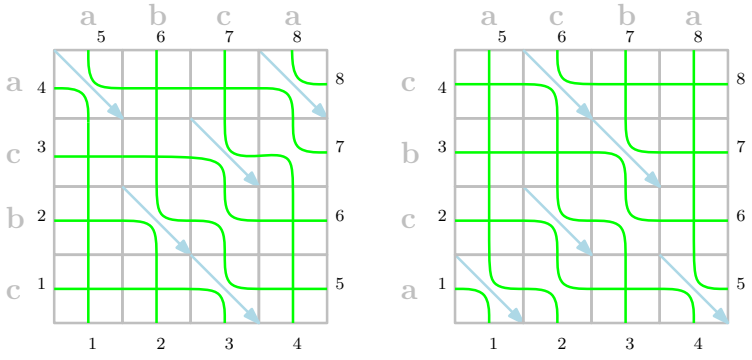


Figure 6.14: Highest-scoring paths for reversals of input strings



Seaweed permutation:
{ 3, 2, 6, 1, 7, 5, 4, 8 }
Reverse permutation:
{ 8, 4, 5, 7, 1, 6, 2, 3 }
Inverse of reverse:
{ 5, 7, 8, 2, 3, 6, 4, 1 }
Reverse of inverse of reverse:
{ 1, 4, 6, 3, 2, 8, 7, 5 }

Seaweed permutation:
{ 1, 4, 6, 3, 2, 8, 7, 5 }
Reverse permutation:
{ 5, 7, 8, 2, 3, 6, 4, 1 }
Inverse of reverse:
{ 8, 4, 5, 7, 1, 6, 2, 3 }
Reverse of inverse of reverse:
{ 3, 2, 6, 1, 7, 5, 4, 8 }

Figure 6.15: Example for Lemma 6.6.1

Lemma 6.6.1. *Consider the implicit highest-score matrix $\bar{A}$ for comparing the reversals of both input strings. We can obtain the reverse seaweed permutation $\bar{A}$ as the inverse of the reverse seaweed permutation of A.*

Proof. From the definition of the extended alignment dag, and the fact that the extended alignment dag $G_{\bar{x},*\bar{y}*}$ is the horizontal and vertical mirror image of $G_{x,*y*}$, we get $A(i,j) = \bar{A}(n-j+1, n-i+1)$, since any path that is longest between $v_{0,i-1}$ and $v_{m,j}$ in the alignment dag corresponding to A will contain the same (reversed) edges as a highest-scoring path between vertices $\bar{v}_{0,n-j+1}$ and $\bar{v}_{m,n-i+1}$ in $G_{\bar{x},*\bar{y}*}$ corresponding to $\bar{A}$ (see Figure 6.14).

$$\begin{aligned} A(i,j) &= \bar{A}(n-j, n-i) \\ &= n-i+1-n+j-1 - \sum_{\hat{\imath}>n-j+1,\ \hat{\jmath}<n-i+1} \bar{P}_A(\hat{\imath},\hat{\jmath}) \\ &= j-i - \sum_{\hat{\imath}>i,\ \hat{\jmath}<j} P_A(\hat{\imath},\hat{\jmath}) \end{aligned} \tag{6.4}$$

Therefore, we have

$$\sum_{\hat{\imath}'>n-j+1,\ \hat{\jmath}'<n-i+1} \bar{P}_A(\hat{\imath}',\hat{\jmath}') = \sum_{\hat{\imath}>i,\ \hat{\jmath}<j} P_A(\hat{\imath},\hat{\jmath}). \tag{6.5}$$

We substitute $\hat{\imath}' = n+1-\hat{\jmath}$ and $\hat{\jmath}' = n+1-\hat{\imath}$ (this accounts for the two sequence reversals), and transpose $\bar{P}_A$ to obtain:

$$\sum_{\hat{\imath}>i,\ \hat{\jmath}<j} \bar{P}_A^T(n+1-\hat{\imath}, n+1-\hat{\jmath}') = \sum_{\hat{\imath}>i,\ \hat{\jmath}<j} P_A(\hat{\imath},\hat{\jmath}). \tag{6.6}$$

Since both P_A and $\bar{P}_A$ are permutation matrices, and since (6.6) has to hold for all $(i,j) \in [-\infty : \infty] \times [-\infty : \infty]$, we have $P_A(\hat{\imath},\hat{\jmath}) = \bar{P}_A^T(n+1-\hat{\imath}, n+1-\hat{\jmath}')$. Transposing a permutation matrix is equivalent to finding the inverse permutation, while transforming $\hat{\imath}$ to $n+1-\hat{\imath}$ and $\hat{\jmath}$ to $n+1-\hat{\jmath}$ is equivalent to reversing the permutation. □

An example for Lemma 6.6.1 is shown in Figure 6.15. As a result of this lemma, we can compute seaweeds incrementally by adding characters in the beginning as well as to the end of an input string. When adding a single character, the resulting implicit highest-score matrix can be computed in linear time.

Lemma 6.6.2. *Given the implicit highest-score matrix P_A for strings x and y of lengths m and n, and a character σ, we can compute the implicit highest-score matrix for comparing σx to y in time $O(m+n)$.*

Proof. We can compute the reverse and inverse of a $(m+n)$-permutation in $O(m+n)$ time and space, and therefore obtain the implicit highest-score matrix for comparing $\bar{x}$ and $\bar{y}$. Using the seaweed algorithm, we can compute the implicit highest-score matrix for $\bar{x}\sigma$ compared to $\bar{y}$. By applying Lemma 6.6.1 again, we obtain the implicit highest score matrix for comparing σx and y. □

Theorem 6.6.3. *The alignment plot with window size w for two input strings x and y of lengths m and n can be computed in time $O(mn\sqrt{w})$.*

Proof. Lemma 6.6.2 allows us to re-use the common part of two or more strips of height w in the alignment dag. Using the original $O(mnw)$ method, we need to evaluate mnw cells to obtain all window scores. We now compute k adjacent strips starting with the part of the alignment dag in which they all overlap. These k strips cover a substring of length $w+k-1$ in x. Their overlap in the alignment dag is of height $w-k$. We first obtain $k/2$ implicit highest-score matrices by extending the strip of overlapping cells to the bottom. Each of these highest score matrices needs to be extended upwards to obtain a full strip of height w (see Figure 6.16). To compute the seaweeds for k strips, we now have work

$$c(k) = \underbrace{n(w-k)}_{\text{overlapping cells}} + \underbrace{n \cdot (k-1+1)}_{\text{extend downwards}} + \underbrace{n \sum_{j=1}^{k-1} (k-1-j)}_{\text{extend upwards}}.$$

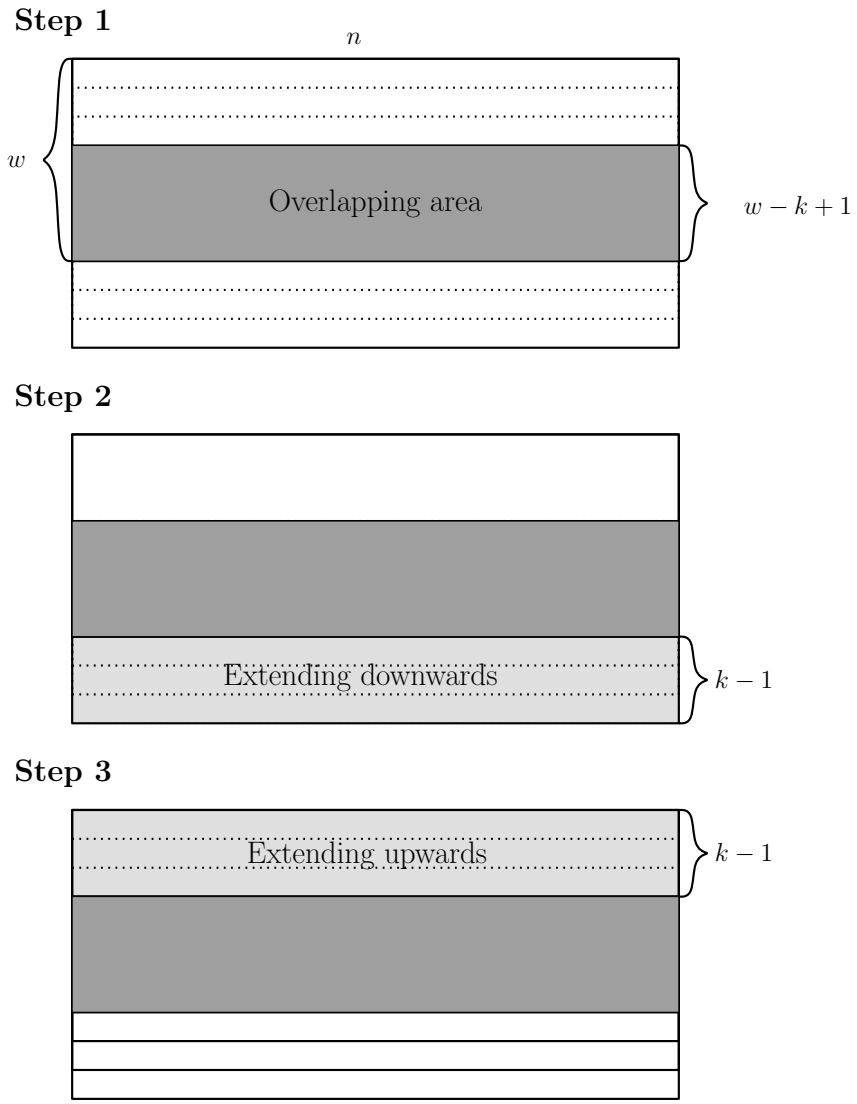


Figure 6.16: Computing seaweeds for k overlapping strips

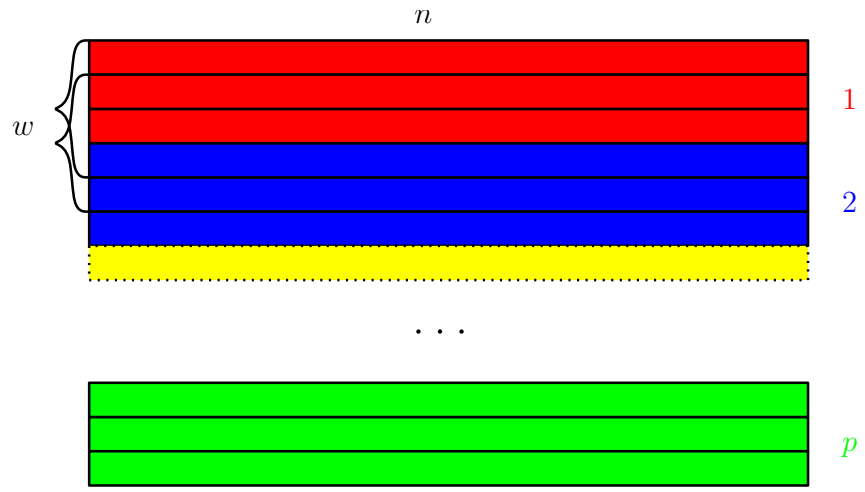


Figure 6.17: Distributing strip computations

The total number of cells we need to process is

$$t(k) = c(k) \cdot \frac{m}{k}.$$

We can compute the minimum number of cells by minimizing $c(k) \cdot \frac{m}{k}$. When using $k = 2\sqrt{2w+2}$, we get

$$t(k) = mn\left(2\frac{w}{\sqrt{2w+2}} + \frac{1}{2}(\sqrt{2w+2}-1) - 2\right)$$

and therefore asymptotic running time $O(mn\sqrt{w})$. □

The optimal value of k to actually use in a computation should be determined experimentally, since this also accounts for the overhead incurred by having to copy and invert the permutations.

6.7 A coarse-grained parallel algorithm

Parallelizing alignment plot computation on a coarse-grained level is straightforward. Assume we have a BSP computer with p processors. We broadcast one of the input sequences to all nodes, and split the other one into p substrings of equal size.

The computation on these substrings can then be performed in parallel, and the resulting scores can be translated to scores for the original problem by adding an offset to each window position. An illustration is shown in Figure 6.17. We obtain parallel running time $W(m,n,w,p) = O(\frac{mnw}{p})$, communication $H(m,n,w,p) = O(\frac{m}{p})$, synchronization $S = O(1)$ and memory $M(m,n,w,p) = O(n + \frac{m}{p})$. We can also distribute the computation using the overlap method from Section 6.6 in the same way to obtain parallel running time $W(m,n,w,p) = O(\frac{mn\sqrt{w}}{p})$.

6.8 Experimental results

In order to evaluate the performance of our method, we have implemented the algorithms from the previous sections. The implementation uses C++ with Intel MMX/SSE instructions for performing vector arithmetic (see Appendix A). We compare an optimized implementation of the seaweed algorithm for Intel processors, and an implementation running on graphics processors by AMD/ATI (see Advanced Micro Devices Inc. [2010]). The algorithms were implemented using C++ and optimized assembly code, and using the Brook+ compiler supplied by AMD for implementing the GPU code discussed in Section 6.5. Experiments with the GPU code took place on a Windows-PC (32-bit) with a 2.4 GHz Core2-Quad processor and 4GB of RAM. The graphics processor used in our test is an AMD/ATI Radeon HD 4870 card with 512 megabytes of memory. We also evaluated the performance of the coarse-grained parallel version of the code on 64-bit Linux systems using MPI and TBB (Intel Corporation [2009]).

