ENGINEERING ALGORITHMS FOR SOLVING GEOMETRIC AND GRAPH PROBLEMS ON LARGE DATA SETS

by

Adan Jose Cosgaya Lozano

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

at

Dalhousie University
Halifax, Nova Scotia
March 2011

© Copyright by Adan Jose Cosgaya Lozano, 2011

DALHOUSIE UNIVERSITY

FACULTY OF COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "ENGINEERING ALGORITHMS FOR SOLVING GEOMETRIC AND GRAPH PROBLEMS ON LARGE DATA SETS" by Adan Jose Cosgaya Lozano in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Dated: March 14, 2011

External Examiner: ____________________

Research Supervisors: ____________________

____

Examining Committee: ____________________

____

Departmental Representative: ____________________

DALHOUSIE UNIVERSITY

DATE: March 14, 2011

AUTHOR: Adan Jose Cosgaya Lozano

TITLE: ENGINEERING ALGORITHMS FOR SOLVING GEOMETRIC AND GRAPH PROBLEMS ON LARGE DATA SETS

DEPARTMENT OR SCHOOL: Faculty of Computer Science

DEGREE: Ph.D. CONVOCATION: May YEAR: 2011

Permission is herewith granted to Dalhousie University to circulate and to have copied for non-commercial purposes, at its discretion, the above title upon the request of individuals or institutions. I understand that my thesis will be electronically available to the public.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

The author attests that permission has been obtained for the use of any copyrighted material appearing in the thesis (other than brief excerpts requiring only proper acknowledgement in scholarly writing), and that all such use is clearly acknowledged.

Signature of Author

Table of Contents

List of Tables . . . . **viii**

List of Figures . . . . **ix**

Abstract . . . . **xi**

List of Abbreviations and Symbols Used . . . . **xii**

Acknowledgements . . . . **xiv**

Chapter 1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 3

1.3 Organization of the Thesis 5

Chapter 2 Memory Hierarchies and Their Impact on Algorithm Design **7**

2.1 Memory Hierarchies 7

2.2 Impact on Traditional Algorithms 8

2.3 I/O-Efficient Algorithms 9

2.4 Algorithm Engineering 10

Chapter 3 Introduction to Skyline Computation **12**

3.1 Overview 12

3.1.1 The Concept of Skyline Query 13

3.2 Skyline Queries in Centralized Environments 14

3.2.1 Algorithms Without Preprocessing 15

3.2.2 Index-based Algorithms 17

3.2.3 Extensions of Skyline Computations 19

3.2.4 Detailed Discussion of Skyline Algorithms 20

3.2.4.1 Divide and Conquer (DC) 20

3.2.4.2 Branch-and-Bound Skyline (BBS) 21
3.3 Skyline Queries in Distributed Environments 24
3.3.1 Skyline Computation in Peer-to-Peer Networks 24
3.3.2 Skyline Computation in Parallel 26
3.4 Shared-Nothing Clusters 29
3.5 Contributions 30

Chapter 4 Parallel Skyline Computation 32
4.1 Overview 32
4.2 Parallel Skyline Algorithm 33
4.2.1 Data Distribution 35
4.2.2 Local Skyline Computation 36
4.2.3 Global Skyline Computation 37
4.2.3.1 Few Local Skyline Points 37
4.2.3.2 Many Local Skyline Points 39
4.2.4 Complexity 40
4.3 A Parallel R-Tree Construction Algorithm 41
4.3.1 A Pointerless R-Tree Representation 42
4.3.2 Parallel R-Tree Construction 44
4.3.3 Querying the Constructed R-Tree 45
4.4 Implementation Details 45
4.5 Performance Evaluation 46
4.5.1 Platforms and Measurements 47
4.5.2 Performance Measurement 47
4.5.3 Datasets 48
4.5.4 Speed-up Evaluation 49
4.5.4.1 Total Speed-up 49
4.5.4.2 Query Speed-up 51
4.5.5 Scale-up Evaluation 55

Chapter 5 External-Memory Skyline Computation 58
5.1 Overview 58

5.2 DFS-Skyline . . . 60
5.2.1 R-tree Construction . . . 60
5.2.2 Skyline Construction . . . 62
5.3 Presorted-BBS . . . 64
5.4 Performance Evaluation . . . 64
5.4.1 Comparison of IM and EM Algorithms . . . 66
5.4.2 Comparison of External Memory Algorithms . . . 69

Chapter 6 I/O-Efficient Algorithms for Massive Graphs . . . 73
6.1 Overview . . . 73
6.1.1 Terminology and Definitions . . . 74
6.1.2 Massive Graphs in the Real-World . . . 76
6.2 State of the Art . . . 79
6.2.1 Difficulties with I/O-Efficient Graph Exploration . . . 79
6.2.2 Techniques for I/O-Efficient Graph Algorithms . . . 80
6.2.2.1 Techniques for Speeding Up Graph Exploration . . . 80
6.2.2.2 Non-Exploration-Based Techniques . . . 82
6.2.3 Engineering of Graphs Algorithms . . . 84
6.3 Contributions . . . 86

Chapter 7 External-Memory Strong Connectivity . . . 88
7.1 Introduction . . . 88
7.2 A Contraction-Based Strong Connectivity Algorithm . . . 89
7.2.1 Preprocessing Phase . . . 89
7.2.2 Contraction Phase . . . 91
7.2.3 Postprocessing . . . 93
7.2.4 Analysis . . . 94
7.3 Implementation Details . . . 95
7.4 Performance Evaluation . . . 99
7.4.1 Environment and Settings . . . 100
7.4.2 Data Sets . . . 101
7.4.3 EmSCC vs. SeSCC . . . 102

7.4.4 Factors Affecting the Performance of EmSCC 103

Chapter 8 External-Memory Topological Sorting 108

8.1 Introduction 108

8.2 Topological Sorting by Iterative Improvement 109

8.2.1 Computing the Initial Numbering 110

8.2.2 Growing the Satisfied Subgraph 111

8.2.3 Analysis 112

8.2.4 Satisfying Local Edges 114

8.3 Other Approaches to Topological Sorting 116

8.3.1 Topological Sorting Using Semi-External DFS 116

8.3.2 Iterative Peeling of Sources and Sinks 116

8.3.3 Divide and Conquer Based on Reachability Queries 118

8.4 Implementation Details 119

8.5 Performance Evaluation 120

8.5.1 Environment and Settings 120

8.5.2 Data Sets 120

8.5.3 Experimental Results 122

8.5.3.1 Comparison of Running Times 122

8.5.3.2 The Effect of the Graph's Structure 126

8.5.3.3 Further Analysis of IterTS 127

Chapter 9 Conclusions 131

Bibliography 133

List of Tables

4.1 Data sets and their skyline sizes. 49

7.1 Maximum sizes of vertex sets processed by SE-DFS 100

7.2 Experimental results with strong connectivity on SE setting . . 103

7.3 Experimental results with strong connectivity on EM setting . 104

8.1 Experimental results with topological sorting algorithms 123

List of Figures

2.1 Memory hierarchy. 8

3.1 A sample dataset and its skyline. 14

3.2 Recursive filtering in DC. 21

3.3 The R-tree for the point set from Figure 3.1. 22

3.4 Sequential query times for DC and BBS 23

3.5 A Shared-nothing cluster. 30

4.1 Illustration of the two phases of Algorithm 1 35

4.2 Partitions of point set over 4 processors 36

4.3 Illustration of pointerless R-tree 43

4.4 Parallel R-tree construction: merging the R-tree pieces. 44

4.5 Synthetic data distributions used in the experiments. 48

4.6 Speedup results on uniform data 50

4.7 Speedup results on hydrological data 52

4.8 Speedup results on anticorrelated data 53

4.9 Break-down of a $2d$ query with 5M points 54

4.10 Scale-up results on uniform data with 5M points 56

4.11 Scale-up results on hydrological data with 5M points 56

5.1 DFSS vs. PRESORTED-BBS 60

5.2 IM and EM results of DFSS and PRESORTED-BBS 67

5.3 Query times of DFSS, PRESORTED-BBS and SFS 70

5.4 DFSS and PRESORTED-BBS break-down of a $6d$ query . . . 71

6.1 Illustration of directed graphs. 76

7.1 Spanning tree and the Euler tour around it 91

7.2 EmSCC: Illustration of a contraction step 93
7.3 EmSCC: Illustration of postprocessing 94
7.4 Illustration of internal-memory representation of the graph . . 99
7.5 Variation of EmSCC total time 106

8.1 IterTS: Illustration of selecting in-edge for a vertex 111
8.2 IterTS: Illustration of heuristic procedure 114
8.3 IterTS: Running time, I/O volume and convergence graphs . 128

Abstract

This thesis focuses on the engineering of algorithms for massive data sets. In recent years, massive data sets have become ubiquitous and existing computing applications, for the most part, cannot handle these data sets efficiently: either they crash or their performance degrades to a point where they take unacceptably long to process the input. Parallel computing and I/O-efficient algorithms provide the means to process massive amounts of data efficiently. The work presented in this thesis makes use of these techniques and focuses on obtaining practically efficient solutions for specific problems in computational geometry and graph theory.

We focus our attention first on *skyline computations*. This problem arises in decision-making applications and has been well studied in computational geometry and also by the database community in recent years. Most of the previous work on this problem has focused on sequential computations using a single processor, and the algorithms produced are not able to efficiently process data sets beyond the capacity of main memory. Such massive data sets are becoming more common; thus, parallelizing the skyline computation and eliminating the I/O bottleneck in large-scale computations is increasingly important in order to retrieve the results in a reasonable amount of time. Furthermore, we address two fundamental problems of graph analysis that appear in many application areas and which have eluded efforts to develop theoretically I/O-efficient solutions: computing the *strongly connected components* of a directed graph and *topological sorting* of a directed acyclic graph.

To approach these problems, we designed algorithms, developed efficient implementations and, using extensive experiments, verified that they perform well in practice. Our solutions are based on well understood algorithmic techniques. The experiments show that, even though some of these techniques do not lead to provably efficient algorithms, they do lead to practically efficient heuristic solutions. In particular, our parallel algorithm for skyline computation is based on divide-and-conquer, while the strong connectivity and topological sorting algorithms use techniques such as graph contraction, the Euler technique, list ranking, and time-forward processing.

List of Abbreviations and Symbols Used

BBS	Branch and Bound Skyline Algorithm
BNL	Block-Nested-Loops Skyline Algorithm
BRT	Buffered Repository Tree
CGM	Coarse Grained Multicomputer
DAG	Directed Acyclic Graph
DC	Divide and Conquer Skyline Algorithm
DFSS	DFS-Skyline Algorithm
DSL	Distributed Skyline Algorithm
EmSCC	External Memory Strongly Connected Components Algorithm
GIS	Geographic Information Systems
IterTS	Topological Sorting by Iterative Improvement Algorithm
LESS	Linear Elimination Sort for Skyline Algorithm
MBB	Minimum Bounding Box
MPI	Message Passing Interface
MST	Minimum Spanning Tree
NN	Nearest-Neighbor Skyline Algorithm
PeelTS	Iterative Peeling of Sources and Sinks Algorithm
PaDSkyline	Parallel Distributed Skyline

ReachTS Divide and Conquer Based on Reachability Queries Algorithm

SeSCC Semi-External Strongly Connected Components Algorithm

SeTS Topological Sorting Using Semi-External DFS Algorithm

SALSA Sort and Limit Skyline Algorithm

SCC Strongly Connected Component

SCSG Strongly Connected Subgraph

SFS Sort-Filter Skyline Algorithm

SSP Skyline Search Space Partitioning Algorithm

STXXL Standard Template Library for Extra Large Data Sets

Acknowledgements

First and foremost, I would like to thank my supervisors Andrew Rau-Chaplin and Norbert Zeh for their guidance and their generous support. I greatly appreciate the time and the attention they have always given me.

I am sincerely grateful to Andrew for introducing me to research work and for being present during my early years at Dalhousie. I would like to also thank Norbert for becoming more a friend than a supervisor. His valuable feedback and attention to detail contributed enormously to the completion of this work. He has been very patient with me and taught me not to rush into a conclusion without having all the facts straight.

I would also like to thank the members of my thesis examination committee for taking the time to read this thesis.

I am also grateful to my parents who, despite the distance, have always expressed their unconditional love, encouragement, and support.

And last but not the least, I would like to thank my wife Monica and daughter Natalia for their love and patience throughout these years. Monica has always been encouraging and understanding, even as I had been mentally immersed in my work. I could not have done it without you.

Chapter 1

Introduction

The main contribution of this thesis is the development of solutions for two types of problems: *geometric problems*, in particular the skyline computation of multi-dimensional data, and *graph problems*, specifically connectivity of directed graphs. These problems are, in terms of used techniques and in terms of focus, unrelated. Nevertheless, the motivation behind both problems is the same, namely, challenges in processing large amounts of data.

1.1 Motivation

Technological advances in areas such as communications, sensors, high performance computing, and digital storage technology, together with the ubiquitous use of computers, have made it possible to acquire and store massive collections of data. Consider the following examples. Wal-Mart's customer transactions database contains over 2.5 petabytes of data which is used for strategic decision making [9]. Google's search engine provides fast text search by storing and indexing over 26 billion web pages [8]. A telecommunications company, such as AT&T, is able to collect and store billions of call detail records per day [51], easily making its storage requirements reach several tera- to petabytes. Online geographic information systems, such as Microsoft Bing Maps [7] and Google Earth [4], store terabytes of satellite images, street maps and terrain data. Advances in biotechnology have made it possible for some research centers to generate terabytes of genomic sequencing data [38]. These are but a small sample of an increasing number of application areas where massive amounts of data are collected and stored. This availability of high-quality data presents opportunities for new discoveries in scientific research or to make better business decisions. What is needed are the tools to analyze these terabyte and petabyte-size data sets.

The development of algorithms able to process massive data sets efficiently is challenging. There are two main problems to overcome. (1) Massive data sets often do not fit in the computer's main memory, forcing the algorithm to retrieve its data from disk. This takes significantly longer than accessing data in main memory, and the computation grinds to a halt. This problem can be overcome by accessing and processing the data in large blocks. Traditional internal-memory algorithms do not exhibit the necessary access locality to facilitate this. Hence, there is a need for "I/O-efficient" algorithms designed specifically with access locality in mind, in order to alleviate the I/O bottleneck. (2) While data set sizes increase rapidly in many application areas, processor speeds no longer increase. Therefore, in order to keep up with increasing input sizes, and continue to process these inputs in a reasonable amount of time, it becomes necessary to utilize many processors simultaneously. This requires the design of parallel algorithms that distribute the work over the available processors.

The work developed in this thesis exploits parallelism and I/O-efficient techniques to approach specific problems in computational geometry and graph theory. Due to their fundamental nature these problems typically arise in decision support systems, web modelling, and many more application areas.

Decision support systems usually work with data stored in archival data warehouses, with the number of records typically being in the order of billions, and sophisticated data analysis is frequently performed through database and OLAP queries [46]. For instance, a database of an international retailer (e.g., Wal-Mart) might consist of all the customer transactions over the world in the course of a year, and most meaningful queries over this data set are multi-criteria queries. An example of such queries is to find all the stores having low profit and high expenses. Even if the output of this query is a small number of records representing the sought stores, the computation still involves large-scale database searches on terabytes or even petabytes of data. In Chapters 4 and 5 we give algorithms for multi-criteria query processing which use parallel computing and I/O-efficient algorithmic techniques to speed up computation.

Web modelling applications perform large-scale data processing to study the topological properties of the World Wide Web. These applications analyze graphs built

from large-scale crawls with sizes that reach several hundreds of gigabytes to terabytes. The resulting graphs are too large to fit in the main memory of a computer. Nevertheless, it is very important to process these graphs as a whole, since information may be lost if they are processed in fragments. For example, an important task in these applications is to identify web communities [87], which are often approximated by cliques or "almost-cliques". As a first approximation that is much easier to compute, at least in internal memory, one may also consider strongly connected components (SCC) to be communities. An SCC of a graph is a set of vertices such that there is at least one directed path from every vertex to every other vertex in the set. The SCCs are traditionally computed using a depth-first search (DFS) traversal. However, in an external setting, in which the data is stored on disk, DFS performs extremely badly since it basically causes one disk access per computation step. On the other hand, if we were to compute only the SCCs of memory-size subgraphs, it would be unlikely that we find all SCCs. Specifically, if the graph contains SCCs that span multiple partitions, these SCCs are not computed using this approach. This problem affects large SCCs beyond the size of main memory but may also affect smaller SCCs if the partition is chosen naively. In Chapter 7 we provide an external-memory SCC algorithm that uses graph contraction to postpone running into this problem.

There exist a number of other application areas where the problems addressed in this thesis commonly arise. These include *DNA sequence analysis*, *social networks* and *geographic information systems* to name a few. Recent multiple sequence alignment algorithms [137, 138] operate on the DNA sequences' de Bruijn graph and reduce the problem to a traversal of an acyclic subgraph of this graph. Similar to web modelling, a typical problem in social networks, is to find communities of people on graphs of social networks. Geographic information systems often need to topologically sort large graphs representing terrains. In all of these applications a challenge exists because of the massiveness of the data. In Chapters 7 and 8 we provide I/O-efficient algorithms for approaching these problems on large graphs.

1.2 Contributions

The main contributions of this thesis are:

Parallel and external-memory skyline computation. Computing the skyline of a set of records is an important operation in applications involving multi-criteria decision making. The goal is to retrieve all objects in a data set that have the property that no other object is better according to all of a given set of criteria. This is a well studied problem in computational geometry. A provably efficient sequential algorithm in arbitrary dimensions and a parallel algorithm in three dimensions have been obtained. More recently, the database community has concentrated on this problem, particularly on fast sequential heuristics in internal memory (see Section 3.2 for details). In this thesis, we propose fast algorithms for parallel and external-memory (I/O-efficient) skyline computations that can handle massive data sets. These algorithms make use of well known techniques such as divide-and-conquer; we also designed a specialized tree representation that can be used to efficiently exchange R-trees between processors in a parallel machine. Our experiments confirm that the algorithms can efficiently process data sets beyond the reach of existing algorithms and can take advantage of parallel architectures to speed up the processing of large amounts of data. This work has been published in [54].

External-memory strong connectivity. Computing the strongly connected components of a directed graph is a fundamental problem in graph analysis. Given a directed graph, a strongly connected component is a maximal subgraph in which every vertex is reachable from every other vertex. There exist provably efficient internal-memory strong connectivity algorithms; however, their performance deteriorates drastically once they have to process massive graphs. I/O-efficient solutions have been proposed for special graph classes, such as directed planar graphs. However, no theoretically I/O-efficient solution is known for general directed graphs. In this thesis we propose a new I/O-efficient algorithm for computing strongly connected components of massive graphs. This algorithm uses graph contraction, which is the most important technique for solving connectivity problems on *undirected* graphs I/O-efficiently. Our experiments confirm that our algorithm can process data sets beyond the reach of existing algorithms efficiently, which demonstrates that, at least as a heuristic, graph contraction is a useful tool for solving connectivity problems on *directed* graphs. The results have appeared in [55].

External-memory topological sorting. Topological sorting is a problem that appears in applications that need to compute an ordering of a set of elements that satisfies a set of application-specific constraints. More specifically, these constraints are modelled by a graph whose vertices represent the elements and with an edge from one vertex to another if the former has to precede the latter. The goal is to compute an ordering of the vertices of the graph such that the tail of every edge precedes its head in the ordering. Such an ordering is called a *topological ordering* and exists if and only if the graph does not contain directed cycles. Similar to computing SCCs, topological sorting is a hard problem for massive directed graphs, whereas in internal memory it can be solved efficiently using depth-first search or even simpler methods. There is no theoretically I/O-efficient algorithm for topologically sorting general directed graphs. However, I/O-efficient algorithms for this problem have been obtained for special graph classes, such as planar graphs. In this thesis we propose an I/O-efficient algorithm specifically designed for computing the topological ordering of massive directed acyclic graphs. We did extensive experiments comparing it with different I/O-efficient algorithms based on existing sequential and parallel approaches. The experimental evaluation confirms that our approach is able to solve the problem on data sets beyond memory size in a reasonable amount of time and it also outperforms its competitors. All algorithms use well known I/O-efficient techniques, such as Euler tour computation, list ranking and time-forward processing. This work has been published in [16].

In this thesis we provide new, practically efficient solutions to a number of problems on massive data sets. A possibly even more important insight of the work on I/O-efficient algorithms for massive graphs is that, at least as a basis for efficient heuristic solutions, some techniques for solving problems on undirected graphs, such as graph contraction, can also be used to solve problems on directed graphs.

1.3 Organization of the Thesis

The remainder of this thesis is organized as follows. In Chapter 2 we will address the memory hierarchy that exists on modern computers and discuss the problems with internal-memory algorithms that ignore the presence of such hierarchy. We will also introduce the external memory (I/O) model of computation. From Chapter 3

onwards, the thesis can be logically divided into two parts, reflecting the nature of the two types of problems being considered. In the first part, comprising Chapters 3, 4 and 5, we present our work on skyline query computation. In the second part, consisting of Chapters 6, 7 and 8, we discuss algorithms for directed graphs.

Chapter 3 provides background information on skyline algorithms. In Chapter 4 we introduce our parallel algorithm for skyline computation. In Chapter 5 we present our external memory algorithm for the same problem. Both chapters also discuss the experimental results we obtained using the algorithms discussed in these chapters.

The focus of the remaining chapters is on algorithms for massive directed graphs. In Chapter 6 we give an introduction to the field of I/O-efficient graph algorithms, including a description of algorithmic techniques used in later chapters. In Chapter 7 we design and implement an algorithm for computing strongly connected components on massive directed graphs. In Chapter 8 we present our work on topological sorting of massive directed acyclic graphs.

Chapter 9, summarizes the results of this thesis and provides an outlook on future work.

Chapter 2

Memory Hierarchies and Their Impact on Algorithm Design

The purpose of this chapter is twofold. First, it discusses why algorithms designed in standard computational models fail to take advantage of the memory hierarchies present in modern computers. Second, it lays the groundwork for the design of I/O-efficient algorithms designed specifically to minimize accesses to the slowest level of the memory hierarchy: the disk(s). The emphasis is therefore not on the details of computer architectures, but rather on general concepts relevant to our work.

2.1 Memory Hierarchies

Computational models abstract how real computers work in order to provide a basis for designing and analyzing algorithms. The most popular model used for algorithm design is the *random access machine* (RAM) model of computation. The RAM model assumes that the computer is equipped with an infinitely large physical memory and that any memory address can be accessed in constant time. While this has proven useful as a tool for studying the computational complexity of a wide range of problems, it fails to capture the real performance of algorithms on modern memory hierarchies.

The reason is that the memory systems of modern computers are composed of a hierarchy of memory layers of different sizes and *widely varying access costs*. Typical layers, arranged here from the bottom up, are external memory, internal memory, L2 cache, L1 cache and registers. The largest and slowest memory is the *external memory*, which is generally supported by hard disk(s) storing from gigabytes to terabytes of data. The *internal (main) memory* is slower than cache memory but nowadays can reach sizes of several gigabytes and is several orders of magnitude faster than external memory. The *cache memory* has a comparably small capacity but allows the CPU to access data very fast; memory systems usually contain at least two levels of cache: L1 and L2. The *registers* are integrated in the CPU and provide the fastest data access. See Figure 2.1 for an illustration.

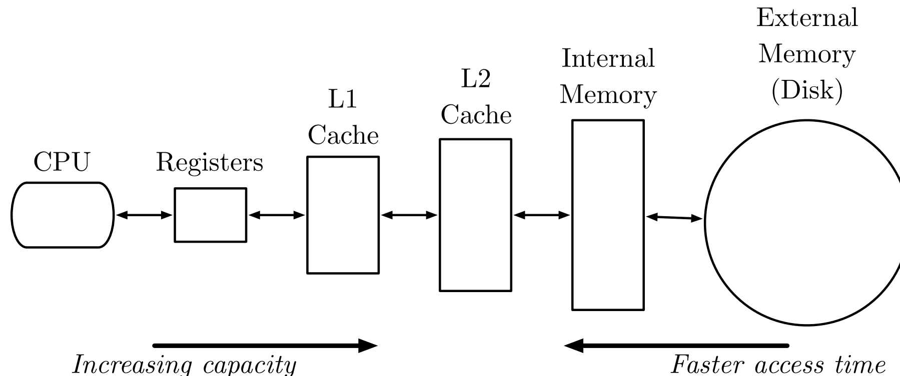


Figure 2.1: Memory hierarchy.

The *bandwidths* of the memory levels—the rates at which they can provide data once it has been addressed—are fairly similar. The *latencies* in addressing data items, on the other hand, differ by orders of magnitude and are the main reason for the high access cost of memory levels far away from the CPU. For instance, disk access may take several milliseconds, whereas memory access takes only nanoseconds, this is a difference of about 10^6.

In order to amortize these latencies over more than one accessed data item, data is transferred between adjacent layers in blocks. The higher the latency the bigger the block size required to amortize it. This approach of amortizing access latencies through blockwise data access is likely to persist at least in the near future. Flash drives, for example, are an emerging alternative to disk-based storage and currently have a much stronger dependence on blockwise data access than disk drives, with differing block sizes for read and write operations and a write block size significantly beyond the block sizes of current hard drives. See [15, 68] for more details on the properties of flash memory devices.

2.2 Impact on Traditional Algorithms

For blockwise data transfer to be an effective amortization strategy, algorithms have to ensure that most of the time data elements they access consecutively are stored in the same block—otherwise each processed element still incurs the full access latency. Traditional algorithms do not exhibit this access locality and they slow down by a factor 10^4–10^6 once they need to access external memory.

Graph algorithms are particularly affected by this phenomenon because the vast majority of these algorithms is based on graph exploration strategies, such as depth-first search (DFS) and breadth-first search (BFS) [53]. During graph exploration it is hard to avoid accessing data in a random fashion since there is no a priori knowledge about the order in which the vertices are to be visited. For example, consider the internal memory DFS algorithm [53]. DFS explores the vertices of a graph one vertex at a time. When visiting a vertex, the algorithm needs to retrieve its adjacency list to identify the vertex's neighbors that it can visit next. This causes one random access per vertex. To choose the neighbor to be visited next, it needs to inspect each neighbor until it finds one that has not been visited before. This causes one random access per edge. Thus, standard DFS does not take advantage of blockwise disk access at all and breaks down on data beyond the size of main memory. Efforts to develop improved DFS algorithms have met with limited success.

2.3 I/O-Efficient Algorithms

Given the performance limitations that traditional algorithms suffer when processing large data sets, the use of computational models that capture the memory hierarchy is very important for designing algorithms for massive data sets. One such model is the *I/O model* (or external memory model) introduced by Aggarwal and Vitter [14] which focuses on minimizing the block accesses to the by far slowest level of the memory hierarchy: disk. The model assumes a single central processing unit (CPU) and two levels of memory: internal memory and disk (i.e., external memory). The CPU can only directly access data in the internal memory. The internal memory has a limited capacity of M data items. The disk is conceptually infinitely large. The disk is partitioned into blocks of B consecutive data items. The transfer of data between internal memory and disk happens using I/O operations, each of which transfers one block of data. A disk access is referred to as an *I/O operation (I/O)*. The measure of performance of an algorithm is the number of disk I/Os performed during its execution.

It is shown in [14] that reading N elements stored contiguously on disk requires $\text{scan}(N) = \lceil N/B \rceil$ I/Os. The number of I/O operations required to sort N data items is $\text{sort}(N) = \Theta(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os. For all realistic values of N, M and B, we have

$\text{scan}(N) < \text{sort}(N) \ll N$. Therefore, there is a significant performance difference between an algorithm performing $\text{scan}(N)$ or $\text{sort}(N)$ I/Os and one performing N I/Os. A large number of algorithms and data structures for the I/O model exist. For instance, for comprehensive surveys of algorithms developed in the I/O model, refer to [19, 101, 130, 131].

2.4 Algorithm Engineering

Despite the many theoretical results obtained in the I/O model, many of the developed algorithms are far too complex to be of practical value, both in terms of their performance and of the difficulty to implement them. To approach this problem, much attention in the I/O-efficient algorithms community has been devoted to the development of practically efficient algorithms. The development of such algorithms has become a discipline in computer science of its own, known as *algorithm engineering* [43, 114].

Algorithm engineering considers the process of developing an algorithm to be a cycle consisting of four steps: design, analysis, implementation and experimental evaluation. Specifically, the algorithm is initially designed and analyzed; the analysis is used to obtain a performance prediction. Next, the algorithm is implemented and experiments are performed using artificial and/or real-world data to evaluate the real performance of the algorithm; the experimental evaluation might provide insights into the limitations of the algorithm as a basis for the development of improved solutions. The development of such an improved solution involves the same four steps of design, analysis, implementation and experimental evaluation—the cycle continues.

A great deal of effort has focused already on engineering I/O-efficient algorithms for large data sets. This work has been very successful and has led to very good libraries of I/O-efficient algorithms. These include the transparent parallel I/O environment (TPIE) developed by Vengroff et al. [25, 129] and the standard template library for extra large data sets (STXXL) by Dementiev et al. [59, 60]. TPIE was the first software project offering implementations of I/O-efficient algorithms and data structures. It contains implementations of I/O-efficient primitives, such as scanning,

merging, sorting and distributing, together with data structures, such as an I/O-efficient priority queue. Several projects have been developed based on TPIE, including Geographic Information Systems (GIS) terrain applications [13,22]. STXXL offers external-memory implementations of STL containers and algorithms that can process huge volumes of data stored on disks. Recently, STXXL has been used in the engineering of I/O-efficient algorithms for solving problems on massive graphs [18,61,118].

Chapter 3

Introduction to Skyline Computation

In this chapter, we discuss the skyline query problem and review a wide range of skyline algorithms that have been developed before. Some of these algorithms which are key to understanding our methods are discussed in more detail. We also introduce the shared-nothing architecture, which is the platform targeted by our parallel skyline algorithm. Finally, we summarize the contributions we make in this part of the thesis.

3.1 Overview

Motivated by rapidly increasing data volumes in many applications, database management systems are increasingly required to provide efficient methods for processing large data sets. The analysis of large high-dimensional data sets require methods to compute meaningful summaries of the data, as humans cannot make sense of massive data sets without such summaries.

Data summarization techniques have been well studied in the database community. Common techniques include finding the minimum or maximum value, computing the average over a set of values and finding the median value. In recent years another summarization method has emerged: the skyline query. In contrast to other methods that consider each criterion (i.e., attribute) in isolation, the skyline computation concerns itself with summarization of a set of criteria and, thus, is often useful in multi-criteria decision making applications. However, sequential skyline algorithms do not scale to massive data sets and, thus, provide poor response times on such data sets. To address this problem we develop parallel and I/O-efficient algorithms for computing the skyline of large data sets, in this part of the thesis. Next, we define the skyline computation problem formally.

3.1.1 The Concept of Skyline Query

Given a collection S of database records, every record in S consists of a set of d attributes. For a numerical attribute, we may consider a record r "better" than another record r' with respect to this attribute if r's attribute value is less than or greater than that of r', depending on what this attribute represents. A *skyline query* retrieves all records that have the property that no other record is better according to all attributes.

More formally, the database records can be represented as a set S of points in d-dimensional space. For a point p and all $1 \leq i \leq d$, we denote its ith dimension by $x_i(p)$. Point p_1 is said to *dominate* point p_2 if $x_i(p_1) \leq x_i(p_2)$, for all $1 \leq i \leq d$, and at least one of these inequalities is strict. A point p is a *skyline point* if it is not dominated by any other point in S. The *skyline* of S, denoted $\text{sky}(S)$, is the collection of all its skyline points. A *skyline query* finds the skyline of the given set S with respect to a given subset of dimensions. Without loss of generality, we can assume that all point coordinates are non-negative, as we can move the origin appropriately to ensure this without changing the set of points that make up the skyline.

Consider for example a collection S of records representing hotels, and the attributes we consider are the *price* of a room and the hotel's *distance* to a conference we plan to attend. We are interested in booking a cheap room close to the conference. If we find that hotel A is both cheaper and closer to the conference than another hotel B, we would definitely choose the former over the latter; we say that hotel A *dominates* hotel B. Ideally, we would like the choice to be the *cheapest* hotel that is also the *closest* to the conference. However, there often exists a trade-off between the criteria we aim to optimize, with no single record being best according to all criteria. In such situations, we want to eliminate those hotels from consideration that are worse according to all criteria than some other; the surviving hotels form the *skyline* of S and are viable candidates to offer the user to choose from.

Figure 3.1 illustrates the previous example, where each point in two-dimensional space corresponds to a hotel record. The room price of a hotel is represented by its y-coordinate, and the x-coordinate specifies its distance to the conference. Hotels p_2, p_3, p_6, p_7, p_8 are dominated by hotel p_1, while hotels p_1, p_4, p_5 are not dominated by

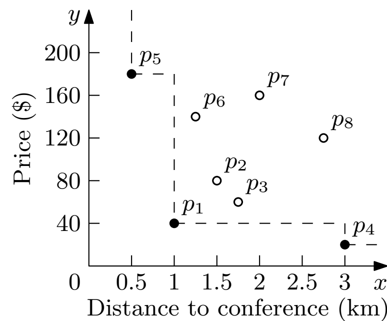


Figure 3.1: A sample dataset and its skyline.

any other hotels. Hence, the latter form the skyline of this collection and constitute the set of candidates we would consider for booking a room.

In the next two sections, we survey relevant work on skyline query algorithms. We divide the review into work in centralized sequential environments and work in distributed settings.

3.2 Skyline Queries in Centralized Environments

There has been a large body of work on skyline computation. The first to study this problem where computational geometers, as discussed for instance in [111]. The set of skyline points is also known as the Pareto frontier in the fields of economics [29, 74] and multi-objective optimization [67]. More recently, work in the database community discuss applications of skyline computations in different application domains, including multi-criteria decision making [39, 86, 109], spatial applications [120], wireless applications [47], mobile environments [80], etc.

A number of algorithms have been proposed in both, the computational geometry and database literature. These algorithms can be roughly divided into two categories. Algorithms in the first category compute the skyline directly without preprocessing; algorithms in the second category first construct an efficient indexing structure on the point set and then use this structure to answer skyline queries. Next, we review the most significant algorithms in both categories.

3.2.1 Algorithms Without Preprocessing

Divide and conquer. In computational geometry, the points in the skyline are called the *maximal elements* of the point set. Kung et al. [89] proposed an optimal algorithm based on divide-and-conquer that finds all maximal elements. Since we considered this algorithm as a candidate due to its good theoretical running time we give a more detailed account of this algorithm in Section 3.2.4.1.

Block-nested-loops. In the database community, Borzsonyi et al. [39] were the first to apply the maximal elements computation in the database context and introduced the skyline operator. In their work they proposed the Block-Nested-Loops (BNL) algorithm, which takes the straightforward approach of comparing every point with every other point in the point set and adding it to the skyline if it is not dominated by any other point. Specifically, BNL sequentially scans the input file containing the point set and keeps a window of skyline candidates in memory. Initially the first point is put into the window. Then each subsequent point p is compared to every candidate in the window. If p is dominated by any of them, it is eliminated and will not be visited again. If p dominates one or more candidates, it is inserted into the window and all those candidates it dominates are removed from the window. Otherwise, p is inserted into the window. Eventually the window may become full. In this case, the rest of the input file is processed differently. As before, if a new point is dominated by a window point, it is eliminated. However, if the new point is not dominated, it is written to a temporary—overflow—disk file (dominated window points are still being removed). The creation of the temporary file means another iteration will be needed to process the overflow points in the file. On a subsequent iteration, the previous temporary file is read as the input. Once a window point has gone through a full "cycle" of comparisons (i.e., it has been compared against all currently surviving points) it can be removed from the window and output as part of the skyline.

Sort-filter skyline. BNL's strategy incurs too many unnecessary comparisons between points that are not part of the skyline. This is because any skyline point has

to be compared against all the points in the window. To reduce the number of comparisons, Chomicki et al. [50] proposed the Sort-Filter Skyline (SFS) algorithm. SFS, in an initial phase, sorts the point set according to decreasing values of a monotone scoring function. This function guarantees that if point p dominates point q, then p has a higher score than q. Afterwards, SFS proceeds as BNL, except now, when a point is added to the window, it is known to be a skyline point. No point following it can dominate it—this is guaranteed by the monotonicity of the scoring function. Thus the point can be output as part of the skyline immediately. As in BNL, once the window gets full, surviving input points must be written to a temporary disk file. If a temporary file was created, another iteration is required. Unlike BNL, the window can be cleared at the beginning of each iteration, since all points have been compared against those skyline points. The temporary file is then used as the input for the next iteration. The preferred scoring function is by volume $\prod_{i=1}^{k} d_i(p)$ or entropy $\sum_{i=1}^{k} \ln d_i(p)$, where $d_i(p)$ refers to the value of point p in dimension i. Overall, the management of the window in SFS is simpler than in BNL and skyline points can be quickly output. Experiments in [50] show that is SFS much faster than BNL, and that it performs less dominance tests.

Linear elimination sort for skyline. As yet another form of BNL, Godfrey et al. [71] introduced the Linear Elimination Sort for Skyline (LESS) algorithm. LESS filters the points via a main-memory skyline-filter window, as does SFS and BNL. Just as SFS, LESS initially sorts the points using a scoring function. However, LESS incorporates two additions already in the sorting step: it integrates an elimination-filter window in the first pass of a standard external merge-sort routine to eliminate some dominated points quickly; and it combines the last merge pass of the sorting algorithm with the first skyline-filter pass.

Sort and limit skyline algorithm. Bartolini et al. [30, 31] also extended the work on BNL and SFS and argued that for suitably chosen scoring functions, it is indeed possible to compute the skyline by looking only at a subset of the sorted input point set. Based on this notion, they proposed the Sort and Limit Skyline algorithm (SaLSa). SaLSa relies on scoring functions that can guarantee that all data points beyond a certain point, the stop point, are dominated by such point.

This way, one can stop fetching points from the input point set, effectively limiting the number of points to be read. Two main factors determine the actual performance of the algorithm: the choice of the scoring function, which might severely influence the number of points to be read, and the strategy for choosing the stop point, which is used to earlier terminate reading points. In their work, they evaluate various scoring functions and according to their study pre-sorting by minimum-coordinate achieves the best performance for their strategy.

