INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

UMI[®]
800-521-0600

Parallel Algorithms in External Memory

By
David Alexander Hutchinson

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Doctor of Philosophy

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

May 1999

© Copyright
1999, David Alexander Hutchinson

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-42793-5

Canada

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis.

Parallel Algorithms in External Memory

submitted by

David Alexander Hutchinson

Dr. Frank Fiala
(Acting Director, School of Computer Science)

Dr. Frank Dehne
(Thesis Co-Supervisor)

Dr. Anil Maheshwari
(Thesis Co-Supervisor)

Dr. Jörg-Rüdiger Sack
(Thesis Co-Supervisor)

Dr. Abhiram Ranade
(External Examiner)

Carleton University

May 1999

Abstract

External memory (EM) algorithms are designed for computational problems in which the size of the internal memory of the computer is only a small fraction of the problem size. The Parallel Disk Model (PDM) of Vitter and Shriver is widely used to discriminate between external memory algorithms on the basis of input/output (I/O) complexity. Parallel algorithms are designed to efficiently utilize the computing power of multiple processing units, interconnected by a communication mechanism. A popular model for developing and analyzing parallel algorithms is the Bulk Synchronous Parallel (BSP) model due to Valiant.

In this work we develop simulation techniques, both randomized and deterministic, which produce efficient EM algorithms from efficient algorithms developed under BSP-like parallel computing models. Our techniques can accommodate one or multiple processors on the EM target machine, each with one or more disks, and they also adapt to the disk blocking factor of the target machine.

We propose new, more comprehensive models for EM and parallel algorithms which consider the total costs incurred by the algorithm including computation, I/O and communication. The new EM-BSP, EM-BSP*, and EM-CGM models combine the features of the BSP and PDM and thereby answer to a challenge posed by the ACM Working Group on Storage I/O for Large-Scale Computing.

We obtain parallel external memory algorithms for a large number of problems including sorting, permutation, matrix transpose, geometric and GIS problems including 3D convex hulls (2D Voronoi diagrams), and various graph problems.

Acknowledgements

I would like to thank all of the people who have helped me to complete the work described in this thesis. Knowing that I cannot thank everyone who has helped me in my studies, and fearing that I would forget someone who has helped a great deal, I have procrastinated as usual in writing these acknowledgements. However, the time has come, so I hope that those who I have not listed will forgive me. I am sure that I will remember all of you tomorrow, after the publication deadline has passed.

I would like to thank my three advisors, Frank Dehne, Anil Maheshwari, and Jörg-Rüdiger Sack for their patience, advice, and for their confidence that I would eventually finish this work.

In particular, I want to thank my initial advisor, Jörg-Rüdiger Sack for taking a gamble on me, for the excellent infrastructure and financial support he provided, and for his critical advice (both on technical and non-technical issues).

I would like to thank Frank Dehne for his help and encouragement in writing papers, and for his unflagging enthusiasm for our results and my prospects.

To Anil Maheshwari, I offer my gratitude for the many hours we spent discussing Computer Science, and other things, over the past several years, and for his unfailing willingness to do so on an impromptu basis. Without his day to day help, encouragement and quiet confidence in our work, I would not have been successful.

I would like to express my appreciation to Wolfgang Dittrich for his collaboration on many nice results and for showing me the power of mathematics in expressing ideas. It has been a real pleasure working with him.

To my external examiner, Abhiram Ranade, and the other members of my examination committee, Thomas Kunz, Ekow Otoo, and Ivan Stojmenovic, thanks are due for their hard work in reading this thesis and for the useful suggestions they made.

I would like to thank my other co-authors, Mark Lanthier, Doron Nussbaum, David Roytenberg, Radu Velicescu, Norbert Zeh, and also Patrick Morin, Kumanan Yogaratnam, for many fruitful discussions and other cooperative work.

For their encouragement, comments, and questions, which led me to new ideas, I would like to acknowledge the help of Lars Arge, Tom Cormen, and Jeff Vitter. To Lars, thanks also for a comprehensive BibTeX file on external memory algorithms. This was a great help to me.

I would like to thank the people of the School of Computer Science for the excellent working environment during my studies. Particular thanks go to Rosemary Carter, Barbara Coleman, Linda Pfeiffer, Maureen Sherman, and Marlene Wilson. Thanks also to the technical staff of SCS who kept the machines running and put up with my many suggestions.

To all of my faculty, graduate and undergraduate colleagues who I have worked with as part of the Paradigm Group, I offer my appreciation for your contributions to a great working environment. Particular thanks go to Mark Lanthier for organizing the coke fund, the pizza schedule, and the hockey games, and to Doron Nussbaum and Pat Morin for organizing the pizza talks.

Finally, I must acknowledge the help and support of a group of people that have made the crucial difference between success and failure for me. First, I would like to thank my parents, for instilling the desire to stretch for new and wonderful things, and for their unconditional support in all of my endeavours. Lastly, I offer very special thanks to my wife Gail and our children Lauren, Julia, Steven and Jessica, for their support and encouragement, for their cheerfulness, and for their love and affection. I could not have continued without your help.

David Hutchinson

Ottawa

May, 1999

This research was partially supported by Almerco, Inc., by the Natural Sciences and Engineering Research Council of Canada, and by pTeran Computing, Inc.

Contents

Abstract	iii
Acknowledgements	iv
1 Introduction	1
1.1 Overview	1
1.1.1 Motivation	1
1.1.2 What Makes a Good EM Algorithm?	3
1.1.3 Effectiveness of EM Techniques	5
1.1.4 Fundamental Assumptions for Implementing Efficient I/O Algorithms	5
1.2 Basic Data Formats on Multiple Disks	6
1.3 Previous EM Research	9
1.3.1 A Brief Survey	10
1.3.2 External Memory Paradigms	14
1.3.2.1 Batch Filtering	14
1.3.2.2 Distribution Sweeping	15
1.3.2.3 Data Structuring	16
1.3.2.4 Simulation	17
1.3.3 Lower Bounds on I/O Complexity	18
1.3.3.1 Sorting Lower Bound	18
1.3.3.2 Permutation Lower Bound	19
1.3.3.3 Matrix Transpose Lower Bound	19
1.3.3.4 A General Lower Bounding Technique	19

1.4	Realistic Parameter Ranges	19
1.5	The Role of RAIDs	20
1.6	Storage Space	21
1.7	Preliminaries - Notation and Terminology	21
1.7.1	Asymptotic Notation	21
1.7.2	BSP Models	22
1.8	A Central Issue: Communication Bottlenecks	22
1.9	Papers Contributing to the Thesis	23
1.10	Claim to Originality or Contribution to Knowledge	25
1.10.1	Collaborations	25
1.10.2	Contributions	25
1.11	Organization of the Thesis	27
2	Models	30
2.1	Overview	30
2.1.1	Organization of This Chapter	31
2.2	EM Models	31
2.2.1	Aggarwal and Vitter Single Disk Model	31
2.2.2	The Parallel Disk Model (PDM)	32
2.2.3	Hierarchical Memory Models	34
2.3	Parallel Computation Models	34
2.3.1	The Parallel Random Access Memory (PRAM) Model	35
2.3.2	The Bulk Synchronous Parallel (BSP) Model	36
2.3.3	The Extended BSP (BSP*) Parallel Processing Model	38
2.3.4	The Coarse Grained Multicomputer (CGM) Model	40
2.3.5	Other Parallel Computing Models	41
2.4	Combining PDM and BSP-like Models	41
2.4.1	Objective: A More Comprehensive Model	41
2.4.2	The EM-BSP Model	42
2.4.2.1	System Model	42
2.4.2.2	Cost model	43
2.4.3	The EM-BSP* Model	44

2.4.4	The EM-CGM Model	44
2.4.5	Goodness Criteria for Parallel EM Algorithms	45
2.4.5.1	Extending the c -optimality Criterion	45
2.4.5.2	Alternative Goodness Criteria for EM-BSP Algorithms	46
3	EM-BSP Algorithms from Simulation	47
3.1	Overview of the Method	47
3.2	Overview and Rationale	55
4	Randomized Simulation	56
4.1	Single Processor Target Machine	56
4.2	Multiple Processor Target Machine	63
5	Deterministic Simulation	71
5.1	Introduction	71
5.2	Balancing Communication	72
5.3	Single Processor Target Machine	77
5.4	Multiple Processor Target Machine	83
5.5	Summary	86
6	Deterministic Simulation with Lower Slackness	88
6.1	Motivation	88
6.2	Overview	89
6.3	The Parallel Algorithm <i>LowSlackRouting</i>	92
6.4	The Parallel Algorithm <i>NumberBlocks</i>	93
6.5	Analysis of <i>LowSlackRouting</i>	96
6.6	The EM Algorithm <i>EMLowSlackRouting</i>	101
6.7	The EM Algorithm <i>EMNumberBlocks</i>	104
6.8	Analysis of <i>EMLowSlackRouting</i>	106
6.9	Single Processor Target Machine	111
6.10	Multiple Processor Target Machine	112
7	Problems of Geometrically Changing Size	115
7.1	Introduction	115

7.2	Reducing I/O Overheads	116
8	New EM Algorithms	122
8.1	Overview	122
8.2	Apparent Reductions in I/O Complexity	122
8.3	Fundamental Problems	123
8.3.1	Sorting	124
8.3.2	Permutation	125
8.3.3	Matrix Transpose	125
8.3.4	Summary	126
8.4	The Multisearch Problem	128
8.4.1	Overview	128
8.4.2	Multisearch on the EM-CGM model	130
8.4.3	EM-BSP* Multisearch	131
8.4.3.1	EM-BSP* Multisearch on a Small Tree	132
8.4.3.2	EM-BSP* Multisearch on a Large Tree	133
8.4.4	Summary	134
8.5	Other New External Memory Algorithms	136
9	Experiments	141
9.1	Buffer Tree Case Study	143
9.1.1	Introduction	143
9.1.2	Description of the Buffer Tree	144
9.1.3	Implementation of the Buffer Tree	147
9.1.4	Evaluating the Implementation	150
9.1.5	Conclusions and Lessons Learned	156
9.2	Simulation Case Study	159
9.2.1	Introduction and Motivation	159
9.2.2	An EM Run-time System for Parallel Algorithms	160
9.2.2.1	One Real Processor	160
9.2.2.2	Multiple Real Processors	166
9.2.3	A Deterministic EM-CGM Samplesort	168

9.2.4	Experimental Results	170
9.2.5	Discussion and Conclusions	170
10	Conclusions and Future Work	173
A	Useful Probabilistic Results	176
A.1	Tail Estimates	176
A.2	Balancing Lemmas	179
A.3	Summary of Notation and Index	183
	Bibliography	188

List of Figures

1.1	A Disk Drive.	7
1.2	Layout of Disk Blocks on a Single Processor Machine.	9
1.3	Layout of Disk Blocks on a Multiple Processor Machine.	10
2.1	The Parallel Disk Model with One Processor.	32
2.2	The Parallel Disk Model with Multiple Processors.	33
2.3	A BSP Computation.	37
2.4	A Parallel Machine with External Memory	43
3.1	The memory of a virtual processor.	49
4.1	Reorganization in Step (2) of SimulateRouting.	59
5.1	Illustration of maximum imbalance in the bin sizes of a processor during superstep A.	75
5.2	Illustration of the layout of message blocks on the D disks.	82
6.1	The division of the $ S $ processors into v^ϵ groups.	90
6.2	The redistribution of blocks for a given bucket.	94
6.3	The tree of processors which numbers the blocks in bucket 0.	96
6.4	The tree for the numbering of blocks in bucket 1.	97
6.5	Imbalance in Item Assignment to Processors	99
6.6	The assignment of messages to local disks on a real destination processor.	105
6.7	Imbalance in Data Received by the Real Processors	110
8.1	Overview of New EM Algorithms	127

8.2	Multisearch example.	129
8.3	Overview of New EM Algorithms (continued)	139
8.4	Overview of New EM Algorithms (continued)	140
9.1	The buffer tree.	144
9.2	Timings for Internal Quicksort and Buffer Tree Sort on Random Inputs.151	
9.3	Timings for Buffer Tree Sort and TPIE Merge Sort.	152
9.4	Number of Block Pushes for Buffer Tree Sort on Random Inputs. . .	153
9.5	Timings for Buffer Tree Sort with Smaller Fanout.	154
9.6	Timings for Buffer Tree Sort with Larger Fanout.	155
9.7	Pipelining sort with generation of inputs.	156
9.8	Running Times of PDM and Iostream I/O Access Methods.	157
9.9	Stevens' measurements on the effects of varying the block size. . . .	158
9.10	Organization of EM Library Services.	164
9.11	System Design of the External Memory Library.	167
9.12	Running times for CGMSampleSort and EM-CGM SampleSort. . . .	171

Chapter 1

Introduction

1.1 Overview

1.1.1 Motivation

In recent years, computer processor speeds have increased dramatically, computer storage device capacities have increased, prices have dropped, and applications have grown in their demands on storage resources. Disk speeds have not kept pace with main memory speeds, however. The bottleneck between main memory and “external memory” (EM) devices, such as disks, threatens to limit the size of very large scale applications [55, 69, 74]. While internal memories have also become much more reliable, cheaper, and larger, we see many new and up-sized applications that demand more storage than is feasible through internal memories. The area of external memory (EM) algorithm design aims to create algorithms which perform efficiently when the internal memory of the computer is significantly smaller than the storage required for the application data. Such applications require the bulk of their data to reside in external memory since, for practical machines, main memory is only a fraction of the required size.

Applications in geographic information systems (GIS), astrophysics, virtual reality, computational biology, VLSI design, weather prediction, computerized medical treatment, 3D simulation and modelling, visualization and computational geometry

Input: We are given a list of N items to be permuted, where N is much larger than the the number of items M that fit into the internal memory of the computer. The permutation to be applied is given as a list of N values.

Output: The permuted list of N items is to be output.

Assumptions: We assume that N is 10^{10} items (10+ GB), M is 5×10^7 items, the average time for a disk access is 10^{-3} seconds, and the disk block size B is 1×10^3 items.

The inputs, the specified permutation, and the resulting output list must all be stored on disk (external memory). In internal memory, the output can be computed in $O(N)$ time. If we apply the same approach to our problem, however, we expend $O(N)$ I/Os in the worst case, or I/O time of $c \times 10^7$ sec. for some constant c . However, it has been shown that permutation in external memory requires only $\Theta(\frac{N}{B} \log_{(M/B)} \frac{N}{B})$ I/Os [4], giving a time of $c' \times 10^4 \log_{5 \times 10^4} 10^7$ seconds for some constant c' . So, assuming similar constants c and c' we have the EM algorithm for permutation running about $B = 1000$ times faster than the internal memory approach. This illustrates the most important factor in a good EM algorithm, which is taking full advantage of the disk blocking factor B .

Example 1.1: Permutation in External Memory

fall naturally into this category. Parallel disk I/O and parallel computing have been identified as critical components of a suitable high performance computer for a number of the Grand Challenge problems (see for instance the Scalable I/O Initiative project [72, 75]).

The time required to perform an I/O operation is typically several orders of magnitude larger than the time for an internal CPU operation. In addition, the time to set up the data transfer (disk head movement, rotational delay) is much larger than the time to actually transfer the data. Data items are grouped into *blocks* and accessed block-wise by efficient external memory algorithms in order to amortize the setup time over a large number of data items. Algorithms developed originally for internal memory models, such as the RAM and PRAM models, are frequently found to be inefficient in the EM environment for this reason. Such algorithms typically have poor “locality of reference,” causing too many I/O operations to be performed

during the execution of an application.

Prior to the evolution of a distinct EM methodology, large problems were often handled via “virtual memory” techniques, which used demand paging to swap pages between disk and internal memory. Unfortunately such techniques are heuristic and the performance can be very poor in difficult cases. As an example, the simple problem of permuting N numbers is offered as motivation for the study of external memory algorithms. See Example 1.1.

As a result, researchers have developed special algorithms optimized for EM computation. In many cases, these have been “hand-crafted”, so the choices for an application developer have been somewhat limited as a result. One objective of the current work, therefore is to identify ways of transferring classes of existing algorithms to the EM domain and therefore take advantage of existing work for this new purpose. Our focus is on algorithms which are provably optimal by some well defined criterion.

1.1.2 What Makes a Good EM Algorithm?

Informally, a good EM algorithm makes full use of the resources at its disposal. This includes consideration of the following factors:

1. the disk blocking factor B , as described above,
2. the available internal memory, captured by the parameter M , the number of data items that fit into internal memory,
3. the ability to perform I/O concurrently to all D available disk drives,
4. full utilization of all p available processors, and
5. efficient use of the communication bandwidth available between processors.

Most existing EM algorithms are based on what has become known as the Parallel Disk Model (PDM), proposed originally by Vitter and Shriver [85]. For optimality, this model requires that EM algorithms use (asymptotically) the minimum number of disk operations required to solve a problem in the worst case. We will refer to this as the *I/O-optimality* criterion. Of course, the *running time* of an algorithm is

a less abstract and more directly relevant issue. While I/O can be the most costly component of an external memory computation if it is not done efficiently, other factors such as computation and communication time can be equally dominant if their costs are not controlled. We will describe new work towards an alternative criterion, which includes other costs such as computation time, and in the case of parallel processors, the communication time required by an algorithm.

There are interesting similarities between parallel algorithms for certain kinds of parallel models, and EM algorithms. We will outline a class of parallel algorithms that also lend themselves to use (or simulation) as efficient external memory algorithms. Parallel algorithms break a problem into subproblems on which computations can proceed independently for some period of time before synchronization, i.e. communication, is required between the processors. Between synchronization events, references to data by a particular processor are restricted to data items currently residing in the local memory of that processor. This is possible because the subproblems in a given parallel step are independent.

In the case of fine grained parallel algorithms the processors may proceed independently for only one machine instruction before data is transferred between them. Such a situation may occur with algorithms based on the various PRAM models, for instance. In contrast, parallel computation models such as the Coarse Grained Multi-computer (CGM) and Extended Bulk Synchronous Parallel (BSP*) provide incentives for coarser granularity in the computation, encouraging a large number of machine instructions to be executed by each processor before inter-processor communication occurs. In this thesis we distinguish between two types of granularity: (1) granularity of the computation, as just described, and (2) granularity of the communication that occurs between processors. We find that the attribute of coarse grained communication in an algorithm allows us to model the communication between processors in the parallel algorithm directly as blocked I/O to and from external memory in a corresponding EM algorithm. The presence of coarse grained computation limits the number of synchronization steps required overall, and therefore limits the number of “context swaps” we need to do when simulating an algorithm. Coarse grained computation is also known in the parallel computing literature as “large slackness”. The

degree of *slackness* of an algorithm is the ratio $\frac{N}{p}$, where N is the problem size and p is the number of processors.

Previous EM research has focussed primarily on the sequential RAM model. Only limited exploration of parallel computing has been done. However, the domain of EM algorithms is tied to large problems and so the applicability of parallel algorithms is clear. The relevance and timeliness of combining EM and parallel BSP-like models is highlighted by the recent publication of a challenge to this effect [30]. In this position statement, Cormen and Goodrich make note of the similarities between parallel and EM computing and the desirability of creating a parallel EM model based on a BSP-like model and the Parallel Disk Model (PDM).

1.1.3 Effectiveness of EM Techniques

Few implementation studies have been done to determine the performance of EM algorithms developed over the past 10 years. The few studies that have been done confirm the effectiveness of these techniques in practice. A study by Chiang [24] reported that selected EM algorithms for computational geometry outperformed conventional techniques for large enough problem sizes, and were “steady and efficient” over the range of problem sizes tested. Cormen and Nicol [34] reported large FFT calculations based on EM techniques outperforming their internal memory counterparts by a factor > 40 in practice on a single disk system and that with 8 disks the EM algorithms outperformed their internal memory counterparts by a factor > 140 .

1.1.4 Fundamental Assumptions for Implementing Efficient I/O Algorithms

Algorithms which are designed to control their I/O costs are sometimes said to be *I/O-aware*. Many conventional operating system services are geared towards optimizing performance for applications which are not I/O-aware. This can be counter productive when an I/O-aware program is running. The literature on optimal EM algorithms describes many algorithms which control the details of their own I/O operations, and implicitly require that the computer operating system “get out of the way”.

A *declustering* technique is a method by which a file of data is spread over multiple disks, either on a single processor, or on a multiple processor system [32]. Two components of such a method are: (1) the *striping unit*: the number of bytes, or perhaps problem items, accessed on each disk by a single parallel I/O operation, and (2) the way in which such units are distributed over the disks. To be effective, I/O-aware programs require the following unusual abilities (Cormen and Kotz [32]):

1. control over declustering (the optimal algorithms control declustering explicitly),
2. control over the size of the striping unit (the optimal algorithms assume that the striping unit is one block),
3. the ability to query the system about the configuration (number of disks, block size, number of processors, amount of physical memory, current declustering method),
4. control over the offset on each disk and the ability to bypass parity operations for independent I/O (e.g. RAID - see Section 1.5), and
5. the ability to turn off the operating system's I/O caching and prefetching behaviours (the optimal algorithms do their own caching and prefetching).

Naturally, for benchmarking work to be accurate, it is also necessary to control the resources consumed by system tasks and other users. The implementation work of Chiang [24] illustrated the difficulties in measuring I/O counts when the operating system was allowed to manage memory and I/O as if the program was not I/O-aware. Our own implementation work ([58], see also Chapter 9) also indicates that these issues can be difficult to address properly. Item (5) in particular proves to be a difficult one.

1.2 Basic Data Formats on Multiple Disks

In this section we briefly discuss the format of data on a disk drive, and the access patterns which we will use.

Figure 1.1: A Disk Drive.

A disk drive can be thought of as a stack of one or more rotating discs, or platters, as illustrated in Figure 1.1. On each surface of each disc an equal number of concentric circular *disk tracks* are recorded. These are usually broken down further into *sectors*, which can be thought of as the basic units of data storage, typically 512 bytes in size. Because tracks at the outer edges of a disc are longer than tracks near the centre, the number of sectors per track varies. Operating systems usually aggregate a number of sectors to form a disk storage block. Data is recorded onto, or read from a disk drive via a set of moveable read/write heads, one per recordable surface. The heads typically move in unison, and are therefore restricted to accessing the same track on each of the discs. The tracks accessible by the heads at a particular position collectively form a *cylinder*. The time to access data on a disk drive is made up of *rotational delay*, or the time for the disk to rotate to the necessary position, plus the *seek time*, which is the time needed to move the heads to the appropriate cylinder.

Typical numbers for a disk drive at time of writing are exemplified by the Seagate Barracuda ST-34571FC Fibre Channel disk.¹ Selected parameters are listed below.

FORMATTED CAPACITY (GB)	_____4.55
AVERAGE SECTORS PER TRACK	_____172 rounded down
ACTUATOR TYPE	_____ROTARY VOICE COIL
TRACKS	_____51,780

¹Courtesy of Seagate Technology, Inc., <http://www.seagate.com/support/>

CYLINDERS	_____	5,178 user
HEADS	_____PHYSICAL_____	10
DISCS (3.5 in)	_____	5
SPINDLE SPEED (RPM)	_____	7,200
AVERAGE LATENCY (mSEC)	_____	4.17
SECTORS PER DRIVE	_____	8,888,923
AVERAGE ACCESS (ms read/write)	_____	9.4/10.4
Drive level with controller overhead		
SINGLE TRACK SEEK (ms read/write)	_____	2.0/2.1
MAX FULL SEEK (ms)	_____	17.5/18.5

In the example, a sector is about $4.55 \times 10^9 / 8888923 \approx 512$ bytes, the time for a complete revolution of the disk is $1/7200 = 140\mu s$, and the average time to access a sector, including both rotational delay and seek time is $9 - 10ms$.

We will use a simplified model of a disk drive. We assume that there is only a single disc with a single recordable surface, and therefore a single head. Our *disk storage block* (we often simply use the term *block*) consumes an entire track of a disk, and therefore, we think of track offset and block number on a disk as synonymous.