As input data for our tests, we used different biological sequence data sets and a fixed window size of 100. The nature of the sequences does not affect the running time of our algorithm, but may affect the impact of the heuristic speedup employed by one of the methods to which we compare the performance results. In all experiments, we used a vertical step size of 5, i.e. we only compare every fifth window in the first input to all windows in the other string. Using the scoring scheme

Table 6.1: Execution times in seconds and speedups

Data Set	**Mikey**	**Berti**	**Jimmy**	**Henry**
Input Size	2712×628	2712×2305	15097×96901	80001×80001
Data-parallel algorithm speedup on Linux/x86_64/1.83GHz Core2-duo, gcc 4.3.1				
Heur	5.1 (÷ 1.0)	41.1 (÷ 1.0)	2677 (÷ 1.0)	11708 (÷ 1.0)
BLCS	3.6 (÷ 1.4)	37.3 (÷ 1.1)	3680 (÷ 0.7)	16191 (÷ 0.7)
Sea-16	1.4 (÷ 3.6)	10.8 (÷ 3.8)	1026 (÷ 2.6)	4514 (÷ 2.6)
Sea-8	0.5 (÷ 10.2)	3.8 (÷ 10.8)	368 (÷ 7.3)	1614 (÷ 7.3)
Sea-8SMP	0.3 (÷ 17.0)	3.4 (÷ 12.1)	210 (÷ 12.7)	821 (÷ 14.3)

Table 6.2: Execution times in seconds using overlapping strips and a GPU

Data Set	**Berti**	**Jimmy**	**Henry**
Input Size	2712×2305	15097×96901	80001×80001
NW-Align	40	2571	11708
Sea-nonoverlap-cpu ($k = 1$)	5.8	554	2410
Sea-nonoverlap-gpu ($k = 1$)	5.1	422	1759
Sea-overlap-gpu ($k = 4$)	4.8	381	1596

Table 6.3: Execution times in seconds and speedups for the MPI Version (Sea-8)

Data Set	**Mikey**	**Berti**	**Jimmy**	**Henry**
Input Size	2712×628	2712×2305	15097×96901	80001×80001
Linux desktop system, Core2-quad 2.66GHz, 64-bit, MPI, gcc 4.3.1				
1 core	0.4 (÷ 1.0)	2.9 (÷ 1.0)	271 (÷ 1.0)	1199 (÷ 1.0)
2 cores	0.7 (÷ 0.6)	1.8 (÷ 1.6)	142 (÷ 1.9)	611 (÷ 2.0)
4 cores	0.7 (÷ 0.6)	1.3 (÷ 2.2)	70 (÷ 3.9)	307 (÷ 3.9)
Apple Mac Pro task farm, 2× dual-core Xeon 3GHz per node, 32-bit				
1 core	2.7 (÷ 1.0)	9.7 (÷ 1.0)	821 (÷ 1.0)	3771 (÷ 1.0)
4 cores	1.5 (÷ 1.8)	3.6 (÷ 2.7)	243 (÷ 3.4)	1061 (÷ 3.6)
8 cores	2.0 (÷ 1.4)	2.8 (÷ 3.5)	150 (÷ 5.6)	666 (÷ 5.7)
16 cores	2.4 (÷ 1.1)	3.4 (÷ 2.9)	94 (÷ 8.7)	392 (÷ 9.6)
32 cores	2.0 (÷ 1.4)	3.7 (÷ 2.6)	55 (÷ 14.8)	227 (÷ 16.6)
IBM Cluster, 2× dual-core Xeon 3GHz/node, QLogic InfiniPath network, gcc 4.1.2				
1 core	0.67 (÷ 1.0)	3.1 (÷ 1.0)	225 (÷ 1.0)	991 (÷ 1.0)
4 cores	0.57 (÷ 1.2)	1.4 (÷ 2.2)	58 (÷ 3.9)	251 (÷ 3.9)
8 cores	0.60 (÷ 1.1)	1.2 (÷ 2.6)	31 (÷ 7.4)	129 (÷ 7.7)
16 cores	1.26 (÷ 0.5)	1.6 (÷ 1.9)	20 (÷ 11.5)	66 (÷ 14.9)
32 cores	–	–	12 (÷ 19.1)	41 (÷ 24.0)
64 cores	–	–	11 (÷ 20.5)	23 (÷ 42.4)

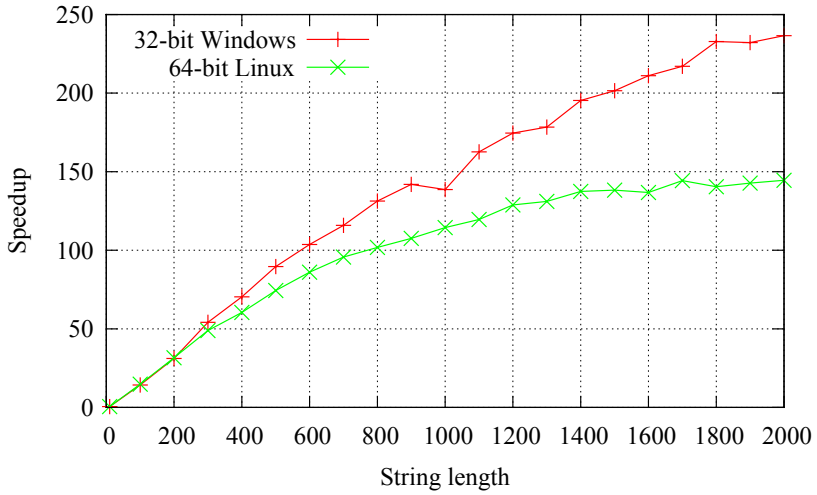


Figure 6.18: Speedup of bit-parallel LCS computation over dynamic programming

as described in Example 6.2.1 induces a window size of 200 due to the constant-size blowup of the alignment dag.

For comparing the results to existing methods, we implemented a bit-parallel LCS computation algorithm (Crochemore et al. [2001]) to compute the pairwise alignment scores ("BLCS"). Figure 6.18 shows the speedup of using this method over LCS computation by the standard dynamic programming algorithm for different string lengths (we compare the running times of the algorithms for computing the LCS of two random input strings of the same length, using 3 bits per character). Our 64-bit implementation of this algorithm can achieve a speedup of around 32 over the standard dynamic programming algorithm for inputs of length 200. Note that our implementation achieves speedups greater than the machine word size since it uses highly optimized assembler code to perform the necessary bit addition operations (see Appendix A), whereas the dynamic programming code has been implemented using C++ to simulate running times comparable to the code used by Ott et al. [2009].

Furthermore, we compared our results to the code used by Ott et al. [2009] ("Heur") which uses the standard dynamic programming algorithm (Wagner and Fischer [1974]; Needleman and Wunsch [1970]) and a heuristic to speed up compu-

tation when a minimum alignment score for a window pair is specified. In "Sea-16", vectors of 16-bit values were used. Using the results from Section 6.4, we can improve this to use 8-bit values, by computing $(200, 2)$-restricted highest-score matrices. The results of our experiments are shown in Table 6.1. We see that the seaweed-based algorithm is fastest for all data sets. We also see that the heuristic employed by Heur makes this algorithm more effective than the BLCS method for long sequences. However, we note that it would be possible to adapt BLCS to make use of the same heuristic speedup. Overall, these results show that the seaweed algorithm is highly competitive against the repeated dynamic programming approach, and that particularly the byte vector version ("Sea-8") is more than seven times faster than the best existing method.

Furthermore, we show performance results for the GPU version of our algorithm in Table 6.2. We see that the new method is significantly faster than its competitor NW-Align, and that using the graphics processor ("Sea-nonoverlap-gpu")[4] also gives improved performance over the already heavily optimized CPU code ("Sea-nonoverlap-cpu") from Section 6.4, particularly for large problem sizes. Furthermore, we see that the overlap method introduced in the last section currently gives a performance improvement of around 10%. However, we believe that our implementation of this method has much potential for improvement by reducing the overhead induced by extending the strips, and thereby achieving a more noticeable improvement. Another improvement to our code could be to execute computations on the graphics processor and on the CPU in parallel. In the current version, the graphics processor is used to pre-compute the implicit highest-score matrices for all overlapping areas in the alignment dag, which are then extended to the full strips of height w on the CPU. This allows us to compare the performance between the graphics processor and CPU implementations. For practical purposes, it would be sensible to implement workload sharing between the CPU and the graphics processor, which would allow us to use both units in parallel for computing and extending the highest-score matrices in the algorithm from Section 6.6.

[4]Note that using the $O(mnw)$ algorithm which does not exploit overlaps is equivalent to setting $k = 1$ in the overlap method.

We also conducted experiments to study the scalability on larger numbers of processors as shown in Section 6.7. Each processor produces a local output file, these are merged in a postprocessing step to obtain the full alignment plot. We obtained good speedup especially for the large datasets both on small and larger parallel systems (see Table 6.3). Note that our sample datasets are still rather small. An interesting application for the algorithm would be whole-genome comparison, which involves much larger input sequences, and hence better speedup on more processors.

Chapter 7

Conclusion and Outlook

7.1 Summary

In this thesis, we have presented a unified approach to modelling and exploiting parallelism in string alignment algorithms. Our approach encompasses multiple types of parallelism, from the bit level to coarse grained parallel computation. We consider string alignment as a special case of semi-local string comparison, which has been recognized as a natural way to decompose string alignment into subproblems that can be distributed for being solved in parallel. The first main result of this thesis uses recent results on semi-local string comparison to achieve scalability in the asymptotic communication requirements for string alignment computation. For this, we propose a revised approach to analyzing BSP algorithms which includes the analysis of communication and I/O cost, as well as of the local memory requirements. We define asymptotic scalability in memory and communication properties, which are important both for algorithms on loosely coupled parallel systems, as well as for obtaining scalable algorithms for modern multi-core and SMP computers. Scalable communication captures the input/output cost of decomposing a problem into subproblems, and achieving scalable memory enables the partitioning of a problem into smaller subproblems which fit into processor caches or into small amounts of local memory. Communication cost can impact the performance of a parallel algorithm even if it is asymptotically dominated by the computation cost, particularly

when communication does not scale as well as computation and large numbers of processors are used, or if large problem inputs need to be distributed using a slow communication network. In this thesis, we show how to achieve scalable memory and communication for computing longest common subsequences, as well as for computing longest increasing subsequences. We show improved theoretical complexities for communication and memory, and demonstrate the practicality of these results using a simple performance model. Table 7.1 shows a summary of our results for comparing two strings of length n in parallel with scalable communication.

The scope of applying semi-local string comparison is not limited to obtaining coarse-grained parallel algorithms. We also show how bit-parallel algorithms for LCS computation relate to semi-local string comparison, and how to obtain efficient algorithms which are parameterized by the similarity or dissimilarity of the input strings. This is achieved by reducing the semi-local string comparison problem to the evaluation of a specific class of comparison networks. By this approach, we can obtain many classical algorithms for LCS computation, including parameterized and bit-parallel algorithms, and also improved bounds for parameterized semi-local LCS computation. Table 7.2 shows a summary of results for parameterized semi-local string comparison of two strings.

In the last part of this work, we study a practical application of semi-local string comparison: computing alignment plots of biological sequences. We show how theoretical methods based on semi-local string comparison can be implemented efficiently to obtain a fast tool for loss-free local sequence comparison. This tool has also been applied at the Systems Biology Centre at Warwick University, and is now part of a web service which provides access to tools for evolutionary sequence analysis. Furthermore, this implementation provides a starting point for an algorithm engineering project to implement fast, reusable software for local string comparison. Another outcome of this last part is the establishment of a relationship between our theoretical model of parallel computation and the practical implementation of algorithms on multi-core computers and graphics processors.

Problem	$W(n,p)$	$H(n,p)$	$M(n,p)$	S	**Slackness**	**Page**
simple unit-Monge matrix multiplication	$O\left(\frac{n^{1.5}}{p}\right)$	$O\left(\frac{n}{\sqrt{p}}\log p\right)$	$O\left(\frac{n}{\sqrt{p}}\right)$	$O(1)$	$n > p^2$	p. 37
	$O\left(\frac{n\log n}{p}\right)$	$O\left(\frac{n}{p}\log p\right)$	$O\left(\frac{n}{p}\right)$	$O(\log p)$	$n > p^3$	p. 47
semi-local LCS	$O\left(\frac{n^2}{p}\right)$	$O\left(\frac{n}{\sqrt{p}}\log p\right)$	$O\left(\frac{n}{\sqrt{p}}\right)$	$O(\log p)$	$n > p^2$	using p. 37
	$O\left(\frac{n^2}{p}\right)$	$O\left(\frac{n}{\sqrt{p}}\right)$	$O\left(\frac{n}{\sqrt{p}}\right)$	$O(\log^2 p)$	$n > p^3$	see p. 56
LIS/permutation string LCS	$O\left(\frac{n\log^2 n}{p}\right)$	$O\left(\frac{n}{p}\log p\right)$	$O\left(\frac{n}{p}\right)$	$O(\log^2 p)$	$n > p^3$	see p. 59

Table 7.1: Results on parallel string comparison

Parameterisation	Running time	Page
Highly dissimilar strings of length n		
Number of matching character pairs r	$O(n\sqrt{r})$	p. 73
Length of the LCS p	$O(np)$	p. 79
Run-length compressed strings of length m and n		
Compressed string lengths $\bar{m}$ and $\bar{n}$	$O(m\bar{n} + \bar{m}n)$	p. 78

Table 7.2: Results on parameterized semi-local string comparison

7.2 Outlook

Building on the results of this thesis, several directions of future research exist. Firstly, we believe that it is possible to improve the number of supersteps required for the work-optimal highest-score matrix multiplication procedure shown in Section 4.3 to $O(1)$. This would make it superior to the method from Section 4.2 in all aspects and improve the synchronization complexity for all the algorithms depending on this procedure by a $\log p$-factor. The challenge this entails is to avoid exponential slackness, i.e. requiring $p < \log n$. However, this improvement is mostly of theoretical interest, since synchronization usually makes up for only a small part of the running time on modern parallel systems (see e.g. Krusche and Tiskin [2006]).

It remains an open question whether it is possible to obtain a generally work-optimal parallel algorithm for the LIS problem running in time $O((n \log n)/p)$ in the comparison-based model, or $O((n \log \log n)/p)$ in the integer arithmetic model. Such an algorithm would be of theoretical as well as practical interest. It is astonishing that no generally work-optimal parallel algorithm for such an "old" problem is known yet. Furthermore, the problem has applications in DNA comparison (Delcher et al. [1999]) and other areas (see e.g. Bar-Yehuda and Fogel [1998]).

It would also be interesting to implement the algorithms shown in Chapter 4 and study their practicality. An immediate application for such an implementation would be extending the tools for alignment plots shown in Chapter 6 to use an

$O(mn)$-time method for computing alignment plots. Similarly, the implementation of highest-score matrix composition algorithms in combination with suffix trees or related data structures (McCreight [1976]; Crochemore et al. [2007]) could be applied to providing an efficient database of fully local (substring to substring) alignments.

Another direction of further work on alignment plots is the extension of our current software to allow comparison on the full-genome scale. Currently, such computations would require a substantial amount of computation time on a large-scale parallel computer. Improving the GPU-based computation code, implementing pre-processing methods, and combining the seaweed method with some type of data compression would allow for more cost-efficient local comparison of larger input sequences.