External memory skyline query algorithm. More recently, Gui et al. [75] presented a simple variant of BNL. This approach keeps the points in the main-memory window sorted by volume (product of their coordinates), in an attempt to reduce the cost of internal memory computation. The intuition is that the points with high possibility to dominate others are on top of the window, and will be compared first, thus reducing the number of comparisons. Nevertheless, since the window has to be kept ordered, the computational cost is still high, particularly for large skylines.

Bitmap. Tan et al. [124] worked on a different approach and presented a novel algorithm for computing skyline points, named Bitmap. The main goal was to output the first skyline points very quickly. To this end, the algorithm maps each point to a bit string, and the skyline is computed using efficient bit operations. Although this method can produce the first skyline points very quickly, it has a major drawback. If the number of distinct values is high for some dimensions, the bitmaps needed would consume huge amounts of memory. Thus, it is appropriate only when the number of possible values is small in each dimension.

3.2.2 Index-based Algorithms

The second group of skyline algorithms exploits special index structures. The main goal of index-based algorithms is to produce skyline points progressively—initial results should be output almost instantly and the output size should gradually increase.

Index. In the same paper where they introduced their bitmap approach, Tan et al. [124] also propose the Index algorithm. This algorithm partitions the d-dimensional point set into d lists, one for each dimension. A point p is assigned to the ith

($1 \leq i \leq d$) list if and only if its coordinate on the ith axis is the minimum of its coordinates. Each list is sorted in ascending order of the minimum coordinate (*minC*, for short) and indexed by a B$^+$-tree along that dimension. The lists are then scanned in a batched form; a batch in the ith list consists of points that have the same ith coordinate (i.e., minC). Initially, the algorithm loads the first batch of each list, and processes the one with the minimum minC. Processing a batch involves (a) computing the skyline of the points inside the batch, and (b) among the computed points, adding the ones not dominated by any skyline points already found, into the skyline list. After processing a batch, the algorithm then chooses from the unprocessed batches with the minimal minC. The algorithm terminates when the current minC is larger than the maximal coordinate of the previously found skyline point. This technique can return skyline points quickly at the top of the lists. However, the lists built for a query with d dimensions cannot be used to solve queries on only a subset of these dimensions.

Nearest-neighbor. Kossmann et al. [86] introduced the Nearest-Neighbor (NN) algorithm. In NN, the point set is indexed by an R-tree. A distance function of the points to the origin of the coordinate system (e.g., L_1-distance norm) is also provided to the algorithm. NN iteratively computes a nearest neighbour point to the origin in a given data space region; the idea is to enforce the computed nearest neighbour points to be part of skyline. The algorithm starts by performing a nearest-neighbor search on the R-tree, to find the point with the minimum distance from the origin; it is shown in [86] that the first nearest neighbor is part of the skyline. The resulting nearest-neighbor point also prescribes a region within which all points are dominated by it; the region is pruned from further consideration. The rest of the data set is partitioned based on the nearest-neighbor point, and the different parts are inserted into a to-do list for subsequent processing. Then NN continues by removing a part from the to-do list and processing it recursively, until the list is empty. The experiments in [86] show that NN outperforms previous skyline algorithms in terms of overall performance and general applicability independently of the characteristics of the point set. However, NN has some shortcomings, such as need for duplicate elimination—different parts may overlap and the overlapping region might cause duplicates in the skyline result—,

multiple R-tree node visits, and huge space overhead.

Branch-and-bound skyline. Motivated by the shortcomings of NN, Papadias et al. proposed the Branch-and-Bound skyline (BBS) algorithm in [108, 109]. BBS, like NN, is based on the nearest-neighbor search technique on the data points, but (unlike NN) visits an R-tree node only if its subtree contributes a skyline point. In Section 3.2.4.2 we provide a detailed explanation of this algorithm.

ZSearch. Recently, Lee et al. [91] utilized the close relationship between the Z-order space filling curve and skyline processing to index data points and efficiently answer skyline queries. This strategy, named ZSearch, first encodes the points in Z-order—points are assigned Z-addresses—and then constructs a novel variant of the B^+-tree called ZBtree. With the points organized in a ZBtree, the skyline search is conducted over the index. Based on Z-addresses, the Z-order curve provides two important properties: (a) monotone order (dominating points are always accessed before their dominated points and, (b) clustering (points ordered by Z-addresses are naturally clustered as regions). These properties facilitate efficient dominance comparisons and space pruning. In their paper they also introduce a suite of algorithms which incrementally maintain skyline results in the presence of changes (insert, delete and update).

3.2.3 Extensions of Skyline Computations

There has also been much work devoted to extending research in skyline computation to new problems and domains. For example, Chan et al. [45] and Yuan et al. [136] extended the computation of skylines in different directions. The former paper proposed a metric that ranks the points based on how often they are part of the skyline in different subspaces; if the skyline is too big, which is often the case in higher-dimensional space, only the k highest-ranked points are reported. This is reasonable, since a very large skyline provides little more (possibly less) information than a representative sample of skyline points. The paper by Yuan et al. [136] focused on answering skyline queries over the whole data cube defined by a given subset of dimensions.

As yet another variant of the traditional skyline query, Sharifzadeh et al. [120] studied the Spatial Skyline Query (SSQ) processing problem. An SSQ, consisting of multiple query points, retrieves data points that are not farther than any other data points, from all query points. More specifically, given a set of data points P and a set of query points Q in a d-dimensional space, an SSQ retrieves those points of P which are not dominated by any other point in P considering a set of spatial derived attributes; for each data point, these attributes are its distances to query points in Q. This computation is relevant for applications such as geographic information systems where the managed data is inherently spatial in nature. [120] provides algorithms for computing SSQs. These methods use the Voronoi diagram, convex hull and Delaunay graph of the data points to find the skyline. The main intuition behind their algorithms is in exploiting the geometric properties of the SSQ problem space to avoid the full examination of all the point pairs in P and Q.

3.2.4 Detailed Discussion of Skyline Algorithms

The skyline algorithms we propose use the divide-and-conquer algorithm by Kung et al. [89] and the branch-and-bound algorithm by Papadias et al. [108] as building blocks. Therefore, we discuss these two algorithms in more detail in the next sections.

3.2.4.1 Divide and Conquer (DC)

The divide-and-conquer algorithm of [89] performs a double recursion on the number of dimensions and the size of the point set. Given a d-dimensional point set S, the first step is to find the median coordinate $\text{med}(d)$ in the dth dimension. The point set S is then divided into two sets L and R around this coordinate, and the skylines of L and R are found recursively. The points in $\text{sky}(L)$ are easily seen to belong to $\text{sky}(S)$, while a point in $\text{sky}(R)$ belongs to $\text{sky}(S)$ if and only if it is not dominated by a point in $\text{sky}(L)$. Since every point in R has a greater x_d-coordinate than every point in L, a point in $\text{sky}(R)$ is dominated by a point in $\text{sky}(L)$ in all d dimensions if and only if this is the case in the first $d-1$ dimensions. This is the basis for the following recursive filtering procedure that removes all points in $\text{sky}(R)$ dominated by points in $\text{sky}(L)$; see Figure 3.2.

First, the median $\text{med}(d-1)$ of the x_{d-1}-coordinates of all points in L is found,

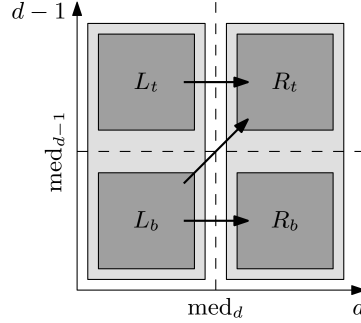


Figure 3.2: Recursive filtering in DC.

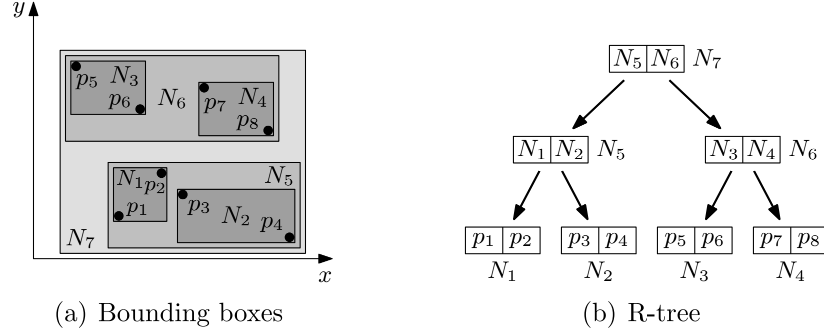
and L and R are partitioned around med$(d-1)$ into sets L_b and L_t, and R_b and R_t, respectively. (Note that the split of R is not necessarily even.) It can be observed that no point in R_b can be dominated by a point in L_t. Hence, it suffices to recursively filter all points in R_b against the points in L_b, and all points in R_t against the points in L_b and L_t. For the filtering of the points in R_t against the points in L_b, it can be observed that every point in R_t has greater x_{d-1}- *and* x_d-coordinates than every point in L_b. Hence, this filtering step has to take only the first $d-2$ dimensions into account. The recursion in this procedure stops as soon as one of the two involved point sets has size at most one or the number of relevant dimensions has reduced to two, at which point the computation can be finished in linear time without recursing any further.

DC has a worst-case running time of $O(N \log^{d-2} N)$ and takes expected linear time on a random point set, where N is the the number of points in S. In practice, the query time can be very high, as shown in Figure 3.4a.

3.2.4.2 Branch-and-Bound Skyline (BBS)

BBS [108] is an index-based algorithm that finds the skyline points using an R-Tree index [35, 76]. An R-tree is a version of a B-tree designed to index geometric objects in two and higher dimensions. The objects are stored in the leaves of the tree; every internal node stores the smallest axis-parallel box that contains all objects in the leaves that are descendants of this node. This box is called the *minimum bounding box* (MBB) of the node. An MBB is represented by the two endpoints of its main diagonal. For $1 \le i \le d$, let $x_i^l(v)$ denote the minimum of the ith coordinates of all points in v's subtree, and let $x_i^u(v)$ denote their maximum. Then, the MBB is represented

by the points $l(v) = (x^l_1(v), x^l_2(v), \ldots, x^l_d(v))$ and $u(v) = (x^u_1(v), x^u_2(v), \ldots, x^u_d(v))$. Throughout the remainder of this part of the thesis, nodes and their bounding boxes will be considered one and the same; it is also assumed that every leaf of the R-tree stores only one object. Note that, while the MBBs along any root-leaf path are properly nested, the MBBs of sibling nodes are not necessarily disjoint. This is true even if the stored objects are points, unless the subsets of points to be stored at the leaves are carefully chosen using a recursive space partition. One way to obtain such a partition is by splitting the dimensions in a round-robin fashion. The tree obtained using this procedure, when applied to the point set in Figure 3.1, is shown in Figure 3.3 and is similar to a k-d-tree [36] for this point set.



(a) Bounding boxes (b) R-tree

Figure 3.3: The R-tree for the point set from Figure 3.1.

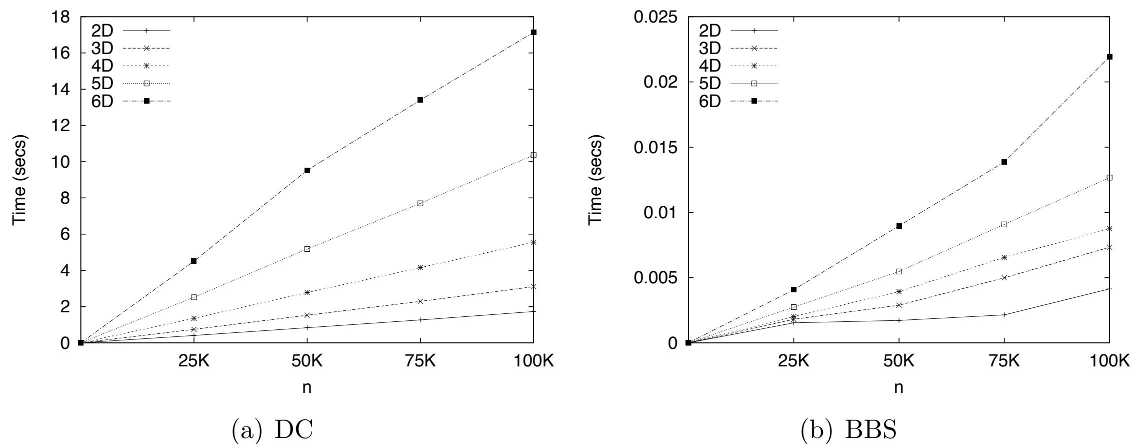
BBS incrementally constructs a list, L, of skyline points it has identified; initially, $L = \emptyset$. The skyline points are found using a traversal of the R-tree. When a node v is inspected, all points in L are checked to decide whether one of them dominates (the bottom-left corner[1]) of v. If so, no point in v's subtree can be a skyline point, and the children of v do not need to be visited; the subtree below v is *pruned*. Otherwise, if v is a leaf, the point stored at v is added to L; if v is not a leaf, its children are visited recursively.

In order to avoid adding a non-skyline point to L (and in order to prune dominated subtrees as quickly as possible), it is necessary to ensure that all skyline points that can dominate a node or point p are added to L before inspecting p. This is achieved by inspecting all non-pruned nodes in order of increasing distance (of their bottom-left corners) from the origin. (Recall that all points have positive coordinates.) To

[1]This terminology borrowed from two dimensions is used throughout the thesis to denote the corner closest to the origin in any number of dimensions.

implement this, a priority queue Q is used. Initially, Q holds only the root of the tree, and its priority is equal to its distance from the origin. Then, while Q is non-empty, the node v with minimum priority is retrieved. If v is dominated by a point in L, the processing of v is finished. Otherwise, if v is a leaf, the point stored at v is added to L; if v is not a leaf, each of its children is tested whether it is dominated by a point in L. Each child that is not dominated is added to Q, with priority equal to its distance from the origin.

Papadias et al. [108] showed that BBS accesses every node at most once and visits a node only if its subtree contains at least one skyline point. They also presented experimental results that demonstrate that the performance of BBS mainly depends on the size of the computed skyline and the dimensionality of the data. This last limitation is inherited from the used index structure, the R-tree. The performance of R-trees does not scale well with the number of dimensions; in practice, however, only a moderate number of dimensions are really relevant [86]. For 2–6 dimensions, the pruning strategy of BBS is highly effective because in this case usually only a small percentage of the given points are part of the skyline. Figure 3.4b shows query times (not including construction of the R-tree) from sequential experiments with different data set sizes.



(a) DC (b) BBS

Figure 3.4: Sequential query times for DC and BBS. Note that the query time for BBS does not include the time to construct the R-tree.

3.3 Skyline Queries in Distributed Environments

Most of the previous work on skyline queries has focused on the development of efficient centralized sequential algorithms that work with data sets that fit in internal memory. Since, however, the data sets to be processed in real-world applications are often of considerable size, there is a need for improved query performance through the use of algorithms that distribute the computation over multiple processing units. For example, in this thesis, we speed up skyline computation by using a shared-nothing multi-processor parallel machine.

One of the earliest work on parallel skyline algorithms is the one by Dehne et al. [57], which proposed an optimal coarse-grained parallel algorithm for computing skylines in three dimensions; however, their approach does not seem to lead to practical algorithms for higher-dimensional point sets.

More recently, there has been a growing interest in parallel skyline computations [54, 69, 110, 128] and distributed skyline computations [133–135]. The primary distinction within parallel and distributed skyline computation is how tightly-coupled the individual processing nodes are. The nodes of a distributed system are physically distributed, i.e., loosely coupled; Subsection 3.3.1 reviews some of the most representative work on skyline computation in this environment. Parallel systems, on the other hand, are tightly coupled—typically in the same box; Subsection 3.3.2 presents some relevant work in this setting.

3.3.1 Skyline Computation in Peer-to-Peer Networks

Distributed skyline algorithm. One of the earlier works on distributed skyline computation is the one by Wu et al. [135]. They proposed the Distributed Skyline (DSL) query algorithm that works on a peer-to-peer (P2P) Content Addressable Network (CAN) [113]. DSL relies on space division techniques to create grid partitions, which are then assigned to the participating servers. Thus, each server is responsible for a disjoint partition. An advantage of the grid partitioning technique is that some partitions may not need to be examined when the query is executed, this way avoiding accessing servers not containing skyline points. DSL focuses on computing constrained skyline queries that are posed within a query range, and consequently

its skyline search method has been designed for such queries. In contrast to our proposed approach, where all servers process the skyline query simultaneously, in DSL each server starts the computation on its local data only after receiving from all servers whose points can dominate its points, their previously computed results. Overall, the focus of this distributed approach is on minimizing the communication cost. However, processors being idle lead to poor balancing. The sequential dependencies of one processor on other processors' results also leads to poor parallelism.

Skyline search space partitioning. Wang et al. [133] proposed the Skyline Search Space Partitioning (SSP) algorithm, which is based on a P2P BATON [81] overlay network. SSP employs the z-curve method for mapping the multidimensional data space to one-dimensional values, that can then be assigned to peers. In this approach, however, a load balancing problem arises. The data access of a skyline query is likely to be skewed towards the part of the space that contains the skyline. Hence, a small number of peers might have to process almost every query since their allocated space is close to or contains part of the skyline. To improve the algorithm's performance, the authors propose some extensions, for instance, allocating relatively smaller space to peers responsible for these regions (by basically partitioning the space and dividing a hyper-rectangular region into smaller, equally loaded, regions). Nevertheless, many servers that do not contribute to the skyline result set are still queried. In [134] the authors extended the previous work by proposing a framework that can be adapted to other P2P network systems.

Both studies above are based on P2P environments and focus on reducing communication by avoiding some nodes to be examined when a query is executed. They do this by using space partitioning techniques. However, in the case of skyline computation in parallel, where all partitions are examined simultaneously, the performance of these techniques degrades. The reason is that many partitions do not contribute to the global skyline result (or their contribution is negligible), resulting in poor load balancing among computational nodes and thus low parallelism.

3.3.2 Skyline Computation in Parallel

Parallel distributed skyline. There has been follow-up work after the parallel algorithm presented in Chapter 4 was published in [54]. Cui et al. [56] proposed the PaDSkyline (Parallel Distributed Skyline) algorithm for computing constrained skyline queries (a constraint on a dimension is a range specifying the user's interest). PaDSkyline works in an environment consisting of different servers which are located at geographically scattered sites and connected via internet. Despite working in a distributed setting the algorithm exhibits some form of parallelism. PaDSkyline partitions all relevant sites into incomparable groups such that the skyline computation in any one group does not depend or affect the computation in any other group. Hence, the skyline computation can be executed simultaneously—in parallel—among all those site groups. On the other hand, within each group, the skyline computation is executed sequentially among the sites; this intra-group computation involves computing the skyline of each site's local data, propagating intermediate results between sites and designating a group coordinator responsible for merging the local results. In addition, within each group, the algorithm optimizes the skyline computation by selecting multiple filtering points based on their overall dominating potential from the skyline of the local data. These points are then sent to other sites, where they help identify more dominated points. Similarly to the peer-to-peer approaches, the strength of this algorithm is that it minimizes the network traffic, in this case by early filtering of dominated points. Nevertheless, load balance is still an issue since, for instance, sites having overlapping partitions are not processed in parallel.

Angle-based space partitioning. Other approaches mainly focus on ways to efficiently partition the input data. For instance, Vlachou et al. [132] proposed a parallel skyline algorithm which uses the hyperspherical coordinates of the points to partition the data. The algorithm initially distributes the points to individual nodes, where a local skyline query on the assigned workload is executed, followed by the computation of the final skyline from the local skyline sets by a coordinator node. The key idea in this strategy is the computation of smaller local skylines, as this reduces the amount of data to be processed by the coordinator. To this end, the data set is initially partitioned using the hyperspherical coordinates of the points; the intuition behind this

approach is that the skyline points are equally spread to all partitions. More specifically, the algorithm maps the d dimensions into hyperspherical coordinates. Then it divides the data points into N partitions (N being the number of available nodes) using angular coordinates. As a result, all points having similar angular coordinates fall in the same partition. By doing this, the average pruning power of points within a partition is increased, thus reducing per-node processing times. Despite the advantage of computing smaller local skylines, the scheme has some limitations. Firstly, no provisions are made for the case when the total size of the local skylines is too big to be efficiently merged by the coordinator node (this is a risk of a bottleneck). Secondly, the reduction of the local skyline sizes is achieved at a more costly way of transforming and distributing the data points, all of which is not reported.

Adaptive distributed skyline computation algorithm. More recently, Valkanas et al. [128] introduced the Adaptive Distributed Skyline Computation (ADISC) algorithm for shared-nothing architectures. ADISC runs both in parallel and cascading (sequential) mode. Initially, the coordinator processor partitions the data set using a grid scheme and assigns a partition to each processor. Then each processor indexes its local copy with an R-tree and reports back arbitrary representative points. Partitions dominated by representatives are excluded from further consideration (they do not contribute to the final skyline). The computation proceeds with the non-dominated partitions. If in cascading mode, the coordinator determines the sequence in which the data propagates among the processors and informs them appropriately; for example, if processor A contains data that processor B can use for pruning away points in its assigned partition, A sends its data to B. Each processor begins computations only after receiving data from all its predecessors in the sequence. In full parallel mode, processors compute the skyline of their own partition simultaneously, and then they send their local results to the coordinator; the coordinator merges the results and computes the global skyline. The algorithm also incorporates a set of optimizations. For example, it exploits the representative points even further to: a) improve parallelism while maintaining progressiveness and, b) remove dependencies between partitions which, as a side effect, reduces network traffic and minimizes the workload of the coordinator. According to [128], this strategy is scalable in terms

of processors and data set size, and is able to handle diverse preferences (e.g., min, max) imposed on attributes.

Skyline computation on other parallel architectures. The skyline computation has also been extended to other parallel architectures, including multi-core [110] and multi-disk environments [69].

Park et al. [110] introduced an algorithm named PSkyline for multi-core architectures. These architectures combine multiple independent cores sharing the computer's internal memory; multi-core architectures offer an added advantage of negligible or low overhead for communications between parallel threads. The intuition behind PSkyline is that the cost of the skyline computation depends heavily on the number of comparisons between points, called dominance tests, which involve only integer or floating-point number comparisons. Since a large number of dominance tests can be performed independently, skyline computation has a good potential to exploit these architectures. PSkyline is based on the divide-and-conquer strategy. Specifically, the algorithm divides the point set into multiple subsets, distributes the subsets to threads and computes the local skyline of each thread separately. Then, it merges local skylines into a global skyline in parallel. PSkyline optimizes thread utilization for both the local computation and the merging process. PSkyline, among the points assigned to each thread, prunes out some of non-skyline points that cannot be contained in the final skyline. In addition, it aggregates the local results, and returns the final skyline by maximizing thread-level parallelism.

Gao et al. [69] proposed an algorithm for a multi-disk architecture. This architecture consists of a single-processor machine with several disks attached to it. In this approach, the point set is distributed over multiple disks; parallel R-trees [83] are used to index the points. Similarly to BBS, the skyline computation also follows the best-first nearest neighbor search paradigm. The main idea is to exploit sufficient parallelism by visiting all relevant R-tree nodes, in multiple disks, simultaneously during the traversal. Additionally, [69] provides several dominance-based pruning optimizations to discard entries—MBBs of R-tree nodes—that are dominated by other points (or other MBBs) from the computation.

Despite the similarities of these two parallel algorithms with our own work, in

that they attempt to use multiple processing units, our work mainly differs from them in that we assume a shared-nothing architecture in contrast with the shared-memory and shared-disk architecture respectively. The problem with using these strategies in shared-nothing environment is, that they employ a very fine-grained parallelism—data is transferred very frequently among processors after small amount of computation—, which is not suitable for shared-nothing systems. For example, if the techniques used in these algorithms were translated to a shared-nothing setting (by replacing the updates done in main memory with messages passed around by the processors), then the inter-processor communication would outweigh the benefit of parallel computation. This observation has been already noted by the authors in [110]. Shared-nothing algorithms are (almost) always coarse-grained—data is communicated infrequently after large amounts of computation.

In the next section we present the details of the parallel architecture we consider.

3.4 Shared-Nothing Clusters

In recent years, there has been a trend in parallel computing to move away from expensive specialized supercomputers to cheaper general purpose clusters made from single or multi-processor personal computers (PC) or workstations. The nodes in a cluster may share memory or disk arrays. If there is neither memory nor disks shared among nodes in clusters, they are called *shared-nothing clusters*, such as the popular, low-cost, Beowulf-style clusters [1] consisting of standard PCs connected via a data switch without any expensive shared disk array. Communication between the nodes of a shared-nothing cluster happens through explicit exchange of messages. Clusters usually host an open-source Unix-like operating system, such as BSD, GNU/Linux, or Solaris. Figure 3.5 shows a shared-nothing cluster computer architecture.

The cluster's middleware consists of tools or interfaces which hide the various hardware and networks (e.g., Ethernet, Myrinet, InfiniBand) from the applications and provide a standard way to share data among nodes. One common piece of middleware is the Message Passing Interface (MPI) [6], which is a standard interface for message passing implementations among processors in a cluster. In a shared-nothing architecture, MPI sends messages by using standard network protocols, such as TCP/IP. MPI is the most widely used message passing framework for parallel

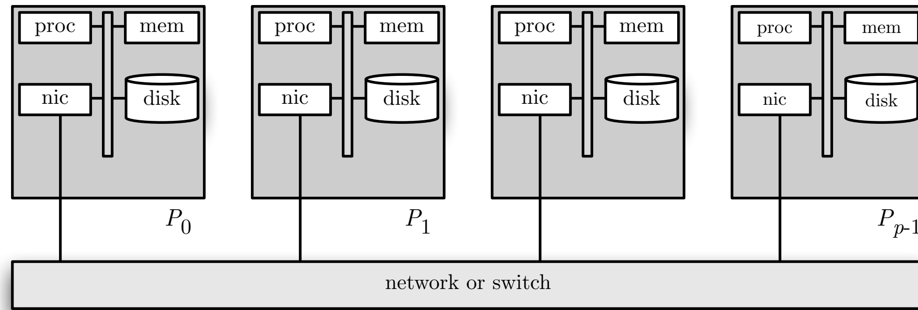


Figure 3.5: A Shared-nothing cluster.

computing across the industry.

3.5 Contributions

Large data sets are becoming ubiquitous; thus, parallelizing the skyline computation and eliminating the I/O bottleneck in large-scale computations is increasingly important in order to retrieve the results in a reasonable amount of time. The bulk of existing skyline algorithms are efficient in a sequential setting, using a single processor, and are not able to efficiently process data sets beyond the capacity of main memory. To approach this problem, a number of algorithms that distribute the computation among multiple computational nodes have been proposed, as discussed in Section 3.3. Our algorithm is one of the earliest to tackle the problem on a shared-nothing parallel machine. There has been follow-up work to our approach, which we also discussed in Section 3.3. However, some of the proposed algorithms work on different architectures than ours and their techniques, if implemented in a parallel setting, lead to workload unbalance and/or excessive inter-processor communication. Other parallel methods focus mainly on partitioning the point set to produce smaller local results, but ignore the cost of computing these partitions.

Our contribution consists of *parallel* and *I/O-efficient* algorithms for computing skylines. For the parallel algorithm, we determined through experimentation the speed-up obtained on a distributed-memory cluster. The results we obtained show that our parallel algorithm can be used effectively to speed up the computation of skylines of large data sets. We also investigated the effect of increasing data dimensionality and input size on the performance of the algorithm. Accordingly, the

algorithm shows good scalability with increasing data set sizes and number of processors. The parallel algorithm (as well as the I/O-efficient algorithms) is based on the branch-and-bound skyline algorithm by Papadias et al. [108]. We investigated the challenges posed by the original algorithm and introduced new ideas aiming to eliminate its I/O bottleneck. The external-memory algorithms can handle data sets beyond the main memory size and can potentially replace the sequential algorithm running on each processor in the parallel approach. We introduce our parallel skyline algorithm in Chapter 4, and the I/O-efficient algorithms in Chapter 5.

Chapter 4

Parallel Skyline Computation

In this chapter, we describe in detail the design and evaluation of a parallel algorithm for skyline computation. Our approach is to distribute the point set to individual processors, where efficient sequential algorithms can be used to independently calculate the skylines of the workload assigned to the processors, followed by a parallel construction of the final skyline from the local skyline sets. We also discuss engineering aspects of the algorithm's implementation, in particular a pointerless R-tree representation that can be easily exchanged between processors. Finally, we present our experimental evaluation on synthetic and real-world data sets. The results we obtained demonstrate that parallel computing can be used effectively to speed up the computation of skylines of large data sets of a moderate number of dimensions, which is what is relevant in practice [86].

4.1 Overview

The computational environment targeted by our algorithm is a shared-nothing parallel architecture consisting of p independent processors $P_0, P_1, \ldots, P_{p-1}$ as discussed in Section 3.4. The algorithm takes as input the point set S and a parameter $k \leq n/p$, and needs to output the entire skyline if $|\text{sky}(S)| \leq k$, or a sample $L \subseteq \text{sky}(S)$ of k skyline points if $|\text{sky}(S)| > k$. The assumption that no more than n/p skyline points are to be reported is reasonable, as p is usually small compared to n, and any skyline or skyline sample of size n/p or bigger can normally be considered to be excessively large to be useful in practice. Moreover, the only place where the algorithm relies on this assumption is in the final step when the entire skyline or skyline sample is collected on processor P_0 to be output.

The algorithm can be outlined as follows: (1) The point set is partitioned into subsets of equal size, each of which is assigned to a different processor. (2) Each processor computes the skyline of its assigned set of points. (3) The processors

collectively filter the remaining points against each other's local skylines, eliminating the points in each processor's subset that are dominated by points assigned to other processors. This collective filtering step can be implemented using a constant number of global communication rounds in all but extremely rare scenarios. For these rare cases, the algorithm includes an alternate version of the collective filtering step that requires between 1 and p communication rounds, depending on the total size of the local skylines and the size of the requested skyline sample.

The discussion of the algorithm is divided into several parts. Section 4.2 starts by giving an overview of the algorithm. Subsequent sections provide further details of it. In particular, Section 4.2.1 discusses the impact of the initial distribution of points among the processors on the performance of the algorithm and argues that a random distribution of the points over the processors is likely to yield good performance (which is confirmed by the experimental results discussed in Section 4.5). Some follow-up work [132] looked at this data distribution problem and proposed a different distribution that leads to smaller local skylines and, thus, has the potential to speed up the collective filtering step; however, that work did not investigate the computational overhead of producing this distribution. Thus, overall is not clear whether their approach actually leads to better performance. Sections 4.2.2 and 4.2.3 explore the two main phases of our algorithm. Section 4.3 discusses a pointerless R-tree representation that allows the construction of an R-tree over a point set to be parallelized. Section 4.4 presents details of the algorithm's implementation. Finally, Section 4.5 examines the results obtained with the algorithm's experimental evaluation.

4.2 Parallel Skyline Algorithm

This section describes our skyline algorithm (see Algorithm 1), referred to throughout the rest of the chapter as Parallel-Skyline. The algorithm exploits that, for any decomposition of the input point set S into subsets $S_0, S_1, \ldots, S_{p-1}$, we have $\text{sky}(S) = \text{sky}(\text{sky}(S_0) \cup \text{sky}(S_1) \cup \cdots \cup \text{sky}(S_{p-1}))$ (see Figure 4.1). Thus, we can compute the skyline of S in two phases. In the first phase, we assign n/p points to each processor, and all processors independently compute the skylines of their local point sets. In the second phase, the processors collectively compute the skyline of the set of these local skyline points. The size of this set is usually substantially reduced

Algorithm 1: PARALLEL-SKYLINE($S_0, S_1, \ldots, S_{p-1}, k$)

Input: The point set $S = S_0 \cup S_1 \cup \ldots \cup S_{p-1}$; set S_i is stored on processor P_i; k is the desired number of skyline points.
Output: A set L of $\min(k, |\text{sky}(S)|)$ skyline points.

// 1st phase: Data partitioning and local computation

1 **for all** $0 \leq i \leq p-1$ **do in parallel**
2 P_i builds an R-tree $R(S_i)$ for S_i;
3 P_i runs BBS on $R(S_i)$ to compute $\text{sky}(S_i)$. Let $S'_i = \text{sky}(S_i)$;

// 2nd phase: Computation of the final skyline

4 Compute the set $L \subseteq \text{sky}(S)$ and collect it in processor P_0 by calling GLOBAL-SKYLINE($S'_0, S'_1, \ldots, S'_{p-1}, k$);

5 **return** a set $L \subseteq \text{sky}(S)$ of size $\min(k, |\text{sky}(S)|)$;

compared to the original input set S.

While the first phase as such is straightforward (Section 4.2.2), the distribution of the points among the processors can have a tremendous impact on the performance of the second phase, as we discuss in Section 4.2.1. In particular, it is desirable to ensure that the total size of the local skylines is as small as possible, as this reduces the amount of data to be processed in the second phase. In fact, the second phase of the algorithm differs depending on whether the union, S', of the local skylines fits in the memory of a single processor; see Algorithm 2.

If $|S'| \leq n/p$, each processor P_i receives a copy of S' and then eliminates all points in $\text{sky}(S_i)$ that are dominated by points in S'. The points in $\text{sky}(S_i)$ that are not eliminated are the points in S_i that belong to $\text{sky}(S)$. Since copying S' to all processors requires one global communication round, and filtering the points in $\text{sky}(S_i)$ requires only local computation on processor P_i, computing $\text{sky}(S)$ from the local skylines of all processors takes one communication round in this case.

If $|S'| > n/p$, no processor has sufficient memory to store S'. In this case, a collective filtering approach is used that may take up to $\lceil p|S'|/n \rceil$ communication rounds, each of which determines for a subset of each processor's local skyline points whether they belong to $\text{sky}(S)$. These two variants of the second phase are discussed in more detail in Section 4.2.3.

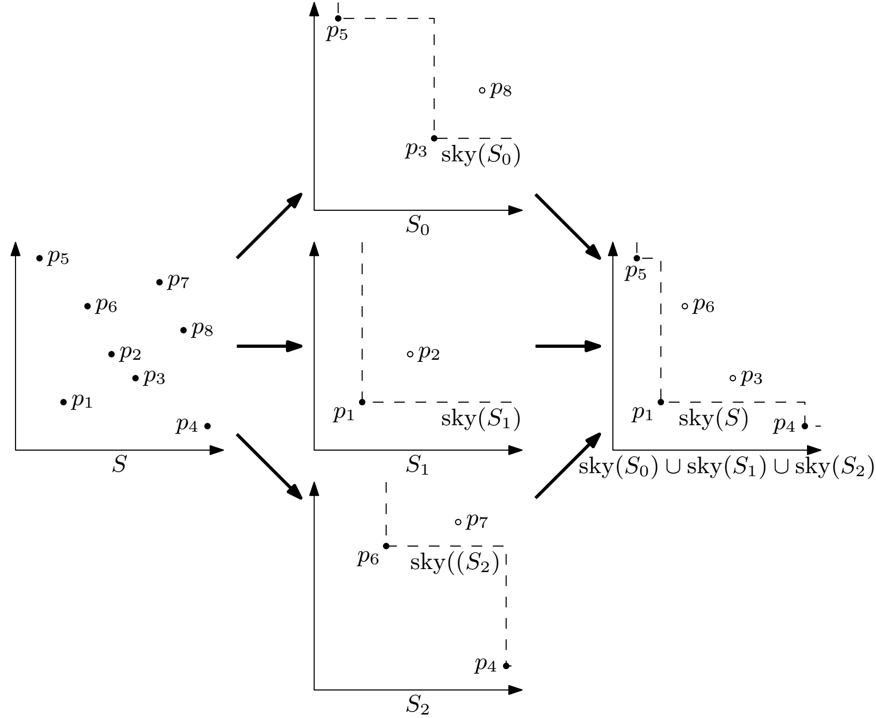
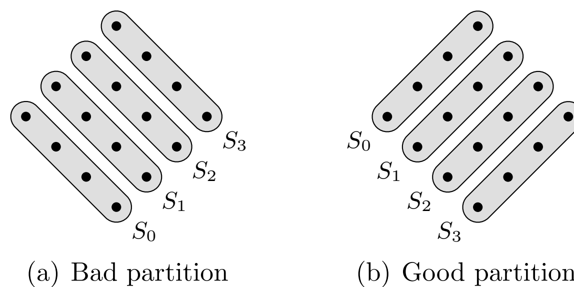


Figure 4.1: Illustration of the two phases of Algorithm 1 for three processors. First the point set is distributed over the processors. Then each processor P_i computes the skyline of its local point set S_i. Finally, $\text{sky}(S)$ is computed as the skyline of $\text{sky}(S_0) \cup \text{sky}(S_1) \cup \text{sky}(S_2)$.

4.2.1 Data Distribution

One aspect affecting the efficiency of Parallel-Skyline is the partitioning method used for distributing the input data set over the processors. More specifically, since the algorithm relies on the union of the local skylines being substantially smaller than the original point set, the assignment of points to the processors can have a tremendous impact on its performance. Consider, for example, the point set in Figure 4.2. If the points are assigned to processors as in Figure 4.2(a), the union of the local skylines is the whole point set. If, on the other hand, every processor receives one of the bottom-left four points, as shown in Figure 4.2(b), the union of the local skylines is the final skyline.

The method used to partition the point sets should ensure that every processor does about the same amount of work in the first phase of the algorithm and that this phase eliminates as many non-skyline points as possible, in order to reduce the cost of the second phase. The cost of the local computation on each processor depends on



(a) Bad partition (b) Good partition

Figure 4.2: Two different partitions of the same point set over 4 processors. In Figure (a), the union of the local skylines is the entire point set; in Figure (b), it is the final skyline.

the number of points in S_i and, for algorithms such as the branch-and-bound skyline algorithm, on the size of the computed skyline. Thus, the partition should ensure that the sets $S_0, S_1, \ldots, S_{p-1}$ are of the same size, which is trivial, and that their skylines are of about the same size. In order to maximize the number of non-skyline points eliminated in the first phase, it is beneficial if the point set assigned to each processor is a representative sample of the whole input.