When multiple disks are present on a single processor, we get the situation illustrated in Figure 1.2. Data storage is similar to a matrix with three dimensions: 1) offset within a block, 2) block offset within a disk, 3) disk number. In the example, the index one varies most quickly, followed by index three, and finally by index two.

When D disks are present on a single processor, we will use the following terms, illustrated via Figure 1.2: A *parallel I/O* is a read or write operation in which $O(D)$ blocks are accessed simultaneously, one from each disk. A *stripe* is a set of D blocks, one per disk, stored at the same track offset on each disk. A *striped I/O* is a parallel I/O which accesses the blocks of a single stripe. An *independent I/O* is a parallel I/O in which the tracks accessed may differ from disk to disk. A *consecutive I/O* is a parallel I/O which accesses items with consecutive labels when the data is laid out as in Figure 1.2. A *staggered I/O* is a parallel I/O in which the block offset changes by a fixed amount between each pair of consecutive disks.

We will use the terminology *consecutive format* to mean that external input or

	Disk number							
	1	2	3	4	5	6	7	8
Stripe 0	1, 2	3 4	5, 6	7, 8	9,10	11,12	13,14	15,16
Stripe 1	17,18	19,20	21,22	23,24	25,26	27,28	29,30	31,32
Stripe 2	33,34	35,36	37,38	39,40	41,42	43,44	45,46	47,48
Stripe 3	49,50	51,52	53,54	55,56	57,58	59,60	61,62	63,64

Figure 1.2: *Layout of Disk Blocks on a Single Processor Machine.*

output data is, or can be, accessed using consecutive I/Os. We use the terms *striped format*, *staggered format*, and *independent format* in a similar manner.

When multiple disks are present on multiple processors, we get the situation illustrated in Figure 1.3. Data storage is now similar to a matrix with four dimensions: 1) offset within a block, 2) block offset within a disk, 3) disk number, 4) processor number. However, in all of our algorithms, we develop independent subproblems at each processor, and a global ordering of items as in Figure 1.3 is not typically required except perhaps as a final step.

1.3 Previous EM Research

The development of external memory algorithms can be viewed as pursuing scalability to the case where internal memory is insufficient to contain the entire problem. Since many of the analyses that have been presented for algorithms in computer science have used asymptotic arguments that assume unit cost access for every problem item, i.e. a uniform random access memory model, there are many, diverse areas where EM considerations can, and perhaps should be applied in order to update these analyses.

Processor number	0				1			
Disk number	0	1	2	3	0	1	2	3
Stripe 0	1, 2	3, 4	5, 6	7, 8	9, 10	11, 12	13, 14	15, 16
Stripe 1	17, 18	19, 20	21, 22	23, 24	25, 26	27, 28	29, 30	31, 32
Stripe 2	33, 34	35, 36	37, 38	39, 40	41, 42	43, 44	45, 46	47, 48
Stripe 3	49, 50	51, 52	53, 54	55, 56	57, 58	59, 60	61, 62	63, 64

Figure 1.3: *Layout of Disk Blocks on a Multiple Processor Machine.*

The number of papers dealing with EM issues has therefore been growing rapidly over the past few years, and it is difficult to provide a comprehensive survey. Vitter [84] currently maintains the most definitive survey on EM issues, and updates it periodically as new results are published.

1.3.1 A Brief Survey

We list here selected research in EM with some relevance to the focus of this thesis. Some of the discussion is repeated in subsequent sections which deal with specific external memory topics.

Floyd [49] is credited with some of the earliest work towards optimal EM algorithms. He studied matrix transposition and permutation algorithms in a paged memory, with a limited set of basic operations.

Aggarwal and Vitter [4] studied sorting on a model which involved a single disk which could read or write D blocks in one operation. They showed matching upper and lower bounds for the number of I/O operations required for sorting, FFT, matrix transpose, and permutation under this model. Their model did not constrain the locations from which the D blocks were to be accessed. In other words, the blocks could be from consecutive block locations on the disk, or they could be widely sep-

arated. In this respect the model was unrealistic, in that no physical disk system allows simultaneous transfer of D blocks under those circumstances.

Vitter and Shriver [85, 86] considered a more realistic model, where each of D parallel disks could simultaneously read or write a single block in a single parallel I/O operation (see Figure 2.1). This model has become known as the Parallel Disk Model (PDM). Since the previous model of Aggarwal and Vitter was more permissive, their lower bounds carry over to the new model. Vitter and Shriver concentrated on randomized algorithms relating to sorting and permutation, and introduced the first optimal randomized algorithms for sorting and related problems in EM on the PDM. They also extended their work from two-level memories [85] to include multi-level memory hierarchies [86]. These papers presented some of the earliest work on parallel processing for EM. They assumed an interconnection network that could do sorting of the records in the internal memories of the p processors in randomized logarithmic time $\tilde{O}(\frac{M}{P'} \log M)^2$. Examples include PRAM-like interconnection networks, cube-connected cycles, hypercube. See Section 2.2.2 for a description of the Parallel Disk Model.

A number of models have been proposed for multi-layer memories: the Hierarchical Memory Model(HMM) [2], the Block Transfer Model(BTM) [3] and the Uniform Memory Model(UMM) [7]. These models have not been heavily used, perhaps due to their complexity [30].

Cormen [28, 29] and Cormen, Sundquist and Wisniewski [35] followed up on an observation of Aggarwal and Vitter [4] that certain types of permutations in EM can be done faster than the general permutation case. Cormen identified a number of such permutations, including matrix transpose, address bit reversal, hypercube routing, vector reversal, perfect shuffle, inverse perfect shuffle, gray code and inverse gray code computations. He showed that certain fundamental operations such as complementing and circular shifting of address bits were common to these problems, and that they could be done faster than general permutations in EM.

Nodine and Vitter [68, 66] studied deterministic sorting algorithms for EM, in both distribution sort and merge sort flavours. They also reported I/O-optimal de-

² M is the total size of the internal memories and P' is the number of processors

terministic sorting algorithms for parallel memory hierarchies [7]. These algorithms are also simultaneously optimal in terms of internal computation for single processors, and for the PRAM. For non-PRAM interconnection networks such as hypercube and cube-connected cycles, only the randomized algorithms of [86] are simultaneously optimal in both I/O and computation. A summary of randomized and deterministic EM sorting techniques is presented in [67].

Goodrich, Tsay, Vengroff and Vitter [54] sketched EM algorithms for a large number of problems in computational geometry, and proposed several useful paradigms, such as *distribution sweeping* and *batch filtering* for designing efficient EM algorithms (see Section 1.3.2 for more information). They focussed on the two-layer single processor version of the PDM, but also included a short discussion of the applicability of multiprocessor and multi-layer models to their work. Many of the computational geometry algorithms proposed in [54] are listed in Section 8.5, where they are compared to new EM algorithms developed using the methods presented in this thesis.

Chiang, Goodrich, Grove, Tamassia, Vengroff and Vitter [25] presented EM versions of many useful algorithms related to graphs. New EM paradigms called *time-forward processing* and *PRAM simulation* are introduced (see Section 1.3.2 for a description of PRAM simulation). A technique for deriving lower bounds for EM problems is also described. A number of the graph algorithms proposed in [25] are listed in Section 8.5, where they are compared to new EM algorithms developed using the methods presented in this thesis.

Motivated by the goal of constructing I/O efficient versions of commonly used internal memory data structures, Arge [8, 9] proposed the data structuring paradigm, and in particular the *buffer tree*. A buffer tree is an external memory search tree, based on the (a, b) -tree [61], which is a generalization of the B^+ -tree [27]. It allows several update operations, such as insert, delete, search, deletemin, and it enables the transformation of a class of internal-memory algorithms to external memory algorithms by exchanging the data structures used. A large number of external memory algorithms have been proposed [8, 9] using the buffer tree data structure, including sorting, priority queues, range trees, segment trees, and time forward processing. These in turn are subroutines for many external memory graph algorithms, such as

expression tree evaluation, centroid decomposition, least common ancestor, minimum spanning trees, ear decomposition. There are a number of major advantages of the buffer tree approach. It applies to a large class of problems whose solutions use search trees as the underlying data structure. This enables the use of many normal internal memory algorithms, and “hides” the I/O specific parts of the technique in the data structures. Several techniques based on the buffer tree, e.g. time forward processing, are simpler than competitive EM techniques, and are of the same I/O complexity, or better, with respect to their counterparts.

The major innovation of the buffer tree is the addition of buffers to each internal node of the tree and lazy insert, delete and query strategies. The basic idea is that requests (query, insert, delete) accumulate in the buffer of an internal node until it is worthwhile to perform the necessary I/O operations to move them to the next level of the tree. See Section 9.1 for a more detailed description of the buffer tree.

Using the buffer tree, Arge et al. [11] presented I/O-optimal algorithms for a number of problems related to GIS, including map overlay, red-blue line segment intersection, trapezoidal decomposition, triangulation of simple polygons, planar point location. General techniques presented include an external memory version of fractional cascading using the buffer tree, and an extended external segment tree.

Vengroff and Vitter [83] describe external memory algorithms for three dimensional range searching and query processing.

In [24] Chiang studied the performance of algorithms to report the intersections of orthogonal line segments. The study compared the optimal EM algorithm reported in [54] with various versions of a similar internal memory algorithm which used balanced trees. Various memory sizes and distributions of the data features were tried. The conclusion was that the EM algorithm was both efficient and reliable, i.e. it handled all of the cases in an acceptable fashion, and significantly outperformed the other techniques when the internal memory was small relative to the problem size. He used some interesting techniques for generating his test data, and his tests highlighted some of the difficulties with running I/O-aware algorithms on an operating system that is designed for non I/O-aware applications.

TPIE (Transparent Parallel I/O Interface) is a library of I/O-aware functions

and services designed for building I/O-efficient applications (see Vengroff [80, 81] and Vengroff and Vitter [82]). Many of the I/O-optimal algorithms and techniques in the literature are included. At the time of writing, the buffer tree and related applications are exceptions.

Barve, Grove and Vitter [15] examined the practicality of EM sorting techniques and proposed a randomized sorting technique that is I/O-optimal, yet simpler in practice than previous EM sorting methods, summarized in [67].

ViC* (Colvin and Cormen [26]) is language for coding data parallel programs with out-of-core (external) variables. The ViC* compiler provides the necessary input/output calls to allow a user program to manipulate huge data objects without explicitly coding the I/O requests. The compiler makes use of multiple disks and its knowledge of program structure to reduce the I/O costs borne by the program.

Cormen and Hirschl [31] reported the implementation of and timing results for external memory radix sort and BMBC permutations on a single processor with multiple disks.

Cormen and Nicol [34] reported on the development of I/O-optimal FFT applications on a single processor system with one or multiple disks, and Cormen, Wegman, and Nicol [36] implemented FFT calculations on multiple processors with multiple disks.

1.3.2 External Memory Paradigms

A number of general techniques or “paradigms” for obtaining efficient EM algorithms have been identified by various researchers in EM. In this section we describe several which are relevant to the research proposed here.

1.3.2.1 Batch Filtering

Batch filtering is a technique first described by Goodrich et al. [54]. Given $O(N)$ queries and a layered decision DAG (directed acyclic graph) of size $O(N)$ stored on disk, batch filtering permits the queries to be answered in an optimal $O(N/B)$ I/O operations. Batch filtering can be viewed as a technique for performing *multisearch*

(see Section 8.4) for N queries on a DAG of size $O(N)$ on a single processor.

We assume a single processor machine, with internal memory size M , for $M > B^2$. A decision DAG with a single source node is stored in external memory, level by level in consecutive format. The queries are partially sorted in order of the successor node they wish to visit on the current level of the graph. Since there is a single source node of the DAG, the queries can initially be in arbitrary order. The algorithm reads the first block of the current level of the DAG (e.g. a multi-way node of size B), and the first block of queries for this node from the disk. The queries destined for this node are processed sequentially, block by block. Each query can be directed in one of B possible ways by the current node, and the algorithm keeps B corresponding block buffers in memory. Each block buffer represents the head of a list of output blocks destined for a given successor node on the next level of the graph. Query blocks are read as necessary until a query requires a different graph node. Full output blocks are written to disk(s) and concatenated to the end of the appropriate list, depending on their destination. When the first query with a different successor is encountered, we are done with the current graph node, and can read the next one. This can continue until the level is finished. For the next level, the output lists are concatenated to form the input for the next round.

1.3.2.2 Distribution Sweeping

The Distribution Sweeping technique was introduced by Goodrich et al. [54].

The input for a geometric problem (e.g., orthogonal line intersection) is stored in external memory. The technique first divides the input into $O(M/B)$ slabs, i.e. it is typically sorted along one dimension of the data, and divided evenly into $O(M/B)$ buckets. The slabs are scanned (one block from each slab can simultaneously exist in memory) along a second dimension of the data. This typically requires a second sort to arrange the data elements within each slab in increasing order by the second dimension. The algorithm then “solves” or “processes” those subproblems which consist of interactions between slabs. More concretely, this amounts to processing data elements that span a slab. When the scan is finished, all of the inter-slab interactions have been processed. The algorithm then proceeds recursively on each

slab.

This technique has been used as the basis for an I/O-optimal EM algorithm to report all intersections of orthogonal line segments [54]. Implementation and performance of the orthogonal line intersection algorithm are reported in [24, 23]

1.3.2.3 Data Structuring

One of the first I/O-optimal data structures in EM was described by Arge [8, 9]. The buffer tree is based on the (a, b) -tree [61], which is a generalization of the B^+ -tree [27]. The major difference between the buffer tree and the (a, b) -tree is the addition of large buffers to each tree node. The buffers collect requests, which may include insert, delete, and query operations, as they navigate down the tree from the root to the appropriate leaves. A request r in the buffer of node q does not advance to the next level of the tree unless there are sufficient other requests in the buffer to pay for the costs of reading in the children of q and “bucket sorting” the requests in q ’s buffer according to q ’s splitters.

Unlike B^+ -tree based applications, requests against a buffer tree data structure are not satisfied immediately. Processing proceeds in a “lazy” manner, and results are reported only when, and in the order in which, requests are forced down to the leaf level of the tree. Once a block of requests reaches a leaf, processing of those requests continues to completion. Only filtering of the requests through the buffer part of the tree is done in a lazy manner. As requests can get “stuck” in the buffers of the buffer tree if there are insufficient other requests to flush them out, it is generally necessary to “force empty” the buffers after all requests have been inserted, before all requests are fully processed. This involves scanning through the buffers and generating sufficient dummy requests to make the requests descend to the leaf level of the tree.

The buffer tree is suitable for processing subproblems which tolerate the lazy behaviour of the buffer tree, i.e. they do not require their output in a particular order (see [48]).

The buffer tree has been implemented as part of the LEDA-SM project (see Crauser et al. [37]). The performance of the external memory priority queue based on the buffer tree was compared to the performance of an external priority queue

derived from the internal memory radix heap data structure [5]. The authors report that the new structure based on radix heaps outperforms the buffer tree for the range of inputs tested, but they note that unlike radix heaps, the the buffer tree permits the priority data type to be a non-integer. They also found their implementation of buffer tree sort to be slower than implementations of other external memory sorting algorithms that they tested.

1.3.2.4 Simulation

Simulating parallel algorithms in EM is one of the primary ideas in this thesis. There appear to have been three forms in which the simulation paradigm has appeared in the EM literature:

- **Localizing parallel references algorithmically:**

A class of parallel algorithms from which good EM algorithms can be derived is illustrated by an algorithm described briefly in [54]. The authors identify a PRAM algorithm for searching the dictionaries in a fractional cascaded binary tree which has the property that the p PRAM processors access data in a compact pattern spanning $O(p/B)$ blocks at each PRAM step. In general, if a PRAM algorithm has the property that the processors in each step access the shared memory at locations that are grouped into clusters of size $\Omega(B)$, then a single processor external memory algorithm can be derived in a straightforward way by simulating the activities of the p PRAM processors.

- **Localizing parallel references by sorting:**

Chiang et al. [25] describe a simulation of PRAM algorithms based on sorting PRAM instructions and references at each parallel computation step. The sort ensures that the data references that would be made by the p PRAM processors are localized in a series of consecutive memory locations, and therefore can be accessed efficiently as a series of disk blocks. The sorting technique must also be an efficient EM sort. Using this technique, I/O-optimal EM algorithms can be derived from certain work-inefficient PRAM algorithms for problems with a “geometrically decreasing size” property.

- **Simulating BSP algorithms:** This idea was articulated independently by Sibeyn and Kaufmann [77] and by Dehne, Dittrich and Hutchinson [40]. The BSP model of parallel computation was proposed in [79] and is discussed in Chapter 2.

In [77], the authors propose an approach for simulating 1-optimal BSP algorithms to produce efficient EM algorithms. Their work is presented in the context of a single disk uniprocessor EM machine, but they suggest that the concept can be extended to multiple disks. They do not explain, however, how to accommodate the blocking factor, which is an intrinsic issue in efficient I/O design, nor do they provide mechanisms for handling multiple disks or multiple physical processors on the target EM machine.

1.3.3 Lower Bounds on I/O Complexity

A number of important results regarding lower bounds for EM algorithms are listed in Sections 1.3.3.1 to 1.3.3.4. The reader is referred to the survey by Vitter [84] for a more complete treatment.

1.3.3.1 Sorting Lower Bound

Perhaps the most fundamental is the lower bound shown for sorting by Aggarwal and Vitter [4]. This result states that the I/O complexity of sorting is:

$$\text{sort}(N) = \Omega\left(\frac{N}{BD} \log_{\frac{M}{B}} \frac{N}{B}\right) \quad (1.1)$$

Vitter [84] points out why disk striping is non-optimal in terms of I/O complexity. With a single disk, Equation (1.1) becomes $\text{sort}(N) = \Omega\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$. Striping data across D disks is equivalent to increasing the blocksize to BD , giving $\text{sort}_{\text{striping}}(N) = \Omega\left(\frac{N}{BD} \log_{\frac{M}{BD}} \frac{N}{BD}\right)$, which is larger than (1.1) by a factor of $\log_{\frac{M}{BD}} \frac{M}{B}$. If D is very large, approaching $\frac{M}{B}$, the base of this logarithm approaches unity, and striping becomes strikingly inferior to using the disks independently for sorting.

1.3.3.2 Permutation Lower Bound

Aggarwal and Vitter also showed that permutation in external memory has I/O complexity:

$$\text{perm}(N) = \Omega\left(\min\left\{\frac{N}{D}, \frac{N}{BD} \log_{\frac{M}{B}} \frac{N}{B}\right\}\right)$$

Cormen [28] showed that special cases of permutation corresponding to affine transformations have lower I/O complexity than the general lower bound. These special classes of permutations can be generated by certain binary matrix transformations on the address bits of the permutation elements.

1.3.3.3 Matrix Transpose Lower Bound

Aggarwal and Vitter [4] showed that the I/O complexity of transposing a $p \times q$ matrix is

$$\Theta\left(\frac{N}{BD} \log_{\frac{M}{B}} \min\{M, p, q, \frac{N}{B}\}\right)$$

1.3.3.4 A General Lower Bounding Technique

Arge et al. [10] showed that in general, assuming a comparison-based model, any problem requiring $\Omega(N \log N)$ comparisons requires $\Omega(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/O operations in the Parallel Disk Model.

1.4 Realistic Parameter Ranges

The general lower bounds of Aggarwal and Vitter and others are extremely useful in designing efficient external memory algorithms, and give an accurate picture of the general case, for all possible values of the parameters.

Our simulation techniques produce external memory algorithms whose constraints restrict them to only a subset of the parameter constellations covered by the lower bounds of the preceding sections. However, we will argue that this subset is a useful

one, and is often encountered in real machines. In this subset, we find that the I/O complexity of sorting, for instance, is only $O(\frac{N}{BD})$, because $\log_{\frac{M}{B}} \frac{N}{B} < c$ for constant c . We will discuss this in more detail in Section 8.2.

1.5 The Role of RAIDs

The performance advantages of parallel disk drives are recognized by the growing popularity of *RAID* arrays (Redundant Array of Inexpensive Disks), see [70, 22]. RAID technology permits an array of R individual disk drives, each with block size B to be accessed as a single, logical disk with blocksize between $\frac{RB}{2}$ and RB , depending on the *RAID level* used. The RAID levels are designed to provide varying degrees of error detection and correction, based on the observation that the mean time between failures (*MTBF*) for an array which depends on D disks is $\frac{1}{D}$ times the MTBF for one of the disks. RAID level 0 is the simplest case, where read and write operations of BD items access a block of application data on each of the $R = D$ disks. RAID levels 1 to 6 introduce redundancy of one form or another to cope with the eventuality of disk or operating system failure. Thus read and write operations of BD items access a block of application data on each of D disks, where $D < R$. Redundancy is used in RAID levels 1 to 6 to permit data to be recovered if a disk failure occurs.

In this thesis, we are concerned with algorithms that are capable of efficiently using the large block sizes that are required for efficient use of disks (including RAID), but the reliability issues are beyond the scope of our discussions. For some of our algorithms, we also find that the reads and writes are not striped, i.e. the blocks accessed on each disk are on different (logical) tracks. It is not clear how useful a RAID array is for these algorithms, as we may need to bypass the RAID logic in order to access the disks independently.

Several of the RAID levels calculate checksums for error correcting purposes on a stripe by stripe basis. Non-striped writes may modify a block in up to D different stripes, necessitating the recalculation of their checksums and resulting in large I/O and computational overhead. While non-striped access to disk promises a theoretical advantage in I/O complexity (see Section 1.3.3), striped I/O may in practice be more

efficient if RAID technology is used.

In general, in this thesis, we assume that the disks are accessible independently according to the EM-BSP and PDM models. We note when the access to the disks is striped, but otherwise we pay little attention to the issues surrounding the use of RAIDs for implementing these algorithms.

1.6 Storage Space

The amount of disk space required by an external memory algorithm is important. For very large problems, even small constant factors in space consumption can be crucial to the practicality of an algorithm. The algorithms considered in this thesis generally require space which is linear in the problem size. Exceptions include the EM-BSP algorithms derived from the CGM parallel segment tree, and the buffer tree technique [8], whose implementation is considered in Chapter 9. Both increase space consumption by a logarithmic factor. For the simulation techniques that are considered in Chapters 3, 4, 5, and 6, the space consumption is linear in the problem size. In this thesis we focus on asymptotic analyses of our techniques. While the precise constants are of interest to a practitioner, they are beyond the scope of our treatment here.

1.7 Preliminaries - Notation and Terminology

1.7.1 Asymptotic Notation

We define the asymptotic notation $O(f(N))$, $o(f(N))$, $\Omega(f(N))$, $\omega(f(N))$, in the usual way [19, 33]. The following definitions are adapted from [19].

Let f, g be any functions such that $f : \mathcal{N} \rightarrow \mathcal{R}^+ \cup \{0\}$, and $g : \mathcal{N} \rightarrow \mathcal{R}^+ \cup \{0\}$.

1. $O(f(N)) = \{t : \mathcal{N} \rightarrow \mathcal{R}^+ \cup \{0\} \mid (\exists c \in \mathcal{R}^+)(\exists n_0 \in \mathcal{N})(\forall n \geq n_0)[t(N) \leq cf(N)]\}$
2. $o(f(N)) = \{t : \mathcal{N} \rightarrow \mathcal{R}^+ \cup \{0\} \mid (\exists c \in \mathcal{R}^+)(\exists n_0 \in \mathcal{N})(\forall n \geq n_0)[t(N) < cf(N)]\}$
3. $\Omega(f(N)) = \{t : \mathcal{N} \rightarrow \mathcal{R}^+ \cup \{0\} \mid (\exists c \in \mathcal{R}^+)(\exists n_0 \in \mathcal{N})(\forall n \geq n_0)[t(N) \geq cf(N)]\}$

4. $\omega(f(N)) = \{t : \mathcal{N} \rightarrow \mathcal{R}^+ \cup \{0\} \mid (\exists c \in \mathcal{R}^+)(\exists n_0 \in \mathcal{N})(\forall n \geq n_0)[t(N) > cf(N)]\}$
5. $\Theta(f(N)) = O(f(N)) \cap \Omega(f(N))$

We extend the notation to expressions containing asymptotic notations as follows. Let g be any function such that $g : \mathcal{N} \rightarrow \mathcal{R}^+ \cup \{0\}$. Let $\mathcal{U}_1$ and $\mathcal{U}_2$ represent any set of functions, such as one of $O(f(N))$, $o(f(N))$, $\Omega(f(N))$, or $\Theta(f(N))$. Let $\oplus$ represent any binary operator.