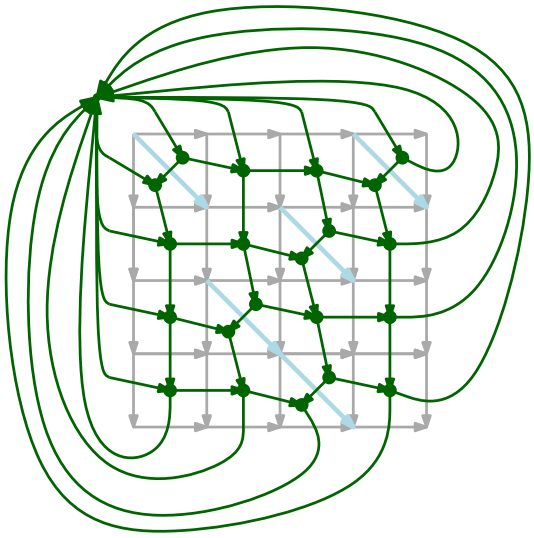


Figure 7.1: Dual graph of the alignment dag

Finally, all work on string alignment in this thesis is based on algorithms for computing longest common subsequences, since the methods for semi-local string comparison by Tiskin [2010a] are limited to highest-score matrices which are unit-Monge only. One possibility of extending these methods could be to consider the seaweeds from Definition 3.5.4 as paths in the dual graph of the alignment dag (see Figure 7.1). An interesting direction of future research would be to study the exact

relationship between seaweeds and their representation as paths in this dual graph. Many problems in planar graphs have efficient solutions involving paths in the dual graph (see e.g. the work by Khuller et al. [1993], and Klein [2005]), and using these methods might allow generalizing the seaweed method to alignment dags with more general weights.

Appendix A

A vector library using MMX/SSE

A.1 Introduction

We implement vector parallelism using a parallel vector function library similar to Framewave (The Framewave Group [2009]) or Intel's performance primitives (Intel Corporation [2010b]). This library provides optimized vector operations for implementing the seaweed algorithm. By design choice, our library does not use multithreading, we make use of multithreaded programming on a higher level of parallel algorithm implementation (see Appendix B). The main goal of this library is to provide an efficient basis for implementing word-RAM style algorithms in a high level language. Our library provides functions that work on vectors of k-bit integers. It provides functions for efficient storage, allocation, access, and manipulation of these vectors. Since the requirements for string comparison algorithms range from generic operations, like bit shifts, logical operations and arithmetic, to specific ones for implementing bit-parallel LCS algorithms, the decision was made to implement a custom library. Our library is portable to most current platforms running on Intel processors, namely Windows (Microsoft Corporation [2010]), Linux (Wikipedia [2010]), and MacOS X Darwin (Apple Inc. [2010]). Our implementation mainly uses the C++ programming language (see Stroustrup [1987] for reference).

Some parts have been implemented using optimized assembler code for the Intel architecture (Fog [2010] gives a wealth of material on this topic). Our library employs a few advanced techniques using C++ templates (see Alexandrescu [2001] for an introduction, and Vandevoorde and Josuttis [2003] for reference) to provide good performance for a range of library parameters. The aim of this section is to give an highlight the capabilities of this library, and give an overview of key issues in its implementation.

A.2 Programmer's interface

The vector library is composed of two parts: A high level abstraction of vector operations, and a low level implementation part using assembler language and C code.

The programmer's interface is given by the following classes.

1. `template <BYTE value_bits> class IntegerVector`: Objects of this type encapsulate functionality for dealing with words of `value_bits` bits.

2. `class BitString`: This class is equivalent to `IntegerVector< 1 >`

3. `template<int _bpc, UINT64 _mapval = 1, bool _invert = true> class CharMapping`: This class implements character match mappings for bit-parallel string comparison algorithms. Let σ be a `_bpc`-bit character, and x a string of n `_bpc`-bit characters. An object of type `CharMapping` will map any such character σ to a `BitString` b with

$$b[i] = \begin{cases} 1 & \text{if } x[i] = \sigma \\ 0 & \text{otherwise.} \end{cases}$$

We will now give descriptions for the `IntegerVector` and `CharMapping` classes.

A.2.1 Class `IntegerVector`

We allow specifying the number of bits per word as a template parameter. Variable `value_bits` specifies how many bits are contained in each word in the vector.

```
template <BYTE value_bits>
class IntegerVector {
public:
typedef IntegerVector<value_bits> my_type
```

Depending on `value_bits`, we declare two constants. Constant `msb` has only the most significant bit set, i.e. $\texttt{msb} = 2^{\texttt{value\_bits}-1}$. Constant `lsbs` has all bits in a word set, i.e. $\texttt{lsbs} = 2^{\texttt{value\_bits}} - 1$.

```
static const UINT64 msb = (static_cast<UINT64>(1))
                                  << (value_bits - 1);
static const UINT64 lsbs = 2*msb - 1;
```

We give a standard constructor, which initializes an empty vector of a given size and a copy constructor.

```
IntegerVector(size_t words = 0);
IntegerVector(my_type const & v);
```

Another way of initializing a vector is as a *slice* of another vector. This operation allows performing operations locally on smaller parts of a bigger vector to save computation time.

```
IntegerVector(my_type & data, size_t _vword_len);
```

Furthermore, we can construct a vector from a string. This constructor translates the input string lexicographically into numbers. Character `'A'` is translated into 1, `'B'` into 2, and so on. Every ASCII character smaller than `'A'` is translated into a zero.

```
IntegerVector(const char * string);
IntegerVector(std::string const & s);
```

The following functions are used to initialize all words in the vector to either zero, or to have all bits set.

```
void zero();
void one();
```

We also supply a few common string operations like reversal, resizing, assignment, substring extraction and concatenation.

```
void reverse();
void append(my_type const & a);
void resize(size_t words);
my_type & operator= (my_type const& v);
/**
 * Extract characters (*this)[start, end]
 * (range including start and end)
 */
void extract_substring(size_t start,
    size_t end, my_type & target) const ;

/**
 * Compatibility with std::string::substr. extract length
 * characters, starting at position
 */
my_type substr(size_t position, size_t length) const ;
```

The following functions return the size of the vector and allow direct access to its raw data. We will discuss in Subsection A.3.5 how this can be used to only manipulate slices of a vector.

```
/**
 * return the number of words in this vector */
size_t size() const ;
/**
 * direct data access pointer */
char * data() const ;
/**
 * direct data access size */
size_t datasize() const;
```

```
/**
 * move the start of the data slice of the vector. Only valid
 * if constructed using data from another vector, and if
 * value_bits divides 64 bits. */
void shiftstart(int offset);
```

Since all words are stored consecutively and may not be aligned to byte or word boundaries, element access is implemented by means of a proxy class which handles reading and writing. This allows accessing the vector elements in the same fashion as a standard C++ array at the expense of some overhead for proxy class creation. When optimizing code, such accesses may be replaced by calls to direct calls to the functions `put` and `get`.

```
UINT64_proxy<value_bits> operator[] (size_t ofs) const
void put(size_t pos, int value) ;
int get(size_t pos) const ;
```

We provide a set of functions which combine vectors element by element. The first such function generates a match mask in the current vector: given two strings `s1` and `s2`, it performs the following operation: $(*\texttt{this})[\texttt{i}] = 0$ if $\texttt{s1}[\texttt{i}] \neq \texttt{s2}[\texttt{i}]$, and $(*\texttt{this})[\texttt{i}] = \texttt{lsbs}$ otherwise. The counterpiece to this operation replaces all values in the current vector with other values $(*\texttt{this})[\texttt{i}] = \texttt{vy}[\texttt{i}]$ if $\texttt{mask}[\texttt{i}] = \texttt{lsbs}$.

```
void generate_match_mask(const my_type & s1,
                         const my_type & s2);
void replace_if(my_type const & mask, my_type const & vy);
```

We also provide a compare-exchange function. This function exchanges elements `(*this)[i]` and `t[i]` if and only if $(*\texttt{this})[\texttt{i}] > \texttt{t}[\texttt{i}]$.

```
void cmpxchg(my_type & t);
```

The following functions perform element-wise operations as follows: $(*\texttt{this})[\texttt{i}] = (*\texttt{this})[\texttt{i}] \circ \texttt{v}[\texttt{i}]$ where $\circ \in \{\&, |, \hat{}\}$. We also provide bitwise negation.

```
my_type & operator&= (my_type const& v);
my_type & operator|= (my_type const& v);
my_type & operator^= (my_type const& v);
```

```
void negate() ;
my_type operator~() const ;
```

Another type of operation combines each vector element with one constant `j`: $(*\texttt{this})[\texttt{i}] = (*\texttt{this})[\texttt{i}] \circ \texttt{j} \quad \text{where} \quad \circ \in \{+, -, \&, |, \hat{}\}$.

```
my_type & operator+= (UINT64 j);
my_type & operator-= (UINT64 j);
my_type & operator&= (UINT64 j);
my_type & operator|= (UINT64 j);
my_type & operator^= (UINT64 j);
```

The following functions provide a shorthand way to set and reset specific bits in every element in the vector.

```
void set_bits(UINT64 bits);
void and_bits(UINT64 bits);
void reset_bits(UINT64 bits);
```

The following functions provide some specific element-wise functionalities required by some of the algorithms implemented in this thesis.

```
/**
 * Test in each word if a specific bit is set.
 * Set each word that has the bit set to lsbs, all others
 * to 0 */
void bittest(int c = 0) ;
/**
 * Saturated increment, only increments a word
 * if it does not have all bits are set already */
void saturated_inc() ;
/**
 * Set all words to lsbs which are equal to zero,
 * reset all other words to zero. */
void test_zero() ;
```

The following operations regard the vector as a long $n \cdot$`value_bits`-sized integer. We start with bit shifts of the entire vector, ignoring word boundaries. The following operations can be used to shift all vector data by a given number of bits.

```
my_type & operator<<= (int shift);
my_type & operator>>= (int shift);
```

The following function implements the basic addition operation for bit-parallel LCS computation introduced by Crochemore et al. [2001]: (∗`this`) = ((∗`this`)&`v`) + ((∗`this`)& ∼ `v`). Apart from this, we also define standard addition and subtraction operations.

```
my_type & add_cipr(my_type const & v, BYTE carry = 0);
my_type & operator+= (my_type const& v);
my_type & operator-= (my_type const& v);
```

Subtraction and (special) addition operations might leave a carry, which can be queried using the following function.

```
bool carry() const;
```

The following functions implement vector-wide bit counting using a lookup table (see Manku [2010] for a discussion, and Jr. [2003], as well as Beeler et al. [1972] for further reference).

```
size_t count_bits() ;
size_t count_zeros() ;
};
```

A.2.2 Class `CharMapping`

This class implements character match mappings. A `CharMapping` is constructed from a string, and is derived from the standard C++ `std::vector` class. After construction, the `vector` contains $2^{\texttt{bpc}}$ different `BitString` objects which indicates the occurrences of each character from the alphabet in the string `_str` the mapping was constructed from.

```
template<int _bpc, UINT64 _mapval = 1, bool _invert = true>
```

```
class CharMapping : public std::vector< BitString > {
public:
    CharMapping(IntegerVector<_bpc> const & _str);
};
```

A.3 Vector operations and their efficient implementation

In this section, we describe various optimisations which were used implementing the class `IntegerVector`. We start describing the storage format of the vectors. Figure A.1 shows the storage format for our vectors. We use alignment to 256-bit (8 byte) boundaries, which allows using all types of MMX and SSE instructions, and ensures good memory access performance. We use padding to 256-bit aligned end addresses. This allows implementing bit shifts and other operations which may carry data outside the vector more efficiently. Based on this storage format, we

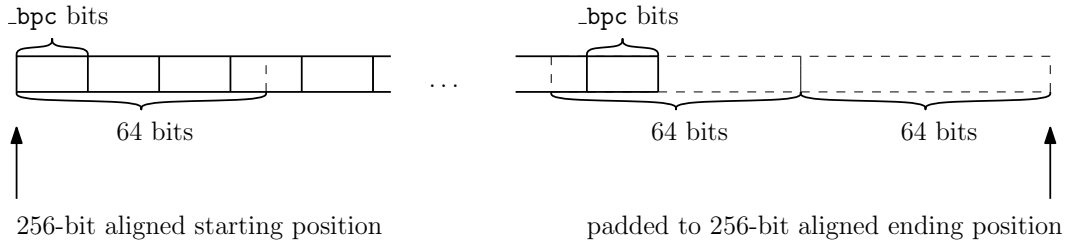


Figure A.1: Class `IntegerVector` data storage format

provide efficient implementations of various word-RAM/MPRAM operations. The main advantages of providing such optimized implementations are:

- Hand-written assembler code can be tuned to use registers only for counting and other overhead arithmetic. This maximizes vector operation throughput, since the only data transferred from main memory or higher-level caches is the data that is actually manipulated.

- Lower overhead for function calls. Modern compilers may generate code for generating stack frames, storing local variables, etc. We can avoid this by optimizing register and stack usage manually.

The effort for doing this is worthwhile, since our vector routines are used at the core of very computation-intensive code, and are called very frequently. Lower per-call overhead and maximum data throughput are essential for achieving the best performance possible. We will show a few of these optimized functions here. Note that each function has been implemented in different versions for different operating systems (Windows and Linux/MacOS use different calling conventions, see Fog [2010]), and on 32 or 64-bit programming modes. These function implementations are stored in separate source files, the correct file for the current platform is selected at compile time.

A.3.1 Addition with carry

A first operation which is commonly used in many of our algorithms is addition of two integers, retaining the carry if the result exceeds the integer's range. When implementing this operation using portable C++ code, it is very hard to ensure that the result will run efficiently. Consider the following piece of code which performs addition with carry on two 32-bit integers.

```
INT32 add_with_carry_generic(INT32 a, INT32 b, BYTE * c) {
    INT32 carry = 0;
    // the first time we might create an overflow is when
    // adding the carry from *c
    carry = a == -1;
    a+= (INT32)*c;

    // if both msbs are set, we also get an overflow
    carry |= ((a & b & 0x80000000) != 0) ? 1 : 0;
    *c = carry;
    return a + b;
}
```

The problem with this code is that it will almost certainly be translated into machine code which includes branching instructions. This is not efficient, especially considering that the Intel architecture provides a simple "add with carry" instruction. On a 32-bit platform, the following assembler code performs the same operation more efficiently.

```
public_c_symbol add_with_carry32
;PROC a:DWORD, b:DWORD, _c:ptr
  push esi
  mov eax, [esp+8]       ; [a]
  add eax, [esp+12]      ; [b]
  setc dl                ; remember first carry
  mov esi, [esp+16]      ; [&c]
  movzx ecx, byte [esi]
  add eax, ecx
  setc cl
  or cl, dl              ; carry if either of the additions
  and cl, 1              ; generated an overflow
  mov byte [esi], cl
  pop esi
  RET
```

For adding 1000000 integers, we get the following execution times. On 32-bit Windows (2.33 GHz Core2 Duo processor, Visual C++ 8.0 compiler), the code runs in 0.013 seconds for the generic version and 0.0084 seconds for the optimized assembler code, making for a speedup of 1.55. For 64-bit integers, the difference is more substantial on this system with 0.024 seconds for the generic implementation, and 0.0125 seconds for the assembler version, giving a speedup of 1.92. On 64-bit Linux using gcc 4.3.1, the difference is smaller, having a runtime 0.0059 seconds for the generic version, 0.0057 seconds for the assembler code, therefore having more or less equal performance for 32-bit integers. For 64-bit integers, we get 0.0087 seconds for the generic code and 0.0078 seconds for the assembler version, making it 1.12 times faster. In conclusion, our optimized version is never slower than the generic code,

and allows for system-dependent performance gains between 10% and 90%, which can save a significant amount of computation time for a frequently-used function.