A simple and effective approach that achieves all these goals is to (uniformly) randomly distribute the points over the processors, assigning n/p points to each processor. By partitioning this way, it is likely that S_i follows the same distribution and is representative of S. Since each subset holds a random sample of S, they are expected to produce approximately the same number of skyline points and a similar percentage in each of them is expected to belong to the final skyline. This partitioning scheme is easy to implement and does not add much overhead. Hash partitioning, which uses a hash function based on some subset of a point's coordinates to map the point to one of the processors, is another alternative that is likely to produce good results for an appropriately chosen hash function. Furthermore, as our framework is not restrictive in the use of a particular distribution method, alternative techniques proposed elsewhere [77, 132] can be potentially applied.

4.2.2 Local Skyline Computation

Our algorithm in principle allows the use of any sequential skyline method to perform the local computation on S_i. However, in practice, the choice of the sequential algorithm has a substantial impact on the algorithm's performance. We confirmed this, by

experimenting with two sequential algorithms representative of the two categories of skyline methods (see Section 3.2). These are the *Divide-and-Conquer* (DC) algorithm of Kung et al. [89] and the *Branch-and-Bound Skyline* (BBS) algorithm of Papadias et al. [108]. DC is a sequential algorithm that uses the divide-and-conquer paradigm to compute the skyline without preprocessing. BBS is an index-based algorithm that finds the skyline using an R-tree index [35, 76].

In theory, BBS has a worst-case running time of $O(N^2)$, whereas DC takes only $O(N \log^{d-2} N)$ time, where N is the the number of points in S. In practice, however, the query time of DC can be very high. According to [108], the performance of BBS mainly depends on the size of the computed skyline and the dimensionality of the point set. This last limitation is inherited from the used index structure, the R-tree. The performance of R-trees does not scale well with the number of dimensions; in practice, however, only a moderate number of dimensions are really relevant [86]. For 2–6 dimensions, BBS is highly effective because, in this case, usually only a small percentage of the given points are part of the skyline. Given its good practical performance, our parallel algorithm uses BBS for the computation of the local skylines.

4.2.3 Global Skyline Computation

Given the local skylines $\text{sky}(S_0), \text{sky}(S_1), \ldots, \text{sky}(S_{p-1})$ computed in the first phase of the algorithm, the second phase computes $\text{sky}(S)$ from these local skylines. This is done using Algorithm 2 (referred to as Global-Skyline), which handles the common case when $S' = \text{sky}(S_0) \cup \text{sky}(S_1) \cup \cdots \cup \text{sky}(S_{p-1})$ contains at most n/p points differently from the rare case when $|S'| > n/p$. The next two subsections discuss these two cases in detail.

4.2.3.1 Few Local Skyline Points

If $|S'| \leq n/p$, Global-Skyline stores a copy of S' on each processor P_i, and processor P_i then eliminates all points from $\text{sky}(S_i)$ that are dominated by points in S'. The remaining points in $\text{sky}(S_i)$ are the ones that belong to $\text{sky}(S)$, while all removed points do not.

To perform this filtering step, processor P_i uses an R-tree T over the points in S'.

Algorithm 2: GLOBAL-SKYLINE($\text{sky}(S_0), \text{sky}(S_1), \ldots, \text{sky}(S_{p-1}), k$)

Input: Local skylines $\text{sky}(S_0), \text{sky}(S_1), \ldots, \text{sky}(S_{p-1})$; set $\text{sky}(S_i)$ is stored on processor P_i; k is the desired number of skyline points.
Output: A set of $\min(k, |\text{sky}(S)|)$ skyline points.

1 Let $S'_i = \text{sky}(S_i)$ and $l_i = |S'_i|$;
2 **for all** $0 \le i \le p-1$ **do in parallel**
3 $\quad$ P_i builds an R-tree $R(S'_i)$ for S'_i;
4 Compute $l = \sum_{i=0}^{p-1} l_i$ using an all-to-all communication;
5 **if** $l < \frac{n}{p}$ **then**
6 $\quad$ Use an all-to-all communication to combine trees $R(S'_0), R(S'_1), \ldots, R(S'_{p-1})$ into one tree T and store it on each processor;
7 $\quad$ **for all** $0 \le i \le p-1$ **do in parallel**
8 $\quad\quad$ P_i uses T to discard all points in S'_i that are dominated by a point in $S' = S'_0 \cup S'_1 \cup \cdots \cup S'_{p-1}$; let g_i be the number of points remaining in S'_i;
9 **else**
10 $\quad$ Let $g = 0$ be the current size of $\text{sky}(S)$;
11 $\quad$ Let $g_i = 0$ be the number of points contributed by P_i to $\text{sky}(S)$;
12 $\quad$ **while** $g < k$ and not all sets S'_i are empty **do**
13 $\quad\quad$ **for all** $0 \le i \le p-1$ **do in parallel**
14 $\quad\quad\quad$ P_i selects (any selection will do) and removes a set S''_i of $(l_i/l) \cdot (n/p)$ points from S'_i;
15 $\quad\quad$ Use an all-to-all communication to construct the set $S'' = \bigcup_{i=0}^{p-1} S''_i$ on each processor;
16 $\quad\quad$ **for all** $0 \le i \le p-1$ **do in parallel**
17 $\quad\quad\quad$ P_i marks all points in S'' dominated by a point in $\text{sky}(S_i)$ using $R(S'_i)$;
18 $\quad\quad$ Use an all-to-all communication to send all marked points back to their original processors;
19 $\quad\quad$ **for all** $0 \le i \le p-1$ **do in parallel**
20 $\quad\quad\quad$ P_i removes the received points from S''_i and adds the remaining points in S''_i to its local portion of $\text{sky}(S)$, increasing g_i by the number of added points;
21 $\quad\quad$ Use an all-to-all communication to compute $g = \sum_{i=0}^{p-1} g_i$ on each processor;
22 Use an all-to-all communication to collect all skyline points on processor P_0;

For every point $p \in \text{sky}(S_i)$, processor P_i traverses T starting at the root. For every visited node v, it inspects the children of v. If there is a child w whose top-right corner dominates p, all points in w's subtree dominate p, and the traversal can be stopped, reporting that p does not belong to $\text{sky}(S)$. If no such child w is found, the search continues on all children of v whose bottom-left corners are not dominated by p. The search ignores the children whose bottom-left corners are dominated by p because all points in these subtrees are dominated by p and, thus, cannot dominate p.

The implementation of this strategy requires one global communication round to send one copy of S' to every processor. Then every processor can locally build its R-tree T and use T to remove those points from $\text{sky}(S_i)$ that are dominated by points in S'. This naive implementation, however, duplicates the R-tree construction p times, essentially not parallelizing this part of the algorithm at all. As it turns out, this is the most time-consuming part even of a one-processor version of the algorithm, making it the most important part to parallelize. Section 4.3 discusses how this is done by distributing the construction of T over all p processors. In particular, every processor builds a subtree of T over $\text{sky}(S_i)$. After copying these local pieces to all processors using an all-to-all communication, each processor can very quickly build T from the received pieces. Experimental results confirm that this parallelization of the R-tree construction leads to a tremendous speed-up of the skyline algorithm.

4.2.3.2 Many Local Skyline Points

If $m = |S'| > n/p$, S' does not fit on a single processor, and the strategy described above cannot be applied. In this case, each processor P_i randomly divides $\text{sky}(S_i)$ into $r = \lceil mp/n \rceil$ subsets $S_{i,1}, S_{i,2}, \ldots, S_{i,r}$ of equal size, and the filtering proceeds in r rounds. In the jth round, every processor P_i sends the jth subset $S_{i,j}$ of $\text{sky}(S_i)$ to all other processors. Observe that no processor receives more than n/p points. After receiving the set $S'' = S_{0,j} \cup S_{1,j} \cup \cdots \cup S_{p-1,j}$, each processor P_i marks all points in S'' that are dominated by points in $\text{sky}(S_i)$. Once this is done, every processor sends the points it has marked back to the processors they came from, which can be done in another all-to-all communication. To complete the jth round, every processor P_i now eliminates all the marked points it has received from $\text{sky}(S_i)$, as they are not part of $\text{sky}(S)$. All unmarked points in $S_{i,j}$, on the other hand, belong to $\text{sky}(S)$. This

process stops once at least k members of $\mathrm{sky}(S)$ have been identified or all r rounds have been completed. At this point, the identified skyline points are collected on P_0, and $\min(k, |\mathrm{sky}(S)|)$ of them are returned. Note that the number of identified points is no more than $k + n/p$ because each round identifies at most n/p skyline points and less than k points had been identified by the end of the previous round. Thus, since $k \leq n/p$, all identified points can be collected on P_0.

In each round, the marking of points in S'' that are dominated by points in $\mathrm{sky}(S_i)$ is done similarly to the filtering of dominated local skyline points when $|S'| \leq n/p$. Before the first round of the collective filtering procedure, every processor P_i builds an R-tree $R(S'_i)$ over the points in $S'_i = \mathrm{sky}(S_i)$. In every round, processor P_i uses $R(S'_i)$ to decide for each point $p \in S''$ whether there exists a point in S'_i that dominates p. The required traversal of $R(S'_i)$ is identical to the traversal of T described in Section 4.2.3.1.

Note that this alternate version of the second phase is required only in extreme cases because, as shown by Bentley et al. [37], the expected size of the skyline of a random point set is $\mathrm{O}(\log^{d-1} n)$, which is typically less than n/p, and typical point sets can be expected to behave much like random point sets in this respect. The experimental results discussed in Section 4.5 confirm this, as the algorithm never invoked this version of the second phase during the experiments.

4.2.4 Complexity

Even though our focus was not on obtaining a theoretically efficient parallel skyline algorithm, it is still useful to have an understanding of the worst-case performance of our algorithm. In this section we analyze its complexity in the Coarse Grained Multicomputer (CGM) model of computation.

The CGM model [57] views the computer as consisting of p processors with $\mathrm{O}(n/p)$ local memory and connected by some arbitrary interconnection network. A CGM algorithm proceeds by alternating between computation and communication rounds. During computation rounds, every processor has access only to data in its local memory. During communication rounds, processors exchange data in an all-to-all fashion, with no processor sending and receiving more than $\mathrm{O}(n/p)$ data.

The term “coarse-grained” reflects the assumption that $n \gg p$, which is realistic

for current parallel machines. For current machines, interprocessor communication is significantly more costly than local computation. Hence CGM algorithms focus on minimizing communication by minimizing the number of communication rounds.

Theorem 4.1. *In the worst case, algorithm 1 computes the skyline for a point set consisting of n elements stored on a p-processor CGM with $O(n/p)$ local memory per processor, $n/p \geq p$, using $O(p)$ communication rounds and $O(n^2)$ local computation, $O(n^2/p)$ per processor.*

Proof. Initially, O(1) communication operations are needed to decide whether the union, S', of the p local skylines fits in the memory of a single processor. If we have few local skylines points, only O(1) communication rounds are needed to combine the locally constructed R-trees. On the other hand, in the case of many skyline points, up to $\lceil p|S'|/n \rceil$ communication rounds are needed to complete the collective filtering approach. Hence the total number of communication rounds needed is bounded by O(p). In terms of local computation, observe that the worst-case cost of BBS on n points is O(n^2). Since each local skyline computation in either of the two phases operates on O(n/p) points, its cost is therefore O(n^2/p^2). Since we have just argued that the algorithm terminates after O(p) rounds, this implies that the total local computation cost per processor is O(n^2/p). □

Note that the bound in Theorem 4.1 is a worst-case upper bound. Our experiments show that the performance in practice is much better. In particular, we never ran into a situation where the O(1)-communication round version of the global skyline computation was insufficient, and BBS is very fast in practice due to pruning of R-tree nodes.

4.3 A Parallel R-Tree Construction Algorithm

As discussed in Section 4.2.3.1, one of the main bottlenecks in any R-tree-based skyline algorithm is the construction of the R-tree, while the computation of the skyline using the R-tree is comparably fast. Thus, it is imperative to parallelize the R-tree construction. This section describes how to do just this.

The key is a representation of the R-tree that allows a portion of an R-tree constructed on one processor to be used without modification on another processor.

Recall that we are working on a distributed-memory parallel architecture, where each processor has its own private address space. Therefore, constructing an R-tree on one processor using a standard pointer-based representation and then sending the R-tree to another processor would result in the tree possibly being stored in different memory locations on the sending and receiving processors. It is possible to adjust the pointers, but this would require a traversal of the entire tree by each processor, negating the performance gain achieved by distributing the R-tree construction. For this reason, we propose a pointerless R-tree representation that can be exchanged between processors without any need to adjust the representation on the receiving processor.

Section 4.3.1 describes the pointerless R-tree representation. This representation is not restricted to be used only in the context of the skyline algorithm but can be used in any scenario where the construction of a static tree structure needs to be distributed over multiple processors. It is, however, described using the construction of an R-tree T for a point set S as a concrete example. Section 4.3.2 discusses how to use this representation in a parallel R-tree construction algorithm, and Section 4.3.3 discusses how to implement the skyline filtering procedure using the constructed R-tree.

4.3.1 A Pointerless R-Tree Representation

The structure of an R-tree over a point set is defined by two parameters: the *leaf size* l determines the number of points to be stored at a leaf, and the *fanout* f determines the maximum number of children per internal node. Every node v of the tree also stores its MBB (see Section 3.2.4.2).

The pointerless R-tree representation consists of two arrays, T and S. Array S stores the point set and is divided into chunks of size l, each of which contains the points stored at a leaf of the tree. Array T holds the nodes of the tree, where every node v is represented as a quadruple $(l(v), u(v), \text{children}(v), \deg(v))$; $l(v)$ and $u(v)$ are the two opposing corners of the MBB of v. If v is an internal node, then $\text{children}(v)$ is the index of the leftmost child of v in T, $\deg(v)$ is the number of children of v, and these children are stored in $\deg(v)$ consecutive cells in T, starting with index $\text{children}(v)$. If v is a leaf, then $\text{children}(v)$ is the index of the first point

in S associated with v, $\deg(v)$ is the number of these points, and all points associated with v are stored in $\deg(v)$ consecutive cells in S, starting with index children(v). Deciding whether a node is an internal node or a leaf is a matter of index arithmetic. See Figure 4.3 for an illustration of this data structure.

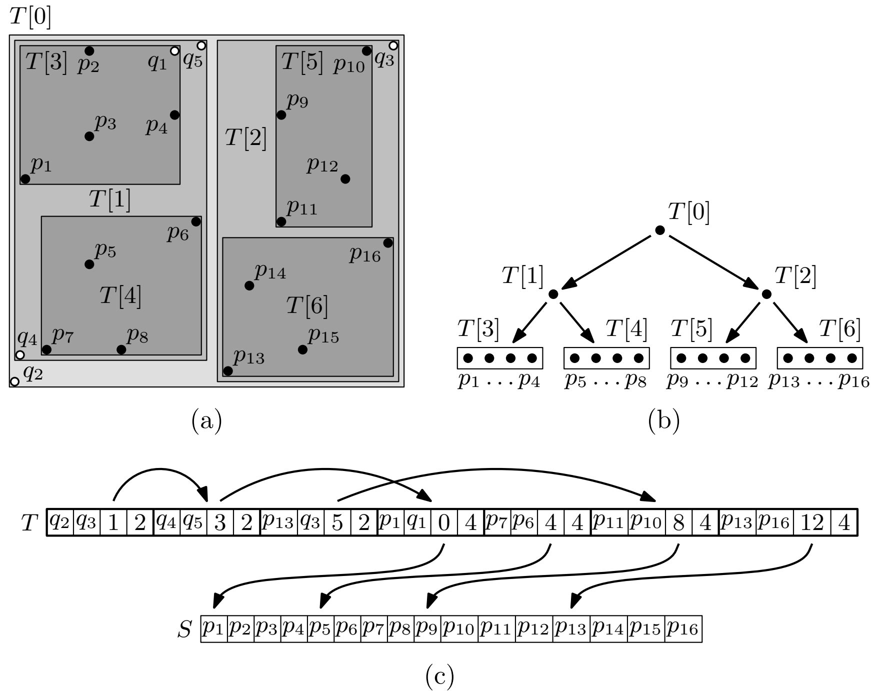


Figure 4.3: Figure (b) shows an R-tree over the black points in Figure (a). Figure (a) also shows the MBB's of the R-tree nodes in Figure (b); the white points are additional points used to represent these MBB. Figure (c) shows the two arrays representing the tree in Figure (b). Array T is partitioned into records of four cells storing the bottom-left and top-right corners of the node's MBB, the index of the first child or point of the node, and the number of children or points associated with the node.

This array representation of an R-tree can be computed in a bottom-up fashion. We can assume that array S is already populated with the relevant points, as the input has to be provided in some fashion, and an array of points is a natural input representation. To populate array T, we construct the levels of the tree bottom-up, starting with the leaves. First, we scan S backwards and add a new leaf node v at the end of T, for every l points in S we read. In the process, we compute the information to be stored at v, including its MBB. Once we have created the leaf level, we scan the nodes on the current level backwards and add a node to the next level, for every f nodes on the current level. Again, we can compute the information to be stored at

every constructed node in the same way as before. We continue in this manner, until the current level has only one node, the root of T.

4.3.2 Parallel R-Tree Construction

The R-tree representation in Section 4.3.1 can be exchanged between processors without becoming invalid, as array indices act as pointers relative to the beginning of the array. Hence, this representation can be used to parallelize the construction of the R-tree T discussed in Section 4.2.3. First, the point set S is distributed among the p processors or, as in the context of the skyline algorithm, is given already distributed among the processors. Then each processor builds an R-tree T_i over its subset, S_i, of points using the pointerless R-tree representation described in Section 4.3.1. To provide every processor with an R-tree over the entire point set, the trees $T_0, T_1, \ldots, T_{p-1}$ are sent to all processors using an *all-to-all* communication. After copying the local pieces to all processors, the only task left for every processor is to assemble these individual trees into the final R-tree T. To this end, each processor creates a "cap" M on top of trees $T_0, T_1, \ldots, T_{p-1}$. This cap M is an R-tree over the bounding boxes associated with the roots of trees $T_0, T_1, \ldots, T_{p-1}$. Every leaf of M points to the physical locations of the two arrays representing a tree T_i. The internal nodes of M are then constructed bottom-up from the leaves of M in the same manner as the construction of the internal nodes of T in Section 4.3.1. This construction is illustrated in Figure 4.4.

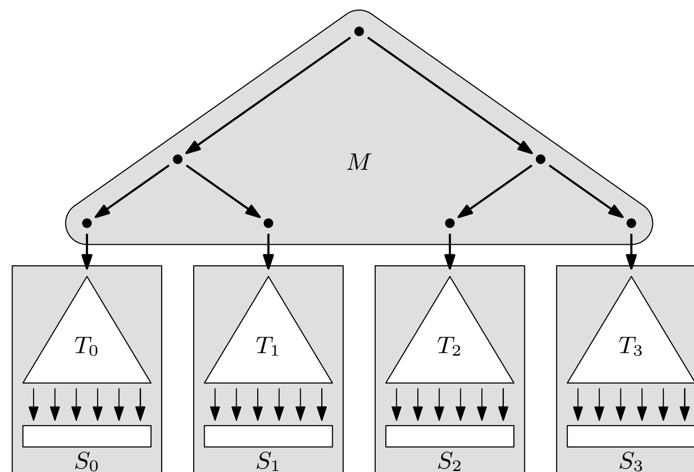


Figure 4.4: Parallel R-tree construction: merging the R-tree pieces.

While the construction of M on each processor still means the duplication of work, the amount of duplicated work is minimal, as M has less than $2p$ nodes. On the other hand, the construction of the majority of the nodes of T, in the bottom part of T, takes full advantage of the p processors.

4.3.3 Querying the Constructed R-Tree

Answering queries with respect to the constructed R-tree always amounts to following a set of root-leaf paths in the tree, which can be done by following pointers from parents to children. Given the constructed representation of T, every root-leaf path consists of three parts: a subpath in M, a subpath in T, and the set of points associated with the leaf at the bottom end of the path. Following pointers from parents to children inside M or T amounts to some simple index arithmetic. Following the pointer from the last node in M to the root of the visited subtree T_i and following the pointer from a leaf of T_i to the corresponding set of points is equally easy, as long as every leaf of M and T is marked as such.

4.4 Implementation Details

We implemented our parallel skyline algorithm in C++ using the Message Passing Interface (MPI) library [6], which is a standard that defines a communication API for distributed memory architectures. The rest of this section discusses implementation choices made in different parts of the algorithm.

R-tree construction. The performance of R-tree queries depends on the amount of overlap between the bounding boxes of nodes that are not ancestors of each other. In order to minimize the overlap between the bounding boxes in each tree T_i, each processor sorts the points in S_i in Hilbert order [79] before constructing T_i over S_i. It is well known that this ordering preserves locality and produces R-trees with little overlap between bounding boxes. In our experiments, Quicksort is used to sort the points, in combination with the comparison function provided by Doug Moore's Hilbert Mapping Library [105].

The random distribution of points among the processors, on the other hand, almost certainly leads to tremendous overlap between the bounding boxes of the roots

of trees $T_0, T_1, \ldots, T_{p-1}$. While this is not desirable as far as the R-tree construction is concerned, it seems unavoidable because the alternative is to distribute the points over the processors in Hilbert order. We do not follow this approach as there is computational overhead involved in distributing the data based on this order. Moreover, we wanted to take advantage of the simpler random distribution method in order to obtain small local skylines.

Communication. Another key element in the performance is the communication required to move large packets of data across the network. Since our algorithm is designed to work in communication rounds, every communication round is implemented using *mpi-all-to-all-v* in MPI. The all-to-all operation greatly reduces the overhead of communication between processors.

4.5 Performance Evaluation

We evaluated the performance of PARALLEL-SKYLINE using an extensive set of experiments. These experiments focus on two scenarios: The first one considers the overall performance of the entire algorithm; this is relevant for single query computations. The second scenario looks at the performance of the query procedure once the R-tree has been constructed; this shows how applications benefit from building an R-tree in a preprocessing phase and then ask queries on different subsets of dimensions. In all experiments, we therefore constructed the R-tree on all 6 dimensions independently of the query dimensions, and answered the queries on subsets of between 2 and 6 dimensions using this structure.

Note that the query procedure includes: (a) the local BBS-query of the first phase, and (b) the collective-filtering of the second phase. The construction of R-trees over the local point sets assigned to the processors in the first phase is considered preprocessing. However, while the evaluation of the query procedure excludes the construction of the R-trees in Phase 1, the computation of the R-trees to complete Phase 2 is included in the query cost (i.e., running time of the query procedure). This R-tree construction during Phase 2 is necessary for each individual query, that is, Phase 2 of each query does not benefit from the preconstructed R-trees for the entire point set.

As discussed here, the experiments confirmed that the algorithm achieves good speed-up and scales well, allowing the processing of datasets beyond the reach of a single node in the cluster. Next we describe our test environment, the evaluation criteria we consider and the data sets used in our experiments. Then, in Sections 4.5.4 and 4.5.5, we discuss the results of our experiments.

4.5.1 Platforms and Measurements

The Parallel-Skyline algorithm was evaluated on a 32-node Beowulf-style cluster with 1.8GHz Intel Xeon processors. Each node was equipped with 1GB of RAM and two 40GB 7200 RPM IDE disk drives. The operating system on each node was Linux RedHat 7.2 as part of a ROCKS cluster distribution. The code was compiled using gcc 2.95.3 and MPI/LAM 6.5.6 using optimization level -O3. Communication between the nodes was provided by a Cisco 6509 GigE switch. The code was a faithful implementation of Algorithm 1 using the pointerless structure from Section 4.3 to represent the R-tree.

All timing results denote the wall clock time taken by the algorithm to complete, measured from the start of the first process in our parallel algorithm till the termination of the last process.

4.5.2 Performance Measurement

For a parallel algorithm, the goal is to speed-up the computation proportional to the number of processors used. This increase in performance can be translated into two performance measures for parallel algorithms [63]:

1. **Speed-up:** Given a fixed input size, the ideal running time of the algorithm on p processors is a $1/p$ fraction of the running time on a single processor. In this case, the algorithm is said to achieve *linear speed-up*. Thus, the speed-up of the algorithm measures the reduction in time needed to process the same amount of data using more than one processor.

2. **Scale-up:** The scale-up of the algorithm, on the other hand, measures the amount of data that can be processed in a given amount of time using p processors. Ideally, one would hope that this is p times as much data as using a

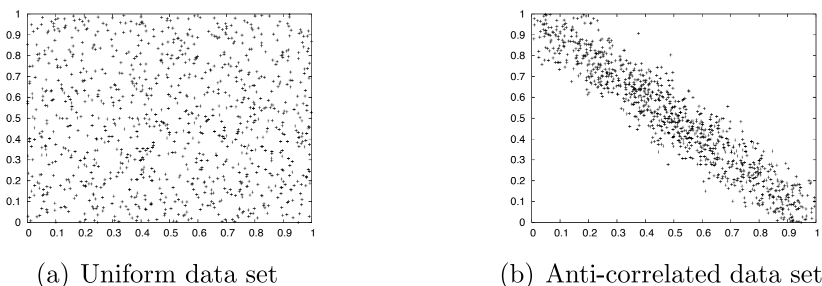
single processor.

These two metrics are the simplest and most commonly used measures of parallel performance. Our experiments evaluate PARALLEL-SKYLINE with respect to both measures.

4.5.3 Datasets

As previous evaluations of skyline algorithms in the literature [39, 45, 56, 86, 108, 110, 133, 136], the algorithm was evaluated on synthetic and on real data sets with between 2 and 6 dimensions. The synthetic data sets were uniformly distributed random point sets (see Figure 4.5(a)) and anti-correlated random point sets (see Figure 4.5(b)). Uniformly distributed points should exhibit the behavior analyzed by Bentley [37], and thus, for moderate numbers of dimensions as considered in our experiments, should have small skylines. Anti-correlated point sets, on the other hand, should have most of their points in their skylines and, thus, represent a good tool for testing how well the algorithms can deal with large skylines. These synthetic data sets were generated using a data generator used in previous evaluations of skyline algorithms and provided by the authors of [39].

The real-world data was taken from the HYDRO1k Elevation Derivative Database [44] and contained geographic data that provides hydrological information on a continental scale. The size of this data set was approximately 29,000,000 records.



(a) Uniform data set (b) Anti-correlated data set

Figure 4.5: Synthetic data distributions used in the experiments.

For evaluating the speed-up, synthetic and real data sets of 1,000,000 and 5,000,000 records were used. Table 4.1 shows the sizes and skyline sizes of the point sets used in the experiments. The evaluation of the scale-up used synthetic data between 2 and 6 dimensions and up to 80,000,000 records.

(a) Uniform

Dim.	Size 1M	Size 5M
2	13	17
3	97	138
4	623	631
5	1819	3110
6	5371	9069

(b) Anti-correlated

Dim.	Size 1M	Size 5M
2	11	22
3	414	477
4	2799	4417
5	14475	25328
6	97891	212865

(c) Hydrological

Dim.	Size 1M	Size 5M
2	636	1115
3	1611	2298
4	2913	4561
5	5802	9440
6	8781	14095

Table 4.1: Data sets and their skyline sizes.

4.5.4 Speed-up Evaluation

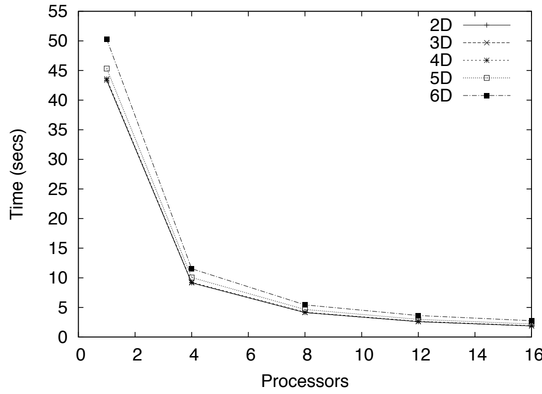
This section discusses in detail the speed-up results of our algorithm. To do this, Subsection 4.5.4.1 studies the total speed-up (the performance of the entire algorithm) and Subsection 4.5.4.2 analyzes the query speed-up (the performance of the query procedure once the R-tree has been constructed). The experimental results with uniform data are shown in Figure 4.6, with hydrological data in Figure 4.7 and with anti-correlated data in Figure 4.8.

4.5.4.1 Total Speed-up

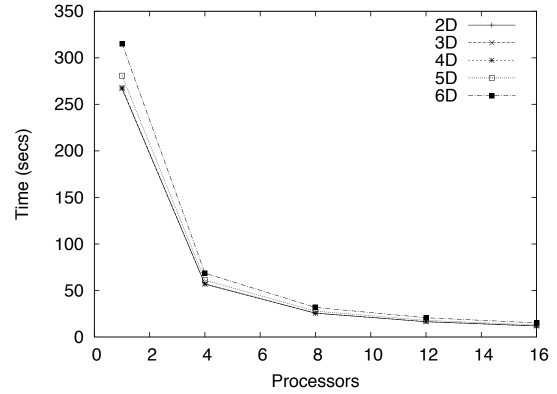
Based on the results obtained with *uniform*, *hydrological* and *anti-correlated* data with 1 million and 5 million points (see Figures 4.6a, 4.6d, 4.7a, 4.7d, 4.8a and 4.8d) we conclude that the total processing time (which includes the local R-tree construction, communication cost, and query cost) of a d-dimensional skyline query is significantly reduced when more processors are added to the computation.

Consistent with our results for the total time, the speed-up of the algorithm with uniform and hydrological data, shown in Figures 4.6b, 4.6e, 4.7b and 4.7e, also improves as we increase the size of the input and hence the total amount of work to be performed. The total speed-up curves in all dimensions are above the linear speed-up curve, indicating that the algorithm parallelizes effectively, achieving superlinear speed-up. The most likely reason for this behavior is the effect that cache memory has on the running time of the algorithm. With a larger overall cache size due to multiple processors, more data fits in the caches; thus, memory access time is reduced considerably.

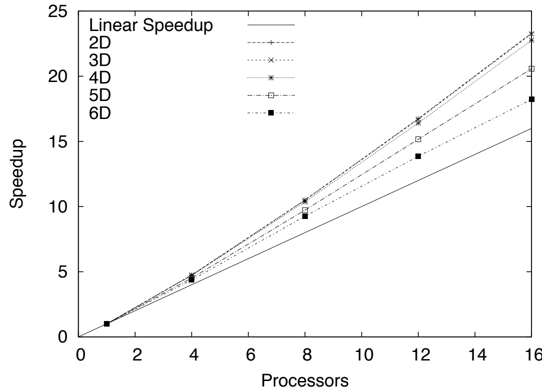
Computing skylines on anti-correlated data sets is much more challenging than the



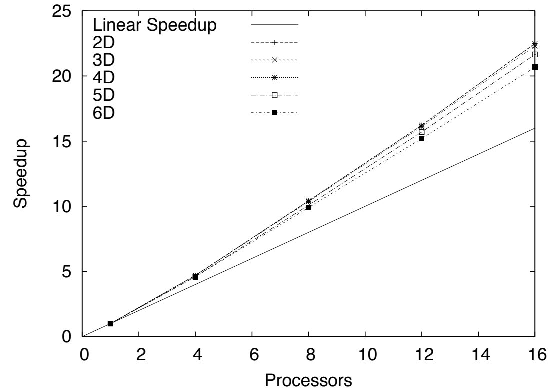
(a) Total wall clock time (1M points)



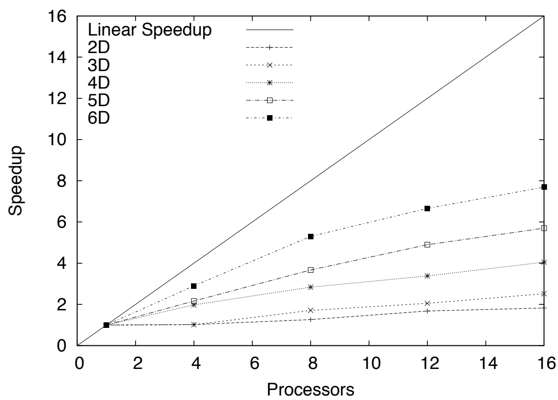
(d) Total wall clock time (5M points)



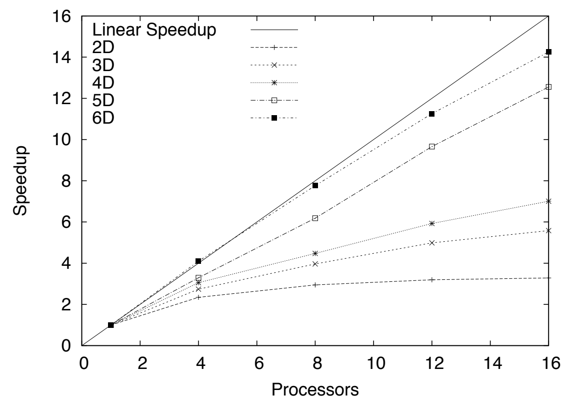
(b) Total relative speedup (1M points)



(e) Total relative speedup (5M points)



(c) Query relative speedup (1M points)



(f) Query relative speedup (5M points)

Figure 4.6: Speedup results on uniform data. The left column (Figures a–c) corresponds to experiments with 1,000,000 points. The right column (Figures d–f) corresponds to experiments with 5,000,000 points

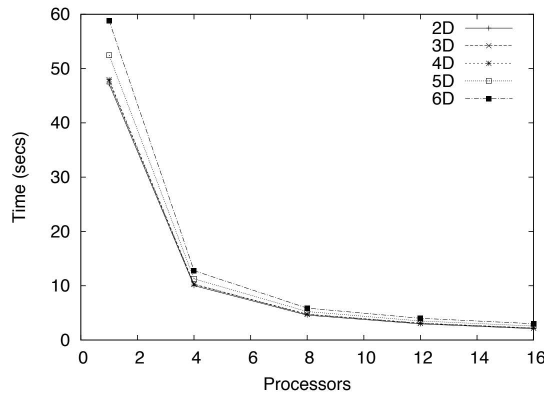
other data sets as reflected by the running times of Figures 4.8a and 4.8d. In this case, the algorithm achieves superlinear speed-up with 2–4 dimensions (see Figures 4.8b and 4.8e). However, with 5 and 6 dimensions, the total speed-up is sublinear. The reason is that, in 5–6 dimensions, the size of the skyline has increased substantially and computing the full skyline requires significantly more time. The next subsection sheds more light into the effects of dimensionality and the size of the skyline on the algorithm's speed-up.

4.5.4.2 Query Speed-up

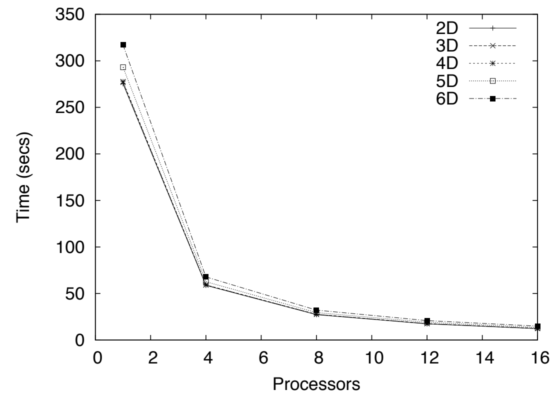
The R-tree construction cost is worthwhile if it can be amortized over a number of queries that can subsequently be answered using this tree. This section studies the performance of the query procedure in terms of the effect that input size and dimensionality have on it, and also studies the hard case presented by anti-correlated data.

Effect of the input size. We first study the effect, on uniform and hydrological data, of increasing the input size from 1,000,000 points (Figures 4.6c and 4.7c) to 5,000,000 points (Figures 4.6f and 4.7f); we can see that the query speed-up on larger inputs is higher. An explanation for this is that there is little work to be done with the smaller data sets, which results in low query times. On the other hand, query processing on a larger data set takes longer, establishing a higher baseline on a single processor with which the running times on multiple processors are compared. It is worth pointing out that in these curves, specifically for 5–6 dimensions, the algorithm again achieves at some point superlinear relative speed-up, which we believe to be the result of cache effects. Given more processors, each processor has to process less data, resulting in a higher fraction of data that fits in cache.