1. $\mathcal{U}_1 \oplus \mathcal{U}_2 = \{t_1(N) \oplus t_2(N) \mid t_1(N) \in \mathcal{U}_1, t_2 \in \mathcal{U}_2\}$
2. $g(N) \oplus \mathcal{U}_2 = \{t_1(N) \oplus t_2(N) \mid t_1(N) = \{g(N)\}, t_2 \in \mathcal{U}_2\}$. $\mathcal{U}_1 \oplus g(N)$ is similar.
3. $a \oplus \mathcal{U}_2 = \{t_1(N) \oplus t_2(N) \mid t_1(N) = a, t_2 \in \mathcal{U}_2\}$. $\mathcal{U}_1 \oplus a$ is similar.

Let $\mathcal{U}$ represent any set of functions, such as those defined above. As is common practice in asymptotic analysis of algorithms, we will also use the notation $g(N) = \mathcal{U}$ to mean $g(N) \in \mathcal{U}$.

1.7.2 BSP Models

In Chapter 2 we discuss the BSP (Bulk Synchronous Parallel) parallel computing model and two related models, BSP* (Extended BSP), and CGM (Coarse Grained Multicomputer), which contain extensions or simplifications of the basic BSP ideas. We will refer to these three models collectively as *BSP-like* models, or simply as *BSP* models. We may refer to a BSP algorithm, with N data and v processors, which communicates exclusively via h -relations of size $h = \frac{N}{v}$ as a CGM algorithm, and a BSP algorithm whose message size is $\Omega(b)$ for some positive integer $b > 1$ may also be referred to as a BSP* algorithm. Please refer to Chapter 2 for more information on the BSP, BSP*, and CGM parallel models.

1.8 A Central Issue: Communication Bottlenecks

The idea of decomposing a problem into smaller subproblems is a common approach in computer science. Divide and conquer algorithms, parallel computing algorithms,

are some examples. Communication between subproblems is required in order for such approaches to be successful. Dijkstra [45] pointed out the similarities between concurrent processes, which “communicate in space” and sequential processes, which “communicate in time”. Message passing is a typical means of communicating in space, and disk files are a common method of communicating between processes whose execution is separated in time.

Optimizing both kinds of communication is naturally an important consideration in controlling the running time of practical algorithms. Recognition of this issue is one of the main differences between the BSP and PRAM parallel computing models. It is the main motivation for the development of the BSP* and CGM parallel computing models. It also provides the primary issues in external memory algorithm design.

An important, common factor in controlling the cost of these types of communication is aggregation or *blocking* of the communicated data. In many real communication systems (e.g. disk input/output, inter-processor message passing), there is a significant overhead cost in preparing the communication system for a data transfer. This may include determining and reserving a communication path in a network, or waiting until disk hardware has been physically adjusted to the appropriate position.

Blocking associates multiple data items with a common block. When a communication operation is required on one of the data items in the aggregation, it is applied to the block. Provided that the algorithm operates on all or a significant fraction of the elements in the block before discarding it, this reduces overall costs by paying the overhead cost only once for the block, instead of once for each of the items actually accessed by the algorithm.

1.9 Papers Contributing to the Thesis

Some of the results described in this thesis have been published as papers with various co-authors. These papers are described briefly below. However, any errors in the current work are mine alone.

- In [40], a randomized technique is presented for simulating BSP* algorithms [17] in external memory (see Section 2.3.3 for a description of the BSP* model of

parallel computation). The resulting algorithm is shown to have blocked I/O to one or multiple disks on a single or multiple processor machine. All I/O is done in parallel to multiple disks if present on each processor, and communication between processors is balanced and is also done block-wise. The authors propose new models for evaluating external memory algorithms on multiple processor machines which address suggestions made by Cormen and Goodrich [30] (see Section 2.4.1 for a description of this “challenge”). This work is covered by Chapters 2 and 4.

- Hutchinson, Maheshwari, Sack and Velicescu [58] (see also Section 9.1) implemented the buffer tree for external sorting and priority queue applications, and suggested modified parameter settings which improved performance in practice. To the author’s knowledge, this was the first full implementation of the buffer tree. Buffer tree sort was found to be well behaved, with the running time increasing linearly over the range of inputs tested. Buffer tree sort easily outperformed the well known quicksort technique using virtual memory.
- Dittrich, Hutchinson and Maheshwari [47] presented external memory, multiple processor algorithms for multisearch and showed them to be efficient using the models previously proposed in [40]. More details of this work can also be found in Section 8.4 of this thesis.
- Dehne, Dittrich, Hutchinson and Maheshwari [42] presented a deterministic technique for simulating CGM algorithms [43] in external memory (see Section 2.3.4 for a description of the CGM model of parallel computation). A number of existing parallel algorithms are shown to have efficient external memory, multiple processor variants, obtained via the simulation technique, and analyzed under the models of [40].
- In [41], Dehne, Dittrich, Hutchinson and Maheshwari explore the transformation of parallel algorithms via the technique of [42] as a way to make existing parallel algorithms scalable to larger problem sizes without using demand paging for handling virtual memory.

1.10 Claim to Originality or Contribution to Knowledge

Section 1.10.2 summarizes the main contributions of this thesis. The claims to originality are subject to the contributions and assistance of various co-authors, which are summarized in Section 1.10.1.

1.10.1 Collaborations

- The randomized simulation technique of Chapter 4 and the EM-BSP, EM-BSP* and EM-CGM models first appeared in [40] and also in [46].
- The buffer tree implementation study described in Section 9.1 appeared earlier as [58].
- External memory multisearch algorithms for the EM-BSP*, described in Section 8.4, first appeared in [46], and later in [47].
- An implementation design and the experiments described in Section 9.2 are also described in [87].
- A summary of the simulation approach, its application to adding scalability to parallel algorithms, and the preliminary experimental results of Section 9.2 were described in [41].
- The deterministic simulation technique of Chapter 5 is described in [42].

1.10.2 Contributions

1. In general terms, this work adds a number of results on parallel processing in communication-aware parallel models to the existing body of external memory research.
2. Our new EM-BSP, EM-BSP* and EM-CGM parallel, external memory computing models combine the features of the external memory Parallel Disk Model

(PDM) with Bulk synchronous Parallel (BSP) parallel computing models, answering a challenge of Cormen and Goodrich in ACM Computing Surveys [30].

3. We propose a general paradigm for producing parallel, external memory algorithms from BSP algorithms. We present detailed analyses of instances of the paradigm, including general derivation techniques. Examples presented include:
 - (a) The BSP* lower bound on message sizes in the parallel computing domain allows blocked I/O in the external memory domain. Using randomization, we also achieve balanced communication to multiple real processors and balanced, independent I/O to multiple disks on each processor
 - (b) Balanced message sizes in a BSP algorithm allows blocked I/O, and balanced communication to multiple processors and striped I/O to multiple disks on each processor without randomization.
 - (c) Adapting a known parallel routing algorithm to our requirements, we show that it converts a BSP algorithm to the requirements of either of techniques 3(a) or 3(b) above.
 - (d) We describe what appears to be a new routing algorithm for parallel algorithms, which routes messages indirectly via a series of rounds, and has an adjustable slackness requirement.
 - (e) Using our new indirect routing technique, we describe a second deterministic approach to obtaining external memory algorithms for an EM-BSP model from existing parallel BSP algorithms. This requires smaller slackness than technique 3(b) above, and so reduces the internal memory size required by the resulting external memory algorithm.
 - (f) We provide an optimization technique for our simulations when the problem size varies geometrically over a series of rounds of the original BSP algorithm.
 - (g) We explain an apparent contradiction between established lower bounds on I/O complexity for various problems, and the I/O complexities we derive for several of our new algorithms.

4. We present a large number of new external memory algorithms obtained from our simulations.
 - Sorting, Permutation, matrix transpose: optimal parallel external memory algorithms existed for models which assumed logarithmic or unit-cost communication. However the new algorithms work on more general models.
 - We present the first parallel, EM multisearch algorithms.
 - We obtain parallel, external memory algorithms for various problems in computational geometry, geographic information systems, and graph theory.
5. We report on the implementation and performance testing of a buffer tree. We believe this to be the first such implementation of this well know EM data structure. We also describe preliminary implementation experiments of one of our simulation techniques, providing evidence towards its validation as a practical technique.

1.11 Organization of the Thesis

In Chapter 2, we survey selected previous work on models for external memory computation and for parallel computation. We highlight the challenge presented by Cormen and Goodrich [30] to combine these models, and we present three new combined models that address many of the issues identified in [30].

In Chapter 3 we describe the general framework of a new simulation technique that creates efficient external memory algorithms from efficient BSP algorithms.

In Chapter 4 we describe a randomized approach which implements the general framework described in Chapter 3. In particular, we focus on how to efficiently write the messages between processors to (parallel) disk(s) and efficiently retrieve them in the next compound superstep. We show that if the simulation is performed on a c -optimal BSP* algorithm the resulting EM algorithm is c -optimal with high probability. The resulting EM-BSP algorithm (in particular, EM-BSP*) acquires

its property of blocked I/O from the attribute of blocked communication present in the original BSP* algorithm from which it was derived. The techniques described in Chapter 4 ensure that this property of blocked communication between tasks is preserved, and that in addition, I/O is done in parallel if multiple disks are present.

In Chapter 5 we describe a deterministic simulation technique for producing an EM-BSP (in particular, an EM-CGM) algorithm from a particular type of BSP algorithm (in particular, a CGM algorithm). This approach requires that each of the v virtual processors generate at least vb items of communication data in every superstep, where the parameter b represents the desired communication block size. This allows us to establish a tight upper and lower bound on the message size to every processor, and allows a simple, efficient deterministic simulation.

In Chapter 6, we describe another deterministic approach, which also generates an EM-CGM algorithm from a CGM algorithm. It requires less communication volume than the methods of Chapter 5. Each of v virtual processors need only generate $v^\epsilon b$ items of communication data in every superstep, where ϵ is a constant, $0 < \epsilon \leq 1$. The simulation requires a constant number of rounds for each communication round of the CGM algorithm, although this constant is larger than in Chapter 5.

In Chapter 7, we describe an extension to the three simulation techniques which allows certain problems, such as list ranking, whose problem size reduces geometrically in each of a series of r rounds to be simulated in a linear (i.e. $O(\frac{N}{B})$) number of I/O operations.

In Chapter 8 a number of new EM-BSP (in particular, EM-BSP* and EM-CGM) algorithms are presented, together with their computation, communication and I/O complexities. Many of these algorithms are obtained from known BSP* and CGM algorithms via the simulations of Chapters 4 and 5. We also describe new multisearch algorithms derived from a known BSP* algorithm.

In Chapter 9, we describe two case studies in implementing EM algorithms. We describe an implementation of the buffer tree technique of Arge [8] and an implementation of the deterministic simulation technique of Chapter 5 applied to a parallel algorithm for Samplesort.

Chapter 10 concludes with a summary of the contributions of this research, and

some suggestions for future work.

In order to assist the reader in understanding the arguments of this thesis, a summary of notation and an index of terms can be found beginning on page 183.

Chapter 2

Models

2.1 Overview

In this chapter, we develop three new models of computation on parallel external memory machines. The new models, which we refer to as EM-BSP, EM-BSP* and EM-CGM, can be viewed as uniting the external memory parallel disk model (PDM) [85] with the BSP [79], BSP* [17] and CGM [73, 43] parallel computing models, respectively. The PDM is described in Section 2.2.2 and the BSP, BSP* and CGM parallel computing models are described in Section 2.3. While the new models each have slightly different parameters, due to their differing heritage, they share a number of common features:

- they capture three contributors to the running time of an algorithm, namely computation, communication, and I/O, and
- the goodness criteria for algorithms evaluated on each of the models are similar and subject to the same issues.

The BSP, BSP*, and CGM models are very similar. Algorithms which are efficient on the BSP* and CGM models are also BSP algorithms. Because of this similarity, we may refer at times to BSP* and CGM algorithms as BSP algorithms, and to EM-BSP* and EM-CGM algorithms as EM-BSP algorithms.

2.1.1 Organization of This Chapter

In Section 2.2 we describe previous models for external memory computation, and in particular, the Parallel Disk Model. Section 2.3 describes several models of computation for parallel machines, including the BSP, BSP* and CGM parallel computing models. Section 2.4 then introduces the new parallel EM models EM-BSP, EM-BSP* and EM-CGM.

In describing each of these models, we identify two fundamental components: the system model, and the cost model.

The system model describes the architecture of the model, and the cost model provides criteria by which competing algorithms on the model can be compared. The cost model consists of a measurement scheme and one or more goodness criteria, which together provide a measure by which two algorithms can be ranked.

2.2 EM Models

In this section we examine some of the external memory models that have been proposed to date.

2.2.1 Aggarwal and Vitter Single Disk Model

This model [4] involves a single processor accessing a single disk. In a single I/O request or operation, D different blocks can be transferred between disk and memory. There is no restriction on the location of the blocks on the disk, and so they may be in consecutive block locations, or widely separated. This is more permissive than real disk systems and so the model is viewed as somewhat unrealistic compared to the later PDM model of Vitter and Shriver. However, Aggarwal and Vitter showed lower bounds for a number of problems using their permissive model, and these lower bound carry over to the PDM. The cost models for the two models are identical (see below).

Figure 2.1: *The Parallel Disk Model with One Processor.* The goodness criterion is simultaneous optimality of I/O complexity and computation complexity.

2.2.2 The Parallel Disk Model (PDM)

System Model The model of Vitter and Shriver [85, 86] has become known simply as the PDM or Parallel Disk Model. See Figure 2.1. The model allows for D disks to be accessed simultaneously, in a single I/O operation. Either striped or independent I/O is allowed. The disks may be attached to a single processor, or individually to different processors, which are interconnected by some network. The basic PDM is independent of the number of processors, but both single and multicomputer versions were introduced in [85].

For the randomized algorithms reported in [85, 86], multicomputer interconnection networks on which sorting could be done in parallel randomized logarithmic time were considered, e.g. cube connected cycles. The later deterministic algorithms of Nodine and Vitter [68] considered only a PRAM-like interconnection between processors.

Cost Model The cost measure used by the PDM is the number of I/O operations required asymptotically by an algorithm $\mathcal{A}$ for a given problem $\mathcal{P}$ as the size of $\mathcal{P}$ grows to infinity. All other costs such as computation time are ignored. Algorithm $\mathcal{A}$ is said to be optimal (we use the term *I/O-optimal*) if the number

Figure 2.2: *The Parallel Disk Model with Multiple Processors.*

of I/O operations required by $\mathcal{A}$ meets the lower bound for the number of I/O operations required to solve $\mathcal{P}$. The following additional rules apply:

- for each I/O operation performed, the model assumes that one block of data can be transferred for each of the D disks,
- for each block transferred, the model assumes that B items of data could have been transferred, where B is the number of items of data that fit into a disk block,
- throughout the computation the model assumes that available internal memory, which is capable of holding M items of data, is fully utilized by the algorithm. For instance, the lower bound for sorting [85] incorporates the fact that internally sorting a memory-load of data is free under the

model. The number of passes over the data and therefore the number of I/O operations is affected by how many items fit into internal memory.

The PDM has been widely used for developing and analyzing external memory algorithms. Much of the research described in Chapter 1 is based on this model of external memory computation.

2.2.3 Hierarchical Memory Models

In a modern computer system, there are potentially many layers of memory, each with a different access time and data unit (“block”) size. These layers form a hierarchy, ranging from those with the fastest access time, such as the on-chip registers of the CPU, through various kinds of cache memory, main memory, to local mechanical disk drives, or even remote disks accessed via a network. In concept there is potential at each interface between layers of the hierarchy to optimize the data transfer algorithmically. A number of models have been developed to capture the relationships between layers of this hierarchy. These include the Hierarchical Memory Model(HMM) of Aggarwal, Alpern, Chandra, and Snir [2], the Block Transfer Model(BTM) of Aggarwal, Chandra, and Snir[3], and the Uniform Memory Model(UMM) of Alpern, Carter and Feig [7]. Due to the added complexity of the multi-level models, relatively few researchers have used them to design algorithms [30]. One exception is [86].

In this thesis we focus primarily on the disk-memory interface, because the speed difference and blocking requirements of this interface are the most severe (offering the most potential for time savings) and because the hierarchical models are significantly more complex.

2.3 Parallel Computation Models

Now we examine some of the parallel computing models that have been proposed. As in the EM models, there are both a system model and a cost model component to each. The system model describes the architecture of the model, and the cost model provides a criterion by which competing algorithms on the model can be compared.

2.3.1 The Parallel Random Access Memory (PRAM) Model

The PRAM model [50] has been used for many years as a model of parallel computation. Its main advantage is its simplicity.

System Model The PRAM Model assumes that p processors are available, and that they are attached to a single, common memory. The processors operate synchronously. In each PRAM step, p instructions are executed in parallel. Any item in the common memory can be accessed at unit cost by any processor. To resolve the issues associated with multiple processors reading and writing a single item in the common memory concurrently, several submodels have been defined:

Concurrent Read Concurrent Write (CRCW): an item can be read or written concurrently in this model. To model the effect of concurrent writes, several further submodels have been proposed.

Exclusive Read Exclusive Write (EREW): neither read nor write concurrency is allowed by the model.

Concurrent Read Exclusive Write (CREW): only read concurrency is permitted by the model.

Exclusive Read Concurrent Write (ERCW): only write concurrency is permitted by the model.

Cost Model The time complexity T_p of a PRAM algorithm $\mathcal{A}$ for problem $\mathcal{P}$ with p processors is defined as the maximum number of memory accesses performed by any processor during the execution of $\mathcal{A}$. The work complexity W_p is $p \times T_p$. The goodness criteria used are typically:

Parallel Time Optimality : Minimize T_p , where $T_p \geq T_s/p$, and T_s is the time complexity of the best sequential algorithm for $\mathcal{P}$.

Parallel Work Optimality : Minimize W_p , where $W_p \geq T_s$, and T_s is the time complexity of the best sequential algorithm for $\mathcal{P}$.

2.3.2 The Bulk Synchronous Parallel (BSP) Model

The BSP Model was proposed by Valiant [79] as a “bridging model for parallel computation”; a standard model on which both hardware and software designs can agree. The stated objective of BSP is to provide a model that is simultaneously useful to hardware designers, algorithm designers, and programmers of parallel computers. It is neither a hardware nor a programming model, but somewhere in between.

System Model : The BSP Model has parameters N , v , g , and L . It consists of v processor/memory components (we use the letter v because these will be *virtual processors* during our simulations - see Chapters 3, 4, 5, 6), a router that delivers messages in a point to point fashion, and a facility to synchronize all processors (prevent processors from proceeding until all have reached a given “barrier” in their code). Each processor has a unique label in the range $0, 1, \dots, v - 1$.

A computation proceeds in a succession of *supersteps* separated by synchronizations, usually divided into *communication* and *computation supersteps*, see Figure 2.3. In computation supersteps processors perform local computations on data that is available locally at the beginning of the superstep and issue send operations. Between computation supersteps, a communication superstep is performed, where each processor exchanges data with its peers, via the router. An *h-relation* is a superstep where $O(h)$ data are sent and received by every processor.

The model anticipates that the routers on different systems may have different characteristics relative to the computation speed of the processors. For explanation purposes, let $\mathcal{N}_{comp} = \{\text{the number of computational operations that can be performed by the processors per unit time}\}$ and $\mathcal{N}_{comm} = \{\text{the number of words that can be delivered by the router per unit time}\}$. The parameters of the BSP model are then defined as follows:

N : the number of items in the problem to be solved

v : the number of processors available

Figure 2.3: A BSP Computation.

$g : \mathcal{N}_{comp}/\mathcal{N}_{comm}$, i.e. the time conversion factor between communication and computation operations. Another way of expressing the meaning of g is as the time required to send a single word of data between two processors, where time is measured in number of CPU operations.

L : the minimum setup time or latency of a superstep, measured in CPU operations. This can involve a number of factors, including the time required for the processors to perform a barrier synchronization.

Cost Model : The cost of a computation on the BSP model is represented by the sum of three quantities, one for computation time, one for communication time, and one for the time required by the processors to synchronize. Let $\mathcal{A}$ be a BSP algorithm, operating on input data of size N items. Let β be its parallel

computation time, let α be the number of words of data sent between processors during the execution of $\mathcal{A}$, and let λ be the number of supersteps required by $\mathcal{A}$ to solve a problem of size N . Then the cost $T(\mathcal{A})$ of the computation is

$$T(\mathcal{A}) = \beta + g \cdot \alpha + \lambda \cdot L$$

The c -optimality goodness criterion was proposed in [52] for the BSP model.

Definition 2.1 *Let $\mathcal{A}^*$ be the best sequential algorithm on the RAM for the problem under consideration, and let $T(\mathcal{A}^*)$ be its worst case runtime. Let $c \geq 1$ be a constant. A c -optimal parallel algorithm $\mathcal{A}$ meets the following criteria:*

- *The ratio ϕ between the computation times of $\mathcal{A}$ and $T(\mathcal{A}^*)/p$ is in $c+o(1)$.*
- *The ratio ξ between the communication time of $\mathcal{A}$ and the computation time $T(\mathcal{A}^*)/p$ is in $o(1)$.*

All asymptotic bounds refer to the problem size n as $n \rightarrow \infty$. We say that a BSP algorithm is one-optimal if it fulfills the requirements of the above definition for $c = 1$.

2.3.3 The Extended BSP (BSP*) Parallel Processing Model

The BSP* model [17] is a special case of the BSP model where each message is assessed a minimum cost equivalent to a message of size b items. The BSP* model therefore gives incentives to send messages of at least b in size. An appropriate value for b in a particular case is determined by the characteristics of the particular router.

System Model : The BSP* model has parameters $N, v, \hat{g}, L, b$. Similarly to BSP, it consists of v processor/memory components, a router that delivers messages in a point to point fashion, and a facility to synchronize all processors in a barrier style. Each processor has a unique label in the range $0, 1, \dots, v - 1$.

As in BSP, a computation proceeds in a succession of supersteps separated by synchronizations, usually divided into *communication* and *computation supersteps*. In computation supersteps processors perform local computations on data that is available locally at the beginning of the superstep and issue send operations. In communication supersteps the send operations are implemented, i.e., the exchange of data between the processors is done by the router.

The parameters of the BSP* model are defined as follows:

N : the number of items in the problem to be solved.

v : the number of processors available.

$\hat{g}$: the time (in number of processor operations) the router needs to deliver a packet of size b (in number of machine words) when in continuous use.

L : the minimum setup time or latency of a superstep, measured in CPU operations.

b : the minimum message size required by the router to achieve its rated throughput.

Cost Model : The cost of a computation on the BSP* model is represented by the sum of three quantities, one for computation time, one for communication time, and one for the time required by the processors to synchronize. Let $\mathcal{A}$ be a BSP* algorithm, operating on input data of size N items. Let β be its parallel computation time, let α be the number of messages sent between processors during the execution of $\mathcal{A}$, and let λ be the number of supersteps required by $\mathcal{A}$ to solve a problem of size N .

Let α_{ij} be the maximum of the number of words sent or received by processor j in the i^{th} communication superstep. The cost of the i^{th} communication superstep is then $t_i = \max_{j=1}^v w_{ij}$ where $w_{ij} = \lceil \frac{\alpha_{ij}}{b} \rceil$ is the maximum of the number of blocks sent or received by processor j in superstep i . Let $\alpha = \sum_{i=1}^{\lambda} t_i$. Then the cost $T(\mathcal{A})$ of the computation is

$$T(\mathcal{A}) = \beta + \hat{g} \cdot \alpha + \lambda \cdot L$$

Note that a message which is shorter than b words incurs the same cost as a message which is b words in length.