Function	Generic	Assembler	Speedup
Windows 32-bit, Visual C++ 8.0			
`add_with_carry32`	0.013 s	0.0084 s	1.55
`add_with_carry64`	0.024 s	0.0125 s	1.92
Linux 64-bit, gcc 4.3.1			
`add_with_carry32`	0.0059 s	0.0057 s	1.03
`add_with_carry64`	0.0087 s	0.0078 s	1.12

A.3.2 Vector addition

Another important operation is considering two vectors as $n \cdot$ `_bpc` numbers, and performing addition-type operations. This is one of the main ingredients in the bit-parallel LCS algorithm by Crochemore et al. [2001].

We implement the following generic operations. The function `vecadd` performs a standard addition, of two vectors interpreted as long numbers. The function `vecadd_cipr` performs the addition operation that is the core of the bit-parallel LCS algorithm by Crochemore et al. [2001], where we compute for two vectors L and M the sum $L' = (L\&M) + (L\& \sim M)$.

```
void vecadd_generic(UINT64 * data1, UINT64 * data2,
    size_t len, BYTE initial_carry /* = 0 */) {
    BYTE carry = initial_carry;
    for (size_t z = 0; z < len; ++z) {
        *data1 = add_with_carry64(*data1, *data2, carry);
        ++data1; ++data2;
    }
}
void vecadd_generic_cipr(UINT64 * M, UINT64 * L,
    size_t len, BYTE initial_carry /* = 0 */) {
    BYTE carry = initial_carry;
    for (size_t z = 0; z < len; ++z) {
```

```
        *L = add_with_carry64(*L & *M, *L & (~(*M)), carry);
        ++M; ++L;
    }
}
```

The following 64-bit assembler versions of these functions can carry out the complete loop using only registers for counting.

```
public_c_symbol vecadd
;PROC data1:PTR, data2:PTR, len:DWORD, carry : BYTE
  ; rdi contains M
  ; rsi contains L
  ; rdx contains len
  ; rcx contains carry
  lahf
  and cl, 1
  or ah, cl ; carry
  xor rcx, rcx
.loop:
  mov r8, [rsi + 8*rcx]
  sahf
  adc [rdi + 8*rcx], r8
  lahf
  inc rcx
  dec rdx
  jnz .loop
  sahf
  setc al
  RET

public_c_symbol vecadd_cipr
; PROC M:PTR, L:PTR, len:DWORD, carry : BYTE
  ; rdi contains M
  ; rsi contains L
  ; rdx contains len
```

```
  ; rcx contains carry
  mov al, cl
  xor rcx, rcx
.loop:
  mov r8, [rsi + 8*rcx] ; r8 = L[i]
  mov r9, [rdi + 8*rcx] ; r9 = M[i]
  mov r10, r9
  not r10
  and r10, r8           ; r10 = V = L & ~M
  and r9,  r8           ; r9  = U = L &  M
  bt ax, 0
  adc r8, r9            ; L' = (L+U)|V
  setc al
  or r8, r10
  mov [rsi + 8*rcx], r8
  inc rcx
  dec rdx
  jnz .loop
  RET
```

When adding vectors of 100000 64-bit words, we get the following times and speedups on the systems described above.

Function	**Generic**	**Assembler**	**Speedup**
Windows 32-bit, Visual C++ 8.0			
`vecadd`	0.00114 s	0.000453 s	2.52
`vecadd_cipr`	0.00134 s	0.000624 s	2.15
Linux 64-bit, gcc 4.3.1			
`vecadd`	0.0004 s	0.000283 s	1.41
`vecadd_cipr`	0.00038 s	0.000295 s	1.3

A.3.3 Vector shifting

Finally, an important vector operation are bit shifts. Optimizing this operation is important for implementing the algorithm shown in Section 6.4, as well as e.g. the algorithm by Boasson et al. [2001] for window-local subsequence matching. The reference implementations of `vecshl` and `vecshr` look as follows.

```
 void vecshl_generic(UINT64 * data, size_t shift, size_t len) {
    UINT64 tmp1 = 0, tmp2;
    for (size_t z = 0; z < len; ++z) {
        tmp2 = *data;
        *data = (*data << shift) | (tmp1 >> (64 - shift));
        tmp1 = tmp2;
        ++data;
    }
}
void vecshr_generic(UINT64 * data, size_t shift, size_t len) {
    for (size_t z = 0; z < len; ++z) {
        *data = (*data >> shift) | (data[1] << (64 - shift));
        ++data;
    }
}

public_c_symbol vecshl
; PROC data1:PTR, bits:DWORD, len:DWORD
  ; rdi contains data1
  ; rsi contains bits
  ; rdx contains len
  xor r8, r8
  mov cx, si
.loop:
  mov rax, [rdi]
  mov r9, rax
  shld rax, r8, cl
  mov r8, r9
```

```
  mov [rdi], rax
  add rdi, 8
  dec rdx
  jnz .loop
  RET

public_c_symbol vecshr
; PROC data1:PTR, bits:DWORD, len:DWORD
  ; rdi contains data1
  ; rsi contains bits
  ; rdx contains len
  dec rdx    ; vector is padded by two QWORDs.
    ; We skip one here so we can read
    ; ahead a little
  mov rax, [rdi]
  mov cx, si     ; move bits to cl
.loop:
  mov r8, [rdi+8]
  shrd rax, r8, cl
  stosq
  mov rax, r8
  dec rdx
  jnz .loop
  RET
```

When bit-shifting 100000 64-bit words, we get the following runtimes. Again, the assembler code gives substantial speedup on the 32-bit platform.

Function	Generic	Assembler	Speedup
Windows 32-bit, Visual C++ 8.0			
`vecshl`	0.0312 s	0.009 s	3.47
`vecshr`	0.0259s	0.011 s	2.35
Linux 64-bit, gcc 4.3.1			
`vecshl`	0.0044 s	0.0038 s	1.16
`vecshr`	0.0044 s	0.0037 s	1.19

A.3.4 Specialized implementation for 8-bit and 16-bit words

Apart from generic functions which work on vectors with words of any bit length, we provide specialized implementations of certain functions for 8-bit and 16-bit words using MMX and SSE2. The implementation uses partial template specialisation (see Vandevoorde and Josuttis [2003]) for the classes `IntegerVector<8>` and `IntegerVector<16>` to make these operations available to the programmer.

We will show the implementation of the compare-and-exchange function as an example. Other functions have been implemented for generating match masks and saturated addition. In particular, all functions necessary for implementing the algorithms from Chapter 6 are available as optimized SSE and MMX versions.

The generic implementation of compare-exchange looks as follows. We compare each pair of vector elements in two vectors, and make sure that the first vector contains the smaller element.

```
void cmpxchg_generic(BYTE * data1, BYTE * data2, size_t len) {
    for (size_t z = 0; z < len; ++z) {
        BYTE t1 = *data1;
        BYTE t2 = *data2;
        if(t1 < t2) {
            *data1 = t2;
            *data2 = t1;
        }
    }
}
```

For a vector of 8-bit words, we can perform the same operation on multiple words in parallel using the following SSE code. Since SSE does not support element-wise comparison and exchange operations directly, we use the difference between saturated subtraction (i.e. subtraction up to a maximum result of zero), and standard subtraction.

```
; elementwise compare and sort
; data1 gets the smaller elements
public_c_symbol cmpxchg_8_mmx
;PROC data1:PTR, data2:PTR, len:DWORD
  ; rdi contains data1
  ; rsi contains data2
  ; rdx contains len

  ; xmm5 == 0ffffff....ff
  pcmpeqd xmm5, xmm5
  ; xmm6 = 00010001....
  movdqa  xmm6, [mmx_one_b]

  xor rcx, rcx
  mov rax, rdx
  shr rdx, 1
  jz .fin
.loop:
  movdqa xmm0, [rsi+rcx]
  movdqa xmm1, [rdi+rcx]

  movdqa xmm2, xmm1
  ; xmm2 = 0   if xmm0 > xmm1 ; xmm2 = xmm1-xmm0 otherwise
  psubusb xmm2, xmm0
  ; xmm2 = xmm0 if xmm0 > xmm1 ; xmm2 = xmm1     otherwise
  paddb xmm2, xmm0

  movdqa xmm3, xmm2
```

```
  pandn xmm3, xmm5   ; xmm3 = !xmm3
  ; xmm3 = -xmm2 (two's complement), i.e.
  ; xmm2 = -xmm0 if xmm0 > xmm1 ; xmm2 = -xmm1 otherwise
  paddb xmm3, xmm6
  ; xmm2 = 0     if xmm0 > xmm1 ; xmm2 = xmm0-xmm1   otherwise
  paddb xmm3, xmm0
  ; xmm2 = xmm1   if xmm0 > xmm1 ; xmm2 = xmm0 otherwise
  paddb xmm3, xmm1

  movdqa [rdi+rcx], xmm3 ; the smaller values
  movdqa [rsi+rcx], xmm2 ; the larger values

  add rcx, 16
  dec rdx
  jnz .loop
.fin:
  RET
```

When running `cmpxchg` on vectors of 800000 bytes, we get the following running times and speedups. The speedup for this operation is greater than for the functions studied so far, since the vector operations can process multiple elements in parallel. Even modern compilers still struggle to recognize and implement this type of parallelism in high-level language code to the same extent as manual optimization.

Function	**Generic**	**Assembler**	**Speedup**
Windows 32-bit, Visual C++ 8.0			
`cmpxchg`	0.00434 s	0.00029 s	15
Linux 64-bit, gcc 4.3.1			
`cmpxchg`	0.0006 s	0.00018 s	3.3

A.3.5 Manipulating slices of vectors

When implementing the seaweed algorithm using vector parallel operations, we encounter the problem that the vectors we have to manipulate may grow or shrink

from step to step with the size of the dynamic programming wavefront (see Section 6.4. We approach this problem by allocating a vector that is able to store the maximum length of a wavefront, and allowing us to restrict the range in which operations have effect by working on a *slice* of the vector (a similar concept has been used in the Boost array library to manipulate parts of multi-dimensional arrays, see Garcia et al. [2010]). In order to prevent overhead for creating exact slices which might not align with the 64-bit storage boundaries required for applying MMX/SSE operations, we only require this range restriction in a weak sense: operations on a vector slice are guaranteed to work on all values contained within the slice, but might also affect values in neighbouring areas.

We can declare vectors and slices thereof as follows.

```
// allocate a vector of 1000 bytes
IntegerVector<8> storage_vector(1000);
// this vector only accesses the first 100 elements of
// storage_vector
IntegerVector<8> slice(storage_vector, 100);

// we can increase the size of the slice to 200
slice.resize(200);
// we can also move the slice through the vector in
// steps of eight bytes
slice.shiftstart(25); // move up 200 = 25*8 characters
```

After these steps, all operations on `slice` will manipulate the 200 characters in `storage_vector` starting at position 200. We do not require the operations to restrict their actions to only these 200 characters. This is not required by our application, where we only work with values within a wavefront which grows and shrinks linearly. Therefore, we do not need to selectively run operations on different parts of a vector while preserving its contents in between, which would complicate the implementation of slices unnecessarily.

A.4 Vector library list of files

Table A.1: Vector library list of files and contents

Filename	Contents
`CharMapping.h`	Declaration and implementation of the `CharMapping class`
`functors.h`	Declaration of functors used in `IntegerVectorFixed.h` to implement element-wise operations on arrays.
`IntegerVector.h`	Declaration of the generic `IntegerVector` class.
`IntegerVectorFixed.h`	Partial specialisations of `IntegerVector<8>`, `IntegerVector<16>`, and `IntegerVector<32>`
`xasmlib.h`	Function prototypes for the low level C and assembler functions.
`xasmlib.c`	Library initialization function implementation.
`machineword_AMD64.asm`	Optimized assembler code for 64-bit Windows platforms.
`machineword_AMD64_sse2.asm`	Optimized SSE2 assembler code for vector functions on 64-bit Windows platforms.
`machineword_x86.asm`	Optimized assembler code for 32-bit Windows/Linux platforms.
`machineword_x86_64.asm`	Optimized assembler code for 64-bit Linux/MacOS X platforms.
`machineword_x86_64_mmx.asm`	Optimized MMX (no SSE) assembler code for vector functions on 64-bit Linux platforms (for reference/comparison use only).

Continued on the next page...

...continued from last page.

Filename	**Contents**
`machineword_x86_64_nommx.asm`	Vector operation assembler code without MMX/SSE for 64-bit Linux platforms (for reference/comparison use only).
`machineword_x86_64_sse2.asm`	Optimized SSE2 assembler code for vector functions on 64-bit Linux platforms.
`machineword_x86_mmx.asm`	Optimized MMX assembler code for vector functions on 32-bit Windows/Linux platforms.
`machineword_x86_nommx.asm`	Assembler code without MMX for vector functions on 32-bit Windows/Linux platforms (for reference/comparison use only).
`machineword_x86_sse2.asm`	Optimized SSE2 assembler code for vector functions on 32-bit Windows/Linux platforms.

Appendix B

A BSP library for C++

B.1 Introduction

Various libraries for BSP-style programming have been implemented, most of them follow the BSPlib standard (Hill et al. [1998]; Bonorden et al. [2003]). However, when MPI (Snir et al. [1995]) and OpenMPI (OpenMP Architecture Review Board [2010] established themselves as the standards for coarse-grained programming, further development of these libraries came to a halt. The most recent BSP programming library is BSPonMPI (Suijlen and Bisseling [2010]), which implements a BSPlib-style message buffer for BSP programming using MPI. In this section, we show a small library for BSP programming which is based on BSPonMPI and Intel's Threading Building Blocks (TBB, Intel Corporation [2009]) which simplifies BSP-style programming on cluster systems with multi-core/SMP nodes. Furthermore, our library provides a layer of abstraction between the number of processors physically available, and the number of processors used by a BSP algorithm. Such "virtual processors" have been implemented in the PUB library (PUB library). However, our approach allows us to make better use of SMP and multicore systems, and uses TBB's functionality for task-based parallel programming for achieving good performance. Furthermore, we can use our library to compile BSP program versions both for MPI-based cluster systems and multi-core desktops from the same source code.