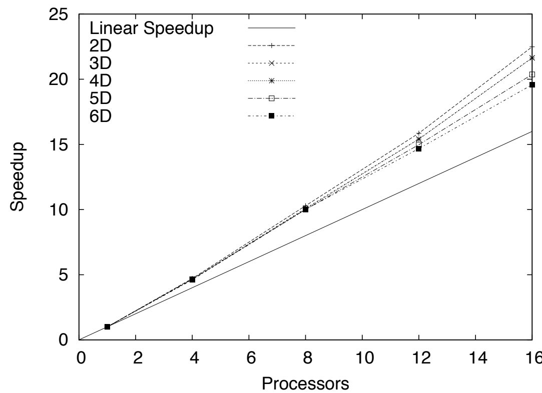
Effect of the dimensionality. In contrast to the case of total speed-up, where the increase in dimensionality tends to decrease the speed-up, the query speed-up now improves with increasing dimensionality and processors (see Figures 4.6c and 4.6f with *uniform* data, and Figures 4.7c and4.7f with *hydrological* data). In these results, the lowest query speed-up values were obtained with data sets of 2–4 dimensions. The



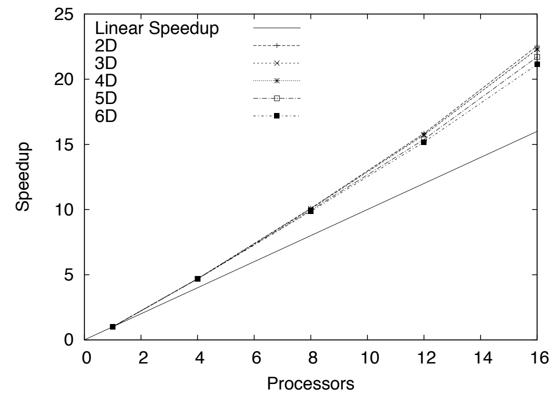
(a) Total wall clock time (1M points)



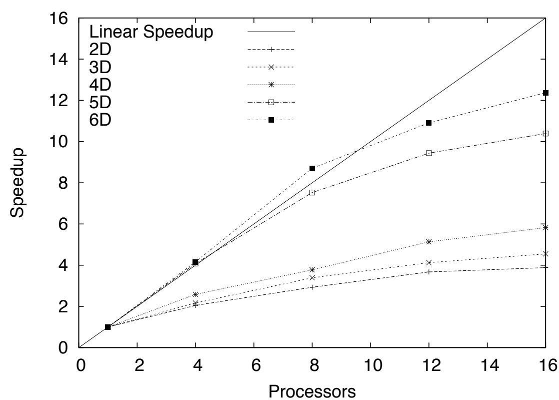
(d) Total wall clock time (5M points)



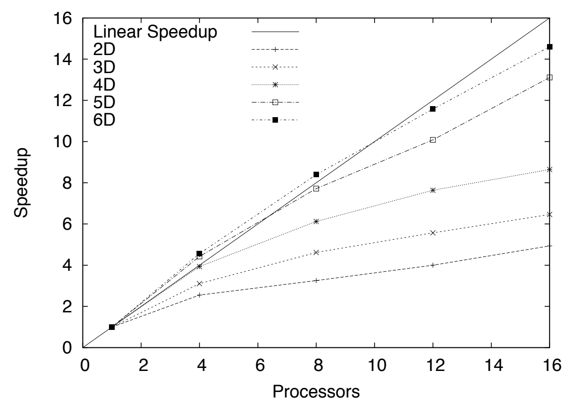
(b) Total relative speedup (1M points)



(e) Total relative speedup (5M points)

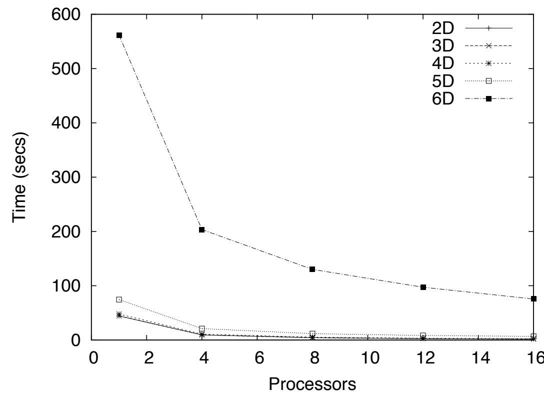


(c) Query relative speedup (1M points)

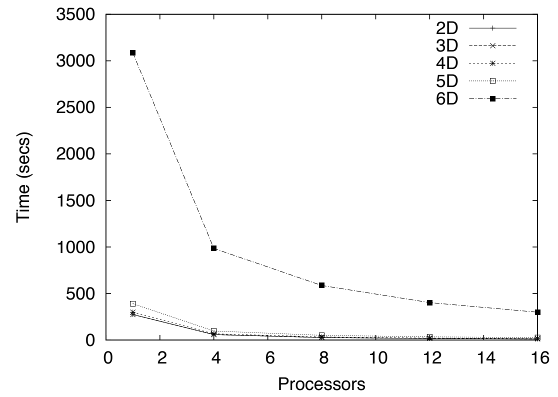


(f) Query relative speedup (5M points)

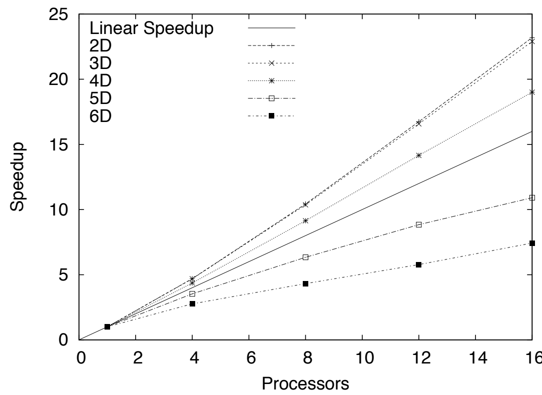
Figure 4.7: Speedup results on hydrological data. The left column (Figures a–c) corresponds to experiments with 1,000,000 points. The right column (Figures d–f) corresponds to experiments with 5,000,000 points



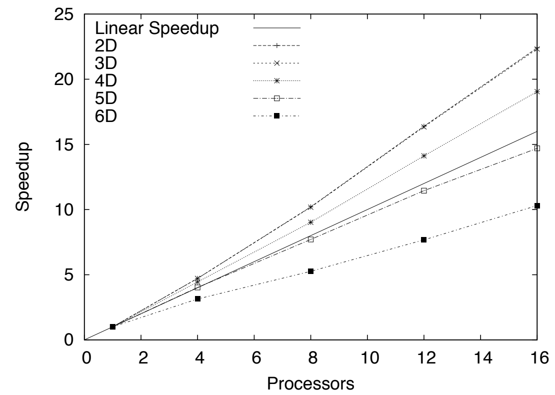
(a) Total wall clock time (1M points)



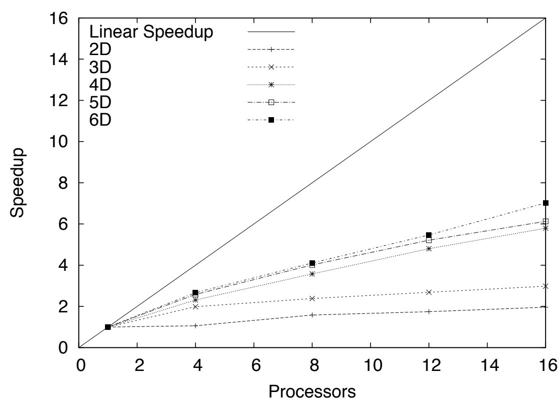
(d) Total wall clock time (5M points)



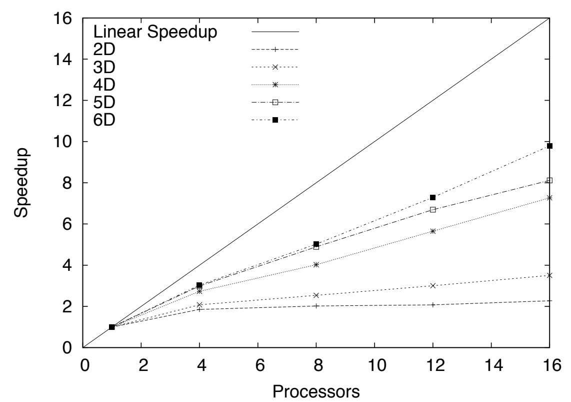
(b) Total relative speedup (1M points)



(e) Total relative speedup (5M points)

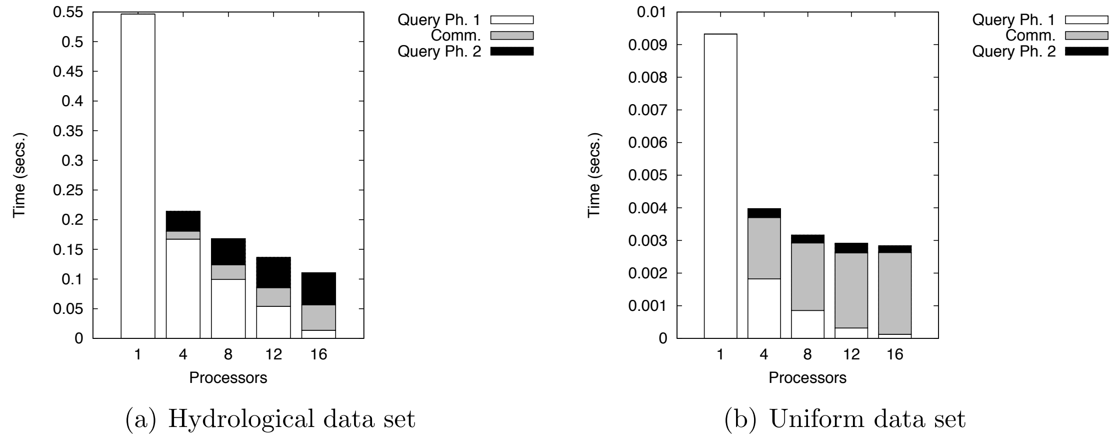


(c) Query relative speedup (1M points)



(f) Query relative speedup (5M points)

Figure 4.8: Speedup results on anticorrelated data. The left column (Figures a–c) corresponds to experiments with 1,000,000 points. The right column (Figures d–f) corresponds to experiments with 5,000,000 points



(a) Hydrological data set (b) Uniform data set

Figure 4.9: Break-down of a $2d$ query with 5,000,000 points

reason is that, in these dimensions, there is little computational work to perform. In 5–6 dimensions, on the other hand, the algorithm yields higher query speed-up values, as there is more work to be done (e.g., a higher number of dimensions means more work per point and an increase in the number of skyline points).

Effect of the skyline size. In the *uniform* distribution with 1,000,000 points we observed low query speed-up, while with the *hydrological* data set and the same number of points, the curves, especially in higher dimensions (5 and 6), show slow but steady increase. The reason for this difference lies in the sizes of the skyline: in 2-d, for example, the skyline of the hydrological data is almost 50 times larger than the one of uniform data (see Table 4.1), which allows better parallelization. Figure 4.9 also provides an explanation for this behavior: when comparing the performance of both data sets for a 2-d query with 5,000,000 points, we can see that the query times are so small for the uniform data set (Figure 4.9b) that adding more processors does not bring much improvement and data communication becomes a significant factor. In contrast, there is more computational work to be done for hydrological data, and this is reflected in Figure 4.9a, where with an increasing number of processors, the communication overhead gradually increases as expected, but the computation is better split among the processors.

Now, to analyze the effect that a very large skyline has on the algorithm, and because it poses a greater challenge, we used *anti-correlated* data (see Figure 4.8).

Previous sequential work has focused on this as a hard case, as, typically, anti-correlated data sets have the largest skyline size. By comparing Figures 4.7c and 4.7f to Figures 4.8c and 4.8f we can see the impact that anti-correlated data has on the performance of the algorithm. The observation here is that, despite the higher computational workload in the anti-correlated case—which intuitively should allow better parallelization—, the query speed-up on anti-correlated data sets is worse than that of hydrological data sets. The reason for the lower query speed-up is that, with anti-correlated data, a much greater number of points survive after the local skyline computations in Phase 1. This means that the timings of Phase 2 are now much higher due to the very large size of the skyline, and the required time to filter the surviving points. It is also worth pointing out that in this case communication is negligible compared to the query times.

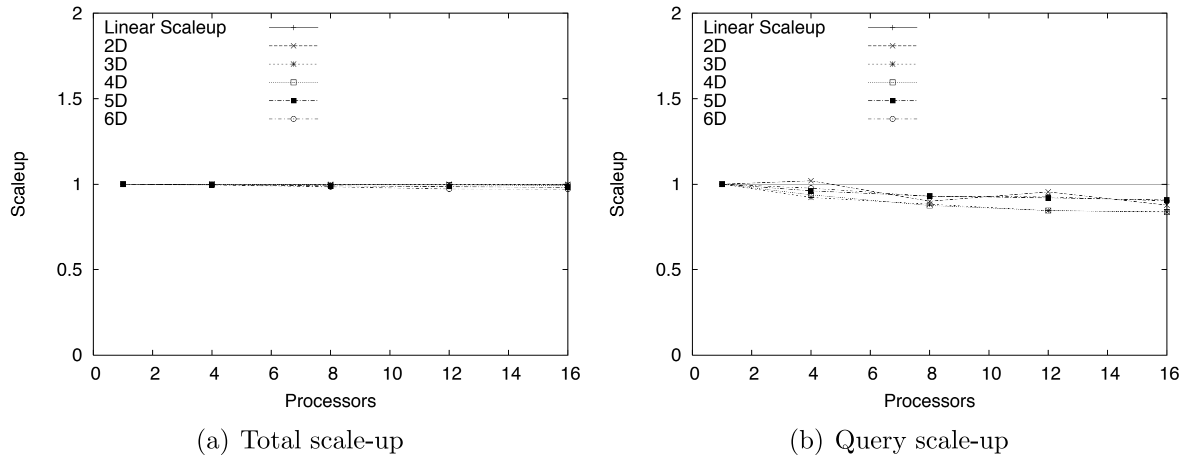
4.5.5 Scale-up Evaluation

Our last experiment focuses on measuring the scale-up of our algorithm, where the size of the processed data set increases proportionally with the number of processors, and one hopes to be able to process p times as much data using p processors as one can process with a single processor in the same amount of time. Figures 4.10 and 4.11 show our experimental results.

Overall, we observe excellent scale-up results for PARALLEL-SKYLINE. The scale-up of the total time, shown in Figures 4.10a and 4.11a for uniform and hydrological data sets respectively, is above .97 for all dimensions in both data sets.

For the query time, with uniform data, the scale-up is above 0.83 for all dimensions and numbers of processors (see Figure 4.10b). For dimension 2, there are slight variations which are likely due to cache effects, and the reduced work to be done in Phase 2 on account of the relatively small skyline. Dimensions 3 and 4 have a similar scale-up behavior, dropping off slightly between 1 and 8 processors, but then holding steady up to 16 processors. For the cases of dimensions 5 and 6, the algorithm achieves excellent scale-up of not less than .9. Similar behavior is reported with hydrological data, where the query scale-up is above 0.81 for all dimensions (see Figure 4.11b). Dimensions 5 and 6 yield the best results with not less that .92.

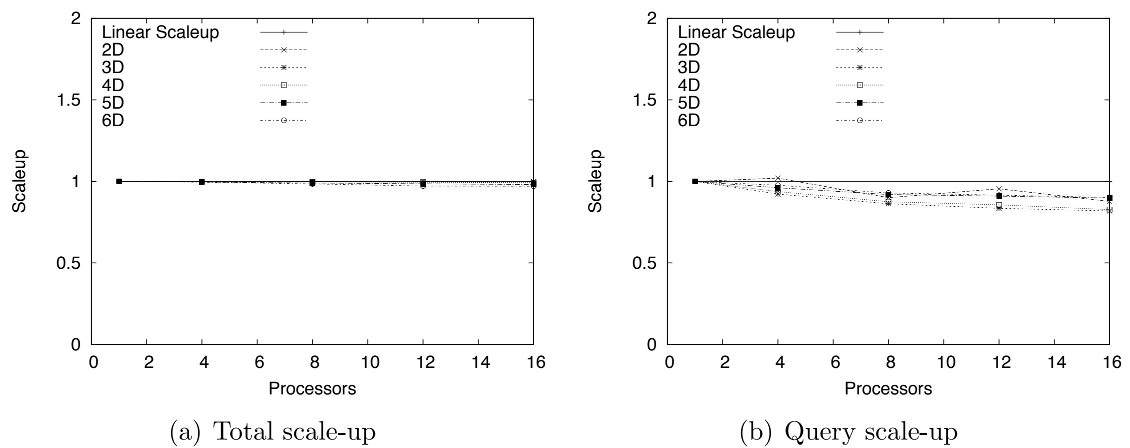
In order to understand why the scale-up for dimensions 2–4 is not as good as for 5



(a) Total scale-up

(b) Query scale-up

Figure 4.10: Scale-up results on uniform data set with 5,000,000 points per processor with Algorithm 1.



(a) Total scale-up

(b) Query scale-up

Figure 4.11: Scale-up results on hydrological data set with 5,000,000 points per processor with Algorithm 1.

and 6 dimensions, we must recall that the network bandwidth is not being scaled as we increase the data size. As we increase the data size per processor, more data has to be moved across the network, and then communication becomes a factor for the overall query times. On the other hand, for dimensions 5 and 6, the communication effect is offset by the fact that sequential computation times are higher in comparison to the time required to exchange data.

Overall, the results shown in this section demonstrate that the Parallel-Skyline algorithm can be used effectively to speed up the computation of skylines of large data sets of a moderate number of dimensions, which is what is relevant in practice [86]. Moreover, the algorithm shows good scalability with increasing data set sizes and number of processors.

Chapter 5

External-Memory Skyline Computation

Our Parallel-Skyline algorithm speeds up the processing of skyline queries tremendously. This result is particularly relevant for the kind of user handling large data warehouses—e.g., with several terabytes of data—which simply cannot be processed efficiently without using parallelism. Other users, such as small companies, may not be able to justify the acquisition of a powerful parallel machine. Yet the data sets of such users may reach hundreds of gigabytes and, thus, are beyond the size of the main memory of a standard PC. Thus, these users can benefit from I/O-efficient methods to answer skyline queries. In this chapter we present two I/O-efficient skyline algorithms. These algorithms are based on branch-and-bound skyline (BBS) and can be considered I/O-efficient versions of BBS. The algorithms can also replace BBS as the algorithm used on each processor in order to increase the input size our parallel skyline algorithm can handle efficiently.

In the following sections, we first introduce and then describe in detail our external memory skyline query algorithms. Afterwards, we demonstrate their effectiveness in processing point sets beyond memory size and show that they outperform internal-memory skyline methods on such data sets.

5.1 Overview

Designing an I/O-efficient version of BBS requires us to develop I/O-efficient implementations for the different steps of the algorithm. For some steps this is entirely trivial, while others pose greater challenges. To explain this in further detail, recall that BBS is a two-part procedure: the first part constructs an R-tree over the given point set, the second traverses the constructed R-tree to extract the skyline points. Also recall that during the traversal, every inspected R-tree node or point is checked against the currently known skyline points to test whether any of them dominates this node or point. This requires a linear scan through the current set of skyline

points for each test and, thus, does not pose any challenges in terms of I/O-efficiency. Moreover, the computed skyline is often small enough to fit in memory, completely removing any concerns about I/O-efficient access to the current set of skyline points. The construction of the array-based R-tree over the given point set S does not pose any challenges either, as it involves sorting the points in S in an appropriate manner and building the tree over S using what amounts to two scans of the given point set (see Section 4.3.1).

The traversal of the R-tree T using a BBS query, on the other hand, is non-trivial, as the query prescribes a specific order in which to visit the nodes of the tree, and this order may bear little resemblance to the order the nodes are stored on disk. Thus, the BBS query may cause a large number of random disk accesses, resulting in a substantial slow-down of the query procedure. The two algorithms presented in this chapter address this I/O bottleneck in BBS by ensuring that the order in which nodes are visited matches the order the nodes are stored on disk.

The first algorithm, DFS-Skyline (DFSS), modifies the order in which the query procedure inspects R-tree nodes. The modified traversal may result in visiting more R-tree nodes than using BBS, but the traversal visits nodes in the order they are stored on disk, allowing the query cost to be bounded by that of a single scan of the R-tree. In practice, the performance of DFSS is even better than this estimation, as many nodes are skipped by the traversal due to pruning of subtrees. Furthermore, this sequential scan takes advantage of disk read-ahead and fast seeks between adjacent sectors, which would not be the case for random disk accesses even if the data was accessed in a blockwise fashion.

The second algorithm, Presorted-BBS, arranges the R-tree nodes so that standard BBS visits the nodes in the order they are stored on disk. Thus, it combines the efficient pruning of BBS with the good I/O behavior of DFSS. This, however, comes at an increased R-tree building cost, as an additional sort of the R-tree nodes is required before applying the query. Our experiments investigate whether the decrease in number of visited R-tree nodes during the query suffices to pay for the increased preprocessing cost.

Both algorithms represent the R-tree using the pointerless R-tree representation from Section 4.3, as its array structure makes it an excellent candidate to be stored

on disk.

5.2 DFS-Skyline (DFSS)

The discussion of the DFS-Skyline algorithm is split into two parts. First, the construction of the R-tree is discussed, including the layout of its nodes on disk. Then, the procedure for querying the R-tree is presented, along with a proof of its correctness.

5.2.1 R-tree Construction

To construct the R-tree for DFSS, the points in the given point set S are initially sorted by their distance from the origin using an I/O-efficient sorting algorithm. Then an array T storing the internal nodes of the R-tree is constructed similar to the construction in Section 4.3.1; however, the ordering of the nodes in the tree deviates from the one described in Section 4.3.1. In particular, the nodes are now stored in the order they are visited by a preorder traversal that visits the children of each node in left-to-right order (and, thus, visits the leaves of the tree by increasing distance from the origin). Figure 5.1 illustrates this.

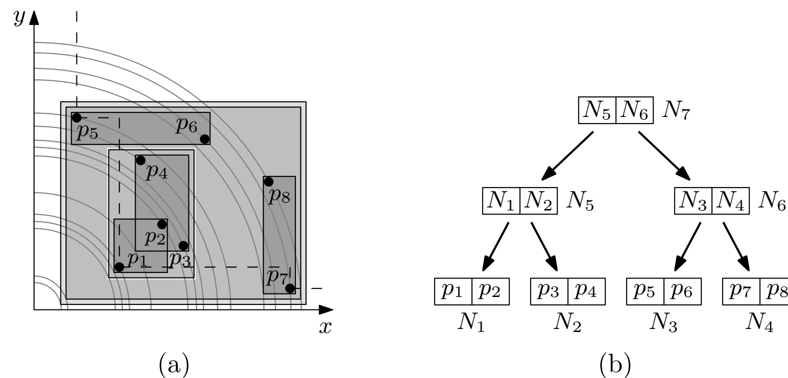


Figure 5.1: DFSS vs. Presorted-BBS. The former stores (and visits) the nodes of T in the order $N_7, N_5, N_1, N_2, N_6, N_3, N_4$. The latter does so in the order $N_7, N_6, N_5, N_1, N_2, N_3, N_4$.

The construction of this layout is fairly easily achieved using a (backward) scan of S, a scan of T to populate T, and a stack R that maintains the roots of the subforest of T constructed so far. The entries of R are quadruples (i, h, j, B), each representing

a node v in T. A node v represented by a quadruple (i, h, j, B) has MBB B, is stored at position i in T, has distance h from the leaf level, and is the jth node in R at this distance from the leaf level.

As in Section 4.3.1, a leaf v is added to T for every l (*leaf size*) consecutive elements read from S. Upon construction, v does not have a parent yet. Hence, a quadruple $q_v = (i, 0, j, B)$ representing v is pushed onto stack R. Components i, h, and B of the quadruple are easy to determine. To determine j, let $q_w = (i', h', j', B')$ be the quadruple on the top of stack R immediately before pushing q_v onto R. If $h' = 0$, then q_w represents a sibling leaf of v, and $j = j' + 1$. Otherwise, it represents a non-leaf node w, and v is in fact the only leaf currently on R; so $j = 1$. After pushing q_v onto R, subtrees of T are merged (if possible) using the following procedure.

If the topmost entry $q = (i, h, j, B)$ on R satisfies $j = f$, the f nodes $v_1, v_2, \ldots, v_f$ corresponding to the f topmost entries $q = q_1, q_2, \ldots, q_f$ on R are roots at the same level h in T, where f is the fanout of the R-tree. These subtrees are merged into a single tree by creating a new node v and making nodes $v_1, v_2, \ldots, v_f$ its children. Entries $q_1, q_2, \ldots, q_f$ are popped from R, and a new entry $q_v = (i', h + 1, j', B')$ representing node v is pushed onto R. Here, i' is the index of node v in T; B' is its MBB, which is easily computed from the MBB's of v's children stored with the popped entries $q_1, q_2, \ldots, q_f$; and j' is computed again by inspecting the entry (i'', h'', j'', B'') on the top of stack R immediately before pushing q_v. If $h'' = h + 1$, then $j' = j'' + 1$; otherwise $(h'' > h + 1)$, $j' = 1$. This may again lead to the j-component of the topmost entry (that is, q_v) being equal to f. If so, this parent addition procedure is repeated until the topmost entry on R has a j-component less than f.

Once all entries in S have been processed in this manner, there may be a list of roots left on R. They are merged into a single tree by repeatedly popping all entries with the same h-component from the stack, creating a new node in T that is the parent of the corresponding node, and pushing a new tuple representing this parent onto R. This is repeated until R contains a single entry, which represents the root of T.

5.2.2 Skyline Construction

Given the representation of the R-tree constructed in the previous subsection, a modified BBS procedure can now be used to compute the skyline of S. This procedure (shown in Algorithm 3) is identical to BBS, except that the nodes of the R-tree may be visited in a different order. Recall from Section 3.2.4.2 that, if a tree node cannot be pruned because the bottom-left corner of its bounding box is not dominated by a skyline point, BBS adds its children to a pool of nodes to be visited next, and it always chooses the node with minimum distance from the origin to visit next.

Algorithm 3: DFS-Skyline(T, r)

Input: A pointerless representation of an R-tree T over a point set S and with root r.

Output: An array L storing the skyline points of S.

```
1  L := ∅;                                            /* Set of skyline points */
2  S := ∅;                                     /* Stack of nodes to be explored */
3  Push(S, r);
4  while S ≠ ∅ do
5      u := Pop(S);
6      if u is not dominated by a point in L then
7          if u is a leaf then
8              forall the points p in u by increasing distance from the origin do
9                  if p is not dominated by a point in L then
10                     Append p to L;
11         else
12             forall the children v of u from right to left do
13                 if v is not dominated by a point in L then
14                     Push(S, v);
15 return L;
```

DFSS, on the other hand, performs a pruned *depth-first* traversal; that is, for a node v that cannot be pruned (using the same condition as used by BBS), DFSS first completes the traversal of the subtree rooted in v's leftmost child, then proceeds to the second child from the left, and so on. Given the order in which the nodes are stored on disk, this left to right traversal means that the algorithm actually visits the nodes in the order they are stored, and the result is a scan of the R-tree that skips

sections corresponding to pruned subtrees. The difference between the two traversal strategies is illustrated in Figure 5.1. The next two lemmas state that this altered order of visiting nodes does not affect the correctness of the procedure and that DFSS computes the skyline of S I/O-efficiently.

Lemma 5.1. DFSS *correctly computes the skyline of the given point set S.*

Proof. DFSS, just as BBS, never prunes a point p in S that belongs to $\text{sky}(S)$. To see why this is true, consider such a point p. Since there is no point in S that dominates p, there is also no point in S that dominates any ancestor of p in T. In particular, there is no such point in the part of $\text{sky}(S)$ constructed before inspecting any given ancestor of p. This implies that no ancestor of p is pruned, and the traversal reaches p and adds it to L. This proves that the list L computed by DFSS is a superset of $\text{sky}(S)$.

To see that $L = \text{sky}(S)$, it remains to show that $L \subseteq \text{sky}(S)$, that is, that only skyline points are added to L. So consider a point $p \notin \text{sky}(S)$. Then there exists a point $q \in \text{sky}(S)$ that dominates p and, hence, has a smaller distance than p from the origin. As shown in the previous paragraph, DFSS adds q to L when it reaches the leaf storing q. By the ordering of the leaves of T, any traversal that visits p must visit q before p because q is closer to the origin than p. Hence, by the time p is visited, q already belongs to L, which prevents the addition of p to L. □

Lemma 5.2. *The cost of* DFSS *is bounded by the cost of sorting S once and sequentially scanning S $2 + 4f/[l(f-1)] \leq 6$ times, as long as $f, l \geq 2$, where f and l are the fanout and leaf-size of the R-tree respectively.*

Proof. The R-tree construction sorts S and then scans S and T once. In addition, every node in T causes two stack operations on R, one Push and one Pop.

The query procedure performs at most one complete traversal of the tree, amounting to another scan of T and S. Hence, S is scanned twice and the cost of manipulating T and R is bounded by that of four scans of T. The number of nodes in T is easily bounded by $(S/l) \cdot f/(f-1)$, which is at most $4S$, as long as $f, l \geq 2$. Summing the different costs now yields the lemma. □

Note that the bound in Lemma 5.2 is a worst-case upper bound. The practical performance is usually much better due to pruning. Yet, even in the presence of

pruning, the algorithm benefits from accessing the nodes in T in the order they are stored on disk, as these accesses can be thought of as a scan of T that skips over certain sections.

5.3 Presorted-BBS

The second approach is a direct I/O-efficient implementation of BBS. The key is to ensure that the order in which BBS inspects points corresponds to the order in which they are stored on disk. Fortunately, once the R-tree has been constructed, this order is easy to determine, as it is fully determined by the distances of the R-tree nodes from the origin. This leads to the following algorithm.

Sort the points in S by increasing distance from the origin and construct the R-tree over the sorted point set using the procedure from Section 4.3.1. Now sort the nodes of the constructed tree by increasing distance from the origin. Then apply BBS to the resulting tree layout.

Again, as the nodes are stored in the order they are visited by BBS, the cost of traversing the constructed tree can be bounded by the cost of a single scan of its node set. The construction of the R-tree using the procedure in Section 4.3.1 involves one scan of S and two scans of T. Unfortunately, the sorting of the nodes of T by increasing distance from the origin requires substantial book-keeping, as in addition to being placed in a new position, every node also has to be informed about the new positions of its children. Thus, sorting the nodes of T requires in fact three sorting passes over T and two scans of T. It is unclear whether the expected gain in the performance of the BBS query compared to the query procedure used by DFSS suffices to compensate for this additional overhead in preprocessing. Investigating this is a focus of our experiments discussed in the remaining sections.

5.4 Performance Evaluation

We conducted a set of experimental studies to compare the performance of our algorithms, DFSS and Presorted-BBS, on synthetic and real data sets. In the experiments discussed in this section, we obtained a substantial speed-up over internal-memory versions of our algorithms. However, in the case of Presorted-BBS, the

reduction in the number of visited nodes was not enough to compensate for the increased preprocessing cost when compared to DFSS, making DFSS the clear winner on datasets beyond memory size. We also compared our algorithms with the Sort-Filter Skyline (SFS) [50] algorithm (see Section 3.2). In SFS, the points are initially sorted according to a scoring function. This sorting procedure arranges the points in ascending order of their scores. Thus, points with lower scores are likely to dominate a large number of points and skyline points are found earlier in the scan of the point set.

Note that the skyline computation with our two algorithms involves (1) sorting the initial point set by increasing distance from the origin, (2) building the array-based R-tree, and (3) querying the R-tree to compute the skyline. The second step also includes, for the case of Presorted-BBS, the time required to sort the bounding boxes, plus the time to recalculate the new positions. In the case of SFS, the skyline computation consists of initially sorting the points with respect to their *volume* (product of their dimensions) values, followed by a scan of the point set to determine the skyline.

Next we describe our test environment, the data sets and the parameters used in our experiments.

Experimental platform. The algorithms, DFSS, Presorted-BBS and SFS, were implemented in C++ using STXXL [59], a library that provides I/O-efficient implementations of the data structures found in the C++ STL. During compilation, optimization level -O3 was used. The algorithms were evaluated on a PC with a 3GHz Intel Pentium 4 processor, 1GB of RAM, an 80GB 7200 RPM IDE disk, and running Fedora Core 6 Linux. For all experiments, all timing results denote the wall clock time taken by the algorithm to complete.

Datasets. We performed experiments using 1–5, 10, 15 and 30 million points in 6 dimensions with *uniform* and *anti-correlated* data. For *hydrological* data we used the same sizes, except for the largest one, which was 29 million instead of 30 million. Despite the fact that our graphs show only the results for 6-d skyline computation, the same relative performance of the algorithm was observed in 2–5 dimensions; 6-d is the computationally most expensive case in our experiments.

Tuning parameters. An `stxxl::vector` was used to implement the pointerless R-tree structure introduced in Section 4.3.1 and the `stxxl::sort` algorithm to perform the sorting step. An `stxxl::vector` is organized as a collection of blocks residing on disk and STXXL allows the specification of the block size for data transfers between disk and memory. We experimented with different block sizes and a block size of 8MB resulted in the best performance. This block size was used throughout our experiments. Two additional parameters control the amount of memory allocated to the LRU pager used by the vector to cache accessed blocks. The first parameter is the page size; data is swapped one page at a time. The second parameter is the number of pages to be cached. We set these parameters to 2 and 4 respectively. The sequential data accesses of the algorithms did not benefit substantially from a bigger cache, but this would have left less memory for the sorting algorithm. Likewise, we experimented with different parameters for laying out the R-tree. Using a fanout of 2 and a leaf size of 500 gave us the best overall running times.

In Section 5.4.1 we compare the performance of the external-memory (EM) algorithms against internal-memory (IM) methods and study the effects of increasing input size and skyline size on the algorithms. Afterwards, in Section 5.4.2, we study the behavior of the external memory algorithms in more detail.

5.4.1 Comparison of IM and EM Algorithms

The starting point in evaluating DFSS and Presorted-BBS, was to implement main memory versions of them, to observe the performance when using virtual memory and compare it with the performance when using an I/O-efficient algorithm instead. We use IM-DFSS and IM-Presorted-BBS to refer to the main memory methods, and EM-DFSS and EM-Presorted-BBS to refer to the external-memory ones.

Effect of the input size. As expected, our internal memory implementations perform very well as long as the data fits in memory; if the structures used fit entirely in memory, which is the case for 1–5 million records, the algorithm finishes in seconds and with a CPU utilization of almost 100% for the *hydrological* and *uniform* data. On

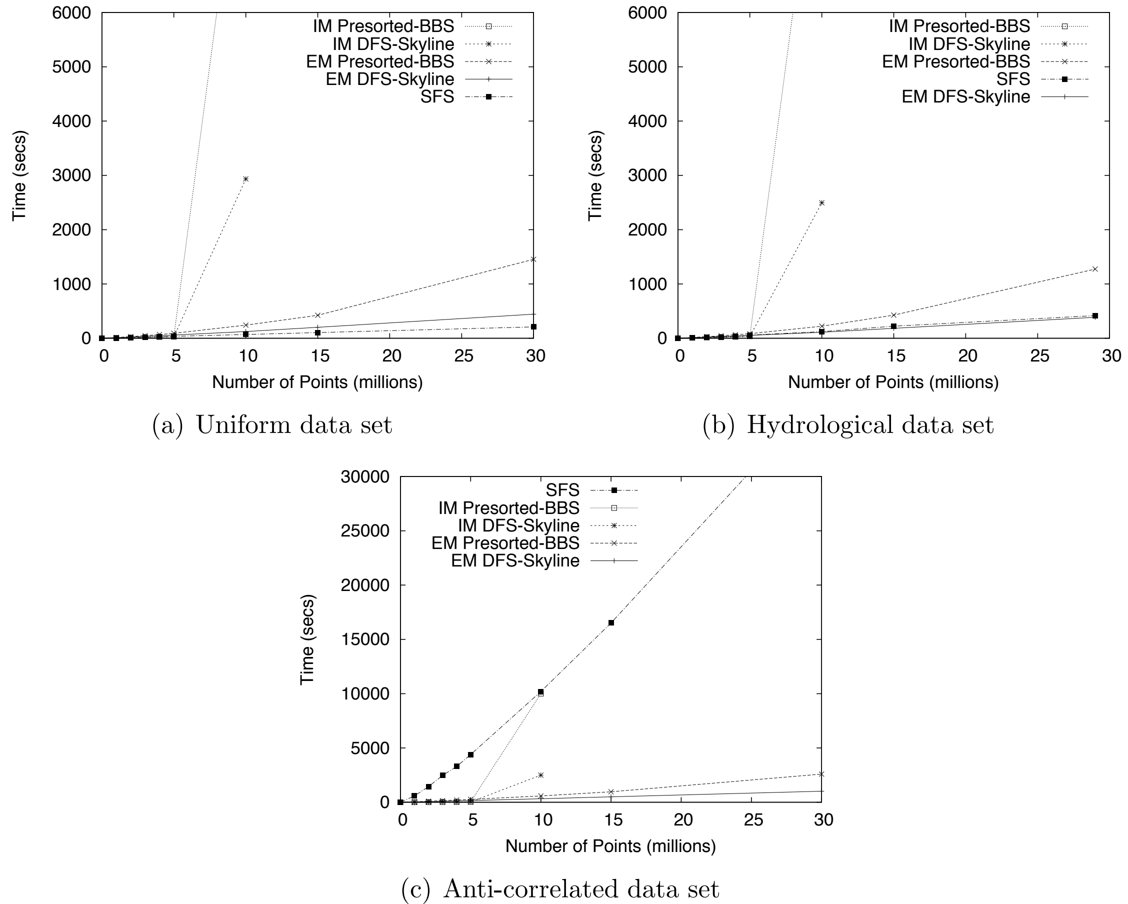


Figure 5.2: Total time results of DFSS, Presorted-BBS and SFS in external memory (EM), and results of internal memory (IM) variants of DFSS and Presorted-BBS. Labels are ordered top-down from fastest to slowest procedure.

the 10M data sets, we can see a sharp rise in the curves due to the use of virtual memory, causing running times to grow rapidly (see Figures 5.2a–c). The virtual memory system allows the address space to be much larger than what can fit in internal memory; if the required address is not in internal memory, a page fault is triggered. After exceeding the internal memory, IM-DFSS and IM-Presorted-BBS rely on virtual memory to handle page management, causing a severe performance degradation due to excessive page faults. For example. IM-DFSS takes 41 minutes to compute the skyline of 10M records of *hydrological* data, and IM-Presorted-BBS takes 2.7 hours approximately to do the same. For *uniform* data, IM-DFSS takes 48 minutes, whereas IM-Presorted-BBS needs 2.7 hours to process 10M records. For larger inputs the implementations were terminated after several hours of running without

having produced any result. In contrast, the running time of the external memory counterparts, continues to grow gradually with increasing input size. For example, EM-DFSS manages to compute the 6-d skyline of 29,000,000 records of real (*hydrological*) data in approximately 6 minutes, and EM-Presorted-BBS takes roughly 21 minutes. For the *uniform* data set and 30,000,000 records, EM-DFSS takes again 5 minutes to produce the skyline, and EM-Presorted-BBS does the same in around 13 minutes.

Data sets with an *anti-correlated* distribution again pose an interesting challenge (Figure 5.2c). The running times with these data sets are substantially greater. This is because compared to uniformly distributed data, anti-correlated distributed data increases the skyline sizes. The internal memory methods always have lower running times while they fit in main memory. On the other hand, we can see them using virtual memory when working on a greater data set. In general, in all the data distributions, we observe the same behavior: the internal memory methods cannot efficiently handle data sets beyond memory size, while the external memory algorithms behave better and their running times grow gradually with the input size.

Effect of the skyline size. Let us consider the effect that the size of the skyline has on the algorithms. Due to the inherent characteristic of the skyline computation, higher dimensionality implies a larger skyline and a larger skyline represents more computational work. By comparing Figures 5.2a–b to Figure 5.2c we can see the impact that a bigger skyline has on the performance of the algorithms. For instance, the performance on the uniform data is always better than the performance on the anti-correlated data because the skyline size has increased substantially. For the anti-correlated data sets used here, the final skyline sets were 97,891, 139,740, 171,562, 190,657, 212,865, 289,307, 345,089, and 460,817 points for 1M, 2M, 3M, 4M, 5M, 10M and 30M, respectively. These sizes are at least ten times bigger than their respective counterparts with the other two distributions. Independent of the preprocessing method, the algorithms must still compare each skyline point against every other one to verify it.

5.4.2 Comparison of External Memory Algorithms

Now that we have established that the internal memory algorithms are not efficient in processing data sets beyond memory size, let us study the performance of the external algorithms in more detail.