The c -optimality goodness criterion is typically used to evaluate an algorithm on the BSP* model. See Section 2.3.2.

We treat the BSP* model as a special case of BSP, differing only in the way the accounting is done for messages shorter than b items.

2.3.4 The Coarse Grained Multicomputer (CGM) Model

The CGM model [73, 43], is another special case of the BSP model. In the spectrum of granularity of parallel computation, the PRAM model is the most fine grained, and the CGM is the most coarse grained model.

System Model : The CGM model [73, 43] has parameters N, v . The model assumes that a collection of v processors are interconnected via a network. Initially, each processor holds an equal portion of the N data items of the problem to be solved, so each processor contains $O(N/v)$ data items. A CGM algorithm is BSP-like, in that it proceeds in an alternating sequence of computation and communication rounds (supersteps). In a single computation round, each processor performs a computation internally on the $O(N/v)$ items in its local memory. Between computation rounds, a communication round is performed, where each processor sends a total of $O(N/v)$ data to its peers and receives $O(N/v)$ data in return. It is this restriction on the communication volume of a superstep that differentiates the CGM model from other BSP models.

Cost Model : A common goodness criterion for CGM algorithms is to simultaneously minimize: i) parallel computation time, and ii) number of supersteps performed. $O(1)$ rounds is the ideal. This has the effect of defining an optimal CGM algorithm for a given problem as being an algorithm that has the

minimum number of supersteps, and performs the same amount of work as an optimal sequential algorithm for the same problem.

In this thesis, we will treat CGM as a model which has parameters N , v , g , L , as in BSP, and but for which each communication superstep consists of one h -relation, for $h = \frac{N}{v}$. The cost of measure of a CGM algorithms in our work will be the sum of the computation, communication and synchronization times, as in BSP. Let $\mathcal{A}$ be a BSP* algorithm, operating on input data of size N items. Let β be its parallel computation time, and let λ be the number of supersteps required by $\mathcal{A}$ to solve a problem of size N . Then the cost $T(\mathcal{A})$ of the computation is

$$T(\mathcal{A}) = \beta + g \cdot \lambda \cdot \frac{N}{v} + \lambda \cdot L$$

A superstep which involves less than $\frac{N}{v}$ data sent or received by each processor is nonetheless charged for a $\frac{N}{v}$ -relation.

2.3.5 Other Parallel Computing Models

A large number of other parallel computing models have been proposed. The LogP model [38] is one of the most frequently mentioned (see for example [30]). This model differs slightly from BSP in that it captures the communication overhead, which is the time spent by a processor sending or receiving a message. A detailed comparison between the LogP and BSP models is presented in [18]. They conclude that BSP and LogP are nearly equivalent in modelling capability. A variant of LogP is the LogGP model [6], which makes special provisions for large messages. Both LogP and LogGP require detailed schedules of the communication activities.

2.4 Combining PDM and BSP-like Models

2.4.1 Objective: A More Comprehensive Model

While the Parallel Disk Model captures computation and I/O costs, it is designed for a specific type of communication network, where a communication operation is

expected to take a single unit of time, comparable to a single CPU instruction. BSP and similar parallel models capture communication and computational costs for a more general class of interconnection networks, but do not capture I/O costs.

The ACM Working Group on Storage I/O for Large-scale Computing [30] presented *“the challenge of synthesizing a coherent model that combines the best aspects of the Parallel Disk Model and Bulk Synchronous Parallel models to develop and analyze algorithms that use parallel I/O, computation and communication.”*

Our EM-BSP, EM-BSP* and EM-CGM models, introduced in Sections 2.4.2, 2.4.3, and 2.4.4 respectively, address many of the issues outlined in that report. In particular, the EM-CGM model, introduced in Section 2.4.4 has a small number of parameters, is not difficult to use, and appears to correctly predict, at least qualitatively, the performance of algorithms designed on the model. See Chapter 9 for preliminary implementation experiments of an EM-CGM Samplesort algorithm.

2.4.2 The EM-BSP Model

We first extend the BSP model to include secondary local memories. The basic idea is illustrated in Figure 2.4. Each processor has in addition to its local memory an external memory in the form of a set of hard disks.

2.4.2.1 System Model

We apply this idea to extend the BSP model to its external memory version *EM-BSP* by adding the following to the standard BSP parameters:

- M is the local *memory size* of each processor,
- D is the *number of disk drives* of each processor,
- B is the *transfer block size* of a disk drive, and
- G is the ratio of local computational capacity (number of local computation operations) divided by local I/O capacity (number of blocks of size B that can be transferred between the local disks and memory) per unit time.

Figure 2.4: *A Parallel Machine with External Memory*

In this thesis, we will use the parameter v to refer to the number of processors of a BSP, and the parameter p to refer to the number of processors of an EM-BSP.

In many practical cases, all processors have the same number of disks. We restrict ourselves to that case, although the model does not forbid different numbers of drives and memory sizes for each processor. We denote the disk drives of each processor by $\mathcal{D}_0, \mathcal{D}_1, \dots, \mathcal{D}_{D-1}$. Each drive consists of a sequence of *tracks* (consecutively numbered starting with 0) which can be accessed by direct random access using their unique track number. A track stores exactly one block of B records.

Each processor can use all of its D disk drives concurrently, and transfer $D \times B$ items from the local disks to its local memory in a single I/O operation and at cost G . In such an operation, we permit only one track per disk to be accessed without any restriction on which track is accessed on each disk. We assume that a processor can store in its local memory at least one block from each local disk at the same time, i.e., $M \geq DB$.

2.4.2.2 Cost model

Like a computation on the BSP model, the computation on the EM-BSP model proceeds in a succession of supersteps. We adapt communication and computation

supersteps from the BSP model and allow multiple I/O-operations during a single computation superstep. For the EM-BSP model, the computation cost, t_{comp} , and communication cost, t_{comm} , are the same as for the BSP model. The total cost of each superstep is defined as $t_{comp} + t_{comm} + t_{I/O} + L$. The term $t_{I/O}$ is the additional I/O cost charged for the superstep, where $t_{I/O} = \max_{j=1}^p \{w_{I/O}^j\}$, and $w_{I/O}^j$ is the I/O-cost incurred by processor j . Recall that each I/O operation costs G time units.

Note that the model gives incentives to access all disk drives using block transfers. For instance, a single processor EM-BSP with D disks is capable of transferring a block of B items to or from each disk in a single I/O operation. An operation involving fewer elements incurs the same cost.

2.4.3 The EM-BSP* Model

The EM-BSP* is derived from the BSP* model in a similar way to how the EM-BSP model is formed from BSP. We add to the BSP* model the additional parameters M (local memory size at each processor), D (number of disk drives at each processor), B (transfer block size for a local disk drive), and G (ratio of local computational capacity to local I/O capacity).

In this thesis, we will use the parameter v to refer to the number of processors of a BSP*, and the parameter p to refer to the number of processors of an EM-BSP*.

The cost of each EM-BSP* superstep is defined as $t_{comp} + t_{comm} + t_{I/O} + L$ where t_{comp} and t_{comm} refer to the computation time and communication time as defined for the BSP* model, and $t_{I/O}$ is defined as indicated above for the EM-BSP model.

We treat the EM-BSP* as a special case of EM-BSP, differing only in its accounting of the costs of short messages.

2.4.4 The EM-CGM Model

The EM-CGM model has the parameters of the CGM plus the following additional parameters: M (local memory size at each processor), D (number of disk drives at each processor), B (transfer block size for a local disk drive), and G (ratio of local computational capacity to local I/O capacity).

In this thesis, we will use the parameter v to refer to the number of processors of a CGM, and the parameter p to refer to the number of processors of an EM-CGM.

As in the CGM, the cost of each communication superstep is identical, and reflects the cost of each processor sending and receiving $\theta(\frac{N}{p})$ data items in each communication superstep. We treat the EM-CGM as a special case of EM-BSP, differing only in this constraint on communication volume. We use the full set of parameters, N , p , g , L , M , D , B , G , as used in the EM-BSP for analysis purposes of the EM-CGM.

2.4.5 Goodness Criteria for Parallel EM Algorithms

2.4.5.1 Extending the c -optimality Criterion

Since even small multiplicative constant factors in runtime are important, we characterize the performance of an EM-BSP algorithm by comparing its run time with an optimal sequential or PRAM algorithm. We measure ratios between runtimes on pairs of models that have the same set of local instructions and adapt the optimality criteria proposed in [52].

Definition 2.2 *Let $\mathcal{A}^*$ be the best sequential algorithm on the RAM for a problem P of size N items, and let $T(\mathcal{A}^*)$ be its worst case runtime. Let $c \geq 1$ be a constant. A c -optimal EM-BSP algorithm $\mathcal{A}$ for p processors meets the following criteria (all asymptotic bounds are with respect to the problem size $N \rightarrow \infty$):*

- *The ratio ϕ between the computation times of $\mathcal{A}$ and $T(\mathcal{A}^*)/p$ is in $c + o(1)$.*
- *The ratio ξ between the communication time of $\mathcal{A}$ and the computation time $T(\mathcal{A}^*)/p$ is in $o(1)$.*
- *The ratio η between the I/O-time of $\mathcal{A}$ and the computation time $T(\mathcal{A}^*)/p$ is in $o(1)$.*

Note that the constraint on η differentiates this definition from the corresponding one for non-external memory parallel algorithms.

We shall say that a EM-BSP algorithm is *one-optimal* if it fulfills the requirements of the above definition for $c = 1$.

2.4.5.2 Alternative Goodness Criteria for EM-BSP Algorithms

The simultaneous satisfaction of the constraints $\phi \in c + o(1)$, $\xi \in o(1)$, and $\eta \in o(1)$ is a stringent requirement to place on an EM-BSP algorithm. It is not always possible to show that the running time of an algorithm is dominated by its computation time, for instance.

A relaxation of this requirement leads to the notion of balancing the costs of computation, communication, and I/O, so that none of them dominates the running time, and the work done in each is asymptotically the same as the work of an optimal sequential algorithm.

We will use the terms *work-optimal*, *communication-efficient* and *I/O-efficient* to describe an algorithm for which $\phi \in O(1)$, $\xi \in O(1)$ and $\eta \in O(1)$, respectively. An algorithm which is work-optimal, communication-efficient, and I/O-efficient, therefore, is one whose running time complexity is no worse than the complexity $T(\mathcal{A}^*)/p$. Constant factors are ignored.

For some problems such as searching a large data structure on disk, the cost of I/O may necessarily dominate the computation and communication time. In this case, we require that the algorithm be *I/O-optimal*, meaning that the number of I/O operations matches the lower bound for the number of I/Os required to solve the problem. We may still be able to show that $\phi \in c + o(1)$ and $\xi \in o(1)$, but if not, we require that the algorithm be work-optimal and I/O-efficient.

Chapter 3

EM-BSP Algorithms from Simulation

3.1 Overview of the Method

The purpose of this chapter is to describe, in general terms, how a suitable BSP algorithm can be executed as an EM-BSP algorithm on a machine with multiple disks.¹ We will call this “simulating a parallel algorithm as an external memory algorithm”. In essence, the simulation is of the message passing that occurs in the parallel algorithm. This approach reflects Dijkstra’s observation [45] that in an abstract sense, communication between processes “in space” (e.g. executing on the processors of a BSP computer) is no different than communication “in time” (in this case, by passing messages between tasks via the disks of an EM-BSP computer). At a less abstract level, however, the simulation of a BSP algorithm for this purpose involves major changes to the way that communication between processes is performed, and certain constraints on the parameters of the problem.

The main ideas of the mapping between a suitable BSP algorithm $\mathcal{A}$ and a corresponding external memory algorithm $\mathcal{A}'$ are as follows:

- A communication operation between processors in $\mathcal{A}$ is replaced by one or more

¹Recall that we treat BSP* and CGM algorithms as particular types of BSP algorithms.

output operations to disk and corresponding input operations from disk in $\mathcal{A}'$,

- We choose values for the parameters of $\mathcal{A}$ that ensure that I/O in $\mathcal{A}'$ can be done blockwise,
- To accommodate multiple disks on the target machine, we choose techniques and parameter values which allow us to use all of the disks in parallel, and
- To accommodate multiple processors on the target machine, we ensure that the communication between processors in $\mathcal{A}'$ is balanced, so no real processor receives or sends much more data than any other.

The execution of a BSP algorithm proceeds as a series of compound supersteps, and can therefore be simulated by repeated application of the simulation steps for a single compound superstep. Message send/receive operations are modeled by disk read/write operations.

We adopt the following terminology: The v processors of the BSP machine will be called *virtual processors*. Each communication superstep will be divided into a *sending superstep* and a *receiving superstep*. During a sending superstep, messages are generated, and during a receiving superstep they are received. A *compound superstep* is composed of a receiving, a computation, and a sending superstep. The *context* of a virtual processor is the local memory it uses, and the *context size* of a virtual processor is the maximum size of its context used during the computation. The maximum context size of all virtual processors is denoted as μ , the maximum size of the data communicated by a virtual processor is represented by γ , and $\gamma = O(\mu)$ (see Figure 3.1).

Outline of the simulation for a compound superstep: A compound superstep for the v virtual processors of a BSP machine is simulated by performing the following steps in a round-robin fashion, for k virtual processors at a time, $1 \leq k \leq v$.

1. *Fetching phase:* Read the context(s) and the messages to be received by the current virtual processors from disk into memory.
2. *Computation phase:* Perform the computations indicated by the BSP algorithm for these k virtual processors in this compound superstep.

Figure 3.1: *The memory of a virtual processor.* The program code is assumed to be of constant size. In many practical cases it is the same for all virtual processors and therefore a single copy is sufficient. The context is of size μ , and consists of the memory portion which is not read-only. The message area is of size γ and is contained in the context.

3. *Writing phase:* Save the current contexts and the messages sent by the current virtual processors on disk.

The Fetching phase (Computation phase, Writing phase) performs the operations necessary to simulate the receiving superstep (computation superstep, sending superstep) of a compound superstep. A compound superstep produces messages which are received in the following compound superstep. The simulation must store the generated messages on disk in such a way that they can be fetched efficiently during the Fetching phase of the next compound superstep. By efficiently we mean in this context that both input and output operations are fully blocked to the disk block size B and that if parallel disks are present they are utilized in parallel, i.e. for D parallel disks, input and output operations are performed D blocks at a time. These requirements can easily be met for the contexts, as we know their maximum size and can preallocate a dedicated area for each, spread across the D disks. For the generated message traffic, however, these requirements may be more difficult, as we may not know the communication pattern for a particular compound superstep.

The communication time of a BSP superstep is $O(g\gamma + L)$, where γ is the maximum size of the data communicated by a virtual processor. Recall that μ is the maximum context size of a virtual processor, hence $\gamma = O(\mu)$. In the following, the context and messages generated by a virtual processor are divided into blocks, and the blocks are spread evenly over the available disks.

Algorithm SeqCompoundSuperstep simulates a single compound superstep of a v processor BSP on a uniprocessor EM-BSP machine. We simulate $1 \leq k \leq v$ virtual processors at a time, and we refer to such a collection of processors as a *batch*. We also use this term to refer to the messages associated with a batch of processors. To maximize the use of available memory, we choose $k = \lfloor \frac{M}{\mu} \rfloor$. Note that $M \geq \mu$. The following outlines the basic ideas of the algorithm. The details are presented in subsequent chapters.

Algorithm 3.1 SeqCompoundSuperstep

Objective: Simulation of a compound superstep of a v -processor BSP on a single processor EM-BSP with D disks.

Input: For each $i \in \{0, \dots, \frac{v}{k} - 1\}$ the blocks of the contexts and arriving messages of batch i (i.e., virtual processors $ik, \dots, (i+1)k - 1$) are spread over the D disks in a parallel format.

Output: (i) The (changed) contexts of the k simulated processors are spread across the disks in striped format. (ii) The messages generated during the compound superstep are grouped by destination into $\frac{v}{k}$ batches, and each batch is stored in a parallel format on the disks.

- (1) for $i = 0$ to $\frac{v}{k} - 1$
 - (a) Read the contexts $V_{ik}, \dots, V_{(i+1)k-1}$ of the k virtual processors of batch i from the disks into memory.
 - (b) Read the packets received by the k virtual processors of batch i from the disks.
 - (c) Simulate the local computation of the k virtual processors of batch i .
 - (d) Write the packets which were sent by the k virtual processors to the D disks.
 - (e) Write the changed contexts V_{ik} to $V_{(i+1)k-1}$ back to the D disks.
- (2) Reorganize the blocks containing the generated messages into consecutive format for each batch of k processors so that they can be accessed in parallel from the disks in the simulation of the next compound superstep.

— End of Algorithm —

Algorithm SeqCompoundSuperstep simulates a single compound superstep of a BSP algorithm. Steps 1(a) and 1(b) correspond to the Fetching Phase, Step 1(c) is the Computation Phase, and Steps 1(d) and 1(e) comprise the Writing Phase.

Details of Steps 1(a) and 1(e): Since we know the size of the contexts of the processors, and the order in which we simulate the virtual processors is static during the simulation, we can distribute the k contexts deterministically. We reserve an area of total size $v\mu$ on the disks, $\frac{v\mu}{BD}$ blocks on each disk, where we store the contexts. We split the context V_j of virtual processor j into blocks of size B and store the i -th block of V_j on disk $(i + j\frac{\mu}{B}) \bmod D$ using track $\left\lfloor \frac{i+j\frac{\mu}{B}}{D} \right\rfloor$. Since the context of each processor is now in consecutive format on the disks, we can easily read and write the contexts of k consecutive processors using D disks in parallel for every I/O operation.

Details of Step 1(b): The details of this step vary slightly depending on the technique used (the randomized simulation of Chapter 4 or the deterministic simulations described in Chapters 5 and 6). In all cases, however, Step (2) of the previous compound superstep guarantees that the blocks which contain the messages destined for the current processors are stored in a reserved area evenly distributed over the disks. Therefore, we can use a similar technique to fetch the messages as we used to fetch the contexts.

Details of Step 1(d) and Step 2: Writing the messages to the parallel disks (Step 1(d)) is more difficult than writing the contexts to the disks. This is because we must ensure that in each I/O operation a single block is written to each disk in parallel (or, asymptotically, $\Theta(D)$ disks receive a block of data). We use the fact that we know the size of the contexts, and that they are all the same size. This makes the swapping of contexts rather straightforward. However, we do not know the size of inter-processor messages in general. They may be of widely differing sizes, from zero items to M items. In the single real processor case the messages can be delivered by sorting. However, we are ultimately interested in an algorithms that works for a multi-processor target machine.

In addition to writing the messages to the disks in parallel, we must also arrange that in Step 1(b) of the next superstep, we can read the messages intended for each processor using all of the disks in parallel. This is the responsibility of Step 2. Steps

1(d) and 2 vary depending on the technique used.

- In Chapter 4, a BSP* algorithm is assumed as input. Step 1(d) involves distributing the communication packets of the BSP* randomly among the available disks. Step 2 then requires several passes over the data to prepare it for Step 1(b) of the next compound superstep.
- In Chapter 5, Step 1(d) involves deterministically splitting the communication from each batch of k processors into v pieces, which are written into fixed locations on the disks. Step 2 in this case is not required.
- In Chapter 6, another deterministic technique is presented. Step 1(d) involves $\lceil \frac{1}{\epsilon} \rceil$ routing stages, for $0 < \epsilon < 1$. In the first $\lceil \frac{1}{\epsilon} \rceil - 1$ stages, the processors deterministically split their outgoing message data into v^ϵ pieces, and the pieces are distributed evenly over the processors of v^ϵ groups. In the final routing stage, the method of Chapter 5 is used. Step 2 is not required.

We now generalize the simulation to $p \geq 1$ processors on the target EM-BSP machine.

We first describe the simulation of a compound superstep of a v -processor BSP with communication time $g\gamma + L$, computation time $\tau + L$ and context size μ on a p -processor EM-BSP, for $p \geq 1$. We assume that $b \geq B$, where b is the message block size of the BSP virtual machine.

Outline of the parallel simulation: As an initial step, $\frac{v}{p}$ virtual processors $i\frac{v}{p}, \dots, (i+1)\frac{v}{p} - 1$ are assigned to each simulating processor i .² As before, the simulation of each compound superstep is composed of a series of rounds. During each of $\frac{v}{pk}$ rounds, a real processor simulates the steps of k virtual processors. Messages sent between virtual processors on different real machines require real communication by the simulation. If these messages are sent directly to their destinations, the traffic may be unbalanced, causing inefficiencies in communication. We describe the techniques used to deal with this problem below.

²This assignment is different for the deterministic simulation described in Chapter 6. See below.

We maintain $\frac{v}{pk}$ batches to store the generated messages. The j^{th} batch contains the messages destined for virtual processors simulated in the j^{th} round of the current compound superstep. Each processor writes its share of the $\frac{v}{pk}$ batches to its local disks.

Algorithm 3.2 ParCompoundSuperstep

Objective: Simulation of a compound superstep of a v -processor BSP on a p -processor EM-BSP.

Input: For every batch j , where $0 \leq j \leq \frac{v}{pk}$, the message and context blocks of the virtual processors $jkp, \dots, (j+1)kp - 1$ are divided among the real processors and their local disks as follows:

- each real processor holds $O(\frac{k\gamma}{B})$ blocks of messages and $\frac{k\mu}{B}$ blocks of context
- each local disk contains $O(\frac{k\gamma}{BD})$ blocks of messages and $O(\frac{k\mu}{BD})$ blocks of context.

Output: The changed contexts and generated messages distributed as required for the next compound superstep.

(1) For $j = 0$ to $\frac{v}{p} - 1$ do

For $i = 0$ to $p - 1$ do in parallel

- (a) *Fetching Phase:* processor i reads the contexts and message blocks pertaining to batch j from its local disks, in parallel.
- (b) *Computing Phase:* processor i simulates the computation supersteps of its current virtual processors and collects all generated messages in its local memory.
- (c) *Writing Phase:* processor i writes the updated contexts and message blocks generated by batch j to its local disks.

(2) In a series of communication supersteps, the real processors exchange and reorganize message data generated in the previous computation rounds.

— End of Algorithm —

Many of the details of algorithm ParCompoundSuperstep are similar to ones previously described for Algorithm 3.1, but there are differences between the three techniques presented in Chapters 4, 5, and 6. We outline the main differences below.

- In the randomized approach of Chapter 4,
 1. The messages blocks of the BSP* are distributed randomly among the real processors,

2. In a separate round of communication, the message blocks are sent to their correct destinations.

This allows us to ensure that the real processors both send and receive the same amount of data with high probability.

In concept, these two communication rounds can be performed for each batch as part of Step 2 of ParCompoundSuperstep. Instead, however, we distribute the message blocks randomly for each round at the end of Step 1(c) and send them to their proper destination as the first part of Step 1(a). See Chapter 4 for details.

- In the deterministic approach of Chapter 5, each round of communication between virtual processors is transformed into two rounds where every message is of size $\Theta(\frac{N}{v^2})$. This ensures that the communication between real processors is balanced, i.e., every real processor sends and receives the same amount of data in each round.
- The deterministic approach of Chapter 6 requires less communication volume (lower slackness) than the method of Chapter 5. This permits the internal memory to be reduced relative to the technique of Chapter 5. Here, the virtual processors are assigned to real processors in a different manner than in the previous cases. Whereas we had previously assigned a range of virtual processors with consecutive labels to each real processor, we now assign the virtual processors to real processors in a round robin fashion. Our routing algorithm computes an ordering over all of the blocks to be sent in each round, and then sends consecutive blocks in the ordering to virtual processors with consecutive labels. We can show that each real processor then receives the same amount of data, within a constant factor, in every communication round.

3.2 Overview and Rationale

In Chapter 4 we describe a randomized approach which allows us to efficiently write the messages to disk and efficiently retrieve them in the next compound superstep. We show in Theorems 4.1 and 4.2 that if the simulation is performed on a c -optimal BSP* algorithm the resulting EM algorithm is c -optimal with high probability. The resulting EM-BSP algorithm (in particular, EM-BSP*) acquires its property of blocked I/O from the attribute of blocked communication present in the original BSP* algorithm from which it was derived. The techniques described in Chapter 4 ensure that this property of blocked communication between tasks is preserved, and that in addition, I/O is done in parallel if multiple disks are present.