B.2 Extending BSPonMPI

The first step towards our new library consisted in modernizing the BSPonMPI library, and introducing an extension in the BSPlib programming model, which is better suited to the model for parallel algorithms introduced in Chapter 2. The main modifications to BSPonMPI to adapt it to our requirements were:

Improving portability: The official BSPonMPI release only supports Linux platforms and uses an old version of the GNU Autoconf toolset (Free Software Foundation (FSF) [2010a]). For our work, the library was adapted to use the modern SCons tool for cross-platform compilation (The SCons Foundation [2010]), and the program code was updated to handle different calling conventions and compilers, and can therefore be used on Windows, Linux, and MacOS X platforms.

Introducing global DRMA: BSPonMPI supplies the standard direct remote memory access operations (DRMA) specified by the BSPlib standard. These operations allow the programmer to allocate and publish a memory segment of a specified size on each processor, and then access this memory from other processors. The programming interface from BSPlib allows the programmer to register/unregister areas of memory as globally accessible, and post requests for reading or writing from/to such a memory segment on a given processor. These requests are fulfilled at the end of a superstep, which is indicated by a call to the `bsp_sync()` function. Our programming interface extension removes the dependency on the number of physical processors from this programming interface. We allow allocation of a segment of global memory, which is accessed in the same way, but which might be distributed arbitrarily between processors.

Compiling without MPI: Another extension to the original code was implementing a dummy library, so BSPonMPI can be compiled without an MPI library to run sequentially on a single node. Apart from testing purposes, this option

allows compiling program versions which run on cluster systems and desktop executables from the same source code.

The programming interface for global DRMA is as follows. Each segment of global memory can be accessed via its handle.

```
typedef int bsp_global_handle_t;
```

We can create and destroy global memory by calling `bsp_global_[alloc|free]`, followed by `bsp_sync` to make the allocation consistent on all processors.

```
bsp_global_handle_t bsp_global_alloc(size_t array_size);
void bsp_global_free(bsp_global_handle_t ptr);
```

Access to global memory is implemented using the following functions, which are directly adapted from the standard BSPlib DRMA functions. An item of data is now read or written from/to a location within the global array, which may reside on the current physical processing node, or somewhere else.

```
void bsp_global_get(bsp_global_handle_t src, size_t offset,
    void * dest, size_t size);
void bsp_global_put(const void * src,
    bsp_global_handle_t dest, size_t offset, size_t size);
```

We also provide unbuffered data access functions, which might start modifying the data already before the end of the current superstep.

```
void bsp_global_hpget(bsp_global_handle_t src, size_t offset,
    void * dest, size_t size);
void bsp_global_hpput(const void * src,
    bsp_global_handle_t dest, size_t offset, size_t size);
```

The current implementation of these functions uses a simple block-cyclic data distribution. For each array, we store a record which contains its total size (`array_size`), and the size of the array stored on each physical processor (`local_size`), which is calculated as

$$\texttt{local\_size} = \lceil \texttt{array\_size}/\texttt{bsp\_nprocs}() \rceil.$$

When accessing an item of data at location `offset`, we calculate the its local position as follows:

```
size_t procs = bsp_nprocs();
size_t gsize = bsp.global_arrays[src].array_size;
size_t target_proc = offset * procs / gsize;
size_t target_idx = offset -
    offset_proc*bsp.global_arrays[src].local_size;
```

In future versions of the library, different distributions could be implemented as well to suit other applications. For our string comparison applications, the cyclic data distribution described above is sufficient.

B.3 C++ library design

Building on the updated version of BSPonMPI, we implemented a small set of template classes which allow simple BSP-style programming on a hybrid parallel system, i.e. a network of workstations with SMP or multi-core processors.

Programming on such a hybrid parallel system comes with a few additional challenges compared to programming in MPI or BSPlib only. The main differences to programming MPI-style are:

- We can dynamically change the number of virtual processors by executing multiple threads on one or more SMP nodes.
- Remote memory access or message transmission is faster when sending data to processes on the same SMP node.

Our BSP class library provides an extension to the BSPlib standard to handle a variable number of virtual processes which can change between supersteps. This simplifies the implementation of recursive BSP algorithms, and also allows implementing overpartitioning (splitting a problem into more subproblems than the number of physical processors, e.g. to fit data into caches) more easily. We distinguish between *(SMP) nodes* and *physical processors* (which might be cores or processors

on a SMP node). Each SMP node can host a number of *virtual processors*. These virtual processors are mapped to the physical processors using the TBB task-based parallel programming framework. In each superstep, the execution of the BSP computation on a virtual processor is mapped to executing a task, which is then mapped to a physical processor by the TBB task scheduler. This approach is more efficient than creating a separate thread for each virtual processor, and has also been applied by the Cilk++ programming language (Intel Corp. [2010]; Leiserson [2009]).

The `DefaultSuperstep` class template provides the basic functionality for implementing BSP computations. This class template provides and implements the following functions to a computation.

```
template <class _context = Loki::NullType>
class DefaultSuperstep {
public:
```

Each local instance of a superstep class on a virtual processor has a *context* object, which stores the information that is local and persistent between supersteps. The context object encapsulates the local memory for a process (see Figure 2.2 on page 11). In normal BSPlib style programming, this would either be stored on the stack or the heap of a compute node. We explicitly encapsulate it here for the following reasons:

- We do not know how many threads we will have on a compute node, and we do not want to restrict data storage to the stack.
- Encapsulating the local state of a virtual process in a class allows implementing process migration and copying more easily.
- We can use the TBB task-based programming framework (Intel Corporation [2009]) for implementing virtual processes efficiently.

The context type can be specified by the programmer as a template parameter. A context object is created for each virtual processor, and for each SMP node, as it can be sensible to share a certain amount of data between all virtual processors running on a single node.

```
typedef _context context_t;
```

The implementation of a BSP computation will overload the `run` method, which will contain the code for executing a single superstep.

```
virtual void run() = 0;
```

This code can then use the following functions to obtain the number of virtual processes, its process id (`pid`), access the context object, and access the global shared memory in a thread-safe fashion.

```
protected:
// number of virtual processors
virtual int bsp_nprocs() const;
// id of current virtual processor
virtual int bsp_pid() const;
// SMP-node local pid
virtual int bsp_local_pid() const;
// get the process context
virtual _context & get_context();
// get the node context
virtual _context & get_node_context();
// Thread safe global DRMA
virtual void bsp_global_get(bsp_global_handle_t src,
    size_t offset,
    void * dest, size_t size);
virtual void bsp_global_put(const void * src,
    bsp_global_handle_t dest, size_t offset, size_t size);
virtual void bsp_global_hpget(bsp_global_handle_t src,
    size_t offset,   void * dest, size_t size);
virtual void bsp_global_hpput(const void * src,
    bsp_global_handle_t dest, size_t offset, size_t size);
};
```

Based on a superstep implementation using the `DefaultSuperstep` class template, each node can instantiate an object of the `Superstep` type, which handles virtual process creation and synchronisation using the TBB task parallelism library.

```
template <
    class _implementation,
    template <class> class _procmapper = ProcMapper
>
class Superstep : public _implementation {
public:
```

Each `Superstep` object uses a `ProcMapper` object to handle the mapping of virtual processors to SMP nodes.

```
typedef _procmapper<typename
    _implementation::context_t> procmapper_t;
typedef Superstep <_implementation, _procmapper
     > my_type;
```

Besides the functions for thread-safe global DRMA, the `Superstep` class implements the following functions.

The function `start()` creates all tasks for the virtual processes according to the `ProcMapper` object, and immediately returns after starting their execution. The superstep tasks will be executed by worker threads in the TBB thread pool.

```
void start();
```

At the end of the superstep, a call to the function `join()` will wait for all tasks to finish, and therefore for all virtual processors to finish the execution of the current superstep.

```
void join();
};
```

The default implementation of the `ProcMapper` class template tries to assign an equal number of virtual processors to each SMP node. It also handles the storage of the context objects for each virtual processor on a SMP node.

```
template <class _context=Loki::NullType>
class ProcMapper {
public:
// create a process mapper which handles the mapping for a
```

```
// given number of processors
ProcMapper(int _processors, int _groups = 1);
// number of virtual processors
int nprocs () const;
// number of processor groups
int ngroups() const;
// how many virtual processors are maximally hosted on a node
int procs_per_node() const;
// how many virtual processors are hosted on this node
int procs_this_node() const;
// return the global pid of a given local process
int local_to_global_pid(int local_pid) const;
// return the group for a given local process
int local_to_global_group(int local_pid) const;
// convert global (1 ... nprocs) pid to local pid
// (i.e. number of process on the current node)
// returns -1 if the process is not resident on the current
// node
int global_to_local_pid(int global_pid, int group = 0) const;
// get context for specific virtual processor
_context & get_context(int local_pid);
// get context for the current SMP node
_context & get_node_context();
};
```

Based on these classes, we construct the following small example.

```
#include "bspcpp/bsp_cpp.h"
#include <tbb/mutex.h>
#include <loki/Typelist.h>
#include <iostream>
```

We syncronize console output using a mutex so the text doesn't get garbled due to multithreaded execution.

```
tbb::mutex g_output_mutex;
```

The following macro simplifies creating superstep classes. We declare a superstep which uses an `int` for local storage, based on the standard `ProcMapper` template.

```
BSP_SUPERSTEP_DEF_BEGIN(Superstep1, int, bsp::ProcMapper)
    tbb::mutex::scoped_lock l;
    l.acquire(g_output_mutex);
    std::cout << "Hello from process "
              << bsp_pid()
              << " out of " << bsp_nprocs()
        << ". I live on node " << ::bsp_pid()
        << ", where i am local pid " << bsp_local_pid() << "."
        << std::endl
        << "Also, I have value " << context
        << ", executing Superstep1" << std::endl;
    context++;
    l.release();
BSP_SUPERSTEP_DEF_END()
```

In our second superstep template, we decrement our context counter, and print the same information as above.

```
BSP_SUPERSTEP_DEF_BEGIN(Superstep2, int, bsp::ProcMapper)
    tbb::mutex::scoped_lock l;
    l.acquire(g_output_mutex);
    std::cout << "Hello from process "
              << bsp_pid()
              << " out of " << bsp_nprocs()
        << ". I live on node " << ::bsp_pid()
        << ", where i am local pid " << bsp_local_pid() << "."
        << std::endl
        << "Also, I have value " << context
        << ", executing Superstep2" <<
    std::endl;
    context--;
    l.release();
BSP_SUPERSTEP_DEF_END()
```

We define our algorithm by creating a typelist that specifies the order in which to create and execute the superstep objects. Through the `FlatComputation` class template, the compiler will then unwind this list and create code which is equivalent to manually calling `run()` on each of these objects, and call `bsp_sync()` between supersteps to initialize data transmission.

```
typedef bsp::FlatComputation<LOKI_TYPELIST_4(
    Superstep1,
    Superstep1,
    Superstep2,
    Superstep2), int>
MyFlatParallelComputation;
```

The remaining code creates computation objects and handles the BSPonMPI initialisation.

```
void runner(void) {
    bsp_begin(-1);

    MyFlatParallelComputation::procmapper_t mapper(2);
    MyFlatParallelComputation::run(mapper);

    bsp_end();
}
int main(int argc, char ** argv) {
    bsp_init(runner, argc, argv);
    runner();
    return 0;
}
```

When running on a single SMP node, this code will produce the following output, which details the assignment of virtual processors.

```
Hello from process 1 out of 2. I live on node 0, where i am local pid 1.
Also, I have value 0, executing Superstep1
Hello from process 0 out of 2. I live on node 0, where i am local pid 0.
...
```

```
Hello from process 0 out of 2. I live on node 0, where i am local pid 0.
Also, I have value 1, executing Superstep2
Hello from process 1 out of 2. I live on node 0, where i am local pid 1.
Also, I have value 1, executing Superstep2
```

When running the same code using two MPI nodes, we will get the following result, having each of the two virtual processes run on a separate MPI node.

```
Hello from process 1 out of 2. I live on node 1, where i am local pid 0.
Also, I have value 0, executing Superstep1
Hello from process 0 out of 2. I live on node 0, where i am local pid 0.
...
Hello from process 0 out of 2. I live on node 0, where i am local pid 0.
Also, I have value 1, executing Superstep2
Hello from process 1 out of 2. I live on node 1, where i am local pid 0.
Also, I have value 1, executing Superstep2
```

B.4 BSP library list of files

Table B.1: BSP C++ library: list of files

Filename	**Contents**
`bspcpp_config.h`	Automatically generated configuration header file.
`examples/computation`	
`bspcomputation.cpp`	BSP computation example
`examples/parameterfiletest`	
`parameterfiletest.cpp`	Test for the `ParameterFile` class.
`examples/timer`	
`timer.cpp`	Source for tool to measure program execution times in microseconds.
`include/bspcpp`	

Continued on the next page...

...continued from last page.

Filename	Contents
`Avector.h`	`Avector` class template implementation: This class provides a resizable array, which can be accessed via a raw data pointer, and is allocated with 8-byte alignment.
`bsp_cpp.h`	This is the main include file for the BSP C++ library-
`CommandLine.h`	Provides an interface to the `CommandLine` class This class has been obsoleted by using `boost::program_options`, but is still used in parts of the code.
`Computation.h`	`FlatComputation` class implementation and superstep creation helper macros.
`ParameterFile.h`	Provides an interface to the `ParameterFile` class. This class provides a simple serializable hash-type storage container.
`Permutation.h`	`Permutation` class template: Allows permuting elements of C++ vectors and arrays in place.
`ProcMapper.h`	The `ProcMapper` class allocates virtual processors to SMP nodes and stores virtual processor context variables.
`Superstep.h`	Contains the implementation of the `Superstep` class template.
`include/bspcpp/tools`	

Continued on the next page...

...continued from last page.