Overall performance. By looking at Figures 5.2a–c we observe that SFS outperforms any of the other external algorithms for small skylines; however, as soon as the size of the skyline increases, the performance of SFS degrades. DFSS and Presorted-BBS have the added cost of building an R-tree. On sets with small skylines, there is hardly any benefit from the pruning of the R-tree. Once the skyline gets larger, the pruning of R-tree nodes becomes effective and the added cost of constructing the tree pays off: DFSS and Presorted-BBS outperform SFS.

To see this behavior more clearly, consider a *uniform* distribution (Figure 5.2a). The results indicate that SFS is the fastest algorithm; this is not surprising, as these data sets have the smallest skylines—the skyline of the 30M data set contains approximately 15000 points. In this case, SFS finds the skyline points early in the scan, leading to fewer dominance tests. On the other hand, DFSS and Presorted-BBS need to construct the array-based R-tree—a costly operation with little gain in this case.

With *hydrological* data (Figure 5.2b), DFSS is the now fastest algorithm. The skyline size in this case is much larger (over 23000 points in the 30M case); hence, the performance of SFS starts to degrade due to an increase in the number of dominance tests. In contrast, DFSS now benefits more from pruning away nodes (and points) and not having to perform dominance tests on the whole point set.

Finally, Figure 5.2c shows that, with *anti-correlated* data, SFS's curve spikes up due to the high number of comparisons it has to perform. In contrast, our algorithms have significantly lower running times due to pruning of tree nodes, which translates into fewer comparisons.

Performance of the query procedure. Another performance measure is the query time. The query-only times for the three algorithms is shown in Figure 5.3; these timings do not consider any preprocessing or construction cost, they only reflect

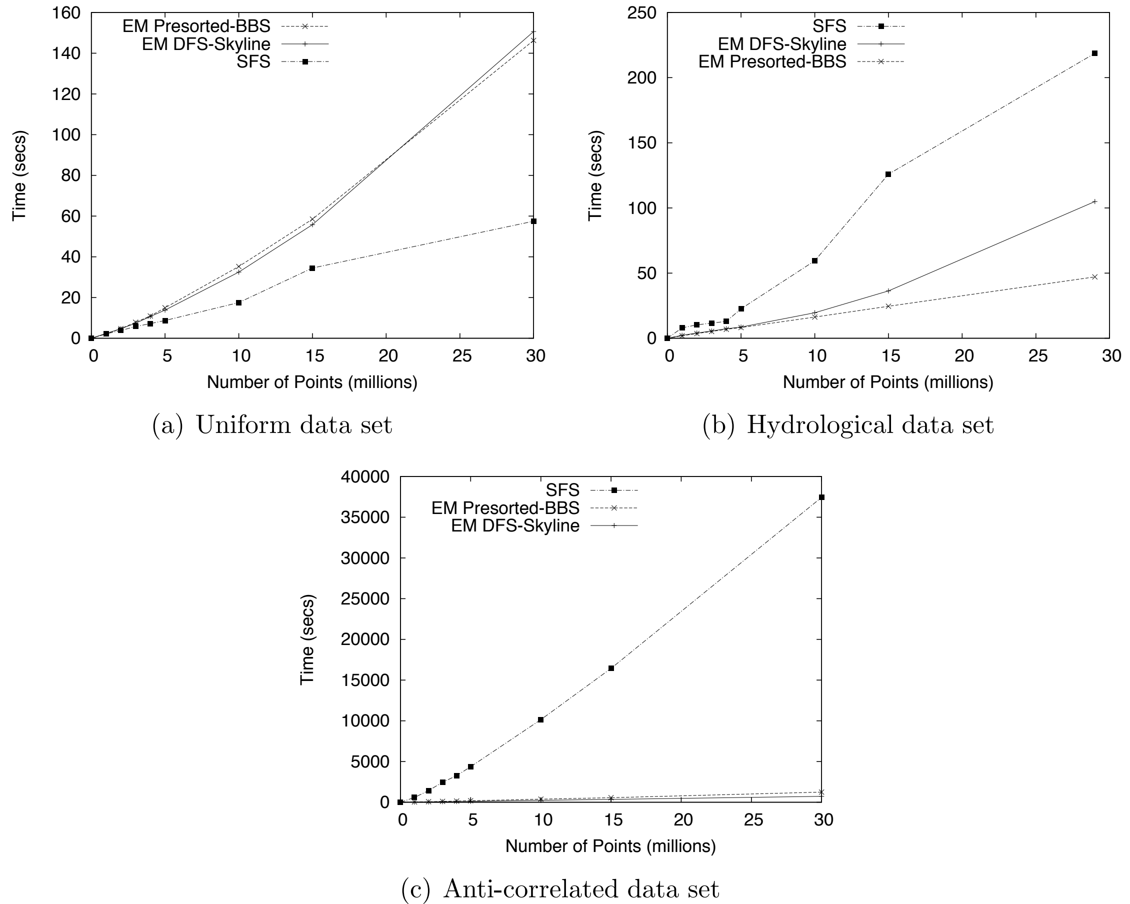


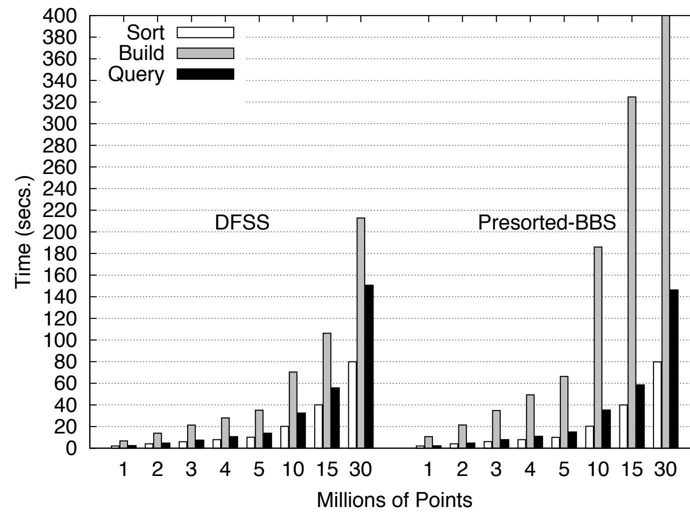
Figure 5.3: Only query time results of EM-DFSS, EM-Presorted-BBS and SFS. Labels are ordered top-down from fastest to slowest procedure.

the cost of the scanning (and computing the skyline) over the point set in the case of SFS, and tree traversal in the case of DFSS and Presorted-BBS. Ultimately, we use this measure to determine if there is any performance gain by the BBS query of Presorted-BBS when compared with the query procedure used by DFSS.

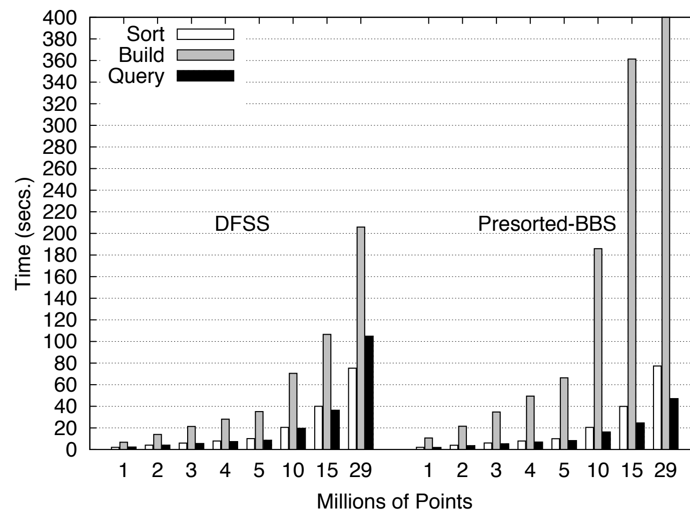
With *uniform* data (Figure 5.3a), SFS is the fastest algorithm due to the small skyline. However, with *hydrological* data (Figure 5.3b), the increase in the skyline's size becomes a factor against SFS's performance as its query times are slower by at least a factor of two than those of DFSS and Presorted-BBS. Overall, with these data sets, the query procedure of Presorted-BBS is slightly faster than DFSS.

For the case of *anti-correlated* data (Figure 5.3c), the query time of SFS is clearly much higher, thus, the lower preprocessing time of it does not compensate for its high

skyline computation cost. On the other hand, our algorithms have much better running times on this distribution, with DFSS being slightly better than PRESORTED-BBS.



(a) Uniform Data



(b) Hydrological Data

Figure 5.4: Break-down of a $6d$ query. Note: for visualization purposes the Build time of PRESORTED-BBS with 29|30M points was cutoff at 400 secs., while its value is 1228 and 1150 for uniform and hydrological data sets respectively.

Comparison of PRESORTED-BBS and DFSS. Given that PRESORTED-BBS is slightly faster than DFSS in the query procedure for uniform and hydrological data sets, we now take a closer look at both algorithms and determine if the lower

query times of Presorted-BBS suffice to compensate for its additional overhead in preprocessing. Figure 5.4 provides insight into this comparison by showing the added preprocessing cost of Presorted-BBS and DFSS using results with uniform and hydrological data. We observe that indeed Presorted-BBS gets faster than DFSS in the query part, but it pays for it with a much bigger increase in the R-tree construction; the sorting part, however, remains exactly the same in both algorithms. For example, the R-tree construction cost with 15M data set is 3 times higher for Presorted-BBS than for DFSS, while the query time is only slightly faster for Presorted-BBS. The additional overhead in preprocessing explains why, in terms of the overall running time, DFSS is much faster than Presorted-BBS.

In general, SFS is the fastest algorithm for small skylines, as is the case with uniform data sets. In contrast, DFSS and Presorted-BBS perform much better with data sets having larger skylines as, for example, with anti-correlated and hydrological data sets. DFSS, overall, is the fastest of our two methods. The experimental evaluation confirmed that, in addition to being scalable, DFSS achieves very good performance with increasing data set sizes.

Chapter 6

I/O-Efficient Algorithms for Massive Graphs

In this chapter, we motivate our work on I/O-efficient algorithms for directed graphs, introduce the terminology and concepts used in this part of the thesis, and give an overview of previous work on I/O-efficient graph algorithms. We close this chapter with a summary of the contributions of this second part of the thesis, which are presented in detail in Chapters 7 and 8.

6.1 Overview

Graphs are ubiquitous in computer science as a means for modelling relationships between entities. Entities are represented by vertices, and relationships between entities by edges. Graphs arise naturally in a wide variety of scientific and real-world applications, including web modelling [41,65,66,87,90], computational biology [126,137,138], and geographic information systems [13,23,58]. For example, search engines analyze the web graph to discover web communities, recent multiple sequence alignment algorithms are based on manipulating the de Bruijn graph of the sequences, and route planning systems model road networks as graphs and find optimal routes by solving shortest path problems on them.

In recent years, the amount of data available in these applications has caused a massive increase in the size of the underlying graphs (see Section 6.1.2 for examples of real-world massive graphs). Due to their size, such graphs cannot be held entirely in memory and need to reside (at least partially) on disk. When working with such large graphs in a disk-based setting, the I/O communication generated by traditional (internal memory) graph algorithms becomes a bottleneck. The reason is that the graph exploration strategies (e.g., depth-first search, breadth-first search) essential in all traditional graph algorithms are inefficient on massive graphs (see Section 6.2.1). This has led to a focus on developing I/O-efficient algorithms for a range of graph problems.

Most previous work on I/O-efficient graph algorithms has focussed on developing provably efficient solutions for *undirected* graphs and special graph classes; this work has led to the development of important techniques for designing I/O-efficient graph algorithms (see Section 6.2.2). Directed graphs pose a much greater challenge and virtually no theoretical nor algorithm engineering results on I/O-efficient algorithms for directed graphs are known, except for special graph classes. This lack of results, both in theory and practice, motivates the study of heuristic approaches for processing directed graphs I/O-efficiently. Most notably, Sibeyn et al. [121] proposed a depth-first search (DFS) heuristic that performs extremely well if the vertex set of the graph fits in memory. Since DFS is the basis for many classical graph algorithms, this heuristic forms the basis for I/O-efficient solutions to a range of problems.

In this part of the thesis we propose I/O-efficient algorithms for solving problems on massive directed graphs that fall into this category of efficient heuristics. In the worst case, their performance is poor, but in practice they perform very well and can efficiently process graphs beyond the reach of existing algorithms, including algorithms based on the DFS heuristic of [121]. Specifically, we address two important problems for directed graphs: computing *strongly connected components* and *topological sorting*. While Sibeyn et al. used these problems merely as examples to demonstrate the efficiency of their DFS procedure, we propose I/O-efficient algorithms specifically for computing strongly connected components and topological sorting. Computing strong connectivity and topological sorting are the two most fundamental connectivity questions one can ask about a directed graph and we develop practically efficient solutions for them. Next, we review terminology and notation used in this and the following chapters.

6.1.1 Terminology and Definitions

A graph is a pair $G = (V, E)$ of a set V of *vertices* and a set E of *edges*. We use $n := |V|$ to denote the number of vertices and $m := |E|$ to denote the number of edges in G. Each edge $e \in E$ is a pair (v, w), for some $v, w \in V$, where we call v and w the *endpoints* of e. If $(v, w) \in E$, then v and w are *incident* with the edge (v, w), and we also say that v and w are *adjacent* or *neighboring* vertices of G. The *degree* of a vertex v is the number of neighbors of v.

A *path* from v to w in a graph G is a sequence of vertices $x_1, x_2, x_3, \ldots, x_n$ such that $(x_i, x_{i+1}) \in E$, for all $1 \leq i \leq n$, $x_1 = v$ and $x_n = w$. A *cycle* is a path where the first vertex and the last one are the same.

We say that $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. In this case, we write $G' \subseteq G$. A *spanning graph* of G is a subgraph of G that contains all vertices of G.

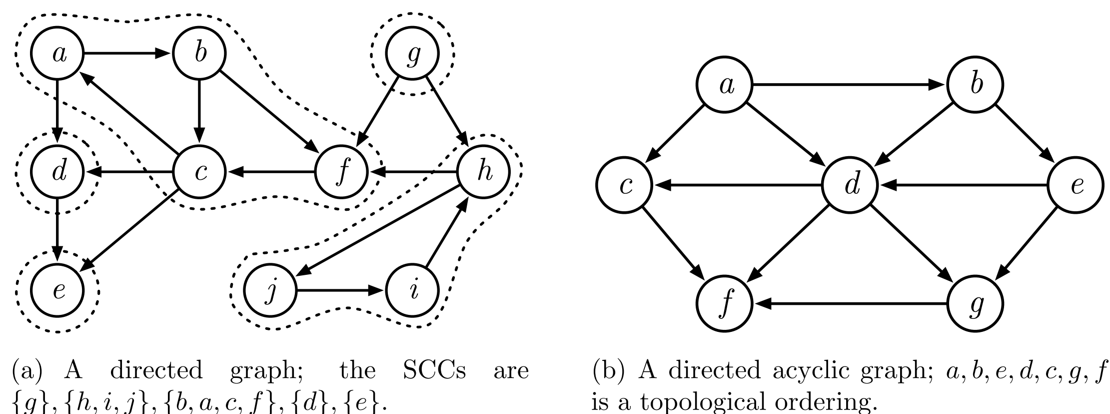
A graph G is called *connected* if every pair of vertices is connected by a path. A maximal connected subgraph of G is called a *component* or a *connected component* of G. An acyclic graph, one containing no cycles, is called a *forest*. A connected forest is called a *tree*. The vertices of degree 1 in a tree are its *leaves*. A *spanning tree* of G is a spanning graph that is a tree. A graph has a spanning tree only if it is connected. A *spanning forest* of G is a spanning graph whose connected components are spanning trees of the connected components of G

We say a graph is *directed* if its edges are *ordered* pairs, that is, edges (v, w) and (w, v) are different edges $((v, w) \neq (w, v))$. If the pairs are unordered, that is, edges (v, w) and (w, v) are considered to be the same edge $((v, w) = (w, v))$, then the graph is *undirected*.

For an edge $e = (v, w)$ of a directed graph, we call v the *tail* and w the *head* of e. An *in-edge* (resp. *out-edge*) of v is an edge with v as its head (resp. tail). An *in-neighbor* (resp. *out-neighbor*) of v is the tail (resp. head) of an in-edge (resp. out-edge) of v. The *in-degree* (resp. *out-degree*) of v is the number of its in-edges (resp. out-edges).

A directed graph $G = (V, E)$ is *strongly connected* if, for every vertex pair (v, w), there exists a directed path from v to w. The *strongly connected components* (SCCs) of a graph are its maximal strongly connected subgraphs (SCSGs). Figure 6.1a shows a directed graph whose strongly connected components have vertex sets $\{g\}$, $\{h, i, j\}$, $\{b, a, c, f\}$, $\{d\}$ and $\{e\}$. All pairs of vertices in each component are mutually reachable. On the other hand, the vertices d and e, for example, belong to different SCCs since vertex d cannot be reached from vertex e.

A *topological ordering* of a directed graph $G = (V, E)$ is an ordering of its vertices such that for every edge $(v, w) \in E$, v precedes w in the ordering. See Figure 6.1b. A graph may have more than one topological ordering. For example a second topological



(a) A directed graph; the SCCs are $\{g\}, \{h, i, j\}, \{b, a, c, f\}, \{d\}, \{e\}$.

(b) A directed acyclic graph; a, b, e, d, c, g, f is a topological ordering.

Figure 6.1: Illustration of directed graphs.

ordering for the graph in Figure 6.1b is a, b, e, d, g, c, f. A directed graph has a topological ordering if and only if it is *acyclic*, that is, does not have any directed cycles.

Note that topological ordering and strong connectivity are orthogonal concepts, as the SCCs of an arbitrary directed graph form a directed acyclic graph (DAG).

6.1.2 Massive Graphs in the Real-World

I/O-efficient graph algorithms are motivated by, and find applications in, a range of application areas dealing with massive graphs. The following is a list of examples of such applications with a particular focus on ones that deal with directed graphs and may benefit from the algorithms developed in Chapters 7 and 8.

Web graphs. Web crawls represent snapshots of the World Wide Web and can produce graphs with billions of edges. In these graphs vertices represent the web pages and edges are the hyperlinks between them. A large amount of research has focused on studying these graphs [41, 65, 66, 87, 90] in order to get some insight into the web's topological properties.

A typical problem in the analysis of web graphs is to identify web communities, which is the problem of finding clique-like structures. As a first approximation one may also consider strongly connected components to be communities. Some studies have analyzed the distribution of the sizes of the strongly connected components in the graph, and discovered that the degree distributions follow a

power-law, and the graph has a bow-tie structure [65, 66].

Computing PageRank [40] (the basis of Google's search engine) is also considered to be a challenging problem with respect to web graphs. PageRank is a measure used for computing the relative importance of web pages for searching. Since the web graph is continuously evolving, a problem here is to efficiently compute PageRank on such graphs [48, 62].

Terrain graphs. Remote sensing technology has permitted the acquisition of enormous amounts of high-resolution terrain data. For example, NASA's Earth Observing System Data and Information System (EOSDIS) holds a collection of satellite data comprising 4.2 petabytes [10]. Laser based LIDAR scanning technology can map the earth's surface at a very high resolution; for example, a 2m resolution map of Denmark is about 1.7 terabytes in size [104]. Geographic Information Systems (GIS), typically store terrains as elevation models, either in the form of Triangulated Irregular Networks (TINs) or as two-dimensional grids. In a TIN, the terrain is represented by a planar triangulation, naturally forming a graph. In a grid, the terrain can be viewed as a grid graph whose vertices correspond to grid cells and and whose edges connect vertices that correspond to neighboring cells.

Terrain analysis is central to a range of important GIS applications concerned with the effects of topography. Typical problems here are flow routing and flow accumulation [23, 104]. Solving such problems can, for example, reveal areas areas susceptible to floods or predict the location of streams. Searching for the optimal route is also important in GIS applications for route planning, where road networks are modelled as graphs and shortest-path-like computations are performed on them [58].

Graphs in GIS are often planar, which has motivated a significant amount of work in I/O-efficient algorithms for planar graphs (e.g., [24, 27, 28, 95]).

Social networks. Social networks are usually represented in terms of graphs with the vertices representing entities (e.g., individuals, organizations) and the edges representing ties or relationships (e.g., friendship, trust, common interest, financial transactions). Social networking web sites such as Facebook and LinkedIn

generate massive graphs and are continuously evolving. LinkedIn has over 80 million members [5] and Facebook alone currently has over 500 million active users [3] and continues to grow.

Similar to web graphs, a typical problem in the analysis of social network graphs is maximum clique detection [11] to find communities of people [82]. In financial transaction networks, grouping people based on such information can be potentially useful in detecting money laundering rings [92]. Security intelligence applications use this information as well to identify key players in terrorist networks [52].

Call graphs. Telephone call graphs are formed by logging phone calls over a certain period of time. In this graph, vertices represent telephone numbers and there is a directed edge between two vertices if there has been a call from one number to another within the observed time frame. The call graphs created by telecommunications companies can be massive. For example, a company like AT&T generates 4 billion phone call records per day—with over 400 million unique telephone numbers—including 2 billion text messages [51], this amount of data easily makes the size of these graphs reach several terabytes.

A typical problem on telephone call graphs is identifying local communities (e.g., by detecting the presence of bipartite cores) [107]. Services providers can use this information to target them with better incentives for retention. Another application is identifying fraudulent behaviour [33, 51]. Some studies have also reported on various topological properties of these massive call graphs, including degree distributions, strongly connected components, and bipartite cores [107].

Biological graphs. Due to the scientific advances in DNA analysis, more precisely in the field of genomics, large genomes (e.g. human, mouse) are completely available. For example, the Broad Institute sequencing center [2] generates large-scale genomic data in the order of 50 billion nucleotide bases a year [38]. Graphs provide a powerful way for modelling this kind of biological data (e.g., biological pathways [126] and protein interaction networks [112]). In these graphs, vertices represent cellular entities (proteins, genes, mRNA, etc.) and edges correspond to interactions between them.

A problem arising in the study of these graphs is approximate graph matching [126, 127]. For example, protein interaction networks for individual species are often matched to determine similarities and differences across species.

Another important problem in comparative genomics is multiple sequence alignment. Recent multiple sequence alignment algorithms [137, 138] operate on the sequences' de Bruijn graph and reduce the problem to a traversal of an acyclic subgraph of this graph. Knuth Reinert's group in Berlin is developing a library of computational biology algorithms [72].

6.2 State of the Art

Massive graphs pose major challenges mainly because the graph exploration strategies at the heart of the vast majority of graph algorithms seem inherently inefficient in external memory. Section 6.2.1 discusses the main difficulties for I/O-efficient graph exploration. As a result, problems on massive graphs have been attacked from two different directions: on one hand, attempting to develop I/O-efficient graph exploration methods and, on the other hand, solving graph problems using alternative strategies not based on graph exploration. Section 6.2.2 gives an overview of techniques used in both approaches.

6.2.1 Difficulties with I/O-Efficient Graph Exploration

Traditional internal-memory graph algorithms often analyze the structure of a graph using graph exploration. Breadth-first search (BFS) and depth-first search (DFS) are two such exploration strategies that are easy to implement, are fast as long as the graph fits in memory, and surprisingly provide much information about the structure of the graph. The latter is true particularly for DFS. On the other hand, not much is known about implementing these exploration strategies I/O-efficiently.

There are two main challenges associated with using traditional graph exploration methods for computation on graphs stored on disk. (1) Remembering visited nodes results in one disk I/O per edge in the worst case. (2) Random accesses to the adjacency lists of the vertices may result in one disk I/O per vertex. Thus, typical implementations of DFS, for example, end up performing $O(n+m)$ I/Os in the worst

case, which is extremely inefficient on graphs beyond memory size.

The lack of locality in their data access patterns is a problem for all graph exploration strategies, at least on directed graphs. Due to the strong reliance on such graph exploration strategies in traditional graph algorithms, even simple problems, such as topological sorting, become challenging on massive graphs.

6.2.2 Techniques for I/O-Efficient Graph Algorithms

Due to the inefficiency of classical graph exploration algorithms in an external setting, problems on massive graphs have been addressed using two different approaches: on one hand, attempting to develop I/O-efficient graph exploration algorithms and, on the other hand, solving graph problems using alternative techniques. This section reviews the techniques used in both approaches. Specifically, Subsection 6.2.2.1 discusses techniques for speeding up graph exploration, and Subsection 6.2.2.2 explores non-exploration-based techniques.

6.2.2.1 Techniques for Speeding Up Graph Exploration

The techniques presented in this subsection address the two problems with graph exploration: remembering visited vertices and random access to adjacency lists.

The buffered repository tree. The buffered repository tree (BRT) [42] addresses the problem of remembering visited vertices in graph traversals. It allows the insertion of edges and the extraction of all edges with a given tail. A visited vertex v inserts all its in-edges into the BRT and can distinguish which of its out-edges lead to previously visited vertices by extracting all edges with tail v. Using this data structure Buchsbaum et al. [42] obtained directed BFS and DFS algorithms with I/O complexity of $\mathrm{O}((n + \frac{m}{B}) \log n)$.

Clustering. For directed graphs, the BFS and DFS algorithms of [42] are the best known, both because there is no known method for tracking visited vertices than the BRT and because there is no known method for avoiding random accesses to adjacency lists. The same is true for undirected DFS. For undirected BFS and shortest paths, it is easier to track previously visited vertices, making the random accesses to adjacency

lists the only problem. This led to an undirected BFS algorithm with I/O complexity $O(n+\text{sort}(m))$ [106] and a single-source shortest path algorithm with I/O complexity $O(n + \frac{m}{B}\log_2 \frac{n}{B})$ [88]. To overcome the problem with random accesses to adjacency lists in BFS, Mehlhorn and Meyer proposed a clustering-based approach [98]. The idea is to form $o(n)$ groups of vertices that are close to each other in the graph. When the first vertex in such a vertex cluster is visited, the adjacency lists of all vertices in the cluster are loaded into a hot pool. Adjacency lists are accessed by scanning the hot pool. Thus, this approach trades random accesses for a higher scanning cost of adjacency lists (in the hot pool). By choosing the parameters of the cluster partition carefully, this leads to a BFS algorithm with I/O complexity $O(\frac{\sqrt{n \cdot m}}{B} + MST(n, m))$, where $MST(n, m)$ is the cost of computing a (minimum) spanning tree of the graph. Meyer and Zeh extended these ideas to shortest paths [102, 103]. These ideas seem entirely ineffective for DFS on directed graphs.

Graph separators and divide-and-conquer. The algorithms discussed so far are efficient for dense graphs ($m = \Omega(Bn)$), with I/O complexity of $O(\text{sort}(m))$ or $O(\frac{m}{B}\log n)$. The (mostly unsolved) challenge is obtaining efficient algorithms for sparse graphs. For some special graph classes, most notably planar graphs, graph separators have proven useful for obtaining (nearly) I/O optimal algorithms for BFS [21], DFS [24], shortest paths [93] and a number of other problems [26–28]. Planar graphs, have the property that there exists a set of $O(n/\sqrt{r})$ "separator vertices" whose removal breaks the graph into connected components of size at most r. For $r = B^2$ and $M \geq B^2$, each such piece fits in memory. By processing each piece in turn, one can construct a graph on the separator vertices that has $O(n)$ edges and captures the interaction between these vertices. For example, for shortest path computations, each separator vertex has the same distance from s—the source vertex—in this compressed graph as in the original graph. The compressed graph is dense, which allows the efficient solution of a range of problems on this graph. The final solution is then obtained by processing each memory-sized subgraph of the original graph in turn. For example, for shortest paths, the distances from s to every vertex in such a piece can be computed from the distances of the separator vertices on the boundary of the piece.

For DFS on planar *directed* graphs, a different type of separator, called a path separator, has been employed to recursively partition the graph in a balanced fashion, leading to a divide-and-conquer algorithm for this problem, with I/O complexity $O(\text{sort}(n) \log \frac{n}{M})$ [28].

6.2.2.2 Non-Exploration-Based Techniques

Motivated by the difficulty of I/O-efficient graph exploration, a number of techniques to solve graph problems without using graph exploration have been developed, particularly for undirected graphs. These techniques are often borrowed from parallel algorithms for the same problem. In this section, we review them and discuss their applicability.

Graph contraction. The key idea in graph contraction is to reduce the size of the input graph G while preserving the properties of interest (e.g., connectivity, planarity). This is achieved by identifying edge-disjoint subgraphs of G, contracting each subgraph and representing it by a graph of smaller size (typically a single vertex). Commonly, this procedure is applied recursively until the number of vertices is reduced to memory size, at which point an efficient semi-external algorithm is used to solve the problem on the contracted graph—a semi-external algorithm can process the edges of the graph I/O-efficiently if the vertices fit in memory. A solution for G is constructed by undoing the contraction. The basic operation is edge contraction, which means that the edge is replaced with a vertex and all edges incident to either one of the endpoints are updated.

Graph contraction was initially studied in the context of parallel algorithms and was extended to external memory for solving connectivity problems on undirected graphs. For example, it has proven useful in the development of I/O-efficient algorithms for computing the connected components [12,49,106,123], biconnected components [49] and minimum spanning trees [12,21,61], where the connectivity information is preserved during edge contraction. Graph contraction has also been used to compute list ranking I/O-efficiently [49] (see below).

The graph contraction approach works remarkably well in external memory due to the fact that vertices do not need to be processed in any particular order, and each

contraction step can be implemented efficiently. However, for the case of directed graphs, the challenge is how to select the edges to contract while maintaining the properties of interest. The main problem is that edge contraction does not necessarily preserve the reachability of vertices in a directed graph: for two vertices v and w with no path from v to w in the original graph, a directed path from v to w may exist in the contracted graph.

List ranking. A list L is a collection of vertices $x_1, x_2, \ldots, x_n$, such that each vertex x_i, except the last one (tail), stores the ID of its successor $succ(x_i)$ in L, no two vertices have the same successor and every vertex can reach the tail by following the successor pointers. In the list ranking problem we want to compute, for every vertex x_i, its distance from the tail (or head) of L, i.e., the number of edges on the path from the head of L to x_i or from x_i to the tail of L.

A number of algorithms solving this problem using $\mathrm{O}(\mathrm{sort}(n))$ I/Os have been proposed [49, 123]. The practically fastest one is due to Sibeyn [123].

Euler tour. An Euler tour of a tree $T = (V, E)$ traverses every edge exactly twice, once in each direction. Such a traversal produces an ordering of the vertices or edges capturing the structure of the tree, and can be computed quite easily using $\mathrm{O}(\mathrm{sort}(n))$ I/Os.

The Euler tour technique and list ranking have also been used to break a spanning tree of a graph into $\mathrm{O}(n/\mu)$ pieces whose vertices have distances at most μ from each other. This is exactly the clustering used in the BFS algorithm of [98] discussed earlier.

List ranking, in combination with the Euler tour technique can be used to solve a wide range of problems on trees [94]. One example is the rooting of an undirected tree T, which is to direct all edges of T away from a root vertex given as part of the input. Given a rooted tree T, the Euler tour technique and list ranking can be used to compute a preorder and postorder numbering of the vertices of T, or the sizes of the subtrees rooted at the vertices of T, and a number of other problems.

Time-forward processing. Time-forward processing [20, 49] is a technique used to solve the following "graph evaluation" problem: given a DAG each of whose vertices

has a label $\phi(x)$, process its vertices in topologically sorted order and compute for each vertex x, a new label $\psi(x)$ from $\phi(x)$ and the ψ-labels of its in-neighbours. A simple example of this type of problem is the evaluation of a Boolean circuit represented as a DAG: $\phi(\cdot)$ assigns a Boolean function to each vertex, turning it into a logical gate; $\psi(x)$ is the output of the gate represented by vertex x, given the inputs it receives from its in-neighbours.

Arge developed a practical $\mathrm{O}(\mathrm{sort}(n))$ algorithm [20] which makes use of a priority queue for solving this problem, improving on an earlier solution by Chiang et al. [49]. The topological sorting algorithm we propose in Chapter 8 uses the time-forward processing algorithm of Arge as building block; we therefore discuss it in more detail next.

The basic idea in the priority-queue based approach is that when we compute the value for a vertex v, we insert the computed result of the function into the priority queue with priority w, for each edge (v, w) in the DAG. When we process vertex w, all elements with lower priority have already been processed, since we processed the vertices in topological order, and we can extract the inputs for vertex w from the priority queue, using DeleteMin operations.

Time-forward processing requires the vertices of the DAG to be given in topologically sorted order. Since no I/O-efficient topological sorting algorithm is known to date, time-forward processing has been applied only in situations where a topological ordering of the vertices can be obtained by using secondary information about the structure of the DAG. Nevertheless, it has been used successfully in solutions to a number of problems, including shortest paths on planar graphs [96], BFS and DFS on outerplanar graphs [95], and flow computations on grid graphs [23].

6.2.3 Engineering of Graphs Algorithms

The techniques discussed above give us the tools to approach a wide variety of problems for undirected graphs, but none of them allow us at the moment to make any progress on directed graphs at least in the worst case. Nevertheless, in many application areas dealing with massive graphs, particularly web modelling and the study of other types of social networks, the graphs are directed, and much valuable information is discarded if the edge directions are ignored. Additionally, while much theoretical

work has focused on solving graph problems I/O-efficiently using these techniques, much less is known about the practical performance of the developed algorithms even on undirected graphs. The main reason is their algorithmic complexity and the constant factors involved. In many cases there are no publicly available implementations of the used primitives—list ranking, Euler tour construction, etc. This makes implementing any I/O-efficient graph algorithm a formidable task, as it requires the implementation not only of the actual algorithm to be tested but also of a wide range of more elementary building blocks.

Results for directed graphs. The lack of provably efficient algorithms for solving problems on general directed graphs has motivated work on heuristic techniques. The most successful effort so far in this regard is the semi-external DFS algorithm by Sibeyn et al. [121]. Since DFS is a central building block used in many classical graph algorithms, the algorithm of [121] provides a general tool for solving problems on directed graphs efficiently if the vertices fit in memory. If, on the other hand, the size of the vertex set exceeds the memory size, the performance of the algorithm deteriorates to that of an internal-memory DFS algorithm. For directed graphs, the semi-external DFS algorithm by Sibeyn et al. [121] is the only work we are aware of that focuses specifically on solving fundamental problems on directed graphs.

Results with undirected graphs. In contrast to the lack even of practical results for general directed graphs, the set of implementations of algorithms for undirected graphs is growing. The existence of libraries of I/O-efficient algorithms has also contributed to the implementation of I/O-efficient algorithms for undirected graphs. Examples of these libraries include TPIE [25, 129] and STXXL [59, 60] which allow the implementation of I/O-efficient algorithms using high-level primitives and provide fundamental data structures such as stacks, priority queues and search trees. For example, by building on top of STXXL, several efforts have been made to engineer and evaluate the practical performance of algorithms for problems on undirected graphs in recent years. Ajwani et al. [17, 18] compared experimentally the BFS algorithms of Munagala and Ranade [106] and Mehlhorn and Meyer [98], and demonstrated that the BFS algorithm of Mehlhorn and Meyer achieves very good performance on a number of graph classes if implemented carefully. Dementiev et al. [61] engineered

a Minimum Spanning Tree (MST) implementation based on ideas from the external connected components algorithm by Sibeyn [123]. Their algorithm is theoretically inferior to the MST algorithms of [12, 21, 49] but performs extremely well in practice. Schultes [118] provided implementations for computing connected components and spanning forests also based on ideas from Sibeyn's algorithm. Meyer and Osipov [100] engineered a practical shortest path algorithm for general undirected sparse graphs.

Other related work includes a large body of work on preprocessing large graphs, particularly road networks, for fast shortest-path queries. The most recent results in this area, which include references to the earlier papers, include [32, 70, 73, 115].

6.3 Contributions

Computing strongly connected components and topological sorting are important building blocks for algorithms in a number of application areas. For example, sequence analysis using de Bruijn graphs first identifies and eliminates cycles in these graphs and then topologically sorts the resulting graph to prepare it for further processing [137, 138]. Traditionally, these problems are approached using graph exploration strategies. However, as we have seen in this chapter, graph exploration is very difficult in external memory, and thus, these problems become challenging on massive graphs.

In this part of the thesis we engineer algorithms for computing the strongly connected components and topological sorting of massive directed graphs. These are the two most fundamental connectivity questions one can ask about a directed graph. Our goal is (a) to provide fast algorithms for these problems that can handle larger graphs than a semi-external approach [121] and outperform any internal-memory algorithm on large inputs, and (b) to identify the techniques that are also useful for solving problems on directed graphs, at least for designing heuristics.

The SCC algorithm presented in Chapter 7 is based on graph contraction: it identifies and contracts strongly connected subgraphs until the graph is small enough to fit completely in memory, at which point we can compute its strongly connected components using a standard DFS-based algorithm. We investigate its behavior on a variety of graph classes. The results presented here show that our algorithm outperforms the method of [121] and can efficiently process bigger graphs than their

semi-external approach. Our results demonstrate that graph contraction, which is the key to success for connectivity problems on undirected graphs, can be used at least as a heuristic to solve strong connectivity. The results have appeared in [55].

In Chapter 8 we present our topological sorting algorithm. It starts by computing an initial ordering of the vertices that satisfies a subset of the edges. Through iterative improvement, which uses techniques such as time-forward processing, the set of satisfied edges is grown until the obtained numbering satisfies all edges, that is, is a topological ordering. The results we present in this thesis show that our algorithm is not only competitive to the method of [121] in a semi-external mode, but it is also able to process large graphs where neither the vertices nor the edges fit in internal memory. We also compare our algorithm to external-memory adaptations of parallel and internal-memory topological sorting algorithms that can be considered natural competitors to our approach. The results have recently appeared in [16].

Chapter 7

External-Memory Strong Connectivity

Identifying the SCCs is one of the most fundamental structural questions one can ask about a directed graph. In this chapter we present a contraction-based algorithm for computing the SCCs of a massive directed graph. The intuition behind our algorithm is that we hope to reduce the size of the input graph G until it fits in memory by repeatedly finding and contracting non-trivial strongly connected subgraphs. By contracting strongly connected subgraphs into a single vertex we do not change the connectivity properties of the input graph. Hence, once the graph fits in memory, we can use an efficient internal-memory strong connectivity algorithm to finish computing the SCCs of G.