In Chapter 5 we describe a deterministic simulation technique for producing an EM-BSP (in particular, an EM-CGM) algorithm from a particular type of BSP algorithm (in particular, a CGM algorithm). This approach requires that each virtual processor generate at least vb items of communication data in every superstep, where the parameter b represents the desired communication block size. This allows us to establish a tight upper and lower bound on the message size to every processor, and allows a simple, efficient deterministic simulation.

In Chapter 6, we describe another deterministic approach, which also generates an EM-CGM algorithm from a CGM algorithm. It requires less communication volume than the method of Section 5. Each virtual processor need only generate $v^\epsilon b$ items of communication data in every superstep, where ϵ is a constant, $0 < \epsilon \leq 1$. The simulation requires a constant number of rounds for each communication round of the CGM algorithm, but this is a larger constant than in Section 5.

In order to assist the reader in understanding the arguments of this paper, a summary of notation and an index of terms can be found beginning on page 183.

Chapter 4

Randomized Simulation

4.1 Single Processor Target Machine

We will now describe the details of the BSP* to EM-BSP* simulation. In the following, we use the terms consecutive format, striped format, and parallel format, which were discussed in Section 1.2. We will also refer to another type of parallel format, linked format, which will be described below.

The communication time of a BSP* superstep is $O(\hat{g}_b^\gamma + L)$, where γ is the maximum size of the data communicated by a virtual processor. Recall that μ is the maximum context size of a virtual processor, hence $\gamma = O(\mu)$. In the following, the context and messages generated by a virtual processor are divided into blocks, and the blocks are spread in consecutive format over the available disks.

Algorithm SeqCompoundSS-4.1 simulates a single compound superstep of a v processor BSP* on a uniprocessor EM-BSP* machine with internal memory of size M , and D disks. We simulate k virtual processors at a time, and we refer to such a collection of processors, or to the messages they generate, as a *batch*. Of course, $M \geq k\mu$.

Algorithm 4.1 SeqCompoundSS-4.1

Objective: Simulation of a compound superstep of a v -processor BSP* on a single processor EM-BSP* with D disks.

Input: For each $i \in \{0, \dots, \frac{v}{k} - 1\}$ the blocks of the contexts and arriving messages of group i (i.e., virtual processors $ik, \dots, (i+1)k - 1$) are spread over the D disks in consecutive format,

so that they can be accessed in parallel.

Output: (i) The (changed) contexts of the k simulated processors are spread across the disks in consecutive format. (ii) The messages generated during the compound superstep are grouped by destination into $\frac{v}{k}$ groups, and each group is stored in consecutive format on the disks.

- (1) for $i = 0$ to $\frac{v}{k} - 1$
 - (a) Read the contexts $V_{ik}, \dots, V_{(i+1)k-1}$ of the k virtual processors of group i from the disks into memory.
 - (b) Read the packets received by the k virtual processors of group i from the disks.
 - (c) Simulate the local computation of the k virtual processors of group i .
 - (d) Write the packets which were sent by the k virtual processors to the D disks.
 - (e) Write the changed contexts V_{ik} to $V_{(i+1)k-1}$ back to the D disks.
- (2) Perform *SimulateRouting*: Reorganize the blocks containing the generated messages into consecutive format for each group of k processors so that they can be accessed in parallel from the disks in the simulation of the next compound superstep.

— End of Algorithm —

Algorithm SeqCompoundSS-4.1 simulates a single compound superstep of the BSP* algorithm. Steps 1(a) and 1(b) correspond to the Fetching Phase, Step 1(c) is the Computation Phase, and Steps 1(d) and 1(e) comprise the Writing Phase.

Details of Steps 1(a) and 1(e): Since we know the size of the contexts of the processors, and the order in which we simulate the virtual processors is static during the simulation, we can distribute the k contexts deterministically. We reserve an area of total size $v\mu$ on the disks, $\frac{v\mu}{BD}$ blocks on each disk, where we store the contexts. We split the context V_j of virtual processor j into blocks of size B and store the i -th block of V_j on disk $(i + j\frac{\mu}{B}) \bmod D$ using track $\left\lfloor \frac{i+j\frac{\mu}{B}}{D} \right\rfloor$. Since the context of each processor is now in consecutive format on the disks, we can read and write the contexts of k consecutive processors using D disks in parallel for every I/O operation.

Details of Step 1(b): Step (2) of the previous compound superstep guaranteed that the blocks which contain the messages destined for the current processors are stored in a reserved area evenly distributed over the disks. Therefore, we can use a similar technique to fetch the messages as we used to fetch the contexts.

Details of Step 1(d): As described in Chapter 3, managing the generated message traffic may be more difficult than the management of the contexts, as we may not

know the communication pattern for a particular compound superstep. After the Computation Phase, all messages sent by the current group of k processors in the current compound superstep have been generated and stored in internal memory. The coarse-grained nature of the BSP* algorithm results in large messages, which are as long or longer than the block size B . We cut the messages into blocks of size B . Each block is labelled with the destination address from its original message. In $k\gamma/B/D$ rounds, we write the blocks out to the disks. In each round a group of D blocks $b_i, 0 \leq i \leq (D-1)$, is written in parallel to the disks by choosing a random permutation π of $\{0, 1, \dots, (D-1)\}$ and writing block b_i to disk $\pi(i)$.

The blocks are partitioned into D buckets on the disks, depending on their destination address. Each bucket contains the blocks destined for $\frac{\gamma}{D}$ consecutive virtual processors. In order to maintain the buckets, the simulation uses a table of D pointers on each disk. The i^{th} entry in the table on a disk points to the head of a list of blocks of bucket i that have been written to that disk. Whenever we write a block of bucket i to disk D_j , we allocate a free track on D_j and concatenate it to the list for bucket i . For convenience, we shall refer to the format just described for the blocks in a bucket as *linked format*.

Clearly, we can read all the blocks composing a bucket stored on D disks in linked format in $O(\frac{k\gamma}{BD})$ parallel I/O operations, provided each disk contains the same number of blocks. In Lemma A.4 we will show that with high probability the blocks of each bucket are uniformly distributed over the disks.

Algorithm SimulateRouting provides the details of Step 2 of SeqCompoundSS-4.1.

Algorithm 4.2 SimulateRouting

Objective: Reorganize the blocks of messages from the previous computation superstep into consecutive format (Step 2 of Algorithm 4.1).

Input: The D buckets (of messages) stored on the disks in linked format.

Output: The $\frac{\gamma}{k}$ groups (of messages) stored on the disks in consecutive format.

- (1) *Allocate space for a copy of bucket i on disk i , for $i = 0, \dots, (D-1)$. Within each copy, reserve space for each of its groups. Read the buckets from the disks in parallel and write them back, one bucket per disk. For the j -th parallel read/write we perform the following:*
 - for $d = 0$ to $D-1$ in parallel do*
 - Read block b_d belonging to bucket d from disk $((d+j) \bmod D)$. Write block b_d to disk d on the next available track for the group to which it is addressed.*

Figure 4.1: *Reorganization in Step (2) of SimulateRouting.* Before: each disk contains the message blocks for $\frac{v}{D}$ processors, divided into $\frac{v}{kD}$ groups. After: the messages for each group are spread over the disks in consecutive format.

- (2) *From each disk, in parallel, read a block of each bucket, writing the blocks back to the disks so that each bucket is in consecutive format.*

for $j = 0$ to $\frac{v\gamma}{BD}$

for $d = 0$ to $D - 1$ in parallel do

read the j^{th} block from disk d and write it to disk $(d + j) \bmod D$ on track $d\lceil \frac{v\gamma}{D^2B} \rceil + \lfloor j/D \rfloor$

— End of Algorithm —

After Step 1 of Algorithm SimulateRouting, all messages that will be received by a group of k processors are stored in one region on the same disk. After Step 2, the blocks are stored in consecutive format. In fact they are stored in fixed locations like the blocks of the context. See Figure 4.1. This is possible because we know that each virtual processor receives and sends messages of total size $\leq \gamma$.

Lemma 4.1 *Steps 1(a), 1(b), and 1(e) of algorithm SeqCompoundSS-4.1 have computation time $O(v\mu)$, memory $\Theta(k\mu + BD)$, I/O-time $O(G\frac{v\mu}{DB})$, and disk space $O(\frac{v\mu}{BD})$ blocks per disk.*

Proof. We have to read and write $\lceil \frac{k\mu}{B} \rceil$ blocks of context (Steps 1(a) and 1(e)) for the simulation of each simulated superstep. In Step 1(b) the arriving messages occupy

$\lceil \frac{k\gamma}{B} \rceil$ blocks. Since in these cases the blocks are arranged on the disks in consecutive format, we can access D tracks in parallel at a time. Hence, we need the following number of I/O operations for one simulated superstep:

$$\sum_{j=1}^{v/k} \left\lceil \frac{k(\mu + \gamma)}{BD} \right\rceil \leq \frac{v}{k} \frac{k(\mu + \gamma)}{BD} + \frac{v}{k}$$

Since $\gamma = O(\mu)$, the number of I/O operations is in $O(\frac{v\mu}{BD})$, and this implies $O(v\mu)$ additional local computation steps.

Since the memory of the simulating processor must hold the context of k virtual processors and a parallel disk track, $M = \Theta(k\mu + BD)$. $\square$

The blocks representing simulated message traffic are divided into buckets by the simulation. The contents of a bucket are written to the available disks in a series of write cycles. In each write cycle, at most one block is written to any disk.

Lemma 4.2 *The computation time of algorithm `SimulateRouting` is $O(l \cdot \gamma v)$ and its I/O time is $O(l \cdot G \frac{v\gamma}{BD})$ with probability at least $1 - e^{-\Omega(l \cdot \frac{\bar{v}}{B})}$ for $k\gamma \geq BD$, $\bar{v} \geq D \log D$, and constant $l \geq 6.4$.*

Proof. For the purposes of the proof we assume that the maximum amount of communication is required. This can be accomplished by the introduction of dummy blocks if necessary. Each bucket contains $R = \frac{v\gamma}{BD}$ blocks since each of the D buckets contains the messages destined for $\frac{v}{D}$ virtual processors.

By Lemma A.4, a given disk contains more than $l \frac{R}{D}$ records of a given bucket with probability at most $\exp(-\Omega(l \cdot \frac{R}{D}))$, for constant $l \geq 6.4$. Let X denote the event that any disk contains more than $l \frac{R}{D}$ blocks of any bucket. There are D drives and D buckets, so with $k\gamma \geq BD$, and $\bar{v} \geq D \log D$, we have

$$\begin{aligned} \Pr[X] &\leq D^2 \exp(-\Omega(l \frac{R}{D})) \\ &\leq D^2 \exp(-\Omega(l \frac{v\gamma}{BD^2})) \\ &\leq \exp(-\Omega(l \frac{vBD}{kBD^2} - \log D)) \\ &\leq \exp\left(-\Omega(l \cdot \frac{\bar{v}}{D})\right) \end{aligned}$$

After D iterations of Step 1 in algorithm `SimulateRouting`, D blocks per bucket have been moved. Each disk contains less than $\frac{v\gamma}{D^2B}$ blocks of each bucket with high probability. Thus, after $D \frac{v\gamma}{D^2B}$ iterations, all blocks have been moved.

In Step 2, $\frac{v\gamma}{BD}$ iterations are performed. During each iteration, a parallel read and a parallel write operation are performed.

Thus, the total I/O-time of algorithm `SimulateRouting` is $O(G \frac{lv\gamma}{BD})$ and the total computation time is $O(lv\gamma)$ with high probability. $\square$

Lemma 4.3 *A compound superstep of a v -processor BSP^* with computation time $\tau + L$, communication time $\hat{g}\gamma/b + L$, and local memory of size μ can be simulated on a single processor $EM\text{-}BSP^*$ with D disks and internal memory of size M in computation time $v\tau + O(l \cdot v\mu)$ and I/O time $O(l \cdot G \frac{v\mu}{BD})$ with probability at least $1 - \exp(-\Omega(l \cdot \frac{\bar{v}}{D}))$ for constant $l \geq 6.4$, $\bar{v} \geq D \log D$, $M \geq k\mu + BD$, $b \geq B$, $\bar{v} = \frac{v}{k}$, and arbitrary integer $k \leq \frac{v}{p}$.*

Proof. The local memory of a virtual processor is of size μ , which is sufficient for incoming messages. Allowing in addition for a parallel disk track in memory implies $M \geq k\mu + BD$ memory in the $EM\text{-}BSP^*$ machine.

The disk space needed by the simulation is the total context size $v\mu$, which includes space for incoming messages. By Lemma A.4, the communicated data is evenly distributed over the disks with high probability. Therefore we need in total $O(v\mu BD)$ space on each disk with high probability.

Step 1(c) of algorithm `SeqCompoundSS-4.1` consumes $v\tau$ computation time overall. For each group of k virtual processors, $k\gamma/b$ messages are generated. This adds $O(v\gamma)$ computation time overall.

During Step 1(d) of algorithm `SeqCompoundSS-4.1`, a permutation can be generated in $O(D)$ time, so the computation time for each batch is $O(D \frac{v\gamma}{BD} + k\gamma)$ and I/O time $O(\frac{k\gamma}{BD})$. Therefore, Step 1(d) contributes computation time $O(v\gamma)$ and I/O time $O(G \frac{v\gamma}{BD})$ for the whole superstep.

By Lemma 4.1 and Lemma 4.2 the computation time and I/O-time respectively for Steps 2, 1(a), 1(b) and 1(d) are $O(v\mu)$ and $O(G \cdot l \frac{v\mu}{BD})$. Since $\gamma = O(\mu)$, these

dominate the costs of Step 1(d), so overall, computation time is $v\tau + O(l \cdot \mu v)$ and I/O-time is $O(G \cdot l \frac{v\mu}{BD})$ with probability at least $1 - \exp(-\Omega(l \cdot \frac{\bar{v}}{D}))$. $\square$

Lemma A.5 allows us to exploit the independence of the random experiments performed during each compound superstep in order to prove that the success probability for the entire simulation is as large as for the simulation of a single compound superstep.

Theorem 4.1 *A v -processor BSP* algorithm $\mathcal{A}$ with communication time $\hat{g}\alpha/b + \lambda L$, computation time $\beta + \lambda L$, and local memory μ can be simulated as a single processor EM-BSP* algorithm $\mathcal{A}'$ with computation time $v\beta + O(l \cdot v\lambda\mu)$ and I/O time $O(l \cdot G\lambda \frac{v\mu}{BD})$ with probability at least $1 - \exp(-\Omega(l \cdot \frac{\bar{v}}{D}))$ for suitable constant $l \geq 6.4$, $M \geq k\mu + BD$, $\bar{v} \geq D \log D$, $b \geq B$, $\bar{v} = \frac{v}{k}$, and arbitrary integer $k \leq v$.*

In particular, if $\mathcal{A}$ is 1-optimal and $\beta = \omega(\lambda\mu)$, $G = BD \cdot o(\frac{\beta}{\mu\lambda})$, the simulation results in a 1-optimal EM-BSP algorithm.*

Proof. Since we assume that the amount of communication each BSP* processor performs per superstep is bounded by its memory size μ , we can conclude that the communication time of each compound superstep is bounded by $\hat{g}\mu/b + L$.

From Lemma 4.3, a compound superstep of $\mathcal{A}$ with communication time $\tau + L$ and computation time $\hat{g}\mu/b + L$ can be simulated on an EM-BSP* machine with D disks and local memory $\Theta(k\mu)$ in computation time $v\tau + O(l \cdot v\mu)$ and I/O time $O(l \cdot G \frac{v\mu}{BD})$ with probability at least $1 - \exp(-\Omega(l \cdot \frac{\bar{v}}{D}))$.

The computation time required to simulate the computation steps of $\mathcal{A}$ is $v\beta$. The computational overhead is $O(l \cdot \lambda v\mu)$, which is asymptotically smaller than $v\beta$ for $\beta = \omega(\lambda\mu)$ and constant l .

Lemma A.5 allows us to exploit the independence of the random experiments performed during each compound superstep in order to prove that the success probability for the entire simulation is as large as for the simulation of a single compound superstep. Since the worst case runtime of a compound superstep is at most D times the average runtime, the theorem follows from Lemma A.5, with $m = \frac{\bar{v}}{D}$, $x = D$, and $\bar{v} \geq D \log D$. $\square$

4.2 Multiple Processor Target Machine

In this section, we generalize the simulation to $p \geq 1$ processors on the target EM machine.

We first describe the simulation of a compound superstep of a v -processor BSP* with communication time $\hat{g}\gamma/b + L$, computation time $\tau + L$ and context size μ on a p -processor EM-BSP*, for $p \geq 1$. We assume further that $b \geq B$, where b is the message block size of the BSP* virtual machine.

Outline of the parallel simulation: As an initial step, $\frac{v}{p}$ virtual processors $i\frac{v}{p}, \dots, (i+1)\frac{v}{p} - 1$ are assigned to each simulating processor i . As before, the simulation of each compound superstep is composed of a series of rounds. During the j^{th} of $\frac{v}{pk}$ rounds processor i simulates the steps of the k virtual processors $i\frac{v}{p} + jk, \dots, i\frac{v}{p} + (j+1)k - 1$. Messages sent between virtual processors on different real machines require real communication by the simulation. If these messages are sent directly to their destinations by the simulation the traffic may be unbalanced, causing inefficiencies in communication. Therefore, we distribute the messages randomly among the processors after each round. After the last round of a compound superstep, each processor reorganizes the messages it has received so that during the fetching phase of the j^{th} round of the next compound superstep it can read the messages destined for the virtual processors $jkp, \dots, (j+1)kp - 1$ in parallel from disk and direct them to the correct simulating processor.

We maintain $\frac{v}{pk}$ batches to store the generated messages. The j^{th} batch contains the messages destined for virtual processors simulated in the j^{th} round of the current compound superstep. Each processor writes its share of the $\frac{v}{pk}$ batches to its local disks. The main difficulty is the efficient maintenance of the batches so that the packets which are needed during the j^{th} simulation round can be read in parallel from the disks.

Algorithm ParCompoundSS-4.3 is executed by each real processor i , for $0 \leq i \leq p-1$, for each compound superstep.

Algorithm 4.3 ParCompoundSS-4.3

Objective: Simulation of a compound superstep of a v -processor BSP* on a p -processor EM-BSP*.

Input: For every batch j , where $0 \leq j \leq \frac{v}{pk}$, the message and context blocks of the virtual processors $jkp, \dots, (j+1)kp-1$ are divided among the real processors and their local disks so that each disk contains $O(\frac{k\gamma}{BD})$ blocks of message data, and $O(\frac{k\mu}{BD})$ blocks of context.

Output: The changed contexts and generated messages distributed as required for the next compound superstep.

- (1) For $j = 0$ to $\frac{v}{pk} - 1$ do
 - (a) (*Fetching Phase*): Processor i reads the contexts for group j from its local disks in parallel, and then reads any message blocks pertaining to group j from its local disks in parallel and sends them to the appropriate simulating processors.
 - (b) (*Computing Phase*): Processor i simulates the computation supersteps of its current virtual processors and collects all generated messages in its local memory.
 - (c) (*Writing Phase*): Processor i splits all generated messages into packets of size b (the message size) and sends each packet to a randomly chosen processor. Processor i writes the contexts for group j back to its local disks.
- (2) For $i = 0$ to $p - 1$ do in parallel: processors i reorganize the received batches using algorithm *SimulateRouting* so that each batch is evenly distributed over the local disks.

— End of Algorithm —

Many of the details of algorithm *ParCompoundSS-4.3* are similar to ones previously described for Algorithm 4.1, so we focus on the differences. In each round a processor receives $\frac{k\gamma}{b}$ packets with high probability. In Step 1(c), each processor cuts the packets it receives into blocks of size B . The blocks are written to the disks, using a random permutation of disk numbers as before. Depending on their destination address, the blocks are maintained in D buckets, which are divided among the disks as in Step 1(d) of the single-processor simulation. Each bucket contains the blocks for $\frac{v}{pk}/D$ batches. As before, a table of D pointers is maintained for each disk. As before, we can show that each disk will receive the same number of blocks of every bucket with high probability.

Lemma 4.4 *Step 2 of algorithm *ParCompoundSS-4.3* can be done in $O(l \cdot \frac{v\gamma}{p})$ computation time and $O(l \cdot G \frac{v\gamma}{pDB})$ I/O time with probability at least $1 - e^{-\Omega(l \cdot \frac{v}{pD})}$ for suitable constant $l \geq 6.4$, $\bar{v} \geq pD \log(pD)$, $k\gamma \geq BD$, and $B = b$.*

Proof. We begin by considering how the packets belonging to a fixed bucket are distributed among the processors by Step 1(c). A bucket contains the blocks/packets of $\frac{v}{pkD}$ batches and each batch contains $\frac{k\gamma}{b}$ packets. In total, for each bucket, $\frac{v\gamma}{BD}$ packets are distributed randomly among p processors. Since we have D buckets and p processors, the probability that any processor receives more than $R = l\frac{v\gamma}{pDb}$ packets for any bucket is $\exp(-\Omega(l\frac{v\gamma}{pDb} - \ln(pD)))$, from Lemma A.3 with $x = \frac{v\gamma}{B}$ and $y = pD$.

To bound this probability, let $\frac{v\gamma}{pDb} \geq \log(pD)$, meaning

$$v\gamma \geq pDb \log(pD) \quad (4.1)$$

From the statement of the lemma we have

$$\bar{v} \geq pD \log(pD) \quad (4.2)$$

We now show that (4.2) implies (4.1). Since $D \geq 1$ and $B = b$, (4.2) means that

$$\bar{v} \geq p(b/B) \log(pD)$$

Multiplying by BD , and substituting $\gamma \geq \frac{BD}{k}$, we obtain (4.1) as required. With constraint (4.2), therefore, we have probability at least $1 - \exp(-\Omega(l \cdot \frac{v\gamma}{pDb}))$ that each bucket of every processor contains less than $l\frac{v\gamma}{pDb}$ packets. To bound this probability further, to $1 - \exp(-\Omega(l \cdot \frac{\bar{v}}{pD}))$, we need a further constraint,

$$\begin{aligned} \frac{v\gamma}{pDb} &\geq \frac{\bar{v}}{pD} \\ \Rightarrow k\gamma &\geq b \end{aligned}$$

Since $B = b$, this constraint is absorbed by $k\gamma \geq BD$, given in the lemma.

We now turn to the issue of whether the packets for each bucket are balanced across the local disks of each real processor. We know that each processor holds $R = \frac{v\gamma}{pDb}$ packets for each bucket. For a fixed processor we have a similar situation to the single simulating processor case.