Filename	**Contents**
`aligned_allocator.h`	Contains a C++ allocator class which allocates memory segments with a specified alignment.
`singletons.h`	This header file provides an interface for the various singleton variables used within the code.
`spawn.h`	Provides the POSIX `spawnvp` function in a uniform fashion on Windows, Linux and MacOS X.
`utilities.h`	Various utility functions like floor/ceiling division, factorials, stream output operators, etc.
`src`	
`CommandLine.cpp`	Implementation of the (obsoleted) `CommandLine` class.
`ParameterFile.cpp`	Implementation of the `ParameterFile` class.
`singletons.cpp`	Declaration of the various singleton variables (e.g. TBB's `task_scheduler_init` object).
`timing.cpp`	Implementation of platform-independent microsecond timing functions.

Appendix C

Alignment Plot Code Documentation

C.1 Introduction

The main part of the seaweed code consists of various implementations of alignment plot and LCS computation, some unit tests to ensure correct results and tools allow the user to obtain running time measurements. The individual tools will be described in the following sections, followed by instructions for compiling the code and a brief outline of the source tree.

C.2 The alignment plot tools

The three main executables for computing alignment plots are *Alignment*, *WindowAlignment*, and *PostProcessWindows*.

Alignment

This is the original alignment code by Ott et al. [2009]. It takes two sequence files as its input, the first sequence file must contain the parameters for the comparison, which are the first and second step sizes (only window pairs with starting positions aligning at these step sizes are compared), the window length, and the minimum

alignment score threshold. The following example shows such a first input sequence file. In this file, the first step size is 5, the second step size is 1, and the window length is 100.

```
5 1 100 55
TAAGCTAGGGGCCAGGAC...
```

The command line usage on Linux is as follows.

```
$ ./Alignment [firstsequencefile] [secondsequencefile]
```

Example versions of sequence files can be found in the subfolders of the `benchmark` directory. The output of the file consists of two profile files which give the score for the highest-scoring window in each row and column, and a file named `results.txt` which contains locations and scores for all windows scoring above the threshold.

WindowAlignment

This is the main executable for computing alignment plots using the seaweed algorithm. The input and output files have the same format as the ones generated by the window alignment code by Ott et al. [2009] described above, however, the locations of the output files and profiles can be specified separately.

The usage is as follows.

```
$ ./WindowAlignment [firstsequencefile] [secondsequencefile] \
        [resultfilename] [firstprofilename] [secondprofilename]
        {options}
```

Table C.1: WindowAlignment command line options

Filename	Contents
-c	enable checkpointing (resuming if checkpoint file exists for the job)

Continued on the next page...

...continued from last page.

Filename	**Contents**
-m	[lcs|blcs|seaweeds|scores|scoresoverlap]: Choose method to use. **LCS:** This computes the scores by computing the LCS separately for each window pair. **BLCS:** This uses bit-parallel LCS computation to obtain the window scores, again separately for each window pair. **Seaweeds:** This uses the seaweed algorithm from Section 6.4, and counts scores in a sliding window using a queue. **Scores:** This uses the seaweed algorithm from Section 6.4, but computes implicit highest-score matrices to allow faster window score queries. **Scoresoverlap:** This method uses the seaweed algorithm and uses strip overlap as described in Section 6.6 to speed up the computation.
-os	[number] specify overlap size for scoresoverlap method
-legalchars_[a|b]	Specify input alphabet translation. The default is to match characters ABGCTz normally, and mismatch N and x by setting `legalchars_a = ABGCTNxz`, and `legalchars_b = ABGCTxNz`

Postprocesswindows

This is a tool to filter the output of Alignment/WindowAlignment and reduce the window count. It also sorts the output files such that outputs from Alignment and WindowAlignment can be compared using diff.

Command line usage:

```
$ PostprocessWindows [resultfilename] -m [number]
```

The number specified after the `-m` option gives the maximum number of windows which are exported. If more windows are found, only the ones with the highest scores are reported.

C.3 Compiling the code

Prerequisites

The seaweed code can be compiled using a recent version of gcc (> 4.0.0) on Linux and MacOS) or Microsoft Visual C++ (MSVC, 2005 or 2008) on Windows. Intel C++ compiler can be used as well. For compiling the code, the tool SCons is required, which is an automated build environment based on Python (The SCons Foundation [2010]). For compiling the assembler code, the yasm assembler is needed (Johnson and other Yasm developers [2010]). Optionally, STLFilt (see `http://www.bdsoft.com/tools/stlfilt.html`) can be used to simplify C++ error messages. The following libraries are required:

Boost: We use this library for command line parsing, regular expression evaluation and various C++ helper classes (like the `auto_ptr` class for implementing smart pointers. It can be obtained at `http://www.boost.org/` or by installing the corresponding developent packages in Linux. A technical introduction to programming using Boost is given by Abrahams and Gurtovoy [2004].

Loki: This library provides class templates for implementing type lists and other metaprogramming techniques as described by Alexandrescu [2001]. It is available at `http://loki-lib.sourceforge.net/`.

Intel Threading Buildingblocks: This is a high-level library for multithreaded programming. It provides C++ classes for thread synchronization, thread creation, and task-based parallel programming. Sources and precompiled binaries are available at `http://www.threadingbuildingblocks.org/`.

Generic compiling instructions

All executables are suffixed with platform and build mode and placed in the bin directory after compiling. Calling one of the tools on Linux would therefore work as follows:

```
> bin/Alignment_posix_gcc_release ...
```

Scons will try to configure the required libraries and paths automatically, but will need additional input if the libraries shown above are installed in a non-standard path. The following sections give some guidance on how to compile on Windows, Linux or MacOS.

When running Scons for the first time in the source directory, it will print the name of the options file it wants to use on your system:

```
> scons -Q configure=1
To use specific options for this system, use options file \
 "opts_virtualsettembrini-ubuntu_Linux_x86_64.py"
Using options from opts.py
```

The options file contains information on additional paths used by the build system. In most cases, making changes to this file, or supplying an adapted file for each system you need to compile on should be sufficient. The standard filename is 'opts.py', however, the system generates a platform-specific name as well (this is useful for maintaining different builds in the same directory). Once all paths and compile flags have been configured correctly (see below for platform-specific instructions), a release version can be compiled by running:

```
scons -Q mode=release
```

The executables are assigned a suffix according to platform and build mode (valid values are release, debug or profile), and are placed in the `bin/` subdirectory.

Compiling on Windows

The easiest way to compile on Windows is either using one of the supplied SLN files (for VC 2005 or VC 2008), or running Scons in a Visual Studio command line. It might still be necessary to modify the Scons options file. Here is an example.

```
# comment this out if you do not use stlfilt for MSVC
cl_stlfilt = '"..\\stlfilt\\mfilt.bat"'

# uncomment the following line to use MinGw on Windows
# toolset = 'gcc'

# You can use STLFilt with MinGW, too
## gcc_stlfilt = 'gfilt'

# You can specify the boost directory if it is not installed in
# C:\Boost. If boost is installed in its standard location under
# C:\Boost, SCons will find it automatically
win32_boostdir = 'C:\\Users\\peter\\Documents\\Code\\boost_1_36_0';

# This is the path to Loki. You need to unzip, and compile Loki so
# the static libraries are available.
# You might need to check the loki make scripts so they use the same
# runtime library as this code (LIBCMT[D]) to prevent linker warnings
win32_lokidir = 'C:\\Users\\peter\\Documents\\Code\\loki-0.1.6';

# You can get a binary installation of TBB of their website.
# Do not forget to copy the static libraries for your system into
# win32_tbbdir\\lib (since we can't guess this from within the
```

```
# scons script)
win32_tbbdir = 'C:\\Users\\peter\\Documents\\Code\\tbb';
```

Compiling on Linux or MacOS X

The main entries in the options file are:

```
toolset = 'gcc'
# replace this with 'gfilt' if you have STLfilt installed
gcc_stlfilt = 'g++'
```

It is possible to add the libraries to the search path by specifying additional linker and compiler flags:

```
additional_cflags = ' -I/opt/boost/include'
additional_lflags = ' -L/opt/boost/lib'
```

Option File Summary

Table C.2: SCons compile options

Option	Description	Default
`mode`	Build mode: set to debug or release.	`'debug'`
`configure`	Perform automatic configuration before build.	`0`
`icc`	Force using Intel C++	`0`
`profile`	Include debug information also in release version and enable profiling using gprof.	`1`
`use_yasm`	Use yasm instead of nasm.	`1`
`cl_stlfilt`	Path to STLFilt for MSVC.	
`gcc_stlfilt`	Path to STLFilt for gcc.	
`win32_boostdir`	Path to Boost library on Windows.	`''C:\\Boost\\include''`

Continued on the next page...

...continued from last page.

Option	Description	Default
`win32_lokidir`	Path to Loki library on Windows.	`''C:\\Boost\\include''`
`win32_tbbdir`	Path to TBB library on Windows.	`''C:\\tbb''`
`toolset`	Specify compiler and linker tools: gcc | icc | default	`'default'`
`simd_mode`	Specify which SIMD instruction set to use: mmx, sse2 or nommx.	`'sse2'`

C.4 List of files

Table C.3: "Seaweed code" list of files

Filename	Contents
`src/checkpoint`	
`checkpoint.h`	Declarations of the class templates for checkpointing.
`src/gpulib`	
`ATI_Gpulib.h`	Declarations of the GPU-based alignment plot code.
`seaweeds.cpp`	Implementation of the `GPUSeaweeds` class which provides an interface to the GPU-based alignment plot functions.
`seaweeds_gpu.br`	Implementation of the seaweed GPU kernels.
`seaweeds_gpu.cpp`	Automatically generated from `seaweeds_gpu.br`.

Continued on the next page...

...continued from last page.

Filename	Contents
seaweeds_gpu.h	Automatically generated from seaweeds_gpu.br.
seaweeds_gpu_gpu.h	Automatically generated from seaweeds_gpu.br.
src/lcs	
Llcs.h	Implementation of the standard LCS dynamic programming algorithm (Wagner and Fischer [1974])
LlcsCIPR.h	Implementation of bit-parallel LCS computation (Crochemore et al. [2001]).
RationalScores.h	Conversion functions for converting LCS lengths to gapped alignment scores.
src/rangesearching	
BinTree.h	Implementation of range searching using binary search on a sorted list.
IRange.h	Interface definition for data structures that allow range querying.
Range2D.h	Implementation of two-dimensional range searching datastructures.
RangeBenchmark.h	Benchmarking class template to compare the performance of different range searching datastructure implementations.
RangeBenchmark2D.h	Benchmarking class template to compare the performance of different two-dimensional range searching datastructure implementations.

Continued on the next page...

...continued from last page.

Filename	**Contents**
`RangeList.h`	Implementation of range searching by linear-time search.
`RangeTest.h`	Unit test to test the correctness of a range searching data structure implementation.
`RangeTest2D.h`	Unit test to test the correctness of a two-dimensional range searching data structure implementation.
`RangeTree.h`	Range searching using a range tree (see Bentley [1980]; de Berg et al. [2008]).
`src/seaweeds`	
`ScoreMatrix.h`	Definition of the main class to represent highest-score matrices.
`Seaweeds.h`	Implementation of highest-score matrix computation using the seaweed algorithm from Section 6.4.
`sm_ExplicitStorage.h`	Implementation of highest-score matrices in explicit (full $O(n^2)$) representation.
`sm_ImplicitStorage.h`	Implementation of highest-score matrices in implicit representation using the seaweed permutation (see Section 3.6).
`src/tests`	
`gpulibtesting.cpp`	Test code for GPU-based alignment plot computation.
`lcstesting.cpp`	Unit-test for all basic LCS algorithms.
`mwwtesting.cpp`	Unit-tests for the vector library (see Appendix A).

Continued on the next page...

...continued from last page.

Filename	**Contents**
`rangecomparison.cpp`	Performance comparison for the range searching data structures.
`rangecomparison2d.cpp`	Performance comparison for the two-dimensional range searching data structures.
`rangetesting.cpp`	Unit-test for the range searching data structures.
`seaweedtesting.cpp`	Unit-test for the highest-score matrix classes.
`seaweedtesting_reference.cpp`	Automatically generated from `seaweedtesting.cpp`.
`Testing.h`	Generic test class template.
`windowlocalcomparison.cpp`	Benchmark for different algorithms for subsequence matching.
`windowlocaltesting.cpp`	Unit test for the subsequence matching algorithms.
`src/tuning`	
`rangetuning.cpp`	Tool to determine the cut-off for using lists rather than trees for range searching.
`Timing.h`	Class template for creating performance comparisons for different algorithms and problem sizes.
`Tuning.h`	Class template to optimize a single parameter of a given algorithm implementation to achieve optimal performance.
`src/util`	

Continued on the next page...

...continued from last page.

Filename	Contents
rs_container.h	Defines the container class used to store data in all our range searching data structures.
src/windowlocal	
boasson.h	The subsequence matching algorithm by Boasson et al. [2001], implemented using our vector library.
naive.h	Subsequence matching by LCS Computation.
naive_cipr.h	Subsequence matching by bit-parallel LCS Computation.
report.h	Score reporting helper class for subsequence matching.
scorematrix.h	Subsequence matching using the seaweed algorithm and highest-score matrices.
seaweeds.h	Subsequence matching using the seaweed algorithm without storing the full highest-score matrix.
src/windowwindow	
alignmentplotcomputation.h	Alignment plot computation framework class.
postprocesswindows.h	Postprocessing functions to merge the output of multiple processors into one file.
seaweedoverlap.h	Implementation of alignment plots using the seaweed overlap reduction shown in Chapter 6.

Continued on the next page...

...continued from last page.

Filename	**Contents**
`translate_and_print.h`	Output helper class for alignment plot computation.
`windowwindowlcs.h`	Implementation of alignment plot computation using (bit-parallel) LCS, and using seaweeds.
`src/xasmlib`	
`CharMapping.h`	Implementation of the `CharMapping` class, see Appendix A.
`functors.h`	Helper functor classes to carry out various unary and binary operations on containers.
`IntegerVector.h`	Implementation of the `IntegerVector` class, see Appendix A.
`IntegerVectorFixed.h`	Specialisations of the `IntegerVector` class for 8, 16 and 32-bit word sizes. See Appendix A.
`machineword_AMD64.asm`	See Appendix A.
`machineword_AMD64_sse2.asm`	See Appendix A.
`machineword_x86.asm`	See Appendix A.
`machineword_x86_64.asm`	See Appendix A.
`machineword_x86_64_mmx.asm`	See Appendix A.
`machineword_x86_64_nommx.asm`	See Appendix A.
`machineword_x86_64_sse2.asm`	See Appendix A.
`machineword_x86_mmx.asm`	See Appendix A.
`machineword_x86_nommx.asm`	See Appendix A.
`machineword_x86_sse2.asm`	See Appendix A.