Our experiments confirm that the algorithm performs remarkably well in practice. The strongest competitor is the semi-external algorithm by Sibeyn et al. [121]. Our algorithm substantially outperforms the algorithm of [121] on most of the graphs used in our experiments and never performs worse. It thus demonstrates that graph contraction, which is the most important technique for solving connectivity problems on *undirected* graphs I/O-efficiently [12,21,49,106], can be used to solve such problems also on *directed* graphs, at least as a heuristic.

7.1 Introduction

In internal memory the SCCs of a graph can be computed in linear time, e.g., using algorithms by Kosaraju's [53], Tarjan's [125] or Dijkstra's [64]. All of them require a depth-first traversal of the graph and, thus, incur $O(n+m)$ I/Os when run in external memory, as discussed in Section 6.2.1

This chapter is organized as follows. In Section 7.2 we give a detailed description of the algorithm. In Section 7.3 we lay the groundwork for the different primitives used in its implementation. Finally, in Section 7.4 we present our experimental results on various synthetic and real graphs.

7.2 A Contraction-Based Strong Connectivity Algorithm

This section describes our contraction-based SCC algorithm, referred to as EmSCC throughout this chapter. This algorithm consists of two phases: a *preprocessing* phase and a *contraction* phase. The contraction phase looks for SCSGs in the input graph G and contracts each into a single vertex, thereby reducing the size of G without altering its connectivity. This process continues until the graph fits in memory, at which point the algorithm loads it into memory and computes its SCCs using an internal-memory algorithm. In this sense, EmSCC resembles the connectivity algorithm for undirected graphs by Chiang et al. [49]. In the undirected case, however, the graph is *guaranteed* to fit in memory after a logarithmic number of contraction steps, while, in the directed case, the algorithm succeeds only if each round finds sufficiently many and large SCSGs to contract.

The contraction phase searches for SCSGs by loading memory-sized subgraphs of G into memory and computing their SCCs. The preprocessing phase tries to arrange the vertices and edges of G so that the chance of finding non-trivial SCCs in these subgraphs is maximized.

Next we discuss these two phases in more detail. Throughout this discussion, we use n and m to refer to the numbers of vertices and edges in the graph, respectively; M is the size of the main memory. Furthermore, we assume that the input graph is connected. It is not hard, however, to extend the algorithm to disconnected graphs with little or no impact on its performance.

7.2.1 Preprocessing Phase

The preprocessing phase of EmSCC is conceptually simple. It arranges the vertices of G in a list V_0 in the order of their first occurrences along an Euler tour of a spanning tree T of G. See Figure 7.1 for an illustration. It stores the edges in a list E_0, which is the concatenation of "one-sided" adjacency lists of the vertices in V_0 arranged in the same order as the corresponding vertices in V_0. The adjacency lists are one-sided in the sense that an edge (x, y) is stored in the adjacency list E_x of x if $x > y$, and in E_y otherwise; vertices are compared by their positions in V_0.

The contraction phase discussed in Section 7.2.2 below sweeps the two lists V_0 and

Algorithm 4: PREPROCESS(G)

Input: $G = (V, E)$.
Output: Vertex list V_0, edge list E_0.

1 Compute a spanning tree T of G;
2 Compute an Euler tour L of tree T;
3 Compute the rank of every edge e in L;
4 Store in V_0 the vertices of G numbered and ordered according to their first occurrences in the tour;
5 Store in E_0 the concatenation of one-sided adjacency lists of the vertices in V_0;
6 **return** V_0 and E_0;

E_0 in tandem and processes maximal groups of consecutive vertices in V_0 that induce memory-sized subgraphs of G. The memory-sized subgraphs it processes correspond to segments of the Euler tour. Intuitively, the ordering of the vertices in V_0 produced by the preprocessing phase should ensure that the processed subgraphs are connected or have few connected components (in the undirected sense). Assuming sufficiently random edge directions and sufficiently many non-tree edges, this should lead to non-trivial SCCs in the processed subgraphs.

Algorithm 4 shows the high-level procedure for preprocessing. To compute lists V_0 and E_0, our algorithm for preprocessing G has to compute the tree T, its Euler tour, and a ranking of the Euler tour (Steps 1–3). To compute the spanning tree, we use the MST algorithm by Dementiev et al. [61] (setting all edge weights to 1). Sorting and scanning the edge set of T suffices to compute an Euler tour of T. To rank this tour, we use the list ranking algorithm by Sibeyn [123]. Given the ranked tour, the algorithm finds the first occurrence of every vertex of G in the tour by sorting and scanning the node list of the tour, numbers the vertices of G in the order of these occurrences, and places them into V_0 in order (Step 4). The edge list E_0 is constructed by sorting and scanning the edges of G three times: twice to label each edge with the numbers of its endpoints, and once more to arrange the edges in the order described above (Step 5).

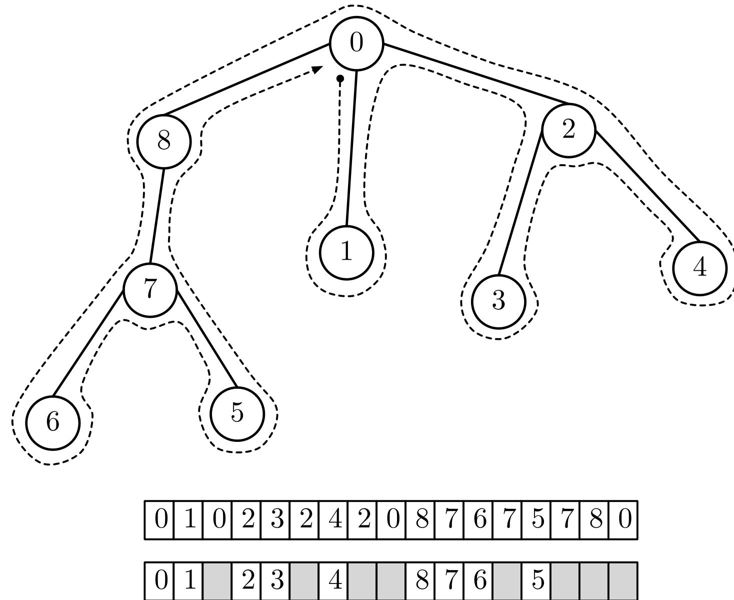


Figure 7.1: Spanning tree and the Euler tour around it (dashed lines), starting and ending at vertex 0. Below is the order of the vertices on the tour and their first occurrence.

7.2.2 Contraction Phase

The contraction phase of EmSCC proceeds in a series of *rounds*, where each round produces a more compressed version of G from the previous version by identifying strongly connected subgraphs of the current graph and contracting them. The high-level procedure of this phase is presented in Algorithm 5. Let $G = G_0, G_1, \ldots, G_r$ be the sequence of graphs produced during the contraction rounds; that is, in round i, the graph G_i is computed from graph G_{i-1}. The algorithm represents each graph G_i using two lists V_i and E_i whose structure is identical to that of V_0 and E_0 described in the previous subsection.

The ith round partitions V_{i-1} into subsets $V'_1, V'_2, \ldots, V'_k$ of consecutive vertices such that each induced subgraph $G'_j := G_{i-1}[V'_j]$ fits in memory. The algorithm loads these subgraphs into memory, one by one, identifies their SCCs and contracts them.

More precisely, the ith round scans V_{i-1} and E_{i-1} in tandem, counting and collecting the vertices and edges in the current subgraph G'_j in memory. Let x be the first vertex in V_{i-1} that belongs to G'_j, and let n_j and m_j respectively be the numbers of vertices and edges currently in G'_j. To decide whether to include the next vertex y in V_{i-1} in G'_j, the algorithm scans E_y and counts the edges whose lower endpoints

Algorithm 5: CONTRACT(G)

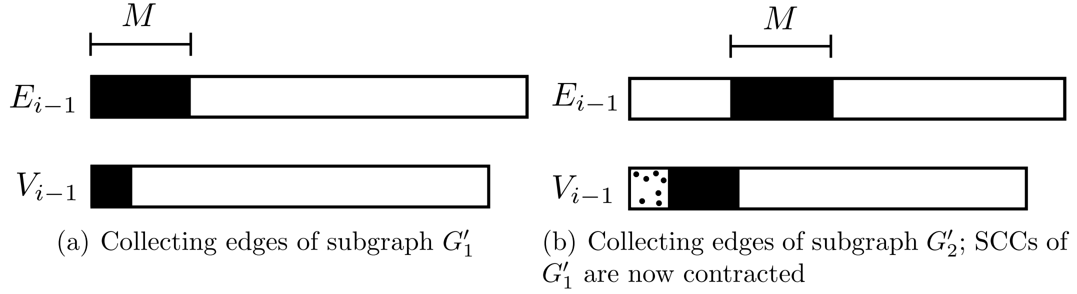
Input: $G = (V_0, E_0)$.
Output: The SCCs of G.

1 Let $i = 0$ and $G_i = (V_i, E_i)$;
// *Contraction rounds*
2 **while** $|G_i| > M$ **do**
3 $i = i + 1$;
4 Partition V_{i-1} into subsets of consecutive vertices that induce subgraphs of size $O(M)$;
5 For each such subgraph, compute its SCCs and compress them;
6 Let V_i and E_i be the new vertex and edge list respectively formed from the compressed graphs;
7 Load G_i into memory and contract;
8 **return** The set of super-vertices representing the SCCs of G_i;

belong to G'_j, that is, are no less than x; let m_y be their number.

If $n_j + 1$ vertices and $m_j + m_y$ edges fit in memory, the algorithm includes y in G'_j and partitions the edges in E_y into two groups: those with lower endpoints no less than x and those with lower endpoints less than x. It loads the former into memory (thereby adding them to G'_j) and appends the latter to an initially empty edge list E''_i to be processed at the end of this round. Then the algorithm proceeds to the next vertex in V_{i-1}.

If adding m_y edges to G'_j would make it exceed the memory size, the algorithm declares vertex y to be the first vertex of G'_{j+1} and appends its entire adjacency list to E''_i. Next it computes the SCCs of G'_j in memory, contracts each SCC it finds into a single vertex, and eliminates parallel edges that result from these contractions. Let us refer to these contracted SCCs as *super-vertices*. During the contraction phase the invariant is maintained that every vertex stores the ID of the super-vertex it has been contracted into. The algorithm writes this mapping information back to V_{i-1} and appends the sorted list of super-vertices to V_i. The edges of the contracted version of G'_j are appended to an initially empty edge list E'_i. This finishes the processing of G'_j, and the algorithm starts to construct G'_{j+1} with y as its first vertex. Let us refer to this procedure of contracting G'_j as a *contraction step*. This contraction step is repeated for all graphs $G'_1, G'_2, \ldots, G'_k$. See Figure 7.2 for an illustration.



(a) Collecting edges of subgraph G'_1

(b) Collecting edges of subgraph G'_2; SCCs of G'_1 are now contracted

Figure 7.2: Illustration of a contraction step in the ith round. Shaded areas represent the subgraphs induced by memory-sized groups of edges.

The ith round ends after the last vertex in V_{i-1} has been consumed. At this point, the algorithm discards the edge list E_{i-1}, but not V_{i-1}, as the information stored in V_{i-1} is necessary to compute the final component labelling of the vertices of G. If the algorithm numbers the vertices of G_i in increasing order as it produces them, V_i already contains the sorted vertex list of G_i. To produce E_i, the endpoints of all edges in E''_i have to be replaced with their corresponding super-vertices in G_i. Since the edges in E''_i are already sorted by their upper endpoints in G_{i-1}, a single scan of V_{i-1} and E''_i suffices to replace those endpoints. To replace the lower endpoints, the algorithm sorts the edges in E''_i by these endpoints and scans V_{i-1} and E''_i again. Finally, it concatenates the resulting list with E'_i, and sorts the concatenation primarily by upper endpoints (in V_i) and secondarily by lower endpoints. A single scan now suffices to filter the edges and thus eliminate duplicates from this list, which produces the edge list E_i of G_i.

7.2.3 Postprocessing

Let G_r be the graph produced by the last round of the contraction phase; that is, G_r fits in memory. Then the algorithm loads G_r into memory and labels every vertex in V_r with the SCC containing it. Finally, by undoing the contractions in the vertex set, one round at a time, the algorithm labels the vertices of G to identify their membership in these SCCs. This is done by iteratively copying the labels from V_i to V_{i-1}, for $i = r, r-1, \ldots, 1$. See Figure 7.3 for an illustration.

To copy the labels from V_i to V_{i-1}, the algorithm sorts the vertices in V_{i-1} by their corresponding super-vertices in V_i. Then the algorithm scans V_i and V_{i-1} to

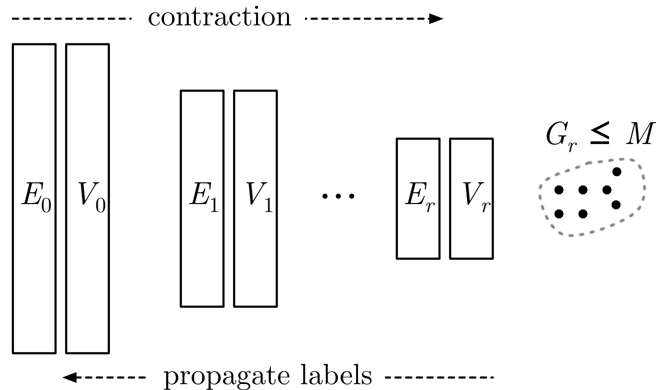


Figure 7.3: Illustration of postprocessing: In the last round G_r fits in memory. We label every vertex in V_r with the SCC containing it, and then propagate these labels back to the original vertices in G.

label every vertex in V_{i-1} with the label of its corresponding vertex in V_i. Finally, the algorithm returns the vertices in V_{i-1} to their original order, in preparation for the next iteration.

It is worth pointing out that our EmSCC algorithm has actually grown out of an initial strategy where we computed a hierarchical decomposition of G based on vertex degrees, similar to k-cores [119], and processed the graphs in this decomposition bottom-up in search for SCSGs. This approach produced better grouping of vertices with slightly better results in the contraction phase; however, constructing the decomposition was too costly and out-weighted the speed-up we obtained in the contraction phase compared to EmSCC.

7.2.4 Analysis

The I/O complexity of the EmSCC algorithm can be split into the costs of preprocessing and contraction. The algorithm starts by constructing a spanning tree T of the input graph G. After T has been built, the algorithm computes an Euler tour around T. Next, the algorithm ranks the vertices in the tour. Finally, it assigns (by scanning and sorting) to each vertex v the rank in the tour of the first occurrence of v. The cost of constructing the spanning tree is $\mathrm{O}(\mathrm{sort}(m)\log(n/M))$ I/Os using the method of [61];. The construction of the Euler tour and list ranking of n elements both take $\mathrm{O}(\mathrm{sort}(n))$ I/Os (see Section 6.2.2.2). Assigning ranks to vertices

and labelling the edges with the assigned ranks requires additional $O(\text{sort}(m))$ I/Os. Hence, the preprocessing costs $O(\text{sort}(n+m)\log(n/M))$ I/Os in total.

Each contraction round i incurs a constant number of scans of $O(n_i)$ vertices and $O(m_i)$ edges, where n_i and m_i are the number of vertices and edges, respectively, that resulted from contractions in round $i-1$, in addition to a number of sort operations on the edges, and the internal memory computation. Scanning the vertex set and the adjacency lists to collect the vertices and edges that induce memory-sized subgraphs takes $O(\text{scan}(n_i+m_i))$ I/Os. Computing SCCs in memory causes no I/Os. Collecting the super-vertices and edges between SCCs that resulted from contractions requires further $O(\text{scan}(n_i+m_i))$ I/Os. Edges between subgraphs are sorted and scanned for remapping and filtering out duplicates, causing $O(\text{sort}(m_i))$ I/Os. Consequently, each round i takes $O(\text{scan}(n_i+m_i)+\text{sort}(m_i)) = O(\text{sort}(m_i))$ I/Os.

The overall running time of the algorithm ultimately depends on two factors: (a) the number of contraction rounds needed and, (b) the amount of contraction achieved in each round. If, for instance, the number of rounds is large but the size of the input graph decreases geometrically due to contractions, then we can bound the cost of our algorithm by $O(\text{sort}(m))$. Similarly, if the number of rounds is constant, the cost is bounded by $O(\text{sort}(m))$ times the number of rounds. However, it may happen that we need many rounds with very little contraction in each one or, as our experiments show, we may not see any contraction at all after certain number of rounds.

7.3 Implementation Details

EmSCC is built on various algorithmic primitives, each of which has an impact on the running time of the algorithm. To compute V_0 and E_0 (Section 7.2.1), for example, the algorithm has to compute a spanning tree, its Euler tour, and a ranking of the tour. As we did not have any implementations of these primitives available, apart from the MST algorithm, we implemented all of them. To compute the spanning tree, we used the minimum spanning tree (MST) algorithm of Dementiev et al. [61]. Computing the Euler tour is fairly simple, as all that is required is sorting and scanning the edge set of the tree [94]. To rank the tour, we implemented the list ranking algorithm of Sibeyn [123]. In this section we discuss implementation choices we made in order to improve the algorithm's performance.

We implemented EmSCC in C++ and using the STXXL library [60], which provides I/O-efficient counterparts of the C++ STL containers and algorithms. In particular, we used STXXL vectors to store the vertex and edge lists of graphs, and the STXXL sorting algorithm to perform all sorting steps in our algorithm.

External-memory graph representation. As already discussed, each graph G_i is represented by a vertex list V_i and an edge list E_i. In our implementation, every vertex in V_i was represented using two integers, one being its own ID, the other one the ID of the corresponding super-vertex in G_{i+1}. Edges were represented as pairs of vertex ID's, that is, using two integers. The only exception was the addition of an extra integer to represent the edge weight up to and including the MST computation. This could have been avoided by modifying the MST implementation to compute an arbitrary spanning tree of an unweighted graph. We did not do this, as the MST computation did not account for a major part of the running time of our algorithm.

MST algorithm. We used the MST algorithm of [61] to compute the spanning tree T in the preprocessing phase. The implementation was available from [117]. The algorithm is a sweeping algorithm, which iteratively removes vertices by contracting the lightest edge incident to each processed vertex. This strategy can be implemented using an external priority queue or using an I/O-efficient bucket structure. The default implementation uses a bucket structure, as it results in slightly better performance; so we had no reason to change this.

Euler tour. To compute the Euler tour of T, we used the standard strategy. We created two copies (x, y) and (y, x) of each spanning tree edge (x, y) and sorted the resulting edge list by their first vertices. Then we scanned the sorted edge list and, for each pair of consecutive edges, (x, y_1) and (x, y_2), incident to the same vertex x, we made edge (x, y_2) the successor of edge (y_1, x) in the Euler tour. This was easily implemented by storing the edges in an STXXL vector and using the STXXL sorting algorithm to implement the sorting step.

List ranking. The list ranking algorithm of [123] is a sweeping algorithm similar to the MST algorithm of [61]. The *down-sweep* removes vertices one by one from

the list until only one vertex remains. For each removed vertex v, its two incident edges are replaced with a weighted edge between v's neighbours; the weight equals the length of the sublist between these two neighbours. The *up-sweep* re-inserts the removed vertices in the opposite order and computes the rank of each vertex v from the rank of one of the two vertices that became adjacent as a result of the removal of v in the down-sweep.

As discussed in [123], this algorithm can be implemented using a bucket structure, similar to the one used in the MST algorithm, to pass information between vertices in the two sweeps. An alternative implementation uses a priority queue and two stacks. Since our focus was not on engineering an optimal list ranking algorithm, we opted for the easier implementation using a priority queue. Moreover, similarly to the MST computation, computing list ranking did not account for a major portion of the running time of the algorithm.

Internal-memory SCC algorithm. In theory, when we talk about I/O complexity, it is not relevant which internal-memory algorithm we use to process the data loaded into memory; in practice, however, the running times of the used internal-memory algorithms can have a significant impact on the algorithm's overall running time. The running times of different linear-time SCC algorithms differ by constant factors and, as argued in [99], it is desirable to use a one-pass algorithm. To compute the strongly connected components of a graph loaded into memory, we used the one-pass strong connectivity algorithm by Djikstra [64]. (For a more recent description of this algorithm, see [28].) The implementation of this algorithm requires two stacks to keep track of partially identified SCCs.

The size of the subgraphs we process in internal memory has a substantial impact on the algorithm's performance. The reason is that the bigger the subgraphs the higher the chances are to find long cycles to contract. As such, we have two implementation choices which, independently, increase the available memory for subgraph processing: use of disk-based stacks in the SCC algorithm and, use of a space-efficient internal-memory graph representation. Thus, in order to minimize the memory requirements of the stacks used in the SCC algorithm, we used STXXL stacks; since n stack operations take $O(n/B)$ I/Os on external stacks, this had almost no impact on

the time spent on stack operations, but it limited the memory footprint of the stacks to 4 pages and thus freed up space for storing the graph in memory. Furthermore, stacks are very I/O-efficient structures [60]. Next, we describe the internal-memory graph representation we use.

Internal-memory graph representation. To maximize the size of the subgraphs that can be processed in internal memory in each round of EmSCC, we used a fairly compact graph representation in internal memory, consisting of two arrays: an edge array and a vertex array. The edge array contained the concatenation of adjacency lists of the vertices. Since the SCC algorithm only needed access to the out-edges of each vertex, only those edges were stored in the adjacency lists. When accessing an adjacency list, it was known to which vertex this adjacency list belonged. Hence, the tail vertex of every edge did not have to be stored explicitly. This allowed us to represent every edge using a single integer storing the head vertex of the edge. Figure 7.4 shows an example of this representation.

We represented every vertex using a two-integer record in the vertex array. The first integer represented the SCC containing this vertex (once identified), the other the index of the first edge in its adjacency list in the edge array. Vertex ID's did not have to be stored explicitly, as a consecutive numbering of the vertices allowed us to use the position of a vertex in the vertex array as its ID.

Since this representation stores edges in a different order than on disk, it was necessary to sort the edges by their tails to construct the internal-memory representation of a graph G'_j from its external one. This required the use of an initial edge representation using both its endpoints during the construction of the internal-memory graph representation. Once the edges were arranged in the right order, we dropped their tail endpoints, thus halving the memory requirements of the representation. Since the ability of our algorithm to identify SCCs improves with the size of the subgraphs it can process in memory, we decided to process subgraphs that occupied all of the available main memory (minus some buffer blocks for caching used by the STXXL vectors) using the *compact* representation. As a result, the initial sorting step required to construct this representation used the STXXL *external* sorting algorithm to sort up to $2M$ data, where M denotes the memory size. Since this is only twice

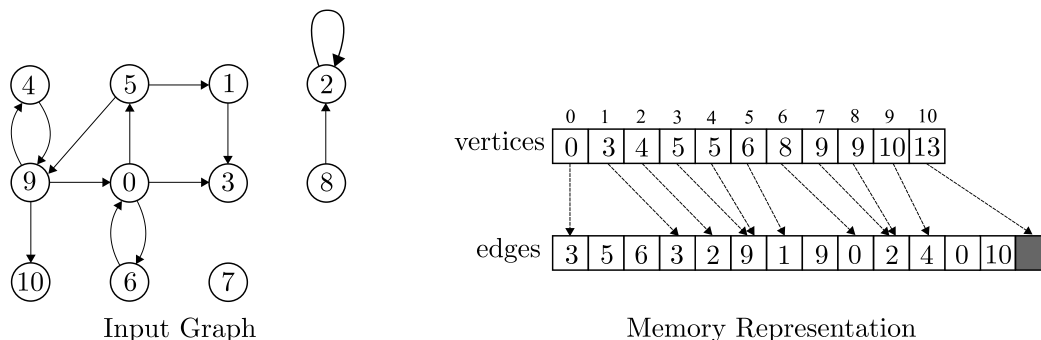


Figure 7.4: Example graph with 11 vertices and 13 edges. Internal-memory representation of the graph. The numbers in the array “vertices” are the starting indexes of the adjacency lists in the array “edges”; the ID of the SCC of the vertex is not shown.

the memory size, sorting such an edge list takes $O(M/B)$ I/Os.

Pipelining. Pipelining is a well known implementation technique that has been used to design faster I/O-efficient algorithms [18, 60, 100]. The idea behind it is to interface a given sequence of processing elements in such a way that the data is passed from one element to another without writing intermediate results to disk. Our implementation of a contraction step (Section 7.2.2) is inspired by pipelining in the sense that, instead of writing the edges of a contracted subgraph back to disk, and using them again only in the next round, we immediately make them part of the next subgraph that is processed in memory. More precisely, once we have contracted a subgraph, we keep its compressed version in memory, merge it with the next group of collected edges, and then proceed to contract the subgraph as usual. We continue in this manner until the compressed version of the current graph occupies a constant fraction of the available memory. This way, before we ever have to write the results of a contraction step back to disk, we perform a more aggressive contraction.

7.4 Performance Evaluation

This section describes the results of an extensive set of experiments designed to evaluate the performance of EmSCC, compared to the performance of the semi-external algorithm by Sibeyn et al. (called SeSCC here). SeSCC uses DFS to compute strongly connected components. Table 7.1 shows the approximate vertex number

Main Memory	Number of Vertices
512 MB	2^{24}
1024 MB	2^{25}

Table 7.1: Maximum number of vertices in a graph that SeSCC can process (without the operating system resorting to virtual memory) using a certain amount of internal memory.

SeSCC can process without using virtual memory, as discussed in [34]. First we describe our test environment and the data sets used in our experiments. Then we discuss the results of our experiments.

7.4.1 Environment and Settings

All experiments were run on a PC with a 3GHz Pentium-4 processor, 1GB of RAM, and one 500GB 7200RPM IDE disk using the XFS file system. The operating system was Fedora Core 6 Linux with a vanilla 2.6.20 Linux kernel. The code was compiled using g++ 4.1.2 and optimization level –O3. All of our timing results refer to wall clock times in minutes.

Since STXXL allows the specification of the block size for data transfers between disk and memory, we experimented with different block sizes between 256KB and 8MB. A block size of 2MB resulted in the best performance, since EmSCC accesses data in a mostly sequential fashion. This block size was used throughout our experiments. Two additional parameters control the amount of memory allocated to the LRU pager used by STXXL vectors to cache accessed blocks. The first parameter is the page size as a multiple of the block size. Data is swapped one page at a time. The other parameter is the number of pages to be cached. We set both parameters to 2, as the mostly sequential data accesses of EmSCC did not benefit substantially from a bigger cache, but this would have left less memory for the graphs to be processed in memory.[1]

[1]Using a single disk, a block size of 2MB and a page size of two blocks is equivalent to using a block size of 4MB and a page size of one block. We chose the former option because we also tested our algorithms using two disks, in which case the blocks of each page can be assigned to different disks. Using two disks, our algorithm experienced a speed-up of about 30%. Since the semi-external algorithm was not able to take advantage of multiple disks, we do not discuss the timings using two disks in detail here.

7.4.2 Data Sets

We tested both algorithms on synthetic graphs and real web graphs. The synthetic graphs were generated using the same data generator used by Sibeyn et al. [121]. The web graphs were produced by real web crawls of the .uk domain, the .it domain, and from data produced by a more global crawl using the Stanford WebBase crawler. They were obtained from `http://webgraph.dsi.unimi.it/`, and their characteristics are shown as part of Tables 7.2 and 7.3. Next we give an overview of the types of synthetic graphs used in our experiments.

Random: These graphs were generated according to the $G_{n,m}$ model; that is, m edges were generated, choosing each edge endpoint uniformly at random from a set of n vertices.

Cycle: The vertices were evenly spaced on a ring, and every vertex had out-edges to its $d = m/n$ nearest neighbours.

Geometric 1D: The vertices were evenly spaced on a ring of length n. Edges were generated by choosing their tails uniformly at random. If u was chosen as the tail of an edge, vertex v was chosen to be the head of this edge with probability proportional to α^d, where $\alpha < 1$ and d is the distance between u and v. In our experiments, we chose $\alpha = 0.9$.

Geometric 2D: The vertices were placed on a $\sqrt{n} \times \sqrt{n}$ grid wrapped around at the edges to form a torus. Edges were generated as for geometric 1D graphs, but d was chosen to be the Manhattan distance between u and v in the grid. Here we chose $\alpha = 0.8$.

Out-star: Given a star degree s, this graph was generated in $\lfloor m/s \rfloor$ rounds. In each round, a tail vertex and s head vertices were chosen uniformly at random. Then edges were added from the tail to the chosen head vertices. We chose $s = 1000$ in our experiments.

In-out-star: This construction was similar to the out-star construction, but half of the rounds directed the generated edges towards the centre of the star. Again, we chose $s = 1000$.

Simple web: This construction started with a small complete subgraph and added new vertices by connecting them to the current graph at random. Afterwards, a small fraction (5% in our case) of random edges were added.

7.4.3 EmSCC vs. SeSCC.

In the following we analyze the behavior of EmSCC and SeSCC for the graph classes described above. Firstly, the discussion focuses on the experimental results with graphs whose vertex set fits entirely in memory; in this case EmSCC outperformed SeSCC for all graph classes. Secondly, the results with graphs having a vertex set much larger than the available memory are examined; here, while SeSCC could not handle any input, EmSCC was able to process most graphs very fast.

Semi-external mode ($|V| \leq M$). Table 7.2 shows the running times of EmSCC and SeSCC on different synthetic inputs and on two of the web graphs for the case where the number of vertices fit into internal memory. For the synthetic graphs with 2^{25} vertices, EmSCC outperformed SeSCC by a factor between 2 and 4. The only exception were random graphs and geometric 2D graphs, where SeSCC took only slightly longer than EmSCC. For the two smaller web graphs, EmSCC outperformed SeSCC by a factor between 3 and 4. As can be observed, the performance of SeSCC depends strongly on the structure of the input graph, whereas (surprisingly) the performance of EmSCC is much more immune to these variations. Sibeyn et al. characterized geometric 1D graphs as being among the hardest inputs for their algorithm, and geometric 2D and random graphs as being among the easiest inputs. This is in line with our observations. On the other hand, cycle graphs were mentioned as easy inputs in [121], while this was the synthetic input that took SeSCC the longest to process in our experiments.

Fully external mode ($|V| > M$). The remaining inputs had at least 2^{26} vertices and were beyond the reach of SeSCC on our hardware, as the vertex set no longer fits in memory (see Table 7.1 for approximate vertex numbers it can process without using virtual memory). Table 7.3 shows the result when $|V| > M$. We ran SeSCC on the smallest of these graphs (with 2^{26} vertices and 2^{29} edges), using virtual memory,

Graph Class	Graph Size n	m	m/n	Time (m) EM	SE	SCCs
Random	2^{25}	2^{29}	16	61	63	12
Cycle	2^{25}	2^{29}	16	58	208	1
Geom-1D	2^{25}	$\approx 2^{29}$ [1]	13.2	51	161	11
Geom-2D	2^{25}	$\approx 2^{29}$ [1]	15.6	58	62	7175
In-out star	2^{25}	2^{29}	16	63	141	22490
Out-star	2^{25}	2^{29}	16	65	109	33m
Simple-web	2^{25}	2^{29}	16	63	113	1.6m
Webgraph	18.5m	298.1m	16.1	29	104	3.8m
Webgraph	41.3m	1,150.7m	25.9	116	517	6.7m

Table 7.2: Experimental results on synthetic and real web graphs with $|V| \leq M$. Notes: (1) For geometric 1D and 2D graphs, m denotes the number of edges requested to be generated. Since the data generator filters duplicate edges for these two graph types, the actual number of edges, is less than m. The ratio m/n in the table reflects this.

and terminated each of these test runs after 12h without SeSCC having produced any result. Since the performance of SeSCC on the semi-external instances of random and geometric 2D graphs was comparable to that of EmSCC, we expected that SeSCC would have the least difficulties to process larger instances of these graph classes, and we let the experiments on these inputs run for 24h. Again, SeSCC did not finish within this amount of time.

In contrast, EmSCC was able to process most of the test graphs in under two hours, while none took more than 2 1/2 hours. The exceptions were the out-star graphs and the sparsest of the in-out-star and simple web graphs. Section 7.4.4 discusses possible reasons why EmSCC could not process these inputs, which sheds some light on its limitations.

7.4.4 Factors Affecting the Performance of EmSCC

A number of factors influence the performance of EmSCC. Here we discuss them in detail.

	Graph Size			Time (m)		
Graph Class	n	m	m/n	EM	SE	SCCs
Random	2^{26}	2^{29}	8	77	—[2]	45173
Random	2^{27}	2^{29}	4	109	—	5.2m
Random	2^{26}	2^{30}	16	133	—	17
Random	2^{27}	2^{30}	8	159	—	90279
Random	2^{28}	2^{30}	4	345	—	10.4m
Webgraph	118.1m	1,019.9m	8.6	124	—[1]	38.5m
Cycle	2^{26}	2^{29}	8	71	—[1]	1
Cycle	2^{27}	2^{29}	4	94	—	1
Cycle	2^{26}	2^{30}	16	120	—	1
Geom-1D	2^{26}	$\approx 2^{29}$	7.2	65	—[1]	45084
Geom-1D	2^{27}	$\approx 2^{29}$	3.8	90	—	5.2m
Geom-1D	2^{26}	$\approx 2^{30}$	13.2	103	—	17
Geom-2D	2^{26}	$\approx 2^{29}$	7.9	70	—[2]	45060
Geom-2D	2^{27}	$\approx 2^{29}$	4.0	91	—	5.2m
Geom-2D	2^{26}	$\approx 2^{30}$	15.6	117	—	18
In-out star	2^{26}	2^{29}	8	79	—[1]	2.6m
In-out star	2^{27}	2^{29}	4	—[3]	—	—
In-out star	2^{26}	2^{30}	16	134	—	44800
Out-star	2^{26}	2^{29}	8	—[3]	—[1]	—
Out-star	2^{27}	2^{29}	4	—[3]	—	—
Out-star	2^{26}	2^{30}	16	—[3]	—	—
Simple-web	2^{26}	2^{29}	8	86	—[1]	10.6m
Simple-web	2^{27}	2^{29}	4	—[3]	—	—
Simple-web	2^{26}	2^{30}	16	133	—	3.2m

Table 7.3: Experimental results on synthetic and real web graphs with $|V| > M$. Notes: (1) experiment terminated after 12h; (2) experiment terminated after 24h; (3) no further compression after a small number of initial contraction rounds, but graph still beyond memory size.

The effect of the available memory. As already mentioned, the ability of our algorithm to process certain graphs is limited by the available amount of main memory. The input graph needs to have few enough SCCs to fit in memory, and the SCCs have to be composed of short enough cycles so we can find them as part of

the memory-sized subgraphs we process. Our inability to process all but one of the out-star graphs nor the sparsest of the in-out-star and simple web graphs reflects these limitations.

The effect of the graph structure. Since we were not able to process these graphs, we can of course only extrapolate from the properties of the graphs in these classes that we *were* able to process. The smallest simple web graph had about 1.6m SCCs, and the smallest out-star graph even had about 33m SCCs. Compared to at most a few hundred SCCs in the smallest cycle, geometric 1D and 2D, and random graphs, these graphs have significantly more SCCs. For the sparser inputs, we suspect that the number of SCCs exploded, preventing us from processing these graphs.

The smallest in-out-star graph had about 22,000 SCCs, which is more than for cycle, geometric 1D and 2D, and random graphs, but significantly less than for out-star and simple web graphs. Therefore, we suspect that our inability to process the sparsest of the in-out-star graphs was not a result of an excessive number of SCCs but rather of SCCs with very long cycles, which we were not able to find using the amount of main memory available on our test machine.

In general, it was apparent that EmSCC is much more immune to variations in the graph structure than SeSCC, as the running time of EmSCC almost only depended on the input size. This behavior can be seen more clearly in Figure 7.5 where, with $m = 2^{29}$ and varying n, all of the synthetic graphs show a very similar performance pattern (except for simple web and in-out-star graphs with $n = 2^{27}$, for which the algorithm was not able to compress the graph down to memory size). On the other hand, SeSCC showed large variations in running times on graphs of the same size but different structure. This is surprising because algorithmically EmSCC explicitly depends on the graph structure (i.e., strongly connected subgraphs having short cycles), while, SeSCC has no such explicit dependence on the graph structure.

The effect of the number of rounds. Another interesting observation we made in our experiments was the lack of a smooth transition between graphs we could process very efficiently and graphs we could not process at all. More precisely, all the graphs we were able to process required one or two contraction rounds, followed by a final round computing the SCCs in internal memory. We believe that much bigger

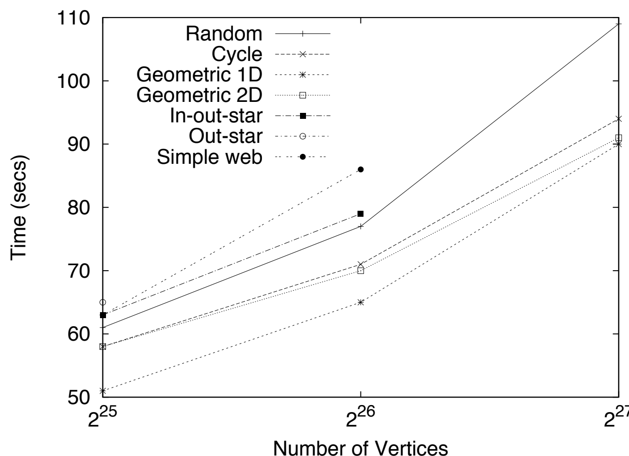


Figure 7.5: Variation of EmSCC total time with increasing n and $m = 2^{29}$.

graphs of the same structure as the graphs we tested *can* be processed, using more rounds, but this is hard to evaluate conclusively, as each such graph would take days to process even using EmSCC, and the generation of such massive graphs also poses major challenges.

The effectiveness of pipelining contraction steps. Let us consider the smallest web graph; in this case EmSCC only needed one round of contractions. With each contraction step we obtained a substantial reduction in the size of the subgraph loaded into memory, and it was not necessary to write the contracted edges back to disk; these edges were few enough so that we were able to incorporate them into the group of edges to be used in the next contraction. This result confirms the success of the pipelining idea. In the case of the second and third graphs, which are the ones with the most SCCs, we were able to achieve over a 90% reduction in the graph size just after the first round of contractions; we needed only one additional round where we loaded the contracted graph into memory and computed the final SCCs.