By Lemma A.4, we know that a fixed drive contains less than $l\frac{R}{D}$ blocks of a fixed bucket with probability at most $\exp(-\Omega(l\frac{R}{D}))$. Let X denote the event that any disk on any processor contains more than $l\frac{R}{D}$ blocks of any bucket. There are p processors, D drives and D buckets. With $k\gamma \geq BD$, we have, similar to the proof of Lemma

4.2,

$$\begin{aligned}
\Pr[X] &\leq pD^2 \exp(-\Omega(l \cdot R/D)) \\
&\leq pD^2 \exp(-\Omega(l \frac{v\gamma}{pbD^2})) \\
&\leq \exp(-\Omega(l \frac{vBD}{kpbD^2} - \log D - \log p)) \\
&\leq \exp(-\Omega(l \frac{\bar{v}B}{pbD} - \log(pD)))
\end{aligned}$$

Now, provided that

$$\begin{aligned}
\frac{\bar{v}B}{pbD} &\geq \log(pD) \\
\Rightarrow \bar{v} &\geq pD \frac{b}{B} \log(pD)
\end{aligned} \tag{4.3}$$

we have

$$\Pr[X] \leq \exp\left(-\Omega(l \cdot \frac{\bar{v}B}{pbD})\right)$$

and for B/b a constant, we have

$$\Pr[X] \leq \exp\left(-\Omega(l \cdot \frac{\bar{v}}{pD})\right)$$

So, in Step 2 each processor can reorganize its packets as required of the input in algorithm ParCompoundSS-4.3 in $O(l \cdot \frac{v\gamma}{p})$ computation time and $O(l \cdot G \frac{v\gamma}{pDB})$ I/O-time with probability at least $1 - \exp(-\Omega(l \cdot \frac{\bar{v}}{pD}))$, provided that $\bar{v} \geq pD \log(pD)$. $k\gamma \geq BD$, $B = b$, $l \geq 6.4$. $\square$

Lemma 4.5 *A compound superstep of a v -processor BSP* with computation time $\tau + L$, communication time $\hat{g}\gamma/b + L$, and local memory size μ can be simulated on a p -processor EM-BSP* in computation time $\frac{v}{p}\tau + O(l \cdot \frac{v}{p}(\gamma + \mu)) + \frac{\bar{v}}{p}L$, communication time $O(l \cdot \hat{g} \frac{v}{p} \frac{\gamma}{b}) + \frac{\bar{v}}{p}L$ and I/O-time $O(l \cdot G \frac{v}{p} \frac{\mu}{BD}) + \frac{\bar{v}}{p}L$ with probability at least $1 - e^{-\Omega(l \cdot \frac{\bar{v}}{pD})}$ for $M \geq k\mu + BD$, $\bar{v} \geq pD \log(pD)$, $B = b$, $k\gamma \geq BD$, $k\gamma \geq b \log p$, $k\gamma \geq \frac{\bar{v}b}{pD}$, $\bar{v} = \frac{v}{k}$, $k \leq \frac{v}{p}$ and suitable constant $l \geq 6.4$.*

Proof. Each of the $\frac{v}{pk}$ batches contains the packets generated by pk simulated processors. We ensure that each batch contains $\frac{pk\gamma}{b}$ packets by creating dummy packets if necessary. We first consider the runtime of Step 1 for a fixed round j :

Step 1(a) : Each processor reads the blocks belonging to batch j from its local disks in I/O-time $O(G\frac{k\gamma}{BD})$. The blocks destined for a common processor are combined into packets of size b , and all packets are then sent to their destination in a single superstep. Each simulating processor sends $O(\frac{k\gamma}{b})$ message packets, since each holds $O(\frac{k\gamma}{B})$ blocks of messages. Each simulating processor receives $\frac{k\gamma}{b}$ packets. Thus an $O(\frac{k\gamma}{b})$ -relation is routed, consuming $O(\hat{g}\frac{k\gamma}{b} + L)$ communication time. The processors can reassemble messages from received packets in linear time since each receives $k\gamma$ data per round and sufficient real memory exists to hold all of the received messages. Each processor then reads the contexts of its k currently simulated processors in I/O-time $O(G\frac{k\mu}{BD})$ from its local disks.

Step 1(b) : The steps of the currently simulated processors are performed, and the contexts are written back to the disks. Since $\gamma = O(\mu)$, the I/O-time is $O(Gl\frac{k\mu}{BD})$ and the computation time is $k\alpha + O(lk\mu) + L$.

Step 1(c) : During step 1(b), message packets of total size $O(k\gamma)$, each one of size b , were generated and stored in the local memories of the simulating processors. Now each processor sends each of its $O(\frac{k\gamma}{b})$ packets to a randomly chosen processor. This corresponds to randomly throwing $p\frac{k\gamma}{b}$ balls into p bins.

With Lemma A.3 for $x = p \cdot \frac{k\gamma}{b}$, $y = p$, we find that each processor receives more than $l \cdot k\gamma/b$ packets with probability at most $\exp(-\Omega(l \cdot \frac{k\gamma}{b} - \ln p))$.

Since these messages are exchanged in one superstep, an $O(l \cdot k\gamma/b)$ -relation is routed, requiring $O(l \cdot \hat{g}\frac{k\gamma}{b} + L)$ communication time, with probability $\exp(-\Omega(l \cdot \frac{k\gamma}{b} - \ln p))$. Thus each processor receives data of size $O(k\gamma)$ which is written to its disks in I/O-time $l \cdot O(G\frac{k\gamma}{BD})$ time, for suitable $l \geq e$, $b = B$, using randomly generated permutations of the disk numbers.

For a single round the I/O-time is bounded by $l \cdot O(G\frac{k\mu}{BD})$, communication time is $O(l \cdot \hat{g}\frac{k\gamma}{b} + L)$ and computation time is $k\alpha + O(kl(\gamma + \mu)) + L$ with probability $\exp(-\Omega(l \cdot (\frac{k\gamma}{b} - \ln p)))$. For

$$k\gamma \geq b \log p \tag{4.4}$$

the probability becomes $1 - \exp(-\Omega(l \cdot \frac{k\gamma}{b}))$.

To bound the probability at $1 - \exp(-\Omega(l \cdot \frac{\bar{v}}{pD}))$, we require that¹

$$\begin{aligned} \frac{k\gamma}{b} &\geq \frac{\bar{v}}{pD} \\ \Rightarrow k\gamma &\geq \frac{\bar{v}b}{pD} \end{aligned} \quad (4.5)$$

Now we consider the runtime for $\frac{v}{pk}$ rounds. The rounds are independent. We introduce $z = \frac{v}{pk}$ independent random variables $X_1, X_2, \dots, X_z$, where X_i represents the cost (communication, computation, and I/O-time) of the i^{th} round. We can apply Lemma A.5 to bound the total communication cost of all $\frac{v}{pk}$ rounds using the substitutions $x = p$, and $m = \frac{\bar{v}b}{pD}$ provided

$$\begin{aligned} m &\geq \log p \\ \Rightarrow \bar{v} &\geq pD \log p \end{aligned} \quad (4.6)$$

In total, we have I/O-time $l \cdot O(G \frac{v\mu}{pBD})$, communication time $l \cdot O(\hat{g} \frac{v\mu}{pb}) + L \frac{v}{pk}$ and computation time $\alpha \frac{v}{p} + O(l \cdot \frac{v}{p} \mu) + L \frac{v}{pk}$ with probability at least $1 - \exp(-\Omega(l \cdot \frac{\bar{v}}{pD}))$.

Incorporating the costs and constraints for Step (2) from Lemma 4.4, constraint (4.6) is absorbed by $\bar{v} \geq pD \log(pD)$. $\square$

Theorem 4.2 states that a c-optimal BSP* algorithm is transformed into a c-optimal EM algorithm by the simulation for the general case of $p \geq 1$ simulating processors.

Theorem 4.2 *Let $l \geq 6.4be$ a constant. Let $\bar{v} = \Omega(pD \log(pD))$, $M \geq k\mu + BD$, $B = b$, $k\gamma \geq BD$, $k\gamma \geq b \log p$, $k\gamma \geq \frac{\bar{v}b}{pD}$, $\bar{v} = \frac{v}{k}$, and $k \leq \frac{v}{p}$.*

A v -processor BSP algorithm $\mathcal{A}$ with communication time $\hat{g}\alpha/b + \lambda L$, computation time $\beta + \lambda L$, and local memory μ can be simulated as a p -processor EM-BSP* algorithm $\mathcal{A}'$ with computation time $\frac{v}{p}\beta + O(l \cdot \frac{v}{p}\lambda\mu) + \frac{\bar{v}}{p}\lambda L$, communication time $O(l \cdot \hat{g}(\frac{v\alpha}{p})) + \frac{\bar{v}}{p}\lambda L$, and I/O time $O(l \cdot G \frac{v}{p}(\frac{\mu\lambda}{BD})) + \frac{\bar{v}}{p}\lambda L$ with probability at least $1 - \exp(-\Omega(l \cdot \frac{\bar{v}}{pD}))$.*

¹If each communication superstep was an $\frac{N}{v}$ -relation, $k\gamma = \frac{N}{v}$, and we get the constraint $N \geq \frac{\bar{v}^2 b}{pD}$.

If $\mathcal{A}$ is also a CGM algorithm then the constraints $k\gamma \geq BD$, $k\gamma \geq b \log p$, $k\gamma \geq \frac{\bar{v}b}{pD}$ can be replaced by $N \geq \bar{v}BD$, $N \geq \bar{v}B \log p$, and $N \geq \frac{\bar{v}^2 B}{pD}$, respectively.

If $\mathcal{A}$ is 1-optimal on the BSP* for $\hat{g} \leq g(n)$, $b \leq b(n)$, $L \leq L(n)$ and $v \leq v(n)$, then $\mathcal{A}'$ is a 1-optimal EM-BSP* algorithm for $\beta = \omega(\lambda\mu)$, $\hat{g} \leq g(n)$, $b \leq b(n)$, $L \leq L(n) \cdot \frac{pk}{v}$ and $G = BD \cdot o(\frac{\beta}{\mu\lambda})$.

Proof. Since we assume that the amount of communication each BSP* processor performs per superstep is bounded by its memory size μ , we can conclude that the communication time of each compound superstep is bounded by $\hat{g}\mu/b + L$.

From Lemma 4.5, each compound superstep of $\mathcal{A}$ with computation time $\tau + L$, communication time $\hat{g}\gamma/b + L$, and local memory size μ can be simulated on a p -processor EM-BSP* in computation time $\frac{v}{p}\tau + O(l \cdot \frac{v}{p}(\gamma + \mu)) + \frac{\bar{v}}{p}L$, communication time $O(l \cdot \hat{g}\frac{v}{p}(\frac{\gamma}{b}) + \frac{\bar{v}}{p}L$ and I/O-time $O(l \cdot G\frac{v}{p}\frac{\mu}{BD}) + \frac{\bar{v}}{p}L$ with probability at least $1 - e^{-\Omega(l \cdot \frac{\bar{v}}{pD})}$ for suitable constant $l \geq 6.4$, $M \geq k\mu + BD$, $\bar{v} \geq pD \log(pD)$, $B = b$, $k\gamma \geq b$, $\bar{v} = \frac{v}{k}$, and $k \leq \frac{v}{p}$.

The computation time required to simulate the computation steps of $\mathcal{A}$ is $\frac{v}{p}\beta$. The computational overhead is λ times the overhead of a single superstep, or $O(l \cdot \lambda v \mu)$, which is asymptotically smaller than $v\beta$ for $\beta = \omega(\lambda\mu)$ and constant l .

Lemma 4.5 showed that the communication between real processors within each superstep is $l \cdot \frac{v}{p}(\frac{\gamma}{b})$, and so the communication for $\mathcal{A}'$ is $l \cdot \frac{v}{p}(\frac{a}{b})$ communication packets.

The I/O volume increases by a factor of λ over the time for a single compound superstep, to $l \cdot \frac{v}{p}\frac{\lambda\mu}{BD}$ I/Os.

Lemma A.5 allows us to exploit the independence of the random experiments performed during each compound superstep in order to prove that the success probability for the entire simulation is as large as for the simulation of a single compound superstep. Since the worst case runtime of a compound superstep is at most D times the average runtime, the theorem follows from Lemma A.5, with $m = \frac{\bar{v}}{pD}$, $x = D$, and $\bar{v} \geq pD \log(pD)$. $\square$

The slackness $\frac{v}{p}$ required by the simulation is controlled by the number of processors and disks we want to employ as well as the desired success probability. The

simulation increases the number of supersteps by a factor of $\frac{v}{pk}$. However we inherit the good communication time from the BSP* algorithm which permits a 1-optimal multiple processor EM-BSP* algorithm.

Chapter 5

Deterministic Simulation

5.1 Introduction

A randomized simulation of BSP algorithms on an EM-BSP machine was described in Chapter 4. This simulation took a BSP* algorithm, and produced an EM-BSP* algorithm, making use of the knowledge that messages of the BSP* are no smaller than a known value, b . In this chapter we describe a deterministic simulation technique that permits a CGM algorithm to be simulated on an EM-CGM machine.

We first consider the simulation of a single compound superstep, and in particular, how the contexts and messages of the virtual processors can be stored on disk and retrieved efficiently in the next superstep. We assume that the size of the contexts of the processors is known, and is the same for each virtual processor. We can therefore store the blocks of the contexts on fixed tracks of the disks. The main issue is how to organize the generated messages on the D disks so that they can be accessed using blocked and fully parallel I/O operations. This task is simpler if the messages have a fixed length. Although a CGM algorithm has the property that $\Theta(\frac{N}{v})$ data is deemed to be exchanged by each processor in every superstep, there is no guarantee on the size of individual messages.

Section 5.2 describes a technique which replaces each communication superstep of a BSP algorithm (not necessarily a CGM algorithm) by two supersteps in which the messages are of uniform size.

In Sections 5.3 and 5.4, we describe how the simulation can be accomplished on target machines with one and multiple processors, respectively.

5.2 Balancing Communication

Algorithm *BalancedRouting* gives us a technique for acquiring messages of nearly uniform size. We will show that we can derive upper and lower bounds on the message size, which allow us to save and retrieve messages on the disks in an efficient and convenient manner.

A total of $\bar{n}$ data is divided evenly among v processors. Each item is labelled with the identifier of a destination processor in such a way that no more than $\bar{h}$ items have the same destination, where $\bar{h} \geq \frac{\bar{n}}{v}$. Algorithm *BalancedRouting* delivers these items to their destination in two rounds of communication, where no message differs in size from any other by more than $v - 1$ items (see Theorem 5.1 below).

Algorithm 5.1 *BalancedRouting* (from [13])

Input: Each of the v processors has $\frac{\bar{n}}{v}$ elements, which are divided into v messages, each of arbitrary length $\leq \frac{\bar{n}}{v}$. Let msg_{ij} denote the message to be sent from processor i to processor j , and let $|msg_{ij}|$ be the length of such a message.

Output: The v messages in each processor are delivered to their final destinations in two balanced rounds of communication, and each processor then contains at most $\bar{h}$ data.

A. For $i = 0$ to $(v - 1)$ in parallel

1. Processor i allocates v local bins, one for each processor

2. For $j = 0$ to $(v - 1)$

3. For $\ell = 0$ to $|msg_{ij}|$

Processor i allocates the ℓ^{th} word of msg_{ij} to local bin $(i + j + \ell) \bmod v$

4. Processor i sends bin j to processor j

———— Superstep Boundary ————

B. For $j = 0$ to $(v - 1)$ in parallel

1. Processor j reorganizes the messages it received in Step 2 into bins according to each element's final destination

2. Processor j routes the contents of bin k to processor k , for $0 \leq k \leq v - 1$

— End of Algorithm —

Observation 5.1 *If $\text{bin}_{\min}$ is the smallest bin created at a processor in step (1) of Superstep A, then the other $(v-1)$ bins can contain at most $1+2+\dots+(v-1) = \frac{v(v-1)}{2}$ more elements than does $\text{bin}_{\min}$ (see Figure 5.1).*

Theorem 5.1 *We are given v processors, and $\bar{n}$ data items. Each processor has exactly $\frac{\bar{n}}{v}$ data to be redistributed among the processors, and no processor is to be the recipient of more than $\bar{h}$ data. The redistribution can be accomplished in two communication rounds of balanced communication: (A) Messages in the first round are at least $\frac{\bar{n}}{v^2} - \frac{v-1}{2}$, and at most $\frac{\bar{n}}{v^2} + \frac{v-1}{2}$ in size, and (B) Messages in the second round are at least $\frac{\bar{h}}{v} - \frac{v-1}{2}$, and at most $\frac{\bar{h}}{v} + \frac{v-1}{2}$ in size.*

Proof. The proof of the maximum message sizes is given by Bader et al. [13]. The proof of the minimum message sizes relies on Observation 5.1; see Figure 5.1.

In round A each processor initially has $\frac{\bar{n}}{v}$ data. At the end of Superstep B, each processor will have at most $\bar{h}$ data.

First, we consider the minimum message size in Superstep A. Due to the round robin allocation mechanism, a given bin after Step 1 will contain at most one more element of a message to processor j than does any other bin. Let us fix any processor i . Consider the bin sizes after all of the messages have been distributed among the bins by processor i (see Figure 5.1). Clearly, all of the bins will contain at least as many elements as the smallest bin, $\text{bin}_{\min}$. Let e_j be the number of extra elements (more than this minimum) in bin j at Step 2. The crucial observation is that if $\text{bin}_{\min}$ is the smallest bin, then the other $(v-1)$ bins can hold at most $1+2+\dots+(v-1) = \frac{v(v-1)}{2}$ extra elements.

Thus,

$$\frac{\bar{n}}{v} = v|\text{bin}_{\min}| + \sum_j e_j$$

Since $\frac{v(v-1)}{2} \geq \sum_j e_j$, we have

$$\begin{aligned} \frac{\bar{n}}{v} &\leq v|\text{bin}_{\min}| + \frac{v(v-1)}{2} \\ |\text{bin}_{\min}| &\geq \frac{\bar{n}}{v^2} - \frac{v-1}{2} \end{aligned}$$

We now turn to the message sizes in Superstep B. The elements which arrive at processor j as a result of Step 2 are the contents of the j^{th} local bins formed in Step 1 at each of the processors 0 through $v - 1$. We can think of the j^{th} local bin of each of the v processors as a component of a single, global superbin, which is the union of the j^{th} local bins of all v processors. Consider only the messages destined for a fixed processor k which are held by each processor i , $0 \leq i \leq v - 1$, prior to Step 1. These are allocated among the superbins, starting with superbin $(i + k) \bmod v$ by Step 1. Superbin j now contains the message which is to be sent from processor j to processor k in Step 4.

In a similar manner to the analysis of superstep A, let E_j be the number of extra elements in superbin j after Step 1. Let $sbin_{\min}$ be the superbin which contains the minimum number of elements after Step 1, and hence $|sbin_{\min}|$ represents the minimum message size in Step 4. When processor k is one of the processors which receives the maximum h data elements, we have $\bar{h} = v|sbin_{\min}| + \sum_j E_j$, and since $\frac{v(v-1)}{2} \geq \sum_j E_j$, we have $|sbin_{\min}| \geq \frac{\bar{h}}{v} - \frac{v-1}{2}$ $\square$

The notion of an h -relation is often used in the analysis of parallel algorithms based on *BSP-like* models (e.g. BSP, BSP*, CGM). An h -relation is a communication superstep in which each of the v processors sends and receives at most h data items. It is typically used in bounding the communication complexity in an asymptotic analysis. Based on this usage of an h -relation, we have:

Corollary 5.1 *An arbitrary h -relation can be replaced by two balanced h -relations whose message size is bounded by $\frac{h}{v} - \frac{v-1}{2}$ and $\frac{h}{v} + \frac{v-1}{2}$.*

Proof. In Theorem 5.1, clearly $\bar{h} \geq \frac{\bar{n}}{v}$. For $h = \bar{h} \geq \frac{\bar{n}}{v}$ we have a message size of at most $\frac{h}{v} + \frac{v-1}{2}$ for each of the two rounds of communication. We can reproduce the precise conditions of Theorem 5.1 by adding dummy items if necessary in Superstep A to ensure that $\frac{\bar{n}}{v} = h$.

We can assume a minimum message size of $\frac{h}{v} - \frac{v-1}{2}$ in the second round because the cost of communication is bounded by the assumption of an h -relation. When every processor is the destination of h data, it does not affect the worst case complexity of the superstep. We can therefore assume that every processor receives h data (by

Figure 5.1: *Illustration of maximum imbalance in the bin sizes of a processor during superstep A.* The light circles represent evenly placed elements and the dark circles are unevenly placed, or “extra” ones. The maximum bin size (bin $v - 1$ in the diagram) is at most $\frac{v-1}{2}$ more than the average, and the minimum bin size (bin 0 in the diagram) is at most $\frac{v-1}{2}$ less than the average.

adding dummy items for the sake of the argument). Hence the minimum message size for any processor in Superstep B becomes $\frac{h}{v} - \frac{v-1}{2}$ without affecting the asymptotic communication cost of the superstep. $\square$

Lemma 5.1 *An arbitrary minimum message size $b_{\min}$ can be assured provided that*

$$N \geq v^2 b_{\min} + \frac{v^2(v-1)}{2} \quad (5.1)$$

where N is the total number of problem items (summed over the v processors).

Proof. From Corollary 5.1, we can achieve a minimum message size $b_{\min}$ provided that $b_{\min} \leq \frac{N}{v^2} - \frac{v-1}{2}$. $\square$

Assurances regarding the minimum message size are particularly relevant to the BSP* model. In Section 5.5 we discuss the use of *BalancedRouting* and Theorem 5.1 for creating BSP* algorithms from BSP algorithms.

In Section 5.3 and 5.4 we describe the simulation of CGM algorithms with the additional properties that *BalancedRouting* provides. Not every CGM algorithm will require balancing, but Lemma 5.2 ensures that we can obtain balanced message sizes when necessary by increasing the number of supersteps by a factor of 2.

Lemma 5.2 *Let A be a CGM algorithm with N data, v processors, and λ communication steps, where $h = \frac{N}{v}$ in every step. The λ communication steps of A can be replaced by 2λ steps of balanced communication in which the minimum message size is $\Omega(B)$ and the maximum message size is $2 \cdot \frac{N}{v^2}$ provided that $N \geq v^2 B + \frac{v^2(v-1)}{2}$*

Proof. The minimum and maximum message sizes follow from Corollary 5.1, with $h = \frac{N}{v}$, and the constraint that $\frac{N}{v^2} + \frac{v-1}{2} \leq 2 \cdot \frac{N}{v^2}$. This is true if $N \geq \frac{v^2(v-1)}{2}$, which is absorbed by (5.1) from Lemma 5.1. $\square$

In many practical EM situations $2 \cdot \frac{N}{v^2}$ will be a significant overestimate of the maximum message size, as $v \ll \frac{N}{v^2}$.

5.3 Single Processor Target Machine

We now turn to the actual simulation results, which rely on a message size of $c \frac{N}{\bar{v}^2}$, where $\bar{v} = \frac{v}{k}$, for a known constant $c \geq 1$. As we have seen, this is guaranteed by algorithm *BalancedRouting* (see Lemma 5.2), for $c = 2$. Not every algorithm will require balancing; see *CGMTranspose* in Section 8 for an example of a CGM algorithm whose messages are already balanced.

For the case of a single EM processor, we simulate a compound superstep of a *BSP-like* (e.g. CGM) algorithm $\mathcal{A}$, using the algorithm *SeqCompoundSuperstep*, shown below. No real communication is required. Lemma 5.3 gives the resulting complexities for a single compound superstep, and Theorem 5.2 summarizes the overall simulation result for a single real processor. For ease of exposition, we assume that k divides v .

Lemma 5.3 *A compound superstep of a v -processor CGM algorithm $\mathcal{A}$ with computation time $\tau + L$, communication time $g \cdot O(\frac{N}{v}) + L$, message size $c \frac{N}{\bar{v}^2}$, for a known constant $c \geq 1$, and local memory size μ can be simulated as a single processor EM-CGM algorithm $\mathcal{A}'$ in computation time $v\tau + O(v\mu)$ and I/O time $O(G(\frac{N}{BD} + \frac{v\mu}{BD}))$ provided that $M \geq k\mu + BD$, $N = \Omega(\bar{v}BD)$, and $N = \Omega(\bar{v}^2B)$, for an arbitrary integer $k \leq v$, and $\bar{v} = \frac{v}{k}$.*

Proof. Since messages are at most $c \cdot \frac{N}{\bar{v}^2}$ in size we can allocate fixed sized slots for them on the disks while preserving an $O(v\mu)$ disk space requirement. $\mathcal{A}'$ requires that $\frac{N}{\bar{v}^2} = \Omega(B)$, so $N = \Omega(\bar{v}^2B)$. The assurance of minimum message size $\Omega(B)$ implies that I/O operations will be blocked. Although a CGM algorithm may occasionally use smaller messages than B , it is charged for an $\frac{N}{v}$ -relation in each superstep as if every processor sent and received h data.

We simulate a compound superstep of $\mathcal{A}$ using algorithm *SeqCompoundSuperstep*, shown below. The algorithm expects the input messages to the virtual processors in the current superstep to be organized (by destination) in a parallel format on the disks, and it also writes the messages generated in the current superstep to the disks in a parallel format. We use the phrase “a *parallel format*” to mean an arrangement of the data that permits fully parallel access to the disks, both for writing the messages,

and for reading them back in a different order in the next superstep. Two instances of a parallel format are the *consecutive* and *staggered* formats. A more intuitive definition appears in Section 1.2.