Continued on the next page...

...continued from last page.

Filename	**Contents**
`Queue.h`	A small class implementing a priority queue for 64-bit integers
`xasmlib.c`	See Appendix A.
`xasmlib.h`	See Appendix A.
`src`	
`pk_config.h`	Automatically generated configuration header.
`PostprocessWindows.cpp`	Main file for the PostprocessWindows tool.
`WindowAlignment_BSP.cpp`	Main file for the WindowAlignment tool.

Bibliography

David Abrahams and Aleksey K. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison Wesley, 2004. ISBN-10: 0321227255.

Michael Abrash. A First Look at the Larrabee New Instructions (LRBni). *Dr. Dobb's Journal*, April 2009. `http://www.ddj.com/hpc-high-performance-computing/216402188`.

Advanced Micro Devices Inc. Stream Computing Resources, `http://ati.amd.com/technology/streamcomputing/index.html`, 2010.

Alok Aggarwal, Maria Klawe, Shlomo Moran, Peter Shor, and Robert Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1):195–208, 1987. doi: 10.1007/BF01840359.

Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass, 1974. ISBN-10: 0201000296.

Alfred V. Aho, Daniel S. Hirschberg, and Jeffrey D. Ullman. Bounds on the complexity of the longest common subsequence problem. *Journal of the ACM*, 23: 1–12, 1976. doi: 10.1145/321921.321922.

Bruce Alberts, Alexander Johnson, Julian Lewis, Martin Raff, Keith Roberts, and Peter Walter. *Molecular Biology of the Cell*. Garland Science, 5th edition, 2007. ISBN-10: 0815341067.

David Aldous and Persi Diaconis. Longest increasing subsequences: From patience sorting to the Baik-Deift-Johansson theorem. *Bulletin of the AMS*, 36:413–432, 1999. doi: 10.1090/S0273-0979-99-00796-X.

Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001. ISBN-10: 0201704315.

Lloyd Allison and Trevor I. Dix. A bit-string longest-common-subsequence algorithm. *Information Processing Letters*, 23(6):305–310, 1986. doi: 10.1016/0020-0190(86)90091-8.

Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. New data structures for orthogonal range searching. In *Proceedings of FOCS'00*, pages 198–207, 2000. doi: 10.1109/SFCS.2000.892088.

Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215: 403–410, 1990. doi: 10.1006/jmbi.1990.9999.

Carlos E. R. Alves, Edson N. Cáceres, Frank K. H. A. Dehne, and Siang W. Song. Parallel dynamic programming for solving the string editing problem on a CGM/BSP. In *Proceedings of SPAA'02*, pages 275–281, 2002. doi: 10.1145/564870.564916.

Carlos E. R. Alves, Edson N. Cáceres, Frank K. H. A. Dehne, and Siang W. Song. A parallel wavefront algorithm for efficient biological sequence comparison. In *Proceedings of ICCSA'03*, volume 2668 of *Lecture Notes in Computer Science*, pages 249–258, 2003a. doi: 10.1007/3-540-44843-8_27.

Carlos E. R. Alves, Edson N. Cáceres, and Siang W. Song. A BSP/CGM algorithm for the all-substrings longest common subsequence problem. In *Proceedings of IPDPS'03*, pages 1–8, 2003b. doi: 10.1109/IPDPS.2003.1213150.

Carlos E. R. Alves, Edson N. Cáceres, and Siang W. Song. A coarse-grained parallel algorithm for the all-substrings longest common subsequence problem. *Algorithmica*, 45(3):301–335, 2006. doi: 10.1007/s00453-006-1216-z.

Carlos E. R. Alves, Edson N. Cáceres, and Siang W. Song. An all-substrings common subsequence algorithm. *Discrete Applied Mathematics*, 156(7):1025–1035, April 2008. doi: doi:10.1016/j.dam.2007.05.056.

Alberto Apostolico. String editing and longest common subsequences. In *Handbook of Formal Languages*, volume 2, pages 361–398. Springer-Verlag, 1997. ISBN: 3-540-60648-3.

Alberto Apostolico and Concettina Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2(1):315–336, 1987. doi: 10.1007/BF01840365.

Alberto Apostolico, Mikhail J. Atallah, Lawrence L. Larmore, and Scott McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM Journal on Computing*, 19(5):968–988, 1990. ISSN 0097-5397. doi: 10.1137/0219066.

Alberto Apostolico, Gad M. Landau, and Steven Skiena. Matching for run-length encoded strings. *Journal of Complexity*, 15(1):4–16, 1999. doi: 10.1006/jcom.1998.0493.

Apple Inc. Apple MacOS X, 2010. `http://www.apple.com/macosx/`.

Vladimir L. Arlazarov, E. A. Dinic, D. A. Kronrod, and I. A. Faradzev. On economical construction of the transitive closure of a directed graph. *Soviet Mathematics - Doklady*, 11:1209–1210, 1970.

Reuven Bar-Yehuda and Sergio Fogel. Partitioning a sequence into few monotone subsequences. *Acta Informatica*, 35(5):421–440, 1998. doi: 10.1007/s002360050126.

Michael Beeler, Ralph William Gosper, and Richard Schroeppel. HAKMEM. Memo 239, Artificial Intelligence Laboratory, MIT, 1972.

Jon Louis Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23(4):214–229, 1980. ISSN 0001-0782. doi: 10.1145/358841.358850.

Sergei Bespamyatnikh and Michael Segal. Enumerating longest increasing subsequences and patience sorting. *Information Processing Letters*, 76:7–11, 2000. doi: 10.1016/S0020-0190(00)00124-1.

Gianfranco Bilardi, Andrea Pietracaprina, Geppino Pucci, and Francesco Silvestri. Network-oblivious algorithms. In *Proceedings of IPDPS'07*, pages 1–10. IEEE, 2007. doi: 10.1109/IPDPS.2007.370243.

Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*. Oxford University Press, 2004. ISBN 0-19-852939-2.

Guy E. Blelloch. *Vector Models for Data-Parallel Computing*. MIT Press, 1990. ISBN 0-262-02313-X.

Guy E. Blelloch, Phillip B. Gibbons, Yossi Matias, and Marco Zagha. Accounting for memory bank contention and delay in high-bandwidth multiprocessors:. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):943–958, 1997. doi: 10.1109/71.615440.

Guy E. Blelloch, Rezaul Alam Chowdhury, Phillip B. Gibbons, Vijaya Ramachandran, Shimin Chen, and Michael Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In Shang-Hua Teng, editor, *Proceedings of SODA'08*, pages 501–510, 2008. doi: 10.1145/1347082.1347137.

Luc Boasson, Patrick Cégielski, Irène Guessarian, and Yuri Matiyasevich. Window-accumulated subsequence matching problem is linear. *Annals of Pure and Applied Logic*, 113(1–3):59–80, 2001. doi: 10.1016/S0168-0072(01)00051-3.

Olaf Bonorden, Ben Juurlink, Ingo von Otte, and Ingo Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187—207, 2003. doi: 10.1016/S0167-8191(02)00218-1.

Gerth Stølting Brodal. Finger search trees. In D.P. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*. Chapman& Hall/CRC, 2005. ISBN-10: 1584884355.

Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, 2004. doi: 10.1145/1186562.1015800.

Horst Bunke and János Csirik. An algorithm for matching run-length coded strings. *Computing*, 50(4):297–314, 1993. doi: 10.1007/BF02243873.

Rainer E. Burkard, Bettina Klinz, and Rüdiger Rudolf. Perspectives of Monge properties in optimization. *Discrete Applied Mathematics*, 70(2):95–161, 1996. doi: 10.1016/0166-218X(95)00103-X.

Timothy M. Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. *Algorithmica*, 50(2):236–243, 2008. doi: 10.1007/s00453-007-9062-1.

Timothy M. Chan and Mihai Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *Proceedings of SODA'10*, pages 161–173, 2010.

Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9:251–280, 1990. doi: 10.1016/S0747-7171(08)80013-2.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press and McGraw-Hill Book Company, 2001. ISBN 0-262-03293-7; 0-07-013151-1.

Maxime Crochemore and Ely Porat. Computing a longest increasing subsequence of length k in time O(n log log k). In *Visions of Computer Science - BCS International Academic Conference, 2008*, pages 69–74. British Computer Society, 2008. URL `http://www.bcs.org/server.php?show=ConWebDoc.22851`.

Maxime Crochemore, Costas S. Iliopoulos, Yoan J. Pinzon, and James F. Reid. A fast and practical bit-vector algorithm for the longest common subsequence problem. *Information Processing Letters*, 80(6):279–285, 2001. doi: doi:10.1016/S0020-0190(01)00182-X.

Maxime Crochemore, Gad M. Landau, and Michal Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. In *Proceedings of SODA'02*, pages 679–688, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. ISBN 0-89871-513-X.

Maxime Crochemore, Costas S. Iliopoulos, and Yoan J. Pinzon. Speeding-up Hirschberg and Hunt–Szymanski algorithms for the LCS problem. *Fundamenta Informaticae*, 56(1–2):89–103, 2003.

Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, New York, NY, USA, 2007. ISBN 0521848997.

Margaret O. Dayhoff, Robert M. Schwartz, and Bruce C. Orcutt. A model of evolutionary change in proteins. In Margaret O. Dayhoff, editor, *Atlas of Protein Structure*, volume 5(Suppl. 3), pages 345–352. National Biomedical Reasearch Foundation, Silver Spring, Md., 1979.

Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, third edition, 2008. ISBN-10: 3540779736.

Pilar de la Torre and Clyde P. Kruskal. Submachine locality in the bulk synchronous setting. In *Proceedings of Euro-Par'96*, pages 352–358, 1996. doi: 10.1007/BFb0024723.

Arthur L. Delcher, Simon Kasif, Robert D. Fleischmann, Jeremy Peterson, Owen White, and Steven L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

Robert P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51:161–166, 1950.

Ran Duan and Seth Pettie. Fast algorithms for (max, min)-matrix multiplication and bottleneck shortest paths. In *Proceedings of SODA '09*, pages 384–391, 2009.

David Eppstein, Zvi Galil, Raffaele Giancarlo, and Giuseppe F. Italiano. Sparse dynamic programming I: linear cost functions. *Journal of the ACM*, 39(3):519–545, 1992. ISSN 0004-5411. doi: 10.1145/146637.146650.

Paul Erdős and George Szekeres. A combinatorial problem in geometry. *Compositio Mathematica*, 2:463–470, 1935.

Agner Fog. Software optimization resources, 2010. `http://agner.org/optimize/`.

Steven Fortune and James Wyllie. Parallelism in random access machines. In *ACM Symposium on Theory of Computing (STOC '78)*, pages 114–118, New York, 1978. ACM Press.

Free Software Foundation (FSF). Autoconf - GNU Project, 2010a. `http://www.gnu.org/software/autoconf/`.

Free Software Foundation (FSF). Diffutils – gnu project, 2010b. `http://www.gnu.org/software/diffutils/`.

Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of FOCS'99*, pages 285–297. IEEE Computer Society Press, 1999. doi: 10.1109/SFFCS.1999.814600.

Brent Fulgham. The computer language benchmarks game, 2010. `http://shootout.alioth.debian.org`.

Ronald Garcia, Jeremy Siek, and Andrew Lumsdaine. Boost.MultiArray, 2010. `http://www.boost.org/doc/libs/1_42_0/libs/multi_array/doc/index.html`.

Thierry Garcia and David Semé. A load balancing technique for some coarse-grained multicomputer algorithms. In *Proceedings of the SCA 21st International Conference on Computers and Their Applications, Seattle, Washington, USA (CATA06)*, pages 301–306, 2006.

A. J. Gibbs and G. A. McIntyre. The diagram: A method for comparing sequences. Its uses with amino acids and nucleotide sequences. *European Journal of Biochemistry*, 16:1–11, 1970.

Leslie M. Goldschlager. A unified approach to models of synchronous parallel machines. *Journal of the ACM*, 29(4):1073–1086, 1982. doi: 10.1145/800133.804336.

Dan Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997. ISBN-10: 0521585198.

Dan Gusfield, K. Balasubramanian, and Dalit Naor. Parametric optimization of sequence alignment. *Algorithmica*, 12:312–326, 1994. doi: 10.1007/BF01185430.

Phuong Hoai Ha, Philippas Tsigas, and Otto J. Anshus. The synchronization power of coalesced memory accesses. In *Proceedings of DISC '08*, pages 320–334. Springer-Verlag, 2008. doi: 10.1007/978-3-540-87779-0_22.

Torben Hagerup. Sorting and searching on the word RAM. In *Proceedings of STACS'98*, volume 1373 of *LNCS*, pages 366–398. Springer-Verlag, 1998. doi: 10.1007/BFb0028575.

John M. Hammersley. A few seedlings of research. In *Proceedings of 6th Berkeley Symposium on Mathematical Statistics and Probability*, 1972.

Steven Henikoff and Jorja G. Henikoff. Amino acid substitution matrices from protein blocks. In *Proceedings of the National Academy of Sciences*, pages 10915–10919, 1992. Published as Proc. Natl. Acad. Sci., USA, volume 89, number 22.

Jonathan M. D. Hill, William F. McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob Bisseling.

BSPlib: The BSP programming library. *Parallel Computing*, 24, 1998. doi: 10.1016/S0167-8191(98)00093-3.

Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975. ISSN 0001-0782. doi: 10.1145/360825.360861.

Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM*, 24(4):664–675, 1977. doi: 10.1145/322033.322044.

Daniel S. Hirschberg. Serial computation of Levenshtein distances. In A. Apostolico and Z. Galil, editors, *Pattern Matching Algorithms*, chapter 4, pages 123–141. Oxford University Press, 1997. ISBN-10: 0195113675.

James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977. doi: 10.1145/359581.359603.

Heikki Hyyrö. Bit-parallel LCS-length computation revisited. In *Proceedings of AWOCA'04*, 2004.

Heikki Hyyrö, Kimmo Fredriksson, and Gonzalo Navarro. Increased bit-parallelism for approximate string matching. In *Proceedings of Experimental and Efficient Algorithms, Third International Workshop, WEA 2004*, volume 3059 of *LNCS*, pages 285–298. Springer, 2004. doi: 10.1007/978-3-540-24838-5_21.

Intel Corp. Intel Cilk++ Software Development Kit, 2010. `http://software.intel.com/en-us/articles/intel-cilk/`.

Intel Corporation. `http://www.intel.com`. URL `http://www.intel.com`.