Overall, the experimental results demonstrated that, while EmSCC is not guaranteed to be theoretically efficient, in practice it is very fast on the graphs it was able to process, and it extends the size of the problems that can be tackled. The algorithm outperforms the semi-external algorithm of [121] as it was able to process a wide range of input graphs (which are the ones used already in [121]) faster, including some real-world webgraphs. Moreover, EmSCC was able to process graphs

whose vertex set did not fit in memory.

In spite of being able to efficiently process larger graphs than SeSCC can handle, the ability of EmSCC to process certain graphs is limited by the available amount of main memory. Some graphs have just too many SCCs, which means that no amount of contraction will make them fit in memory. Other graphs have SCCs with only long cycles, and finding these cycles requires us to load them into memory, which again may not be feasible. Nevertheless, we can expect to be able to process most interesting synthetic and real graphs, as has been the case of most of the graphs presented in this section.

Chapter 8

External-Memory Topological Sorting

In this chapter we present an I/O-efficient algorithm, IterTS, for the problem of topological sorting of directed acyclic graphs (DAGs). This is a problem for which no provably I/O-efficient solution is known for general DAGs, while optimal algorithms for planar graphs [27], graphs of bounded treewidth [97] and near-planar graphs [78] exist. The strategy of IterTS can be summarized as follows. An initial numbering of the vertices in the DAG is computed; even a random numbering is expected to satisfy at least half the edges in the DAG, where we call an edge satisfied if its tail has a smaller number than its head. IterTS then applies an iterative procedure which corrects the initial numbering to satisfy more and more edges until all edges are satisfied.

To evaluate IterTS, we compared its running time to that of three competitors: PeelTS, an I/O-efficient implementation of the standard strategy of iteratively removing sources and sinks; ReachTS, an I/O-efficient implementation of a recently proposed parallel divide-and-conquer algorithm based on reachability queries; and SeTS, standard DFS-based topological sorting built on top of a semi-external DFS algorithm. In our evaluation on various types of input graphs, IterTS consistently outperformed PeelTS and ReachTS, by at least an order of magnitude in most cases. SeTS outperformed IterTS on most graphs whose vertex set fits in memory. However, IterTS often came close to the running time of SeTS on these inputs, and SeTS was not able to process graphs whose vertex set was beyond the main memory size, while IterTS was able to process such inputs efficiently.

8.1 Introduction

Topological sorting is solved in internal memory in linear time by either of two methods: repeatedly removing sources (in-degree-0 vertices) [84,85] or using DFS [53,125]. Unfortunately, both algorithms access the vertices in an unstructured fashion and,

thus, usually perform one random disk access per vertex when processing inputs beyond the size of main memory.

The rest of the chapter is organized as follows. In Section 8.2, we describe our new algorithm. In Section 8.3 we describe other algorithms we considered reasonable competitors and implemented, in order to compare their performance with that of our algorithm. In Section 8.4 we present some implementation details. In Section 8.5 we discuss our experimental setup and results.

8.2 Topological Sorting by Iterative Improvement

In this section we describe our algorithm for solving the topological sorting problem, referred to as ITERTS throughout the rest of the chapter. Given a numbering $\nu(\cdot)$ of the vertices of the input DAG G, we call an edge *satisfied* if its tail receives a lower number than its head; otherwise, we say the edge is *violated*. The *satisfied subgraph* of G is a DAG G_ν whose vertex set is V and whose edge set consists of all edges of G satisfied by $\nu(\cdot)$.

The general strategy of the algorithm can be described as follows. Given a DAG $G = (V, E)$, the first step of our algorithm is to compute a numbering $\nu_0(\cdot)$ of the vertices in V. After computing the initial numbering $\nu_0(\cdot)$ and its corresponding satisfied subgraph G_{ν_0}, we proceed in iterations, each of which computes a new numbering $\nu_i(\cdot)$ from the previous numbering $\nu_{i-1}(\cdot)$, with the goal of increasing the number of satisfied edges. The computation of $\nu_i(\cdot)$ from $\nu_{i-1}(\cdot)$ ensures that $\nu_i(\cdot)$ satisfies strictly more edges than $\nu_{i-1}(\cdot)$. Thus, the algorithm is guaranteed to terminate, slowly in the worst case, quickly in practice.

The description of the algorithm is divided into four parts. In Section 8.2.1, we describe how to compute the initial numbering $\nu_0(\cdot)$. In Section 8.2.2, we discuss the computation in each iteration. In Section 8.2.3, we analyze the I/O complexity of the algorithm. Finally, in Section 8.2.4, we discuss a heuristic improvement that led to a tremendous performance improvement of our algorithm.

8.2.1 Computing the Initial Numbering

Throughout the algorithm, we assume the input graph G has only one source s. If this is not the case, we introduce a new source vertex and connect it to each of the original sources. Then we compute an out-tree of s, that is, a spanning tree of G whose root is s and all of whose edges are pointing away from s. Since G is acyclic, such a spanning tree T_0 can be obtained by having every vertex $x \neq s$ choose an arbitrary in-edge to be included in T_0. The set of all selected edges forms a spanning tree of G.

In the tree thus constructed, we now choose an arbitrary left-to-right ordering of the out-edges of every vertex in T_0. Then we proceed to compute two numberings $\nu_l(\cdot)$ and $\nu_r(\cdot)$ of its vertices. Both are preorder numberings; $\nu_l(\cdot)$ numbers the subtrees of each vertex in left-to-right order, while $\nu_r(\cdot)$ numbers the subtrees in right-to-left order. Both numberings satisfy all tree edges and one of them satisfies at least half of the non-tree edges. We choose our initial numbering $\nu_0(\cdot)$ to be the one that satisfies more edges. Next we describe how to carry out this procedure using O(sort(m)) I/Os.

Constructing T_0. After sorting the edges of G by their heads, a scan of this edge list suffices to choose one in-edge for each vertex and, if there is more than one vertex without in-edges, add a new source s and connect it to each original source. Thus, T_0 can be constructed using O(sort(m)) I/Os.

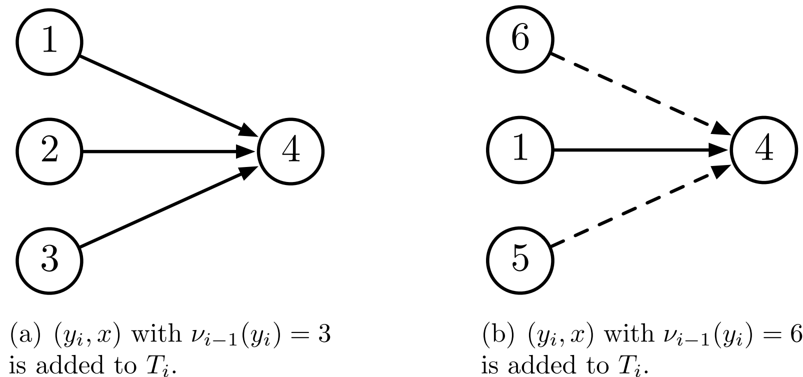
Computing $\nu_l(\cdot)$ and $\nu_r(\cdot)$. The numberings $\nu_l(\cdot)$ and $\nu_r(\cdot)$ are easily computed by first constructing an Euler tour around T_0 and then ranking the nodes in the tour using an I/O-efficient list ranking algorithm, all of which takes O(sort(n)) I/Os.

Computing $\nu_0(\cdot)$. It suffices to sort and scan the vertex and edge sets of G to label every edge with the numbers assigned to its endpoints by $\nu_l(\cdot)$ and $\nu_r(\cdot)$, and count the number of edges satisfied by each numbering, in order to choose $\nu_0(\cdot)$. This takes O(sort(m)) I/Os.

8.2.2 Growing the Satisfied Subgraph

Each iteration of the algorithm now computes a new numbering $\nu_i(\cdot)$ from the current numbering $\nu_{i-1}(\cdot)$ so that $\nu_i(\cdot)$ satisfies strictly more edges than $\nu_{i-1}(\cdot)$. We do this in two phases.

In the first phase, we compute an out-tree T_i of s and compute a numbering $\nu'_i(\cdot)$ such that $\nu'_i(x) \geq \nu_{i-1}(x)$, for all $x \in V$, and every edge in T_i is satisfied. During the construction of the tree T_i, every vertex $x \neq s$ chooses an in-edge (y, x) for inclusion in T_i so that $\nu_{i-1}(y)$ is maximized. Figure 8.1 shows an illustration of this rule.



(a) (y_i, x) with $\nu_{i-1}(y_i) = 3$ is added to T_i.

(b) (y_i, x) with $\nu_{i-1}(y_i) = 6$ is added to T_i.

Figure 8.1: Selection of in-edge for vertex x with $\nu_{i-1}(x) = 4$. Each vertex displays its corresponding value of $\nu_{i-1}(\cdot)$. Dashed lines denote violated edges. The left side shows a case where x has only satisfied in-edges, whereas on the right side it has violated in-edges.

In the second phase, we compute a numbering $\nu''_i(\cdot)$ by processing the subgraph $G_{\nu_{i-1}}$ of G satisfied by $\nu_{i-1}(\cdot)$. This numbering satisfies $\nu''_i(x) \geq \nu'_i(x)$, for all $x \in V$, and satisfies all edges of $G_{\nu_{i-1}}$. We obtain the new numbering $\nu_i(\cdot)$ by ordering the vertices in G according to $\nu''_i(\cdot)$ and then numbering the vertices of G in order.[1] Next we describe the computation of $\nu'_i(\cdot)$, $\nu''_i(\cdot)$, and $\nu_i(\cdot)$ in detail.

Computing $\boldsymbol{\nu'_i(\cdot)}$. To construct the tree T_i, we proceed similarly to the construction of T_0, choosing one in-edge (y, x) per vertex $x \neq s$ to be included in T_i. This time, however, we choose each such edge (y, x) so that $\nu_{i-1}(y)$ is maximized. Similar to the construction of T_0, this construction can be carried out by sorting the edges

[1]The orderings defined by $\nu_i(\cdot)$ and $\nu''_i(\cdot)$ are identical, but $\nu''_i(\cdot)$ may not assign unique numbers to vertices and may assign numbers greater than N.

of G by their heads and then scanning the edge list to choose the in-edge of each vertex to be included in T_i. (Recall that each edge (y, x) is labelled with the numbers $\nu_{i-1}(x)$ and $\nu_{i-1}(y)$ of its endpoints, making it easy to identify the in-edge (y, x) of each vertex x that maximizes $\nu_{i-1}(y)$.) Next we construct an Euler tour of T_i and apply list ranking to compute a preorder numbering of T_i, which is also a topological ordering of T_i. We sort the vertices of T_i in this order and then apply time-forward processing to compute, for every vertex $x \in T_i$, $\nu'_i(x) := \max(\nu_{i-1}(x), \nu'_i(p_i(x)) + 1)$, where $p_i(x)$ denotes x's parent in T_i. The sorting and scanning of the vertex and edge sets of G, and the application of the Euler tour technique, list ranking, and time-forward processing to T_i take $O(\text{sort}(m))$ I/Os in total.

Computing $\nu''_i(\cdot)$. In the second step, we sort the vertices according to $\nu_{i-1}(\cdot)$ and the edges of $G_{\nu_{i-1}}$ by their tails. Then we apply time-forward processing to $G_{\nu_{i-1}}$, which is possible because $\nu_{i-1}(\cdot)$ defines a topological ordering of $G_{\nu_{i-1}}$ (by definition, $\nu_{i-1}(\cdot)$ satisfies all edges in $G_{\nu_{i-1}}$). For every vertex, we compute $\nu''_i(x) := \max(\{\nu'_i(x)\} \cup \{\nu''_i(y) + 1 \mid (y, x) \in G_{\nu_{i-1}}\})$, which ensures that $\nu''_i(\cdot)$ satisfies every edge of $G_{\nu_{i-1}}$. This takes $O(\text{sort}(m))$ I/Os.

Computing $\nu_i(\cdot)$. To prepare for the next iteration, we compute $\nu_i(\cdot)$ by ordering the vertices in G by $\nu''_i(\cdot)$ and then numbering them in order. Using a constant number of sorting and scanning passes, we label every edge with the numbers of its endpoints and accordingly classify the edge as satisfied or violated. This takes $O(\text{sort}(m))$ I/Os.

8.2.3 Analysis

From the above discussion, it follows that the initialization and each iteration of the algorithm take $O(\text{sort}(m))$ I/Os. Thus, the I/O complexity of the whole algorithm depends on the number of iterations the algorithm executes. The following lemma bounds this number of iterations.

Lemma 8.1. ITERTS *takes at most $l - 1$ iterations to satisfy all edges in G, where l is the length of the longest path in G.*

Proof. For a vertex x, let $\text{l-dist}(x)$ be the length of the longest path from s to x in G. We prove by induction on i that all in-edges of vertices x with $\text{l-dist}(x) \leq i + 1$ are

satisfied by $\nu_i(\cdot)$. Thus, $\nu_{l-1}(\cdot)$ satisfies all edges of G if l denotes the length of the longest path in G.

The base case, $i = 0$ is trivial because $\nu_0(s) = 1$, while $\nu_0(x) > 1$, for all $x \neq s$. Hence, all out-edges of s are satisfied by $\nu_0(\cdot)$, which is a superset of the in-edges of all vertices x with l-dist$(x) \leq 1$.

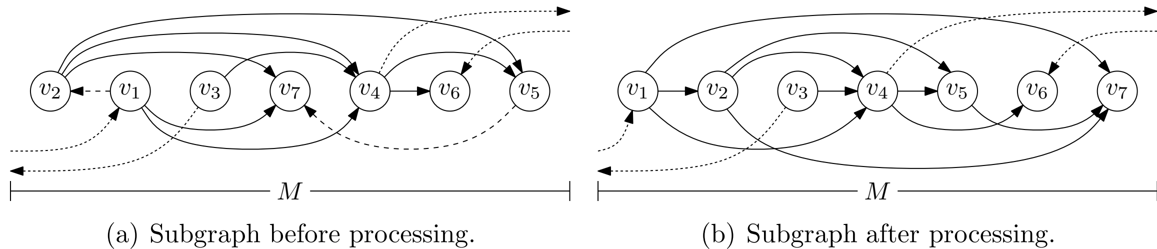
So assume the claim holds for $i < k$. We need to prove it for $i = k$. It suffices to prove that $\nu''_k(\cdot)$ satisfies all in-edges of vertices x with l-dist$(x) \leq k+1$ because $\nu_k(\cdot)$ is obtained by ordering the vertices by $\nu''_k(\cdot)$ and then numbering them in order. Thus, if $\nu''_k(x) < \nu''_k(y)$, we also have $\nu_k(x) < \nu_k(y)$.

First we prove that $\nu''_k(x) = \nu'_k(x) = \nu_{k-1}(x)$, for all x with l-dist$(x) \leq k$. Since every in-neighbour y of such a vertex x satisfies l-dist$(y) \leq k$ and $\nu_{k-1}(x)$ satisfies every in-edge of x, this implies that $\nu''_k(y) = \nu_{k-1}(y) < \nu_{k-1}(x) = \nu''_k(x)$, that is, $\nu''_k(\cdot)$ satisfies all in-edges of x. We prove this claim by induction on l-dist(x).

For l-dist$(x) = 0$, we have $x = s$ and $\nu''_k(s) = \nu'_k(s) = \nu_{k-1}(s) = 1$ because s is the source of $G_{\nu_{k-1}}$ and the root of T_{k-1}. For $0 <$ l-dist$(x) \leq k$, we have $\nu'_k(x) = \max(\nu_{k-1}(x), \nu'_k(p_k(x)) + 1)$. However, we have l-dist$(p_k(x)) <$ l-dist(x) and, hence, $\nu'_k(p_k(x)) = \nu_{k-1}(p_k(x))$. Furthermore, $\nu_{k-1}(p_k(x)) < \nu_{k-1}(x)$ because $p_k(x)$ is an in-neighbour of x and $\nu_{k-1}(\cdot)$ satisfies all in-edges of x. This implies that $\nu'_k(x) = \nu_{k-1}(x)$. Similarly, we have $\nu''_k(x) = \max(\{\nu'_k(x)\} \cup \{\nu''_k(y) + 1 \mid yx \in G_{\nu_{k-1}}\})$. Every in-neighbour y of x in $G_{\nu_{k-1}}$ satisfies l-dist$(y) <$ l-dist(x). Hence, by the induction hypothesis and because $\nu_{k-1}(\cdot)$ satisfies the edge yx, $\nu''_k(y) = \nu'_k(y) = \nu_{k-1}(y) < \nu_{k-1}(x) = \nu'_k(x)$, and $\nu''_k(x) = \nu'_k(x) = \nu_{k-1}(x)$.

Now consider a vertex x with l-dist$(x) = k+1$, and let y be an arbitrary in-neighbour of x. The parent $p_k(x)$ of x in T_k is chosen so that $\nu_{k-1}(p_k(x)) \geq \nu_{k-1}(y)$. Hence, $\nu'_k(x) \geq \nu'_k(p_k(x)) + 1 = \nu_{k-1}(p_k(x)) + 1 \geq \nu_{k-1}(y) + 1$. We also have $\nu''_k(x) \geq \nu'_k(x)$, that is, $\nu''_k(x) > \nu_{k-1}(y)$. On the other hand, since y is an in-neighbour of x, we have l-dist$(y) \leq k$ and, hence, $\nu''_k(y) = \nu_{k-1}(y)$. Thus, the edge yx is satisfied by $\nu''_k(\cdot)$. Since this argument applies to all in-edges of vertices x with l-dist$(x) = k+1$, and we have already shown that $\nu''_k(\cdot)$ satisfies all in-edges of vertices x with l-dist$(x) \leq k$, this proves that $\nu''_k(\cdot)$ satisfies all in-edges of vertices x with l-dist$(x) \leq k+1$. This finishes the proof. □

By Lemma 8.1, IterTS is guaranteed to terminate, after at most $n-2$ iterations.



(a) Subgraph before processing.

(b) Subgraph after processing.

Figure 8.2: Local topological ordering of memory-sized subgraphs. In (a), vertices are arranged by $\nu_i''(\cdot)$. Solid edges are satisfied edges in the subgraph, dashed edges are violated edges in the subgraph, and dotted edges have only one endpoint in the subgraph, that is, are not local. In (b), the vertices are rearranged to ensure that all edges local to the subgraph are satisfied.

For many graphs, the longest path has length significantly less than $n - 2$, guaranteeing a faster termination of the algorithm. Even for graphs with long paths, our experiments show that, in practice, IterTS terminates much faster than predicted by Lemma 8.1.

8.2.4 Satisfying Local Edges

By our analysis in the previous subsection, the cost of our algorithm depends crucially on the number of iterations it needs to satisfy all edges in the DAG. Furthermore, each iteration is costly since it involves computing a number of primitives, including Euler tour, list ranking and time-forward processing. In this section, we discuss a heuristic that helped us reduce the number of iterations in our algorithm significantly.

The idea is to immediately satisfy violated edges whose endpoints are "not too far apart" in the current ordering. To this end, we add the following step between computing $\nu_i''(\cdot)$ and numbering the vertices of G in order to compute $\nu_i(\cdot)$. We sort the vertices by $\nu_i''(\cdot)$ and the edges of G by the maximum positions of their endpoints in this sorted list. Then we scan the vertex and edge lists to greedily partition the vertex list into maximal contiguous sublists such that the vertices in each sublist induce a subgraph with at most M vertices and edges. We load each such subgraph into memory, topologically sort it, and then rearrange the vertices in the subgraph according to this topological ordering. We do not change the relative order of vertices in different subgraphs. This strategy ensures that all edges within each memory-sized subgraph are satisfied, while satisfied edges between subgraphs remain satisfied. This

is illustrated in Figure 8.2. Once we have rearranged the vertices in each memory-sized subgraph in this fashion, we compute $\nu_i(\cdot)$ by numbering the vertices of G in order as before. Using this heuristic, the edges satisfied by $\nu_i(\cdot)$ are a superset of the edges satisfied by $\nu''_i(\cdot)$. Algorithm 6 presents the pseudocode of IterTS and shows where this heuristic fits in IterTS's procedure (Steps 9–12).

Algorithm 6: IterTS(G)

Input: $G = (V, E)$.
Output: A topological ordering of G.

```
1  Let i = 0 and G_i = G;
   // Computing the initial numbering:
2  Construct an out-tree T_0 of G and compute numberings ν_l(·) and ν_r(·) of T_0;
3  Let ν_0(·) be the numbering that satisfies more edges of G;
4  Compute G_{ν_0} ⊆ G using ν_0(·);
   // Growing the satisfied subgraph iteratively:
5  while G_{ν_i} ≠ G do
       // Phase 1:
6      Construct an out-tree T_i of G and compute a preorder numbering of T_i;
7      Process T_i in preorder and compute ν'_i(·);
       // Phase 2:
8      Process G_{ν_{i-1}} in order of ν_{i-1}(·) and compute ν''_i(·);
       // Satisfying local edges:
9      Arrange V and E according to ν''_i(·) appropriately;
10     Partition V into sublists that induce subgraphs of size O(M);
11     Compute a topological ordering of each subgraph in memory;
12     Rearrange the vertices in each subgraph according to this ordering;
       // Preparing for next iteration:
13     Compute ν_i(·) by numbering the vertices in order;
14     Compute G_{ν_i} ⊆ G using ν_i(·);
15 return The numbering ν(·) representing the topological ordering of G;
```

The additional cost of this heuristic improvement is O(sort(m)) I/Os per iteration. Indeed, in order to identify the memory-sized subgraphs, we need to label every edge with the positions of its endpoints in the vertex list sorted by $\nu''_i(\cdot)$ and then sort the edges by their endpoints with higher position in the list. This takes O(sort(m)) I/Os.

Given the sorted vertex and edge list, we scan the list and construct the memory-sized subgraphs greedily, starting with the first vertex in the list. When considering a vertex x for inclusion in the current subgraph H, we scan the list of edges that have x as their higher endpoint. Such an edge has both endpoints in H if and only if the lower endpoint of the edge succeeds the first vertex in H in the sorted vertex list. Thus, we can count the number of edges the addition of x would add to H. If the total number of edges in H is still less than M, we include x in H and proceed to considering the next vertex. Otherwise, we make x the first vertex of the next subgraph. This is similar to the construction of memory-sized subgraphs in our strong connectivity algorithm discussed in Chapter 7.

8.3 Other Approaches to Topological Sorting

There are other approaches to topological sorting that are worth considering, as they are either well known or were proposed with I/O efficiency or parallelism in mind and, thus, may achieve better performance than IterTS, at least on certain inputs. In our experiments, we compared the performance of these algorithms to the performance of IterTS.

8.3.1 Topological Sorting Using Semi-External DFS

A classical method for topological sorting is to perform DFS on the DAG and number the vertices in reverse postorder [53]. Building this strategy on top of the semi-external DFS heuristic of [121], one obtains an algorithm for topological sorting that should be very efficient as long as the vertex set of the graph fits in memory. We refer to this algorithm as SeTS.

8.3.2 Iterative Peeling of Sources and Sinks

Another classical method for topological sorting works by iteratively removing sources (in-degree-0 vertices) and sinks (out-degree-0 vertices). In each iteration, the algorithm identifies all sources and sinks of the current DAG and numbers them, sources up from 1, sinks down from N. Then these vertices are removed, which makes some of their neighbours sources or sinks to be removed in the next iteration. This procedure

is repeated until no vertices remain.

A naive implementation of this strategy incurs one random access per edge to test, for each neighbour of a removed vertex, whether it becomes a source or sink as a result of this removal. In contrast, in our experiments, we used the following more I/O-efficient implementation which should be fast for random DAGs and should perform reasonably well even for some other graph classes. We refer to this approach as PeelTS. It consists of two phases: a *preprocessing* phase and a *peeling* phase. The peeling phase iteratively removes the sources and sinks of the DAG I/O-efficiently. The preprocessing phase arranges the vertices (and their adjacency lists) of the DAG so that the locality of data accesses during the peeling phase increases; PeelTS uses an Euler tour of a spanning tree T of the input DAG to order the vertices and their adjacency lists by their depth in T.

Preprocessing. We assume the initial DAG G has only one source, which is easily achieved by adding a new source and connecting it to the original sources. Then we compute an out-tree T of the source as in Section 8.2. We label the vertices in G with their in- and out-degrees and with their depths in T. This information can be computed using the Euler tour technique and list ranking. Now we sort the vertices and their adjacency lists by their depths in T. Let L be the resulting list. By laying out the graph this way, we expect accesses to the vertex list, during the peeling phase, to occur in a relatively local portion. This intuition is confirmed, for some graph classes, in the experimental evaluation.

Peeling. This phase proceeds in a series of iterations. Each iteration starts with the current set of sources and sinks, which we call S^+ and S^- respectively. The task of each iteration is to (a) number the vertices in S^+ and S^-, and (b) identify the new set of sources and sinks passed into the next iteration. To this end, we sort the out-edges of the sources in S^+ by their heads and the in-edges of the sinks in S^- by their tails. Now we scan L from the beginning until we have seen all out-neighbours of S^+ and from the end until we have seen all in-neighbours of S^-. For each out-neighbour of a vertex in S^+, we decrease its in-degree by one and, if its in-degree is now 0, retrieve it and its adjacency list to be used as part of S^+ in the next iteration. We process in-neighbours of S^- analogously.

In order to avoid the vertex and edge lists from becoming too sparse as a result of the removal of vertices and edges, which would contribute unnecessarily to the cost of scanning L, we compact these lists periodically. For some load factor $0 < \alpha < 1$, we call a sublist of L *α-sparse* if less than an α-fraction of the elements in the sublist are unprocessed. In each iteration, we find the longest α-sparse prefix and suffix of the prefix and suffix of L scanned in this iteration, and we compact these two sublists by storing the unprocessed elements in them consecutively. In our implementation, we chose $\alpha = 5\%$, which we determined experimentally gave the best performance.

8.3.3 Divide and Conquer Based on Reachability Queries

In [116], a parallel divide-and-conquer algorithm for topological sorting based on reachability queries is described. We implemented an external-memory version of this algorithm (referred to as ReachTS throughout this chapter).

The algorithm starts by checking if the DAG fits in memory. If it does, it is topologically sorted using an efficient internal-memory algorithm. Otherwise, the following partitioning strategy is applied. First the algorithm arranges the vertices in a random order $x_1, x_2, \ldots, x_n$. Then it uses binary search to find the lowest index k such that vertices $x_1, x_2, \ldots, x_{k-1}$ can reach less than $n/2$ vertices and vertices $x_1, x_2, \ldots, x_k$ can reach at least $n/2$ vertices. Then it computes two sets A and B, where A contains all vertices reachable from $x_1, x_2, ..., x_{k-1}$, and B is the set of vertices reachable from x_k.

Now the following five sets are defined: $S_1 = V \setminus (A \cup B)$, $S_2 = A \setminus B$, $S_3 = \{x_k\}$, $S_4 = B \setminus A$, and $S_5 = A \cap B$. The algorithm recurses on each of the sets in turn and then concatenates the sorted sequences of vertices in $S_1, S_2, \ldots, S_5$ to obtain a topological ordering of G. The correctness of this strategy is shown in [116]. It is also shown that the expected size of each set is $n/2$, making this algorithm terminate after expected $\log n$ levels of recursion.

To find the set of vertices reachable from a set S during the binary search to identify the index k, we use an implementation of directed breadth-first search. This procedure starts by initializing two sets $X := S$ and $Y := S$. The set X is the current BFS level. The set $Y \supseteq X$ is the set of vertices already seen by the BFS. Then it computes a set Z as the set of out-neighbours of vertices in X that are not already

in Y. Afterwards, it sets $Y := Y \cup X$ and $X := Z$. The algorithm iterates this until $X = \emptyset$. The final set Y is the set of vertices reachable from S. Each iteration of this directed BFS procedure can be implemented using $O(\text{sort}(m))$ I/Os. The set Z can be computed by scanning X and the set of edges of G to find all out-edges of vertices in X. Then we sort the set of heads of these edges and scan the sorted list and Y to remove all duplicates and vertices that belong to Y from the sorted list. The result is Z. Since each BFS iteration takes $O(\text{sort}(m))$ I/Os, ReachTS should be efficient if the "diameter" of the graph is low.

8.4 Implementation Details

This section outlines the most important implementation choices made in the different parts of the algorithms presented in Sections 8.2 and 8.3. We implemented IterTS, PeelTS and ReachTS in C++ and using the STXXL library [60], which is an implementation of the C++ STL for external memory computations. For SeTS, we used an implementation provided by Andreas Beckmann [34].

External-memory graph representation. We used STXXL vectors to store the vertex set V and edge set E of the input graphs since they guarantee that the scanning of vertices and edges can be done in $O(\text{scan}(n))$ I/Os and $O(\text{scan}(m))$ I/Os respectively. Every vertex $x \in V$ is represented by its ID and has a label associated with it denoting its current number $\nu(x)$. Every edge (x, y) is represented by the IDs of its two endpoints and by their current numbers $\nu(x)$ and $\nu(y)$. All sorting steps on V and E in our implementation were accomplished using the STXXL sorting algorithm.

Time-forward processing. Time-forward processing (see Section 6.2.2.2) processes the vertices in topologically sorted order and uses a priority queue to simulate the sending of information along the edges of the DAG. For its implementation we used the priority queue provided by STXXL.

Euler tour. We used the standard construction of an Euler tour of a tree (see Section 6.2.2.2), which generates a list of edges incident to each vertex by duplicating each edge and then sorting the edge list. Then a scan of this sorted list suffices to

generate the Euler tour. The edges are stored in an STXXL vector and the STXXL sorting algorithm is used.

List ranking. For list ranking, we used an algorithm of [122]. Ajwani et al. [18] provided an STXXL-based implementation of this algorithm as part of their undirected BFS implementation, and we re-used this code.

Internal-memory graph representation. For processing a graph in internal memory, we used the compact representation described in Section 7.3, which maximizes the size of the subgraphs that can be processed in memory.

8.5 Performance Evaluation

This section presents an experimental study of the behavior of the algorithms proposed in this chapter, using different graph classes. We start with a discussion of our test environment, followed by a description of the data sets used in our tests, and finally an evaluation of the algorithms on these data sets.

8.5.1 Environment and Settings

All experiments were run on a PC with a 3.33GHz Intel Core i5 processor, 4GB of RAM, and one 500GB 7200RPM IDE disk using the XFS file system. The operating system was Ubuntu 9.10 Linux with a 2.6.31 Linux kernel. The code was compiled using g++ 4.4.1 and optimization level –O3. For our experiments we limited the available RAM to 1GB (using the `mem=` kernel option). All of our timing results refer to wall clock times in hours.

8.5.2 Data Sets

We tested the algorithms on synthetic graphs chosen with certain characteristics that should be hard or easy for different algorithms among the ones we implemented. We also ran the algorithms on real web graphs with their edges redirected to ensure that the graphs are acyclic. The number of vertices in the graphs were between 2^{25} and 2^{28}, the number of edges between 2^{27} and 2^{30}. Next we give a description of the graphs used in our experiments.

Random: These graphs were generated according to the $G_{n,m}$ model; that is, m edges were generated, choosing each edge endpoint uniformly at random from a set of n vertices. The edges were directed from lower to higher endpoints.

Width-one: To generate these graphs, we started with a long path of $n - 1$ edges. Then $m-n+1$ random edges with appropriate directions were added according to the $G_{n,m}$ model as for random graphs.

Layered: These graphs consist of $\sqrt{n}$ layers of $\sqrt{n}$ vertices, with random edges between adjacent layers. To generate these graphs, we first chose, for each vertex in a given layer, a random in-neighbour in the previous layer and a random out-neighbour in the next layer. Then we added more random edges between adjacent layers to increase the edge count to m.

Semi-layered: Layered graphs consist of many moderately long paths but are too structured, which makes them extremely easy inputs for PeelTS. Semi-layered graphs aim to have moderately long paths but with less structure. To construct these graphs, we first constructed $q := n^{1/3}$ layered DAGs $G_1, G_2, \ldots, G_q$ consisting of $n^{1/3}$ layers of size $n^{1/3}$ each. Then we added random edges between the DAGs by generating random quadruples (i, j, h, k) with $i < j$ and $h > k$ and, for each such quadruple, adding a random edge from layer h of G_i to layer k of G_j.

Low-width: These graphs were constructed in the same way as layered graphs. However, the number of layers was set to $1,000,000$ in this case and the size of a layer was set to $n/1,000,000$. Moreover, in the first phase of the construction of the graph, which chooses one in- and one out-neighbour per vertex, we connected the ith node in the jth layer to the ith node in layer $j + 1$, thereby starting with $n/1,000,000$ disjoint paths of length $1,000,000$. Then we added random edges between layers as for layered graphs.

Grid: These graphs were formed by taking a $\sqrt{n}$ x $\sqrt{n}$ grid and directing all horizontal edges to the right and all vertical edges down.

Webgraphs: The web graphs were produced by real web crawls of the .uk domain, the .it domain, and from data produced by a more global crawl using the Stanford WebBase crawler. These web graphs were obtained from `http://webgraph.dsi.unimi.it/`. Since they were not necessarily acyclic, we redirected the edges from lower vertex IDs to higher vertex IDs.

8.5.3 Experimental Results

The main goals of our experiments were the following: Compare the algorithms, study how they are affected by the structure of the input graph, and, using the result, recommend which algorithm to use if there is *a priory* knowledge of the graph structure. Table 8.1 shows the running times of the algorithms on different input graphs. In order to bound the time spent on our experiments, we used the following rules.

1. Each algorithm was given an amount of time at least 10 times the time used by IterTS to process the same input. If it did not produce a result in the allocated time, we terminated it. This is indicated by a dash (—) in the table, with superscripts indicating the amount of time given to the algorithm.

2. If IterTS took more than one day to process an input and was consistently faster than another algorithm on smaller inputs, we did not run the other algorithm on this input. This is indicated by stars (***) in the table.

3. Since SeTS is a semi-external algorithm and 2^{26} vertices do not fit in 1GB of memory, we did not run it on larger inputs if it did not finish in the allocated time on the smallest input with 2^{26} vertices (which was the case for all input types).

8.5.3.1 Comparison of Running Times

The initial intuition was that IterTS would perform reasonably well on any of the input graphs, whereas PeelTS and ReachTS would be feasible only on a few of them. In practice, for almost all the inputs, IterTS outperforms both PeelTS and ReachTS. In addition, PeelTS and ReachTS were not able to finish many of the

Table 8.1: Experimental results. Dashes indicate inputs that could not be processed by the algorithm in the allocated time. Superscripts indicate the number of days after which each run was terminated. A superscript of 0 means the run was terminated after 15h. Stars indicate an experiment not performed. Time values are given in hours.

Graph Class	n	m	m/n	IterTS		PeelTS	ReachTS	SeTS
				iters.	time	time	time	time
Random	2^{25}	2^{27}	4	2	0.94	2.71	2.20	0.50
Random	2^{25}	2^{28}	8	5	3.50	8.58	2.39	1.56
Random	2^{26}	2^{28}	4	4	3.47	5.48	4.23	$—^{2}$
Random	2^{26}	2^{29}	8	5	7.48	17.76	10.78	***
Random	2^{27}	2^{29}	4	5	9.22	14.02	9.80	***
Random	2^{28}	2^{30}	4	7	27.13	***	***	***
Width-one	2^{25}	2^{27}	4	4	1.78	$—^{1}$	$—^{0}$	0.05
Width-one	2^{25}	2^{28}	8	6	4.25	$—^{2}$	$—^{2}$	0.08
Width-one	2^{26}	2^{28}	4	8	7.42	$—^{3}$	$—^{3}$	$—^{3}$
Width-one	2^{26}	2^{29}	8	9	13.46	$—^{6}$	$—^{6}$	***
Width-one	2^{27}	2^{29}	4	14	24.90	***	***	***
Width-one	2^{28}	2^{30}	4	19	68.38	***	***	***
Layered	2^{25}	2^{27}	4	2	0.92	2.70	$—^{0}$	0.48
Layered	2^{25}	2^{28}	8	1	0.76	4.62	$—^{1}$	1.17
Layered	2^{26}	2^{28}	4	1	1.02	6.33	$—^{1}$	$—^{1}$
Layered	2^{26}	2^{29}	8	1	1.49	10.76	$—^{1}$	***
Layered	2^{27}	2^{29}	4	3	5.01	25.55	$—^{2}$	***
Layered	2^{28}	2^{30}	4	2	7.14	57.87	$—^{3}$	***
Semi-layered	2^{25}	2^{27}	4	3	1.33	$—^{1}$	2.58	0.34
Semi-layered	2^{25}	2^{28}	8	5	3.26	$—^{2}$	8.02	0.75
Semi-layered	2^{26}	2^{28}	4	5	4.47	$—^{2}$	15.77	$—^{2}$
Semi-layered	2^{26}	2^{29}	8	7	10.08	$—^{4}$	20.83	***
Semi-layered	2^{27}	2^{29}	4	8	14.09	$—^{5}$	66.97	***
Semi-layered	2^{28}	2^{30}	4	9	31.75	***	***	***

... continue on next page

Table 8.1: (continued)

Graph Class	n	m	m/n	ITERTS		PEELTS	REACHTS	SETS
				iters.	time	time	time	time
Low-width	2^{25}	2^{27}	4	1	0.47	—[1]	—[1]	0.35
Low-width	2^{25}	2^{28}	8	1	0.72	—[1]	—[1]	0.93
Low-width	2^{26}	2^{28}	4	1	1.01	—[1]	—[1]	—[1]
Low-width	2^{26}	2^{29}	8	1	1.48	—[1]	—[1]	***
Low-width	2^{27}	2^{29}	4	1	2.03	—[1]	—[1]	***
Low-width	2^{28}	2^{30}	4	1	4.09	—[2]	—[2]	***
Grid	2^{25}	$\approx 2^{26}$	2	1	0.31	4.14	—[1]	0.50
Grid	2^{26}	$\approx 2^{27}$	2	1	0.67	8.46	—[1]	—[1]
Grid	2^{27}	$\approx 2^{28}$	2	1	1.38	18.67	—[1]	***
Grid	2^{28}	$\approx 2^{29}$	2	1	2.84	44.54	—[2]	***
Webgraph	18.5m	298.1m	16.1	3	1.88	—[1]	7.75	1.30
Webgraph	41.3m	1,150.7m	25.9	4	9.06	—[4]	—[4]	3.49
Webgraph	118.1m	1,019.9m	8.6	4	10.13	—[4]	—[4]	—[4]

experiments in the allotted time, that is, ITERTS outperformed them by at least one order of magnitude. Next we discuss our findings in detail.