Consecutive format: We say that a disk read/write operation on D blocks is *consecutive* when the q^{th} block, $0 \leq q \leq D$ is read/written from/to disk $(d + q) \bmod D$ on track $T_0 + \lfloor (d + q)/D \rfloor$, where T_0 is the track used for the first of the D blocks to be read/written, and d is the disk offset (from disk 0) for the first of the D blocks to be read/written.

Staggered format: We say that a disk read/write operation on D blocks involving n messages, each of size at most b' blocks, is *staggered* when the q^{th} block, $0 \leq q \leq (b' - 1)$ of the j^{th} message, $0 \leq j \leq (n - 1)$ is read/written from/to disk $(d + S + q) \bmod D$ on track $T_0 + \lfloor (d + S + q)/D \rfloor$, where T_0 is the track used for the blocks of the 0^{th} message, d is the disk offset (from disk 0) for the first of the D blocks to be read/written, and $\lceil S/D \rceil$ is the number of tracks by which consecutive messages are to be staggered (separated).

Algorithm SeqCompoundSuperstep-5.2 simulates a compound superstep of a v -processor CGM on a single processor EM-CGM with D disks. It simulates k processors at a time, where $1 \leq k \leq v$, provided that the simulating processor has $M = \Theta(k\mu)$ memory.

Algorithm 5.2 SeqCompoundSuperstep-5.2

Input: For each $j \in \{0, \dots, v - 1\}$ the blocks of the context are stored on the disks in consecutive format, and the arriving messages of virtual processor j are spread over the D disks consecutive format.

Output: (i) The (changed) contexts of the v simulated processors are spread across the disks in consecutive format. (ii) The generated messages for each processor in the next superstep are stored in consecutive format on the disks.

1. For $j = 0$ to $\frac{v}{k} - 1$
 - (a) Read the context of virtual processors jk to $(j + 1)k - 1$ from the disks into memory.
 - (b) Read the packets received by virtual processors jk to $(j + 1)k - 1$ from the disks.

- (c) Simulate the local computation of virtual processors jk to $(j+1)k-1$.
- (d) Write the packets which were sent by virtual processors jk to $(j+1)k-1$ to the D disks in the staggered format illustrated in Figure 5.2. See below for details.
- (e) Write the changed context of virtual processors jk to $(j+1)k-1$ back to the D disks (in consecutive format).

— End of Algorithm —

Details of Steps (a) and (e): We reserve an area of total size $v\mu$ on the disks, $\frac{v\mu}{BD}$ blocks on each disk, where we store the contexts. We split the context V_j of virtual processor j into blocks of size B and store the i -th block of V_j on disk $(i + j\frac{\mu}{B}) \bmod D$ using track $\left\lfloor \frac{i+j\frac{\mu}{B}}{D} \right\rfloor$. Since the context of each processor is now in striped format on the disks, we can read and write the contexts of k consecutive virtual processors using D disks in parallel for every I/O operation.

Details of Step (b): The previous compound superstep guaranteed that the blocks which contain the messages destined for the current processor are stored in consecutive format on the disks. Therefore, we can use a similar technique to fetch the messages as we used to fetch the contexts.

Details of Step (d): After the Computation Phase, all messages sent by the current k virtual processors have been generated and stored in internal memory. The coarse-grained nature of the underlying BSP algorithm results in large messages, (see Lemma 5.2) which are as long or longer than the block size B . We cut the messages into blocks of size B . Each block inherits the destination address from its original message. In $\frac{k\gamma}{BD}$ rounds, we write the blocks out to the disks, as described in detail below. Recall that γ is the maximum size of the data sent or received by any virtual processor over all supersteps.

Let b represent the maximum message size, and let b' represent the maximum number of disk blocks per message. Hence, $b' = \lceil \frac{b}{B} \rceil$. Let msg_{ij} represent the message sent from processor v_i to processor v_j in one communication superstep. We will store the messages destined for a fixed processor j in consecutive format, beginning with msg_{0j} and ending with $msg_{v-1,j}$. We ensure that the first block of $msg_{i,j+1}$ is assigned to disk $(b_0 + b') \bmod D$, for $0 \leq j \leq v-2$, where b_0 is the disk number of the first block

for msg_{ij} . In other words, the starting block positions for messages to consecutive processors are appropriately staggered on the disks to ensure that we can write blocks of messages to consecutively numbered processors in a single parallel I/O when $b' \bmod D \neq 0$. Let $T_j = j \cdot \lceil \frac{vb'}{D} \rceil$ be the track offset for msg_{0j} (the first such message destined for processor v_j). Let $d_j = jb' \bmod D$ be the disk offset (from disk 0) for the first block of msg_{0j} . The q^{th} block of msg_{ij} is assigned to disk $(d_j + ib' + q) \bmod D$ on track $T_j + \lfloor (d_j + ib' + q)/D \rfloor$. This storage scheme maintains what we will call the *messaging matrix* across the parallel disks. The messages destined to a particular virtual processor are stored in a band, or stripe of consecutive parallel tracks.

Outgoing message blocks are placed in a FIFO queue for servicing by procedure *DiskWrite*. *DiskWrite* removes at most D blocks from the queue in each write cycle and writes them to the disks in a single write operation. Blocks are serviced strictly in FIFO order. Blocks will be added to the current write cycle and removed from the queue until a block is encountered whose disk number conflicts with that of an earlier block in the current write cycle.

Since the messages destined for any given processor are stored in consecutive format on the disks, we can read the messages received by a virtual processor using D disks in parallel for every I/O operation. Except possibly for the last, each parallel read performed by the simulation of processor v_j will obtain D message blocks. By staggering the first message blocks for consecutive virtual processors across the disks, we can achieve fully parallel writes to the disks.

The scheme just described requires two copies of the messaging matrix because the messages generated by virtual processor i in compound superstep ℓ must be stored on disk before virtual processor $i + 1$ can process the messages generated for it in compound superstep $\ell - 1$.

We can avoid this extra space requirement, however, as follows.

Observation 5.2 *By alternating between {consecutive reads, staggered writes} and {staggered reads, consecutive writes} in successive compound supersteps, the simulation can achieve fully parallel I/O on all message blocks with a single copy of the messaging matrix.*

SeqCompoundSuperstep loads k virtual processors into the real memory at once, requiring that $M \geq k\mu$. Since the messages sent or received in a superstep by a virtual processor are $h = O(\frac{N}{v})$ in total size, we require that $\frac{kN}{vB} = \Omega(D)$ to ensure that our I/O scheme for messages is efficient. This means that $N = \Omega(\bar{v}BD)$, where $\bar{v} = \frac{v}{k}$.

Computation Time: Steps (a) and (e) of algorithm SeqCompoundSuperstep take $O(v\mu)$ computation time overall. In Steps (b) and (d), $O(\frac{N}{v})$ message data is routed for each virtual processor. Over all v processors, this adds $O(N)$ computation time overall, which can be ignored. Step (c) consumes $v\tau$ computation time.

I/O Time: Steps (a) and (e) consume $O(G\frac{v\mu}{BD})$ time, and Steps (b) and (d) consume $O(G\frac{N}{BD})$ time overall. Since $O(N/p)$ message data is sent in each superstep, and $N/p \leq \mu$ we have time $O(G\frac{v\mu}{BD})$ due to I/O, overall.

Thus, overall, the computation time is $v\tau + O(v\mu)$ and the I/O-time is $O(G\frac{v\mu}{BD})$. $\square$

Theorem 5.2 *A v processor CGM algorithm $\mathcal{A}$ with λ supersteps, local memory size μ , running time $\beta + g \cdot O(\frac{N}{v}) + \lambda L$, and message size $\Theta(\frac{N}{v^2})$ can be simulated as a single processor EM-CGM algorithm $\mathcal{A}'$ with time $v\beta + O(\lambda v\mu) + G \cdot O(\lambda\frac{v\mu}{BD})$ for $M \geq k\mu + BD$, $N = \Omega(\bar{v}^2B)$, and $v = \Omega(D)$, for an arbitrary integer $1 \leq k \leq v$, and $\bar{v} = \frac{v}{k}$.*

In particular, algorithm $\mathcal{A}'$ is c -optimal if $\mathcal{A}$ is c -optimal, $\beta = \omega(\lambda\mu)$ and $G = o(\frac{\beta BD}{\lambda\mu})$. Furthermore, algorithm $\mathcal{A}'$ is work-optimal and I/O-efficient if $\mathcal{A}$ is work-optimal and communication-efficient, $\beta = \Omega(\lambda\mu)$ and $G = O(\frac{\beta BD}{\lambda\mu})$.

Proof. We use the results of Lemma 5.3. The computation time required to simulate the computation steps of $\mathcal{A}$ is $v\beta$. The computational overhead associated with the I/O steps (Steps (a),(b),(d),(e)) is $O(\lambda v\mu) + O(\lambda N)$. Since $v\mu > N$ the total computation time is bounded by $v\beta + O(\lambda v\mu)$. When c -optimality is required, we therefore need $\lambda v\mu = o(v\beta)$, or $\beta = \omega(\lambda\mu)$. Note that when $\mu = \Theta(\frac{N}{v})$, we

Figure 5.2: *Illustration of the layout of message blocks on the D disks.* In the example we have $D = 5$ and message size $b' = 2$ blocks. Messages from processor i to processors j , $j+1$, and $j+2$ are shown as shaded rectangles. Messages to consecutively numbered processors are staggered on the disks to permit D blocks to be written in parallel.

can substitute $\beta = \omega(\frac{\lambda N}{v})$ for $\beta = \omega(\lambda \mu)$. For work-optimality, we require that $\lambda v \mu = O(v\beta)$, or $\beta = \Omega(\lambda \mu)$.

The number of I/O operations (Steps (a),(b),(d),(e)) is $O(\lambda \frac{v \mu}{B D}) + O(\lambda \frac{N}{D})$, which is bounded by $O(\lambda \frac{v \mu}{B D})$. For c -optimality, we require the I/O time to be in $o(v\beta)$, which means that $G = o(\frac{\beta B D}{\lambda \mu})$. For I/O-efficiency, we require the I/O time to be in $O(v\beta)$, which means that $G = O(\frac{\beta B D}{\lambda \mu})$.

The messages sent by a group of k virtual processors to any other group must be large enough in total that they fill a disk block, so $N = \Omega(\bar{v}^2 B)$. Also, the messages sent or received by a group in a single superstep must fill a track of the D parallel disks, meaning that $N = \Omega(\bar{v} B D)$, where $\bar{v} = \frac{v}{k}$. These constraints can be combined

into $N = \Omega(\bar{v}^2 B)$, for $v = \Omega(D)$. □

5.4 Multiple Processor Target Machine

For the case of $p \geq 1$ processors on the EM-CGM machine we simulate a compound superstep of a CGM algorithm $\mathcal{A}$ using the algorithm *ParCompoundSuperstep-5.3*, shown below. Unlike in the case of a single real processor, we are now forced to perform real communication between the real processors of the target machine.

Each real processor i , $0 \leq i \leq p - 1$, executes algorithm *ParCompoundSuperstep-5.3* in parallel. For ease of exposition, we assume that pk divides v .

Algorithm 5.3 *ParCompoundSuperstep-5.3*

Objective: Simulation of a compound superstep of a v -processor CGM on a p -processor EM-CGM.

Input: The message and context blocks of the virtual processors are divided among the real processors and their local disks. Each real processor i , $0 \leq i \leq (p - 1)$ holds $O(\frac{N}{pB})$ blocks of messages and $\frac{v\mu}{pB}$ blocks of context, and each local disk contains $O(\frac{N}{pBD})$ blocks of messages and $O(\frac{v\mu}{pBD})$ blocks of context.

Output: The changed contexts and generated messages distributed as required for the next compound superstep.

For $j = 0$ to $\frac{v}{pk} - 1$ do

- (a) Read the contexts of virtual processors ijk to $i(j + 1)k - 1$ from the local disks.
- (b) Read any message blocks addressed to virtual processors ijk to $i(j + 1)k - 1$ from the local disks.
- (c) Simulate the computation supersteps of virtual processors ijk to $i(j + 1)k - 1$, collecting all generated messages in the local internal memory.
- (d) Send all generated messages to the required (real) destination processors. Upon arrival, the messages are arranged within the internal memory of the real destination processor and then written to its disks as in the single processor simulation; see Algorithm 5.2.
- (e) Write the contexts for virtual processors ijk to $i(j + 1)k - 1$ back to the local disks; see Algorithm 5.2.

— End of Algorithm —

Lemma 5.4 *A compound superstep of a v -processor CGM algorithm $\mathcal{A}$ with computation time $\tau + L$, communication time $O(g\frac{N}{v}) + L$, message size $\Theta(\frac{N}{v^2})$, and local memory size μ can be simulated as $\frac{v}{p}$ compound supersteps of a p -processor EM-CGM algorithm $\mathcal{A}'$ in parallel computation time $\frac{v}{p}\tau + O(\frac{v}{p}\mu) + \frac{v}{p}L$, communication time $g \cdot O(\frac{N}{p}) + \frac{v}{p}L$ and I/O time $G\frac{v}{p} \cdot O(\frac{\mu}{BD}) + \frac{v}{p}L$, for $pk \leq v$, arbitrary integer $1 \leq k \leq \frac{v}{p}$, $N = \Omega(\bar{v}^2 B)$, $v = \Omega(D)$, and $\bar{v} = \frac{v}{k}$ provided $M \geq k\mu + BD$ memory is available on each real processor.*

Proof. There are $\frac{v}{p}$ virtual processors resident on each real processor. Each real processor simulates v/p virtual processors of a round of $\mathcal{A}$ in a total of $v/pk = \frac{v}{p}$ supersteps, as k virtual processors are simulated on each real processor in each round of $\mathcal{A}'$.

Communication time: In Step (d), in each round of $\mathcal{A}'$, a real processor receives

- p real messages, one from each real processor,
- $pk \cdot \frac{v}{p}$ virtual messages, one from each of pk virtual processors simulated in this round, to each of its own $\frac{v}{p}$ resident virtual processors,
- $pk \cdot \frac{v}{p} \cdot \frac{N}{v^2} = \frac{kN}{v}$ data items, since each virtual message is $\frac{N}{v^2}$ items in size.

It also sends $\frac{kN}{v}$ data in each round of $\mathcal{A}'$. We have $\frac{v}{pk}$ such rounds, so we have $\frac{v}{pk} \cdot 2k\frac{N}{v} = 2\frac{N}{p}$ data to communicate overall. Therefore, the overall communication time of $\mathcal{A}'$ is $g \cdot O(\frac{N}{p}) + \frac{v}{pk}L$.

I/O time: The I/O time is determined by the cost of swapping contexts plus the cost of simulating the original messaging via I/O. Each group of k processors has context of size $O(k\mu)$, which requires $O(\frac{k\mu}{BD})$ I/O operations to swap in or out of memory. Over $\frac{v}{pk}$ supersteps, the swapping of contexts therefore costs $G\frac{v}{pk} \cdot O(\frac{k\mu}{BD})$. The I/O costs due to messaging between virtual processors are bounded by $O(\frac{kN}{vBD})$ in each superstep, and so the total costs due to virtual messaging over $\frac{v}{pk}$ supersteps is

$G \frac{v}{pk} \cdot O(\frac{kN}{vBD})$. Since the context size $\mu = \Omega(\frac{N}{v})$, the total I/O cost overall is dominated by $G \cdot \frac{v}{p} \frac{\mu}{BD}$.

Computation time: We have $p \leq v$ real processors, so the time to simulate Step (c) is $\frac{v}{p}\tau$. Computational overhead is contributed by $O(\frac{v}{pk}(k\mu + \frac{kN}{v}))$, due to swapping of contexts (Steps (a),(e)) and messaging I/O (Steps (b),(d)). As before, the computational overhead is dominated by the cost of swapping contexts. $\square$

Theorem 5.3 *A v processor CGM algorithm $\mathcal{A}$ with λ supersteps, computation time $\beta + \lambda L$, communication time $g\lambda\frac{N}{v} + \lambda L$, local memory size μ and message size $b = \Theta(\frac{N}{v^2})$ can be simulated as a p -processor EM-CGM algorithm $\mathcal{A}'$ with computation time $\frac{v}{p}\beta + \frac{v}{p}O(\lambda\mu) + \frac{v}{p}\lambda L$, communication time $g\lambda O(\frac{N}{p}) + \frac{v}{p}\lambda L$, and I/O time $G\lambda\frac{v}{p}O(\frac{\mu}{BD}) + \frac{v}{p}\lambda L$ for $M \geq k\mu + BD$, $p \leq v$, $N = \Omega(\bar{v}^2 B)$, and $v = \Omega(D)$, for arbitrary integer $1 \leq k \leq \frac{v}{p}$, and $\bar{v} = \frac{v}{k}$.*

Let $g(N)$, $L(N)$, and $v(N)$ be increasing functions of N . If $\mathcal{A}$ is c -optimal on the CGM for $g \leq g(N)$, $L \leq L(N)$ and $v \leq v(N)$, then $\mathcal{A}'$ is a c -optimal EM-CGM algorithm for $\beta = \omega(\lambda\mu)$, $g \leq g(N)$, $G = o(\frac{\beta BD}{\mu\lambda})$ and $L \leq L(N) \cdot \frac{p}{v}$. $\mathcal{A}'$ is work-optimal, communication-efficient, and I/O-efficient if $\mathcal{A}$ is work-optimal and communication-efficient, $\beta = \Omega(\lambda\mu)$, $g \leq g(N)$, $G = O(\frac{\beta BD}{\mu\lambda})$, and $L \leq L(N) \cdot \frac{p}{v}$.

Proof. We use the results of Lemma 5.4. The computation time required to simulate the computation steps of $\mathcal{A}$ is $\frac{v}{pk}k\beta$. The computational overhead associated with the I/O and communication steps (Steps (a),(b),(d),(e)) is $O(\frac{v}{p}\lambda\mu + \frac{v}{pk}\lambda\frac{N}{v})$ from Lemma 5.4. Since $\mu \geq \frac{N}{v}$, the total computation time is bounded by $\frac{v}{p}\beta + O(\frac{v}{pk}\lambda\mu)$. When c -optimality is required, we need $\beta = \omega(\lambda\mu)$. Note that in many cases $\frac{N}{v} = \Theta(\mu)$. Also, when only work-optimality is required, $\beta = \Omega(\lambda\mu)$ suffices.

From Lemma 5.4 the communication time of the simulation per superstep of $\mathcal{A}$ is $O(g\lambda\frac{N}{p} + \frac{v}{pk}L)$, giving $g\lambda\frac{N}{p} + \frac{v}{pk}\lambda L$ time overall.

The I/O time (Steps (a),(b),(d),(e)) is $G\lambda\frac{v}{p} \cdot O(\frac{\mu}{BD} + \frac{N}{vBD})$, which is bounded by $G\lambda\frac{v}{p}O(\frac{\mu}{BD})$. For c -optimality, we require the I/O time to be in $o(\frac{v}{p}\beta)$, which means that $G = o(\frac{\beta BD}{\lambda\mu})$. For I/O-efficiency we need only that $G = O(\frac{\beta BD}{\lambda\mu})$. Since the number of supersteps increases by a factor of $\frac{v}{pk}$ we require that $L \leq L(N) \cdot \frac{pk}{v}$.

□

Theorem 5.4 shows that Theorem 5.3 holds even if $\mathcal{A}$ has unknown message size, provided that, in addition to the constraints of Theorem 5.3, $B \geq \frac{v}{2}$, and $N \geq \bar{v}^2 B$, $v \geq D$.

Theorem 5.4 *A v processor CGM algorithm $\mathcal{A}$ with λ supersteps, computation time $\beta + \lambda L$, communication time $g\lambda\frac{N}{v} + \lambda L$, and local memory size μ can be simulated as a p -processor EM-CGM algorithm $\mathcal{A}'$ with computation time $\frac{v}{p}\beta + \frac{v}{p}O(\lambda\mu) + \frac{\bar{v}}{p}\lambda L$, communication time $g\lambda O(\frac{N}{p}) + \frac{\bar{v}}{p}\lambda L$, and I/O time $G\lambda\frac{v}{p}O(\frac{\mu}{BD}) + \frac{\bar{v}}{p}\lambda L$ for $M \geq k\mu + BD$, $p \leq v$, $N = \Omega(\bar{v}^2 B)$, $v = \Omega(D)$, and $B \geq \frac{v}{2}$, for arbitrary integer $1 \leq k \leq \frac{v}{p}$, and $\bar{v} = \frac{v}{k}$.*

Let $g(N)$, $L(N)$, and $v(N)$ be increasing functions of N . If $\mathcal{A}$ is c -optimal on the CGM for $g \leq g(N)$, $L \leq L(N)$ and $v \leq v(N)$, then $\mathcal{A}'$ is a c -optimal EM-CGM algorithm for $\beta = \omega(\lambda\mu)$, $g \leq g(N)$, $G = o(\frac{\beta BD}{\mu\lambda})$ and $L \leq L(N) \cdot \frac{p}{\bar{v}}$. $\mathcal{A}'$ is work-optimal, communication-efficient, and I/O-efficient if $\mathcal{A}$ is work-optimal and communication-efficient, $\beta = \Omega(\lambda\mu)$, $g \leq g(N)$, $G = O(\frac{\beta BD}{\mu\lambda})$, and $L \leq L(N) \cdot \frac{p}{\bar{v}}$.

Proof. We use algorithm *BalancedRouting*, which ensures message size $b \leq \frac{N}{v^2}$, provided that $N \geq v^2 b + \frac{v(v-1)}{2}$. If $b = B$, and $B \geq \frac{v}{2}$, this condition is preserved by $N \geq v^2 B$, $v \geq D$ which also satisfy Theorem 5.3. □

5.5 Summary

The result of Corollary 5.1 can be applied to any algorithm which communicates exclusively via h -relations. The concept of an h -relation is relevant primarily with respect to whether it is an assumption in the analysis of the algorithm in question. Typically, a good algorithm has been shown to be asymptotically optimal when the communication volume to and from each processor is bounded by h in each superstep. Using Lemma 5.1 we can additionally ensure any desired minimum communication block size of b at the cost of at most doubling the number of communication rounds

for problems with sufficient slackness. Here we use the term communication to mean either I/O or conventional message passing.

The main results of this chapter are as follows:

1. Using Theorem 5.3, algorithms which have both BSP* and CGM properties can be converted to EM-BSP*/EM-CGM algorithms.
2. Using Corollary 5.1, CGM algorithms can be converted to BSP* algorithms with $b = \frac{N}{v}$.
3. Using Corollary 5.1, conforming BSP algorithms can be converted to BSP* algorithms with $b = \frac{h_{min}}{v} - \frac{v-1}{2}$, where h_{min} is the minimum value of h used in any communication superstep.
4. Using Corollary 5.1 and Theorem 5.3, CGM algorithms can be converted to EM-BSP*/EM-CGM algorithms with $b = \frac{N}{v}$.

The term “conforming” in item (3) above refers to the need for the bounding concept of an h -relation to be a universal assumption in the analysis of the original BSP algorithm for each of its communication rounds. It is convenient, but not necessary, that the same value of h be used in every round.

Chapter 6

Deterministic Simulation with Lower Slackness

6.1 Motivation

Algorithm *BalancedRouting*, described in Section 5, is a method for delivering the messages generated by the v processors of a parallel algorithm. It does this in twice the original number of communication rounds while maintaining balanced message sizes, i.e. $b = \Theta(\frac{N}{v^2})$, where N is the problem size.