Intel Corporation. *Intel Architecture Software Developer's Manual*. Intel, 1999a. URL `ftp://download.intel.com/design/PentiumII/manuals/24319002.PDF`. Basic Architecture.

Intel Corporation. *Intel Architecture Software Developer's Manual*. Intel, 1999b. URL `ftp://download.intel.com/design/PentiumII/manuals/24319102.PDF`. Instruction Set Reference.

Intel Corporation. Intel Threading Building Blocks Website, 2009. `http://www.threadingbuildingblocks.org/`.

Intel Corporation. Intel Threading Building Blocks Reference Manual. Technical Report Document Number 315415-005US, `http://www.intel.com`, 2010a.

Intel Corporation. IPP: `http://software.intel.com/en-us/intel-ipp/`, 2010b.

Joseph JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992. ISBN 0-201-54856-9.

Joseph JáJá, Christian Worm Mortensen, and Qingmin Shi. Space-Efficient and Fast Algorithms for Multidimensional Dominance Reporting and Range Counting. In *Proceedings ISAAC'04*, volume 3341 of *LNCS*, 2004. doi: 10.1007/978-3-540-30551-4_49.

Peter Johnson and other Yasm developers. The Yasm Modular Assembler, 2010. `http://www.tortall.net/projects/yasm/`.

Henry S. Warren Jr. *Hacker's Delight*. Addison-Wesley, 2003. ISBN-10: 0201914654.

Ben H. H. Juurlink and Harry A. G. Wijshoff. The E-BSP Model: Incorporating General Locality and Unbalanced Communication into the BSP Model. In *Proceedings of Euro-Par'96, Vol. II*, pages 339–347, 1996. doi: 10.1007/BFb0024721.

Khronos Group. OpenCL Overview, 2010. `http://www.khronos.org/opencl/`.

Samir Khuller, Joseph (Seffi) Naor, and Philip N. Klein. The lattice structure of flow in planar graphs. *SIAM Journal on Discrete Mathematics*, 6(3):477–490, 1993. doi: 10.1137/0406038.

Philip N. Klein. Multiple-source shortest paths in planar graphs. In *Proceedings of SODA'05*, pages 146–155. SIAM, 2005. ISBN 0-89871-585-7. doi: 10.1145/1070432.1070454.

Donald E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Massachusetts, 1973. ISBN-10: 0201896850.

Jan Krumsiek, Roland Arnold, and Thomas Rattei. Gepard: a rapid and sensitive tool for creating dotplots on genome scale. *Bioinformatics*, 23(8):1026–1028, 2007. doi: 10.1093/bioinformatics/btm039.

Peter Krusche. Experimental evalution of BSP programming libraries. *Parallel Processing Letters*, 18(1):7–21, 2005. doi: 10.1142/S0129626408003193.

Peter Krusche and Alexander Tiskin. Efficient longest common subsequence computation using bulk-synchronous parallelism. In *Proceedings of ICCSA'06, vol. (5)*, volume 3984 of *LNCS*, pages 165–174. Springer, 2006. doi: 10.1007/11751649_18.

Peter Krusche and Alexander Tiskin. Efficient parallel string comparison. In *Proceedings of ParCo'07*, volume 38 of *NIC Series*, pages 193–200. John von Neumann Institute for Computing, 2007. ISBN: 978-3-9810843-4-4.

Peter Krusche and Alexander Tiskin. New Algorithms for Efficient Parallel String Comparison. In *Proceedings of SPAA'10*, pages 209–216, 2010. doi: 10.1145/1810479.1810521.

S. Kiran Kumar and C. Pandu Rangan. A linear space algorithm for the LCS problem. *Acta Informatica*, 24(3):353–362, 1987. doi: 10.1007/BF00265993.

Gad M. Landau. Can dist tables be merged in linear time - An Open Problem. In *Proceedings of the Prague Stringology Conference, Prague, Czech Republic, August 28-30, 2006*, page 1, 2006.

Charles E. Leiserson. The Cilk++ concurrency platform. In *Proceedings of DAC'09*, pages 522–527, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-497-3. doi: 10.1145/1629911.1630048.

Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on Information Theory*, 23(1):75–81, 1976. doi: 10.1109/TIT.1976.1055501.

Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics - Doklady*, 6:707–710, 1966.

Bin Ma, John Tromp, and Ming Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002. doi: 10.1093/bioinformatics/18.3.440.

Jacob V. Maizel and Robert P. Lenk. Enhanced graphic matrix analysis of nucleic acid and protein sequences. *Proceedings of the National Academy of Sciences of the USA*, 78(12):7665–7669, 1981. doi: 10.1073/pnas.78.12.7665.

Colin L. Mallows. Patience sorting. *Bulletin of the Institute of Mathematics and its Applications*, 9:216–224, 1973.

Gurmeet Singh Manku. Puzzle: Fast bit counting, 2010. `http://gurmeetsingh.wordpress.com/2008/08/05/fast-bit-counting-routines/`.

William J. Masek and Michael S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20:18–31, 1980. doi: 10.1016/0022-0000(80)90002-1.

William F. McColl. General Purpose Parallel Computing. In A. M. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation, Proceedings of the 1991 ALCOM Spring School on Parallel Computation*, pages 337–391. Cambridge University Press, 1993. ISBN-10: 052141556X.

William F. McColl. Scalable Computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *LNCS*, pages 46–61. Springer-Verlag, 1995. ISBN-10: 3540601058.

Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976. doi: 10.1145/321941.321946.

Microsoft Corporation. Microsoft Windows, 2010. http://www.microsoft.com/WINDOWS/.

Ernst A. Munter. U.S. Patent 5,216,420, June 1993.

Eugene W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986. doi: 10.1007/BF01840446.

Takaaki Nakashima and Akihiro Fujiwara. A cost optimal parallel algorithm for patience sorting. *Parallel Processing Letters*, 16(1):39–52, 2006. doi: 10.1142/S0129626406002459.

Narao Nakatsu, Yahiko Kambayashi, and Shuzo Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Informatica*, 18(2): 171–179, 1982. doi: 10.1007/BF00264437.

Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search of similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970. doi: 10.1016/0022-2836(70)90057-4.

NVIDIA Corporation. Cuda zone: http://www.nvidia.com/object/cuda_home.html, 2009.

NVIDIA Corporation. NVIDIA CUDA SDK Code Samples, 2010. http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html.

OpenMP Architecture Review Board. OpenMP: http://www.openmp.org/, 2010. URL http://www.openmp.org/.

Sascha Ott, S. Gunawardana, Mike Downey, and Georgy Koentges. Loss-free identifcation of alignment-conserved CRMs. *In preparation.*, 2009.

Yi Pan. Basic Data Movement Operations on the LARPBS Model. In *Parallel Computing Using Optical Interconnections*, pages 227–247. Springer, 1998. doi: 10.1007/978-0-585-27268-9_11.

Wolfgang J. Paul, Peter Bach, Michael Bosch, Jörg Fischer, Cédric Lichtenau, and Jochen Röhrig. Real pram programming. In *Proceedings of Euro-Par 2002*, volume 2400 of *LNCS*, pages 522–531, Paderborn, Germany, 2002. Springer. doi: 10.1007/3-540-45706-2_71.

Emma Picot, Alexander Tiskin, Paul Brown, Peter Krusche, Isabelle Carré, and Sascha Ott. Evolutionary analysis of regulatory sequences (EARS) in plants. *The Plant Journal*, 64(1):165–176, 2010a. doi: 10.1111/j.1365-313X.2010.04314.x.

Emma Picot, Alexander Tiskin, Paul Brown, Peter Krusche, Isabelle Carré, and Sascha Ott. Ears: Evolutionary analysis of regulatory sequences, 2010b. `http://wsbc.warwick.ac.uk/ears/main.php`.

David R. Powell, Lloyd Allison, and Trevor I. Dix. Fast, optimal alignment of three sequences using linear gap costs. *Journal of Theoretical Biology*, 207(3):325–336, 2000. doi: 10.1006/jtbi.2000.2177.

Vaughan R. Pratt, Michael O. Rabin, and Larry J. Stockmeyer. A characterization of the power of vector machines. In *Proceedings of STOC'74*, pages 122–134. ACM, 1974. doi: 10.1145/800119.803892.

PUB library . `http://www.uni-paderborn.de/~bsp/`. URL `http://www.uni-paderborn.de/~bsp/`.

Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003. ISBN 0071232656.

Vijaya Ramachandran. *Foundations of Software Technology and Theoretical Computer Science*, chapter QSM: A general purpose shared-memory model for parallel computation, pages 1–5. Springer Berlin/Heidelberg, 1997. doi: 10.1007/BFb0058018.

Kim R. Rasmussen, Jens Stoye, and Eugene W. Myers. Efficient q-gram filters for finding all epsilon-matches over a given length. *Journal of Computational Biology*, 13(2):296–308, 2006. doi: 10.1089/cmb.2006.13.296.

Laurence Rauchwerger, Ping An, Alin Jula, Silvius Rus, Steven Saunders, Timmie G. Smith, Gabriel Tanase, Nathan Thomas, and Nancy M. Amato. STAPL: An adaptive, generic parallel C++ library. In *Proceedings of LCPC'01*, volume 2624 of *LNCS*, pages 193–208. Springer, 2001. doi: 10.1007/3-540-35767-X_13.

Peter Rice, Ian Longden, and Alan Bleasby. EMBOSS: The European molecular biology open software suite. *Trends in Genetics*, 16(6):276–277, 2000. doi: 10.1016/S0168-9525(00)02024-2.

Claus Rick. A new flexible algorithm for the longest common subsequence problem. *Nordic Journal of Computing*, 2(4):444–461, 1995.

Claus Rick. Simple and fast linear space computation of longest common subsequences. *Information Processing Letters*, 75(6):275–281, 2000. doi: 10.1016/S0020-0190(00)00114-9.

Bruce Eli Sagan. *The symmetric group*. Springer, second edition, 2010. ISBN-10: 1441928693.

Craige Schensted. Longest increasing and decreasing subsequences. *Canadian Journal of Mathematics*, 13:179–191, 1961.

Jeanette P. Schmidt. All highest scoring paths in weighted grid graphs and their application to finding all approximate repeats in strings. *SIAM Journal on Computing*, 27(4):972–992, 1998. doi: 10.1137/S0097539795288489.

Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Pradeep Dubey, Stephen Junkins, Adam Lake, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, Michael Abrash, Jeremy Sugerman, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. *IEEE Micro*, 29(1):10–21, 2009. doi: 10.1109/MM.2009.9.

David Semé. A CGM algorithm solving the longest increasing subsequence problem. In *Proceedings of ICCSA'06, Part V*, volume 3984 of *Lecture Notes in*

Computer Science, pages 10–21. Springer, 2006. ISBN 3-540-34079-3. doi: 10.1007/11751649_2.

David Semé and Sideny Youlou. An Efficient Parallel Algorithm for the Longest Increasing Subsequence Problem on a LARPBS. In *Proceedings of the International Conference on Parallel and Distributed Computing Applications and Technologies*, pages 251–258. IEEE Computer Society, 2007. doi: 10.1109/PDCAT.2007.74.

Hanmao Shi and Jonathan Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992. doi: 10.1016/0743-7315(92)90075-X.

David B. Skillicorn, Jonathan M. D. Hill, and William F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995. ISBN 0262691841.

Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13: 354–356, 1969.

Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1987.

Wijnand J. Suijlen and Rob H. Bisseling. BSPonMPI, 2010. `http://bsponmpi.sourceforge.net/`.

The Arabidopsis Information Resource (TAIR). Arabidopsis gene AT5G61380, 2010. `http://arabidopsis.org/servlets/TairObject?id=133196&type=locus`.

The Framewave Group. Project homepage: `http://framewave.sourceforge.net/`, 2009.

The SCons Foundation. SCons: A software construction tool, 2010. `http://www.scons.org/`.

The University of Warwick. The Centre for Scientific Computing, The University of Warwick, `http://www.csc.warwick.ac.uk`, 2009.

Alexander Tiskin. Efficient representation and parallel computation of string-substring longest common subsequences. In *Proceedings of ParCo'05*, volume 33 of *NIC Series*, pages 827–834. John von Neumann Institute for Computing, 2005. ISBN: 3-00-017352-8.

Alexander Tiskin. Longest common subsequences in permutations and maximum cliques in circle graphs. In *Proceedings of CPM*, volume 4009 of *LNCS*, pages 270–281, 2006. doi: 10.1007/11780441_25.

Alexander Tiskin. Semi-local longest common subsequences in subquadratic time. *Journal of Discrete Algorithms*, 6(4):570–581, 2008a. doi: 10.1016/j.jda.2008.07.001.

Alexander Tiskin. Semi-local string comparison: Algorithmic techniques and applications. *Mathematics in Computer Science*, 1(4):571–603, 2008b. doi: 10.1007/s11786-007-0033-3. See also arXiv: 0707.3619.

Alexander Tiskin. Semi-local string comparison: Algorithmic techniques and applications. Book draft, arXiv: 0707.3619, 2010a.

Alexander Tiskin. Fast distance multiplication of unit-Monge matrices. In *Proceedings of SODA'10*, pages 1287–1296, 2010b.

Alexandre Tiskin. The bulk-synchronous parallel random access machine. *Theoretical Computer Science*, 196(1–2):109–130, 1998. doi: doi:10.1016/S0304-3975(97)00197-7.

Esko Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1–3):100–118, 1985. doi: 10.1016/S0019-9958(85)80046-2.

Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990. ISSN 0001-0782. doi: 10.1145/79173.79181.

Leslie G. Valiant. A bridging model for multi-core computing. In Dan Halperin and Kurt Mehlhorn, editors, *Proceedings of the European Symposia on Algorithms (ESA'08)*, volume 5193 of *LNCS*, pages 13–28. Springer, 2008. doi: 10.1007/978-3-540-87744-8_2.

David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2003. ISBN 0-201-73484-2.

Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974. ISSN 0004-5411. doi: 10.1145/321796.321811.

Xingzhi Wen and Uzi Vishkin. The XMT FPGA prototype/cycle-accurate-simulator hybrid. In *The 3rd Workshop on Architectural Research Prototyping, WARP08*, 2008. June 21-22, 2008, Beijing, China, held in conjunction with ISCA 2008.

Wikipedia. Wikipedia: Linux, 2010. `http://en.wikipedia.org/wiki/Linux`.

Sun Wu, Udi Manber, Eugene W. Myers, and Webb Miller. An $O(NP)$ sequence comparison algorithm. *Information Processing Letters*, 35(6):317–323, 1990. doi: 10.1016/0020-0190(90)90035-V.