Using synthetic graphs. Let us first consider the case of *random* graphs which are usually regarded the easiest. As expected, random graphs proved to be easy instances for all algorithms, with usually a factor of less than two between the running times of ITERTS, PEELTS, and REACHTS. On most of the other input graphs, PEELTS and REACHTS were not able to process any of the inputs in the allotted amount of time, that is, ITERTS outperformed them by at least one order of magnitude on these inputs.

PEELTS was able to process all *layered* and *grid* graph instances we tried. For grid graphs, the running time was still more than 10 times higher than that of ITERTS. Layered graphs are a particularly easy input for PEELTS because the preprocessing

stage of the algorithm arranges the vertices layer by layer, which is also the order in which the peeling phase peels sources and sinks. Thus, each peeling round scans exactly those vertices removed from the graph in this round. Nevertheless, the running time of PeelTS here is still at least 3 times higher than that of IterTS.

We designed the *semi-layered* graphs to eliminate the "orderly" structure of layered graphs and, as expected, the performance of PeelTS broke down on these graphs. ReachTS performed better on semi-layered graphs than on layered graphs, at least on the graphs it was able to process in the allotted time. We suspect that this was the result of somewhat shorter shortest paths in the semi-layered graphs, which made the reachability queries in ReachTS cheaper. Finally, IterTS was the only method able to handle *width-one* and *low-width* graphs.

Using web graphs. The results obtained with web graphs presented a surprise, with ReachTS being able to process one of these graphs in 4 times the time taken by IterTS, while not being able to process the bigger web graphs. PeelTS was not able to process any of these graphs in the allotted time. Overall, this is surprising because we expected these graphs to behave similarly to random graphs, particularly given that the edge directions were essentially chosen randomly. Thus, these graphs should not have posed any challenges for any of the algorithms.

Semi-external setting. On inputs whose vertex sets fit in memory ($n = 2^{25}$), SeTS outperformed IterTS on most inputs, while IterTS was faster on some inputs. Width-one graphs turned out to be particularly easy instances for SeTS. On these inputs, it was nearly two orders of magnitude faster than IterTS. This concurs with the discussion in [121], where it was stated that the semi-external DFS algorithm performs very well for deep DFS trees. In experiments with larger graphs, where the vertices did not fit in memory (see Table 7.1 for approximate vertex numbers SeTS can process without using virtual memory), the performance of SeTS immediately deteriorated and it was not able to process any of these inputs within the allotted time, that is, IterTS outperformed SeTS by at least one order of magnitude.

In summary, we conclude that SeTS is the algorithm that should be used for semi-external inputs, while IterTS is the clear choice on larger inputs. PeelTS and ReachTS were not competitive with either SeTS or IterTS. The following

sections discuss possible reasons why these algorithms could or could not process some inputs, shedding some light on their limitations.

8.5.3.2 The Effect of the Graph's Structure

In general, IterTS's performance is fairly consistent across all graph classes, with width-one and semi-layered inputs being the ones that forced the algorithm to use the most iterations. The other algorithms are much more sensitive to the graph's structure. In this section we discuss the effects that the different graph structures have on the algorithms.

Effect on IterTS. Recall that the running time of IterTS is determined mostly by the number of iterations it needs to satisfy all edges in the graph. With the exception of width-one graphs and the larger semi-layered graphs, the number of iterations needed by IterTS was low, even though the graph structure had some impact on the number of iterations needed. Thus, the performance of IterTS can be considered fairly robust and almost independent of the graph's structure. Width-one graphs and the larger semi-layered graphs posed a greater challenge. Nevertheless, while the upper bound on the number of iterations provided by Lemma 8.1 is between 2^{25} and 2^{28} for the input graphs we tested, IterTS needed less than 20 iterations for all of these inputs and was able to process all our input instances in a reasonable amount of time.

Effect on SeTS. SeTS can be considered equally robust on semi-external instances, even though it benefits from deep DFS trees, as already discussed. In contrast, IterTS benefits from graphs having short paths, even according to the pessimistic prediction of Lemma 8.1. Hence, IterTS is competitive with SeTS, for instance, on semi-external random inputs, while SeTS is significantly faster on width-one graphs.

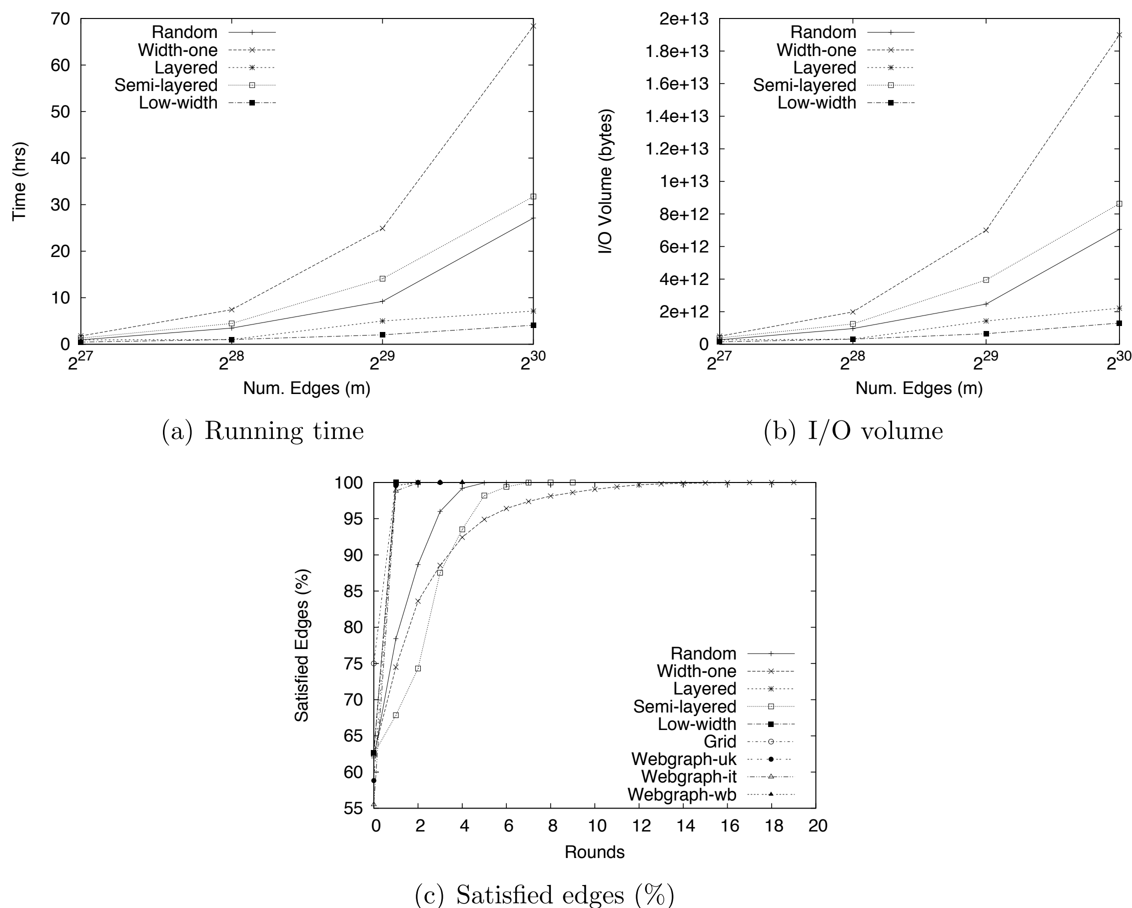
Effect on PeelTS. By definition, PeelTS needs a large number of peeling rounds for graphs with long paths. For example, for the smallest low-width graph, only 5% of the vertices had been removed after 92,000 peeling rounds, while PeelTS needed between 73 and 148 rounds for random graphs. As such, width-one graphs

are intractable for PeelTS since it can only remove one source (and sink) at time. On layered graphs, PeelTS also needed a large number (2898–8194) of rounds. The reason for the good performance of PeelTS on these graphs is that the total cost of the rounds is proportional to the total number of vertices, due to the particular order in which the preprocessing phase arranges the vertices. The same should be true for low-width graphs, which are layered graphs with many small layers. The reason why PeelTS was not able to process them was the large number of peeling rounds, each of which incurred some overhead leading to a cost of 1–5s per peeling round. This overhead could have been eliminated for these graphs, given our knowledge of their structure, but our goal was not to design customized algorithms for individual graph classes.

Effect on ReachTS. ReachTS should perform well on graphs with low diameter and poorly on graphs with long shortest paths, as the most costly part in the algorithm is the BFS-based reachability queries. This intuition is confirmed by its good performance on random graphs and its poor performance on layered, low-width, and grid graphs. For example, the maximum number of BFS levels observed in any reachability query on the random instances was 39, while the smallest low-width graph led to reachability queries with over 1,400 BFS levels before the algorithm was terminated. The performance on semi-layered and width-one graphs, however, contradicts this intuition. Width-one graphs are random graphs, apart from the one path visiting all vertices. So most shortest paths should be short, and the algorithm should perform well, but it did not manage to process any of these instances. Conversely, semi-layered graphs should have fairly long shortest paths; yet, the algorithm performed fairly well on these graphs.

8.5.3.3 Further Analysis of IterTS

Effect of the input size. Figure 8.3(a) shows the running time of IterTS on graphs of different types and sizes but with fixed density. As expected, the running time increased linearly with the input size for layered and low-width graphs, as the number of iterations is nearly independent of the size of the graph. For random, width-one, and semi-layered graphs, the number of iterations required by the algorithm



(a) Running time

(b) I/O volume

(c) Satisfied edges (%)

Figure 8.3: (a) Increase of the running times of ITERTS for graphs with fixed density $m/n = 4$ and increasing m. (b) Increase of the I/O volume. (c) Increase of satisfied edges per iteration for graphs with $n = 2^{28}$ and $m = 2^{30}$.

to terminate increased with the input size, leading to a super-linear dependence of the algorithm on the input size. A nearly identical growth pattern can be seen in Figure 8.3(b) with respect to increasing I/O volume.

Convergence rate. Another interesting factor to consider is how quickly the satisfied subgraph G_ν converged to the whole DAG G. Figure 8.3(c) shows the percentage of satisfied edges as a function of the iteration number for the largest input of each type. As can be seen, with the exception of width-one graphs, the algorithm took only few iterations to satisfy nearly all edges. Even for width-one graphs, 95% of the edges were satisfied after only 6 iterations, and nearly 100% were satisfied after 10 iterations. This implies that, under reasonable assumptions about the ratio between

main memory and disk size, the edges that remained violated after 8–10 iterations fit in memory. It would be helpful to switch to an alternate strategy at this point, which takes advantage of this fact in order to avoid a large number of iterations to satisfy the remaining edges.

One strategy we considered was to identify the set of vertices that cannot be reached by any other vertex having violated in-edges. This way if this "satisfied" vertex prefix is big enough, its removal might reduce the size of the subgraph we have to work with in subsequent iterations. Identifying these vertices requires one time-forward processing pass. Removing them requires sorting and scanning of the vertex and edge sets of the graph. We tried to apply this strategy at different iterations in the run of an experiment; for example, after every iteration, every 2 iterations, and in the case of the two largest width-one graphs, after every 5 iterations. The added cost of identifying the vertices to be removed outweighed the cost reduction resulting from the reduced graph size, and we did not see any performance gain using this strategy.

Effect of satisfying local edges. Our final comment concerns the effect of the local reordering heuristic described in Section 8.2.4 on the running time of the algorithm. It became clear relatively early on that this heuristic speeds up the algorithm tremendously. So we did not run IterTS without the heuristic, except on some of the smaller inputs. For graphs with 2^{25} vertices and 2^{27} edges, we observed a reduction in the number of iterations from between 4 and 21 to between 1 and 3 as a result of the heuristic. The only exceptions were grid graphs, which took one iteration with or without the heuristic, and width-one graphs, which took 4 iterations with the heuristic and which we terminated after 51 iterations without the heuristic.

Overall, the experiments demonstrated that IterTS and SeTS substantially outperform PeelTS and ReachTS and are less susceptible to variations in the graph's structure; though width-one and semi-layered graphs can be considered hard instances for IterTS due to the larger number of iterations they require. While SeTS outperformed IterTS on most inputs whose vertex sets fit in memory, IterTS was able to process larger inputs efficiently, while SeTS was not. As such, we conclude that IterTS is the first algorithm for topologically sorting large DAGs that can efficiently process graphs whose vertex set is beyond the main memory size, while SeTS should

be used on semi-external inputs.

Chapter 9

Conclusions

The primary focus of this thesis was on engineering algorithms that are able to process massive data sets. The problems we studied were skyline computations on multi-dimensional data, and computing strongly connected components and topological sorting of directed graphs. The algorithms we proposed were carefully engineered using fundamental primitives and techniques. Their behavior was empirically evaluated on a large set of benchmark data sets.

Skyline query computation. In Chapters 4 and 5 of this thesis, we proposed skyline algorithms with the aim of querying massive databases. The two main results are a parallel and an I/O-efficient sequential algorithm for computing skylines. Users managing truly massive data warehouses of terabyte to petabyte scale can do so only using large-scale parallel clusters. Our parallel algorithm allows these users to take advantage of the power of these machines to answer skyline queries efficiently. Our I/O-efficient algorithm demonstrates that the same techniques employed in the parallel algorithm also lead to an I/O-efficient algorithm. This algorithm is useful to users processing smaller data sets of a few hundred gigabytes on standard PC hardware.

Algorithms for directed graphs. Chapters 7 and 8 of this thesis focused on solving problems on directed graphs I/O-efficiently. I/O-efficient algorithms for directed graphs are a major frontier in the area of I/O-efficient algorithms. There are almost no theoretical results in this area, nor has there been much previous work on engineering algorithms for directed graphs. The work we presented on strong connectivity and on topological sorting is the first to address these problems from an engineering perspective. Our algorithms are the first that can efficiently process graphs whose vertex sets do not fit in memory. Furthermore, our algorithms demonstrate that a number of techniques, such as graph contraction, the Euler tour technique, list ranking, and

time-forward processing, which were the key to obtaining I/O-efficient algorithms on undirected graphs, can also be used to obtain at least heuristic, practically efficient solutions on directed graphs.

Directions for future work. A potential direction for future research in skyline computation concerns improving the data distribution scheme in the parallel approach. Some follow-up work [132] already looked at better data distribution than ours but they did not consider the cost of the distribution step. As we have seen, for instance, in our strong connectivity and topological sorting algorithms is that, ensuring this kind of improved data distribution—or improved clustering—often comes with a computational overhead. The performance of the algorithm improves only if the benefits of the improved data distribution outweigh the increased cost of computing the distribution.

For strong connectivity, a direction for future work is to overcome the limitations on the types of graph classes the algorithm can process, possibly by combining our strong connectivity and topological sorting algorithms. For topological sorting, an important research direction is to reduce the I/O volume and, thus, the performance of the algorithm. Both algorithms also seem to be (mostly) parallelizable. The combination of I/O-efficiency and parallelism would allow the processing of even larger inputs found, for example, in web modelling applications.

Finally, we hope that the insights we gained in developing our algorithms will also help when engineering algorithms for other graph problems. The techniques used here most likely will not lead to provably efficient solutions for other problems on directed graphs, such as computing shortest paths. Nevertheless, here we demonstrated that essentially the techniques for undirected connectivity problems do work reasonably well in practice in the directed case, so it is possible that the techniques for solving undirected graph exploration problems, such as diameter-based clustering, may also work reasonably well in practice for directed graphs.

Bibliography

[1] Beowulf cluster. http://www.beowulf.org/. Accessed March 21, 2011.

[2] Broad institue. http://www.broadinstitute.org/. Accessed November 26, 2010.

[3] Facebook. http://www.facebook.com/press/info.php?factsheet. Accessed November 11, 2010.

[4] Google earth. http://www.google.com/earth/index.html. Accessed September 15, 2010.

[5] Linkedin. http://press.linkedin.com/about. Accessed November 11, 2010.

[6] The message passing interface (mpi) standard. http://www-unix.mcs.anl.gov/mpi/. Accessed March 21, 2011.

[7] Microsoft bing maps. http://www.bing.com/maps/. Accessed September 15, 2010.

[8] The size of the world wide web. http://www.worldwidewebsize.com/. Accessed September 15, 2010.

[9] Data, data everywhere. In *The Economist*, February 25, 2010. Retrieved online from http://www.economist.com/node/15557443 on September 15, 2010.

[10] Eosdis: Manage unprecedented volumes of earth science data. In *Earth Imaging Journal*, September 2010. Retrieved online from http://www.eijournal.com/05EOSDIS.pdf on November 15, 2010.

[11] J. Abello, P. Pardalos, and M. G. C. Resende. On maximum clique problems in very large graphs. In James M. Abello and Jeffrey Scott Vitter, editors, *External memory algorithms*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 119–130, Boston, MA, USA, 1999. American Mathematical Society.

[12] James Abello, Adam L. Buchsbaum, and Jeffery Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002.

[13] Pankaj K. Agarwal, Lars Arge, and Ke Yi. I/O-efficient batched union-find and its applications to terrain analysis. In *Proceedings of the 22nd ACM Symposium on Computational Geometry*, pages 167–176, 2006.

[14] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[15] Deepak Ajwani, Andreas Beckmann, Riko Jacob, Ulrich Meyer, and Gabriel Moruz. On computational models for flash memory devices. In *Proceedings of the 8th International Symposium on Experimental Algorithms*, pages 16–27, 2009.

[16] Deepak Ajwani, Adan Cosgaya-Lozano, and Norbert Zeh. Engineering a topological sorting algorithm for massive graphs. In *Proceedings of the International Workshop on Algorithm Engineering and Experiments*, 2011.

[17] Deepak Ajwani, Roman Dementiev, and Ulrich Meyer. A computational study of external-memory BFS algorithms. In *Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms*, pages 601–610, 2006.

[18] Deepak Ajwani, Ulrich Meyer, and Vitaly Osipov. Improved external memory BFS implementation. In *Proceedings of the International Workshop on Algorithm Engineering and Experiments*, 2007.

[19] Lars Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–357. Kluwer Academic Publishers, 2002.

[20] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[21] Lars Arge, Gerth Stølting Brodal, and Laura Toma. On external-memory MST, SSSP and multi-way planar graph separation. *Journal of Algorithms*, 53(2):186–206, 2004.

[22] Lars Arge, Jeffrey S. Chase, Patrick Halpin, Laura Toma, Jeffrey S. Vitter, Dean Urban, and Rajiv Wickremesinghe. Efficient flow computation on massive grid terrain datasets. *Geoinformatica*, 7(4):283–313, 2003.

[23] Lars Arge, Jeffrey S. Chase, Patrick Halpin, Laura Toma, Jeffrey S. Vitter, Dean Urban, and Rajiv Wickremesinghe. Efficient flow computation on massive grid terrain datasets. *Geoinformatica*, 7(4):283–313, 2003.

[24] Lars Arge, Ulrich Meyer, Laura Toma, and Norbert Zeh. On external-memory planar depth first search. In *Proceedings of the 7th International Workshop on Algorithms and Data Structures*, pages 471–482. Springer-Verlag, 2001.

[25] Lars Arge, Octavian Procopiuc, and Jeffrey Scott Vitter. Implementing I/O-efficient data structures using tpie. In *ESA '02: Proceedings of the 10th Annual European Symposium on Algorithms*, pages 88–100, London, UK, 2002. Springer-Verlag.

[26] Lars Arge and Laura Toma. Simplified external memory algorithms for planar dags. In Torben Hagerup and Jyrki Katajainen, editors, *Algorithm Theory — SWAT 2004*, volume 3111 of *Lecture Notes in Computer Science*, pages 493–503. Springer Berlin / Heidelberg, 2004.

[27] Lars Arge, Laura Toma, and Norbert Zeh. I/O-efficient topological sorting of planar dags. In *Proceedings of the 15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 85–93, 2003.

[28] Lars Arge and Norbert Zeh. I/O-efficient strong connectivity and depth-first search for directed planar graphs. In *Proceedings of the 44th IEEE Symposium on Foundations of Computer Science*, pages 261–270, 2003.

[29] Nicholas Barr. *The economics of the welfare state*. Oxford University Press, 4. ed. edition, 2004.

[30] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. Salsa: computing the skyline without scanning the whole sky. In *Proceedings of the 2006 ACM CIKM International Conference on Information and Knowledge Management*.

[31] Ilaria Bartolini, Paolo Ciaccia, and Marco Patella. Efficient sort-based skyline evaluation. *ACM Transactions on Database Systems*, 33(4), 2008.

[32] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. In *Proceedings of the 7th International Workshop on Experimental Algorithms*, volume 5038 of *Lecture Notes in Computer Science*, pages 303–318. Springer-Verlag, 2008.

[33] Richard A. Becker, Chris Volinsky, and Allan R. Wilks. Fraud detection in telecommunications: History and lessons learned. In *Technometrics*, volume 52, pages 20–33, February 2010.

[34] Andreas Beckmann. Parallelizing semi-external depth first search. Master's thesis, Martin-Luther-Universität, Halle, Germany, October 2005.

[35] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 322–331, New York, NY, USA, 1990. ACM Press.

[36] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[37] J. L. Bentley, H. T. Kung, M. Schkolnick, and C. D. Thompson. On the average number of maxima in a set of vectors and applications. *Journal of the ACM*, 25(4):536–543, 1978.

[38] Toby Bloom and Ted Sharpe. Managing data from high-throughput genomic processing: A case study. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 1198–1201, 2004.

[39] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Proceedings of the 17th International Conference on Data Engineering*, pages 421–430, 2001.

[40] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the 7th International Conference on World Wide Web*, WWW7, pages 107–117, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B. V.

[41] Andrei Broder, Ravi Kumar, Farzin Maghoul, Prabhakar Raghavan, Sridhar Rajagopalan, Raymie Stata, Andrew Tomkins, and Janet Wiener. Graph structure in the web. In *Proceedings of the 9th International World Wide Web Conference*, pages 309–320, Amsterdam, The Netherlands, The Netherlands, 2000. North-Holland Publishing Co.

[42] Adam L. Buchsbaum, Michael Goldwasser, Suresh Venkatasubramanian, and Jeffery R. Westbrook. On external memory graph traversal. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 859–860, 2000.

[43] Giuseppe Cattaneo and Giuseppe Italiano. Algorithm engineering. *ACM Computing Surveys*, 31:3, September 1999.

[44] cgmLab Portal. Hydro1k elevation derivative database. `http://cgmlab.cs.dal.ca/downloadarea/datasets/`.

[45] C. Y. Chan, H. V. Jagadish, K. Tan, A. K. H. Tung, and Z. Zhang. On high dimensional skylines. In *International Conference on Extending Database Technology*, pages 478–495, 2006.

[46] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Record*, 26(1):65–74, 1997.

[47] Hekang Chen, Shuigeng Zhou, and Jihong Guan. Towards energy-efficient skyline monitoring in wireless sensor networks. In *Proceedings of the 4th European Conference on Wireless Sensor Networks*, pages 101–116, Berlin, Heidelberg, 2007. Springer-Verlag.

[48] Yen-Yu Chen, Qingqing Gan, and Torsten Suel. I/O-efficient techniques for computing pagerank. In *Proceedings of the Eleventh International Conference on Information and Knowledge Management*, CIKM '02, pages 549–557, New York, NY, USA, 2002. ACM.

[49] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.

[50] Jan Chomicki, Parke Godfrey, Jarek Gryz, and Dongming Liang. Skyline with presorting: Theory and optimizations. In Mieczyslaw A. Klopotek, Slawomir T. Wierzchon, and Krzysztof Trojanowski, editors, *Intelligent Information Systems*, Advances in Soft Computing, pages 595–604. Springer, 2005.

[51] AT&T Labs-Research Chris Volinsky. Mining massive graphs for telecommunication applications. `http://www.cs.umd.edu/mlg2010/keynotes/mlg2010-keynote-volinsky.pdf`, July 2010. Accessed November 11, 2010.

[52] Thayne Coffman, Seth Greenblatt, and Sherry Marcus. Graph-based technologies for intelligence analysis. *Communications of the ACM*, 47:45–47, March 2004.

[53] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1990.

[54] Adan Cosgaya-Lozano, Andrew Rau-Chaplin, and Norbert Zeh. Parallel computation of skyline queries. In *Proceedings of the 21st International Symposium on High Performance Computing Systems and Applications*, page 12, 2007.

[55] Adan Cosgaya-Lozano and Norbert Zeh. A heuristic strong connectivity algorithm for large graphs. In *Proceedings of the 8th International Symposium on Experimental Algorithms*, pages 113–124, 2009.

[56] Bin Cui, Hua Lu, Quanqing Xu, Lijiang Chen, Yafei Dai, and Yongluan Zhou. Parallel distributed processing of constrained skyline queries by filtering. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 546–555, Washington, DC, USA, 2008. IEEE Computer Society.

[57] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proceedings of the Ninth Annual Symposium on Computational Geometry*, pages 298–307, 1993.

[58] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering route planning algorithms. *Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation*, pages 117–139, 2009.

[59] R. Dementiev. Stxxl homepage, documentation, and tutorial. `http://stxxl.sourceforge.net`, 2000.

[60] Roman Dementiev, Lutz Kettner, and Peter Sanders. STXXL: Standard library for XXL data sets. *Software: Practice and Experience*, 38(6):589–637, 2007.

[61] Roman Dementiev, Peter Sanders, Dominik Schultes, and Jop F. Sibeyn. Engineering an external memory minimum spanning tree algorithm. In *Proceedings of the 3rd International Conference on Theoretical Computer Science*, pages 195–208, 2004.

[62] Prasanna Kumar Desikan, Nishith Pathak, Jaideep Srivastava, and Vipin Kumar. Divide and conquer approach for efficient pagerank computation. In *Proceedings of the 6th International Conference on Web Engineering*, ICWE '06, pages 233–240, New York, NY, USA, 2006. ACM.

[63] D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[64] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[65] Debora Donato, Luigi Laura, Stefano Leonardi, and Stefano Millozzi. The web as a graph: How far we are. *ACM Transactions on Internet Technology*, 7(1), February 2007.

[66] Debora Donato, Stefano Leonardi, Stefano Millozzi, and Panayiotis Tsaparas. Mining the inner structure of the web graph. *Journal of Physics A: Mathematical and Theoretical*, 41:(12pp), May 2008.

[67] M. Ehrgott. *Multicriteria optimization*. Lecture Notes in Economics and Mathematical Systems. Springer-Verlag, 2000.

[68] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2):138–163, 2005.

[69] Yunjun Gao, Gencai Chen, Ling Chen, and Chun Chen. Parallelizing progressive computation for skyline queries in multi-disk environment. In *Proceedings of the 17th International Conference on Database and Expert Systems Applications*, pages 697–706, 2006.

[70] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contration hierarchies: Faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th International Workshop on Experimental Algorithms*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer-Verlag, 2008.

[71] Parke Godfrey, Ryan Shipley, and Jarek Gryz. Maximal vector computation in large data sets. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 229–240, 2005.

[72] Andreas Gogol-Döring and Knt Reinert. *Biological Sequence Analysis using the SeqAn C++ Library*. CRC Press, 2010.

[73] Andrew V. Goldberg and Renato Werneck. Computing point-to-point shortest paths from external memory. In *Proceedings of the International Workshop on Algorithm Engineering and Experiments*, pages 26–40, 2005.

[74] Bruce C. Greenwald and Joseph E. Stiglitz. Externalities in economies with imperfect information and incomplete markets. *The Quarterly Journal of Economics*, 101(2):229–264, 1986.

[75] Xiangquan Gui, Yuanping Zhang, and Xiaohong Hao. An almost linear I/O algorithm for skyline query. *Journal of Software*, 5(2):235–242, February 2010.

[76] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, New York, NY, USA, 1984. ACM Press.

[77] Un gyu Baek, Sukhyun Ahn, and Seung won Hwang. Dynamic partitioning for parallel skyline computation. In *The 1st International Conference on Emerging Databases*, 2009.

[78] Herman Haverkort and Laura Toma. I/O-efficient algorithms on near-planar graphs. In Jos Correa, Alejandro Hevia, and Marcos Kiwi, editors, *LATIN 2006: Theoretical Informatics*, volume 3887 of *Lecture Notes in Computer Science*, pages 580–591. Springer Berlin / Heidelberg, 2006.

[79] D. Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

[80] Zhiyong Huang, Christian S. Jensen, Hua Lu, and Beng Chin Ooi. Skyline queries against mobile lightweight devices in manets. In *Proceedings of the 22nd International Conference on Data Engineering*, page 66, Washington, DC, USA, 2006. IEEE Computer Society.

[81] H. V. Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton: a balanced tree structure for peer-to-peer networks. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 661–672, 2005.

[82] Jing Jiang, Yafei Dai, and Ben Y. Zhao. Understanding latent interactions in online social networks. In *Proceedings of The 10th ACM SIGCOMM Internet Measurement Conference*, November 2010.

[83] Ibrahim Kamel and Christos Faloutsos. Parallel r-trees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 195–204, New York, NY, USA, 1992. ACM.

[84] Donald E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[85] Donald E. Knuth and Jayme Luiz Szwarcfiter. A structured program to generate all topological sorting arrangements. *Information Processing Letters*, 3(2):64, 1974.

[86] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *Proceedings of the 28th International Conference on Very Large Data Bases*, 2002.

[87] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, and Andrew Tomkins. The web and social networks. *Computer*, 35(11):32–36, 2002.

[88] Vijay Kumar and Eric J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP '96)*, SPDP '96, pages 169–, Washington, DC, USA, 1996. IEEE Computer Society.

[89] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM*, 22(4):469–476, 1975.

[90] Luigi Laura, Stefano Leonardi, Stefano Millozzi, Ulrich Meyer, and Jop Sibeyn. Algorithms and experiments for the webgraph. In *Algorithms — ESA 2003*, volume 2832 of *Lecture Notes in Computer Science*, pages 703–714. Springer Berlin / Heidelberg, 2003.

[91] Ken C. K. Lee, Baihua Zheng, Huajing Li, and Wang-Chien Lee. Approaching the skyline in z order. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 279–290, 2007.

[92] Yuhua Li, Dongsheng Duan, Guanghao Hu, and Zhengding Lu. Discovering hidden group in financial transaction network using hidden markov model and genetic algorithm. *Fourth International Conference on Fuzzy Systems and Knowledge Discovery*, 5:253–258, 2009.

[93] Anil Maheshwari and Norbert Zeh. I/O-optimal algorithms for planar graphs using separators. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 372–381, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[94] Anil Maheshwari and Norbert Zeh. A survey of techniques for designing I/O-efficient algorithms. In *Algorithms for Memory Hierarchies*, volume 2625 of *Lecture Notes in Computer Science*, pages 36–61. Springer-Verlag, 2002.

[95] Anil Maheshwari and Norbert Zeh. I/O-optimal algorithms for outerplanar graphs. *Journal of Graph Algorithms and Applications*, 8:47–87, 2004.

[96] Anil Maheshwari and Norbert Zeh. I/O-efficient planar separators. *SIAM Journal on Computing*, 38(3):767–801, 2008.

[97] Anil Maheshwari and Norbert Zeh. I/O-efficient algorithms for graphs of bounded treewidth. *Algorithmica*, 54:413–469, 2009.

[98] Kurt Mehlhorn and Ulrich Meyer. External-memory breadth-first search with sublinear I/O. In *Proceedings of the 10th European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 723–735. Springer-Verlag, 2002.

[99] Kurt Mehlhorn, Stefan Näher, and Peter Sanders. Engineering DFS-based graph algorithms. `http://www.mpi-inf.mpg.de/~mehlhorn/ftp/EngineeringDFS.pdf`, 2007.

[100] Ulrich Meyer and Vitaly Osipov. Design and implementation of a practical I/O-efficient shortest paths algorithm. In Irene Finocchi and John Hershberger, editors, *Proceedings of the International Workshop on Algorithm Engineering and Experiments*, pages 85–96. SIAM, 2009.

[101] Ulrich Meyer, Peter Sanders, and Jop Sibeyn, editors. *Algorithms for memory hierarchies: advanced lectures*, volume 2625 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, 2003.

[102] Ulrich Meyer and Norbert Zeh. I/O-efficient undirected shortest paths. In *Proceedings of the 11th Annual European Symposium on Algorithms*, pages 434–445, 2003.

[103] Ulrich Meyer and Norbert Zeh. I/O-efficient undirected shortest paths with unbounded edge lengths. In *Proceedings of the 14th European Symposium on Algorithms*, pages 540–551, 2006.

[104] Thomas Mølhave, Pankaj K. Agarwal, Lars Arge, and Morten Revsbaek. Scalable algorithms for large high-resolution terrain data. In *Proceedings of the 1st International Conference and Exhibition on Computing for Geospatial Research and Application*, COM.Geo '10, pages 20:1–20:7, New York, NY, USA, 2010. ACM.

[105] D. Moore. Fast Hilbert curve generation, sorting, and range queries. `http://www.tiac.net/~sw/2008/10/Hilbert/moore/hilbert.c`, 2000.

[106] Kamesh Munagala and Abhiram G. Ranade. I/O-complexity of graph algorithms. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 687–694, 1999.

[107] Amit A. Nanavati, Siva Gurumurthy, Gautam Das, Dipanjan Chakraborty, Koustuv Dasgupta, Sougata Mukherjea, and Anupam Joshi. On the structural properties of massive telecom call graphs: findings and implications. In *Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 435–444, New York, NY, USA, 2006. ACM.

[108] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 467–478, New York, NY, USA, 2003. ACM Press.

[109] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. Progressive skyline computation in database systems. *ACM Transactions on Database Systems*, 30(1):41–82, 2005.

[110] Sungwoo Park, Taekyung Kim, Jonghyun Park, Jinha Kim, and Hyeonseung Im. Parallel skyline computation on multicore architectures. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 760–771, Washington, DC, USA, 2009. IEEE Computer Society.

[111] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer, 1985.

[112] N. Pržulj, D. G. Corneil, and I. Jurisica. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. *Bioinformatics*, 22:974–980, April 2006.

[113] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, New York, NY, USA, 2001. ACM.

[114] Peter Sanders. Algorithm engineering — an attempt at a definition. In *Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, pages 321–340, Berlin, Heidelberg, 2009. Springer-Verlag.

[115] Peter Sanders, Dominik Schultes, and Christian Vetter. Mobile route planning. In *Proceedings of the 16th European Symposium on Algorithms*, volume 5193 of *Lecture Notes in Computer Science*, pages 732–743. Springer-Verlag, 2008.

[116] Warren Schudy. Finding strongly connected components in parallel using $O(\log^2 n)$ reachability queries. In *Proceedings of the 20th Symposium on Parallelism in Algorithms and Architectures*, pages 146–151, New York, NY, USA, 2008. ACM.

[117] Dominik Schultes. External memory minimum spanning trees. `http://algo2.iti.uni-karlsruhe.de/schultes/emmst`, 2003.

[118] Dominik Schultes. External memory spanning forests and connected components. `http://algo2.iti.uni-karlsruhe.de/dementiev/files/cc.pdf`, September 2003.

[119] S. B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269–287, 1983.

[120] Mehdi Sharifzadeh and Cyrus Shahabi. The spatial skyline queries. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 751–762, 2006.

[121] Jop Sibeyn, James Abello, and Ulrich Meyer. Heuristics for semi-external depth-first search on directed graphs. In *Proceedings of the 14th ACM Symposium on Parallel Algorithms and Architectures*, pages 282–292, 2002.

[122] Jop F. Sibeyn. From parallel to external list ranking. Technical report, Max Planck Institut für Informatik, Saarbrücken, Germany, 1997.

[123] Jop. F. Sibeyn. External connected components. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *Lecture Notes in Computer Science*, pages 468–479. Springer-Verlag, 2004.

[124] Kian-Lee Tan, Pin-Kwang Eng, and Beng Chin Ooi. Efficient progressive skyline computation. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 301–310, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[125] Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[126] Yuanyuan Tian, Richard C. Mceachin, Carlos Santos, David J. States, Jignesh M. Patel, and Martin Bishop. Bioinformatics saga: A subgraph matching tool for biological graphs. *Bioinformatics*, 23(2):232–239, 2007.

[127] Yuanyuan Tian and Jignesh M. Patel. Tale: A tool for approximate large graph matching. *Proceedings of the 24th International Conference on Data Engineering*, 0:963–972, 2008.

[128] George Valkanas and Apostolos N. Papadopoulos. Efficient and adaptive distributed skyline computation. In Michael Gertz and Bertram Ludäscher, editors, *SSDBM*, volume 6187 of *Lecture Notes in Computer Science*, pages 24–41. Springer, 2010.

[129] Darren Erik Vengroff. A transparent parallel I/O environment. In *In Proceedings of the third DAGS Symposium on Parallel Computation*, pages 117–134, July 1994.

[130] Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[131] Jeffrey Scott Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2006.

[132] Akrivi Vlachou, Christos Doulkeridis, and Yannis Kotidis. Angle-based space partitioning for efficient parallel skyline computation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 227–238, New York, NY, USA, 2008. ACM.

[133] Shiyuan Wang, Beng Chin Ooi, and Anthony K. H. Tung. Efficient skyline query processing on peer-to-peer networks. In *Proceeding of the IEEE International Conference on Data Engineering*, pages 1126–1135, 2007.

[134] Shiyuan Wang, Quang Hieu Vu, Beng Chin Ooi, Anthony K. Tung, and Lizhen Xu. Skyframe: a framework for skyline query processing in peer-to-peer systems. *The VLDB Journal*, 18(1):345–362, 2009.

[135] Ping Wu, Caijie Zhang, Ying Feng, Ben Y. Zhao, Divyakant Agrawal, and Amr El Abbadi. Parallelizing skyline queries for scalable distribution. In *Proceedings of the 10th International Conference on Extending Database Technology*, pages 112–130, 2006.

[136] Y. Yuan, X. Lin, Q. Liu, W. Wang, J. Xu Yu, and Q. Zhang. Efficient computation of the skyline cube. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 241–252, 2005.

[137] Y. Zhang and M. S. Waterman. An Eulerian path approach to global multiple alignment for DNA sequences. *Journal of Computational Biology*, 10:803–820, 2003.

[138] Yu Zhang and Michael S. Waterman. An Eulerian path approach to local multiple alignment for DNA sequences. *Proceedings of the National Academy of Sciences of the United States of America*, 102(5):1285–1290, 2005.