In the external memory context, this technique permits the messages to be written to, and read from, the local disks of each real processor in parallel with minimal overhead, and also allows us to keep the communication balanced between real processors. It has the disadvantage that it requires large slackness, i.e. $N \geq v^2b$. When we substitute $v = \frac{N}{M}$ into $N \geq v^2b$, we obtain the constraint $N \geq \frac{N^2b}{M^2}$, so *BalancedRouting* requires that

$$M^2 \geq Nb \tag{6.1}$$

This is practical at modest cost on contemporary computers even for very large problem sizes, at least into the tera-item range, but a smaller value of M would be desirable from a theoretical point of view. In this chapter, we will present a

more flexible message balancing and distribution algorithm, which we refer to as *LowSlackRouting*. It works for smaller slackness than *BalancedRouting*, but involves an increase in the number of rounds. For constant ϵ , $0 < \epsilon \leq 1$, *LowSlackRouting* requires

$$N \geq v^{1+\epsilon}b \quad (6.2)$$

Substituting $v = \frac{N}{M}$ into (6.2), we obtain the constraint

$$M^{1+\epsilon} \geq N^\epsilon b \quad (6.3)$$

When $\epsilon = 1$, (6.3) is the same as (6.1), but it can be made more attractive by choosing a smaller value for ϵ . For example, with $b = 10^3$ and a huge, perhaps even impossible problem size of $N = 10^{20}$, *BalancedRouting* requires internal memory size $M > 3.2 \times 10^{11}$, which is currently difficult to achieve. However, with $\epsilon = \frac{1}{2}$, *LowSlackRouting* requires only $M \geq 4.7 \times 10^8$ for the same conditions, and $\epsilon = \frac{1}{3}$ gives $M \geq 6.4 \times 10^5$.

6.2 Overview

We first present a parallel algorithm, *LowSlackRouting*, which implements an h -relation for $h = \frac{N}{v}$ while ensuring that the minimum message size is b items. We then adapt the algorithm to an external memory context, with p real processors, $1 \leq p \leq v$, where each real processor has D local disks, and internal memory $O(\frac{kN}{v})$. We show that the algorithms are valid provided that $N \geq v^{1+\epsilon}b$, where ϵ is a constant, $0 < \epsilon \leq 1$. For ease of exposition we will assume that the quantities v^ϵ and $v^{1-r\epsilon}$ are positive integers for $1 \leq r \leq \frac{1}{\epsilon} - 1$. Such assumptions do not materially affect the analysis.

Overview of the Parallel Algorithm *LowSlackRouting*: Algorithm *LowSlackRouting*, presented in Section 6.3, delivers messages generated in a preceding computational superstep to their destinations. Provided that $\frac{N}{v} \geq v^\epsilon b$, it maintains a

Figure 6.1: The division of the $|S|$ processors into v^ϵ groups. S is a set of processors which are participating in the current routing subproblem. Initially, $|S|$ is the set of all v virtual processors. The shaded areas illustrate the components of a fixed bucket on the various processors.

specified minimum message size b even though the amount of data held by each processor is too small for it to send such a message to each of the v processors in a given round. Instead, *LowSlackRouting* constructs v^ϵ groups of $v^{1-r\epsilon}$ processors, in the r^{th} round, for $1 \leq r < \lceil \frac{1}{\epsilon} \rceil - 1$, and messages for processors within a group are combined. *LowSlackRouting* delivers messages to their eventual destination by routing them indirectly, i.e. through a series of intermediate destinations in a corresponding series of communication rounds. The messages originating from a given processor are divided into v^ϵ local buckets, depending on their intended destination. The local buckets of the $v^{1-r\epsilon}$ processors of a group in the r^{th} round are treated collectively as v^ϵ global buckets. Each global bucket corresponds to one of v^ϵ groups of intermediate destinations. See Figure 6.1. By computing a total order on the blocks in each global bucket, we can distribute the blocks evenly among the virtual processors of that group. We will refer to v^ϵ as the *fanout* of each round of routing, as a group of $v^{1-r\epsilon}$ processors in round r divides its data into v^ϵ groups of $v^{1-(r+1)\epsilon}$ processors in round $r+1$. Message

data intended for a particular processor of a group is differentiated and delivered to its correct destination in a later round. For the last round, *BalancedRouting* is used, as by then its slackness constraint is satisfied.

Lemma 6.1 deals with the difficult case where a small number of large messages and a large number of small messages are generated in a round of a client algorithm. Lemma 6.1 allows us to treat the small messages as if they were of size b without asymptotically affecting the CPU, communication, or I/O costs of algorithm *LowSlackRouting*.

Since *LowSlackRouting* performs an $\frac{N}{v}$ -relation in each superstep, with a minimum communication blocksize of b , it can be considered to be both a BSP* and a CGM algorithm. For our purposes, however, the distinction is not material. We are interested primarily in its use as an external memory algorithm.

Overview of External Memory Algorithm *EMLowSlackRouting*: We now adapt algorithm *LowSlackRouting* for use in an external memory context, i.e., on an EM-BSP machine. *EMLowSlackRouting* is essentially a simulation of *EMLowSlackRouting*. Few changes are needed, as *LowSlackRouting* already has coarse, well behaved communication between its subproblems (virtual processors).

On a single processor, single disk machine, a simple approach exists. We know that each group of x processors in each round will eventually receive $x \cdot \frac{N}{v}$ data, and so prior to each round, we allocate $x \cdot \frac{N}{v}$ space on the disk for each group. Messages are then concatenated to the end of the list of message data for their respective destination group. As each virtual processor is simulated, it can read as many blocks of message data for its group as is necessary to ensure that it processes $\frac{N}{v}$ data.

For an EM machine with $p > 1$ processors, however, it is more difficult to ensure that communication is balanced across physical processors and message data is written to the disks D blocks at a time, where a disk block is B items in size. We will show that allocating the v virtual processors of *LowSlackRouting* to the p real processors of *EMLowSlackRouting* in round-robin order, we can achieve balanced communication between real processors on the target machine, provided that $p = O(v^\epsilon)$, where p is the number of real processors. By requiring that the communication block size of

LowSlackRouting be sufficiently large, i.e., $b = \Omega(BD)$, we can fully utilize D parallel disks on each real processor.

In Section 6.3 we present the parallel algorithm *LowSlackRouting*. In Section 6.6 we present the external memory routing algorithm *EMLowSlackRouting*, which is derived from *LowSlackRouting*.

6.3 The Parallel Algorithm *LowSlackRouting*

We are given N problem items and v processors. The routing is performed in $\lceil \frac{1}{\epsilon} \rceil$ rounds of *LowSlackRouting*. The constant ϵ , $0 < \epsilon \leq 1$ determines the fanout v^ϵ of each routing stage, and b is the number of problem items in a message block. $\mathcal{S}$ indicates the set of processors participating in the routing operation. $\mathcal{S}$ is a two-element vector consisting of the first and last labels of the processors participating in the routing operation. Initially, $\mathcal{S} = (0, v - 1)$. We use $|\mathcal{S}|$ to represent the number of processors in the range indicated by $\mathcal{S}$. Initially, $|\mathcal{S}| = v$ and round $r = 1$.

Algorithm 6.1 *LowSlackRouting* ($\mathcal{S}, \epsilon$)

Input: Each of the $|\mathcal{S}|$ processors has $\frac{N}{v} \geq bv^\epsilon$ elements, partitioned into $|\mathcal{S}|$ messages. Each message may be of arbitrary length $\leq \frac{N}{v}$.

Output: The $O(|\mathcal{S}|)$ messages in each processor are delivered to their final destinations. Each processor receives $\frac{N}{v}$ elements.

0. If $|\mathcal{S}| \leq v^\epsilon$, the processors in $\mathcal{S}$ route their messages to their destination by means of algorithm *BalancedRouting*.
1. The processors in $\mathcal{S}$ organize themselves into v^ϵ groups, each of size $|\mathcal{S}|/v^\epsilon$. See Figure 6.1. Each processor divides its messages into v^ϵ buckets. The j^{th} bucket contains those messages destined to the processors in group j . Each message is subdivided into communication blocks of size b .
2. Each processor in $\mathcal{S}$ determines a rank for each of its blocks as follows:
 - (a) Assign unit weight to each locally-held block.
 - (b) Create a vector T of the local partial sums of the weights in each bucket as follows:

$$T \leftarrow \text{the number of blocks in each bucket } j \text{ with size } \ell, \text{ for } 1 \leq \ell \leq b \text{ and } 1 \leq j \leq v^\epsilon$$

- (c) Call *NumberBlocks* (S, T, ϵ), below. This determines the rank of the first full block and any partial block in each bucket.
 - (d) Label all of the locally-held blocks with their rank. The full blocks in each bucket receive consecutive ranks.
3. The processors addresses each block of local bucket j to the $(i \bmod v^{1-r\epsilon})^{\text{th}}$ processor of group j , where i is the rank of the block. Blocks to a common processor are then concatenated to form a single message and sent. Figure 6.2 illustrates the result of this redistribution process.

———— Superstep boundary: $r \leftarrow r + 1$ ————

- 4. Each processor receives at most one message from each of the $v^{1-(r-1)\epsilon}$ processors which were part of its group in the previous round (Step 3 above).
- 5. Let S_j be the range of processors in group j , $1 \leq j \leq v^\epsilon$. Via v^ϵ parallel recursive calls¹, each processor of S_j routes the blocks it received in Step 4 to their destination processor by executing *LowSlackRouting* (S_j, ϵ), for all $1 \leq j \leq v^\epsilon$.

— End of Algorithm —

6.4 The Parallel Algorithm *NumberBlocks*

Algorithm *NumberBlocks* assigns a rank to every block by applying a parallel prefix computation over unit weights. Blocks are numbered independently within each of the v^ϵ global buckets. The ordering is such that blocks of size ℓ are given ranks lower than any block of size $\geq (\ell + 1)$, for $1 \leq \ell \leq b - 1$. This fact is used by Lemma 6.3 to show that each processor receives nearly the same amount of data as any other. Figures 6.3 and 6.4 illustrate the operation of algorithm *NumberBlocks*.

Algorithm 6.2 *NumberBlocks* (S, T, ϵ)

Input: Each of the $|S|$ processors has v^ϵ local buckets, which contain blocks of at most b items. T is a $b \times v^\epsilon$ array containing partial sums for the weights of each block size within each bucket.

Output: The blocks in each global bucket are each given a rank which is unique within that bucket.

- 1. The processors in S organize themselves into $|S|/v^\epsilon$ ordered groups of size v^ϵ . Let i be the index of the current processor, and let $\lfloor \frac{i}{v^\epsilon} \rfloor$ be the group for the current processor.

¹(possibly via a *fork* for each)

Figure 6.2: *The redistribution of blocks for a given bucket.* Step (3) of *LowSlackRouting* redistributes the blocks in a bucket j evenly among the processors of group j .

2. Processor i sends the b sums for its j^{th} local bucket to processor $t(i) \cdot v^\epsilon + j$.
 ————— Superstep boundary —————
3. Each processor receives b partial sums from each of v^ϵ children. Each processor is now associated with the computation of ranks for a single bucket, i.e. processor i is associated with the computation of ranks for bucket $s(i) = i \bmod v^\epsilon$. Processors $0, v^\epsilon, 2v^\epsilon, \dots, (v^{1-\epsilon} - 1) \cdot v^\epsilon$ now contain partial sums for bucket 0, processors $1, 1 + v^\epsilon, 1 + 2v^\epsilon, \dots, 1 + (v^{1-\epsilon} - 1) \cdot v^\epsilon$ now contain partial sums for bucket 1, etc.
4. Each processor computes b local partial sums from the data it received.
 ————— partial sums travel from leaves to root —————
5. for $r = 1$ to $\frac{1}{\epsilon} - 1$ do
 6. If $i = \lfloor \frac{i}{v^{(r-1)\epsilon}} \rfloor \cdot v^{(r-1)\epsilon} + s(i)$ then processor i sends its local partial sums to “parent” processor $\lfloor \frac{i}{v^{r\epsilon}} \rfloor \cdot v^{r\epsilon} + s(i)$.
 ————— Superstep boundary —————
 7. Each processor computes b local partial sums from the data it received.

```

endfor
———— partial sums travel from root to leaves ————
8. for  $r = \frac{1}{\epsilon} - 1$  downto 1 do
    9. Each processor which was a destination in Step 6 computes the rank of each of
        $b$  components received from each of its  $v^\epsilon$  “children” in Step 6.
    10. Each processor sends  $b$  computed ranks to each of its children.
        ————— Superstep boundary —————
endfor
11. Processor  $i$  sends its  $b$  partial sums to each of  $v^\epsilon$  children:  $(i - s(i)) + 0, (i - s(i)) + 1, \dots, (i - s(i)) + v^\epsilon - 1$ .
— End of Algorithm —
    
```

Description of Algorithm NumberBlocks: We assume a local copy of the loop variable r , not the same one that is used in *LowSlackRouting*. The blocks of each bucket are labelled with sequential numbers using algorithm *NumberBlocks*. *NumberBlocks* orders the blocks according to size and requires two passes through a tree of processors, each pass requiring $\lceil \frac{1}{\epsilon} \rceil$ supersteps, for a total of $\lceil \frac{2}{\epsilon} \rceil$ supersteps, where $0 < \epsilon \leq 1$ is a constant. The numbering is computed via parallel prefix sum computations, using a v^ϵ -way tree over the processors for each bucket. Blocks of size ℓ elements are assigned sequence numbers before blocks of size $\ell + 1$, for all $1 \leq \ell \leq (b - 1)$.

In the first superstep, each processor assigns a weight of 1 to each local block, and computes a partial sum for the weights of the locally held blocks of each size ℓ , $1 \leq \ell \leq b$ in each bucket. In Step 2 each processor i first sends its b partial sums for the j^{th} bucket, for $0 \leq j \leq v^\epsilon$, to processor $\lfloor \frac{i}{v^\epsilon} \rfloor + j$. For each bucket, therefore, $\frac{v}{v^\epsilon} = v^{1-\epsilon}$ destination processors are selected, or v destinations over all buckets. In Steps 3 - 7, each such group of $v^{1-\epsilon}$ processors operates independently on a different bucket. In round $\frac{1}{\epsilon}$ the processors $0, \dots, v^\epsilon - 1$ at the root of the respective v^ϵ trees will compute the sum of the weights of the blocks of size ℓ , for $1 \leq \ell \leq b$ in buckets $0, \dots, v^\epsilon - 1$, respectively.

Steps 8 - 11 perform a second pass through the trees of processors. This consists of $\frac{1}{\epsilon}$ additional rounds, during which the results are propagated back down the trees

Figure 6.3: *The tree of processors which numbers the blocks in bucket 0.*

of processors. In each round r , the processors in level r of the tree return the sequence number of the first block contributing to each of the partial sums received from their children in the first pass.

Figures 6.3 and 6.4 illustrate the tree of processors used for calculating the sequence numbers for blocks in buckets 0 and 1 respectively.

6.5 Analysis of *LowSlackRouting*

Lemma 6.1 *The number of blocks (both full and partially filled) in Step (1) of LowSlackRouting is at most $\frac{2N}{vb}$ per processor.*

Proof. Let us assume that partial blocks are padded with dummy items if necessary to ensure that the size of every local bucket on every processor is a multiple of b items. Since there are v^ϵ buckets per processor, each processor adds less than $v^\epsilon \leq \frac{N}{vb}$ blocks.

Therefore, the total amount of data (both real and dummy) at each processor is at most $\frac{2N}{vb}$ blocks. Note that dummy elements are considered only for purposes of the proof and are not actually sent. $\square$

Figure 6.4: The tree for the numbering of blocks in bucket 1.

Lemma 6.2 *Each virtual processor receives at most $\frac{2N}{vb}$ blocks (some perhaps partially full) in Step 4 of LowSlackRouting.*

Proof. In the r^{th} round, $0 \leq r < \lceil \frac{1}{\epsilon} \rceil - 1$, of *LowSlackRouting*, there are less than $v^{1-r\epsilon}$ extra blocks generated per bucket as a result of padding partial blocks. There are a minimum of $\frac{N}{v^{(r+1)\epsilon}b}$ blocks in a bucket in the r^{th} round, so there are at most $\frac{N}{v^{(r+1)\epsilon}b} + v^{1-r\epsilon}$ blocks per bucket. These are spread evenly over $v^{1-(r+1)\epsilon}$ processors of the group, so each processor receives at most

$$\frac{\frac{N}{v^{(r+1)\epsilon}b} + v^{1-r\epsilon}}{v^{1-(r+1)\epsilon}} = \frac{N}{vb} + v^\epsilon \text{ blocks.}$$

Since $\frac{N}{v} \geq v^\epsilon b$, each processor receives at most $\frac{2N}{vb}$ blocks. $\square$

Lemma 6.3 *Although some of the blocks it receives may be only partly full, each virtual processor receives at most $b - 1$ fewer items than any other processor in any round r of LowSlackRouting, $1 \leq r \leq \frac{1}{\epsilon}$.*

Proof. The sequence numbering of the blocks in a bucket by *NumberBlocks* orders all of the blocks of size ℓ before any block of size $\geq (\ell + 1)$, for $1 \leq \ell < (b - 1)$. The

blocks are then allocated in *round-robin* fashion over the ordered set of processors in the group. The allocation of data items among $v^{1-r\epsilon}$ processors in the r^{th} round via $\frac{2N}{v^{r\epsilon}}$ partially filled blocks can be modelled by a $\lceil \frac{2N/v^{r\epsilon}}{v^{1-r\epsilon}} \rceil \times v^{1-r\epsilon}$ matrix as in Figure 6.5. The block sizes form a sequence $\pi(1), \pi(2), \dots, \pi(i), \pi(i+1), \dots, \pi(2i), \pi(i+1), \dots, \pi(\frac{2N}{v^{r\epsilon}})$. The blocks are allocated row by row from left to right to the processors (columns). The maximum imbalance δ in the number of items per processor is the maximum imbalance in column sums of the matrix.

First we consider the case where every processor receives the same number of blocks. In this case, the maximum imbalance in column sums occurs between columns 1 and $i = v^{1-r\epsilon}$, so,

$$\begin{aligned} \delta &\leq \pi(i) - \pi(1) \\ &\quad + \pi(2i) - \pi(i+1) \\ &\quad + \dots \\ &\quad + \pi\left(\frac{2N}{v^{r\epsilon}}\right) - \pi\left(\frac{2N-i}{v^{r\epsilon}} + 1\right) \end{aligned}$$

We know that the maximum cell value is b , and so δ reaches a maximum value when $\pi(1) = 1$, $\pi(\frac{2N}{v^{r\epsilon}}) = b$, and $\pi(v^{1-r\epsilon}) - \pi(v^{1-(r-1)\epsilon} + 1) = 0$ for all r . This gives $\delta = \pi(\frac{2N}{v^{r\epsilon}}) - \pi(1) \leq (b - 1)$.

In the case where a processor is allocated more blocks than another, the difference in number of blocks can be at most 1. Let q be the last processor to receive an extra block. Then q may have received at most $b - 1$ items more than processors $q+1, \dots, v^{1-r\epsilon}$, and by the previous argument, at most $b-1$ items more than processors $1, \dots, q-1$. $\square$

Lemma 6.4 gives an upper bound on the number of rounds required by *NumberBlocks* for each round of *LowSlackRouting*. We generalize the lemma slightly to account for $q \leq v$ participating processors so we can use it in analyzing *EMNumberBlocks* in Lemma 6.8. Note that we do not actually use *NumberBlocks* in the final routing round, as *BalancedRouting* does not require it.

Lemma 6.4 *Let ϵ be a constant, $0 \leq \epsilon \leq 1$, and let $q \leq v$. Given $q/v^{r\epsilon}$ processors participating in the ranking of blocks in v^ϵ buckets (in Step 2(c) of the r^{th} round of*

Relative Processor Number				
1	2	3	...	$v^{1-r\epsilon}$
1	1	2	...	2
2	3	4	...	8
10	41	77		210
263	263	290		850
1010	1830	2578	...	4096
		.		
		.		
		.		
4096	4096	4096	...	4096

Figure 6.5: Imbalance in Item Assignment to Processors

The columns represent processors and the cell values represent the size of the block in that position. The illustration shows a possible scenario for $b = 4096$.

LowSlackRouting, $0 \leq r \leq \lceil \frac{1}{\epsilon} \rceil - 1$, the rank of each block in each bucket can be determined by algorithm *NumberBlocks* in $2 \times \lceil \frac{1}{\epsilon} - r \rceil$ communication rounds.

Proof. For each bucket, *NumberBlocks* forms a v^ϵ -ary tree of processors over the $q/v^{r\epsilon}$ processors. The height of such a tree is $\lceil \log_{v^\epsilon}(q/v^{r\epsilon}) \rceil \leq \lceil (1-r\epsilon)\frac{1}{\epsilon} \rceil$, since $q \leq v$. The trees for each of the v^ϵ buckets are traversed simultaneously. The parallel prefix ranking operation involves traversing the tree of processors in parallel from leaves to root, and then from the root back to the leaves, giving $2 \times \lceil \frac{1}{\epsilon} - r \rceil$ communication rounds. $\square$

Lemma 6.5 *The number of rounds required by LowSlackRouting is $\lceil \frac{1}{\epsilon} \rceil^2 + 2 \cdot \lceil \frac{1}{\epsilon} \rceil - 1$, for ϵ constant, $0 < \epsilon \leq 1$.*

Proof. The ranking of each block in Step 2 of *LowSlackRouting* in round r , $0 \leq r \leq \lceil \frac{1}{\epsilon} \rceil - 1$ can be done in $2 \times \lceil \frac{1}{\epsilon} - r \rceil$ rounds from Lemma 6.4. The data sent by processor i to processor j is sent to a group of $v^{1-\epsilon}$ processors containing processor j in the 0^{th} round by Step 2 of *LowSlackRouting*. In the r^{th} round, the data sent by processor i to

processor j is sent to a group of $v^{1-r\epsilon}$ processors containing processor j . Therefore, after $\lceil \frac{1}{\epsilon} \rceil - 1$ rounds, the data sent by processor i to processor j is sent to a group containing only processor j . The routing step is performed by *LowSlackRouting*, which takes 2 rounds.

Since there are $\lceil \frac{1}{\epsilon} \rceil - 1$ rounds of *LowSlackRouting* which require the use of *NumberBlocks*, the number of rounds overall is

$$\begin{aligned}
 & 2 + \sum_{r=0}^{\lceil \frac{1}{\epsilon} \rceil - 2} \left(1 + 2 \cdot \lceil \frac{1}{\epsilon} - r \rceil \right) \\
 &= 2 + \lceil \frac{1}{\epsilon} \rceil - 1 + 2 \cdot \sum_{r=0}^{\lceil \frac{1}{\epsilon} \rceil - 2} \lceil \frac{1}{\epsilon} - r \rceil \\
 &= 2 + \lceil \frac{1}{\epsilon} \rceil - 1 + \left(\lceil \frac{1}{\epsilon} \rceil \cdot (\lceil \frac{1}{\epsilon} \rceil + 1) \right) - 2 \\
 &= \lceil \frac{1}{\epsilon} \rceil^2 + 2 \cdot \lceil \frac{1}{\epsilon} \rceil - 1
 \end{aligned}$$

□

Lemma 6.6 *Algorithm NumberBlocks uses $O(\frac{N}{v} + L)$ computation time, and $O(g \cdot v^\epsilon b + L)$ communication time.*

Proof. In each round any processor computes at most b sums from at most $v^\epsilon b \leq \frac{N}{v}$ inputs. The maximum number of items received is $v^\epsilon b \leq \frac{N}{v}$. From Lemma 6.5, the number of rounds used is a constant. □

Lemma 6.7 *LowSlackRouting uses $O(\bar{\lambda} \frac{N}{v}) + O(\bar{\lambda} L)$ computation time, and $O(g \cdot \bar{\lambda} v^\epsilon b + \bar{\lambda} L)$ communication time, where $\bar{\lambda} = \lceil \frac{1}{\epsilon} \rceil^2 + 2 \cdot \lceil \frac{1}{\epsilon} \rceil - 1$.*

Proof. From Lemma 6.3 we can conclude that each virtual processor has $O(\frac{N}{v})$ data in each round, for $b = O(\frac{N}{v})$. Exclusive of *NumberBlocks*, therefore, the computation time and communication time of *LowSlackRouting* are both $\frac{N}{v} + L$ per round. From Lemma 6.6, we get the overall computation time, including *NumberBlocks*, to be $O(\bar{\lambda} \frac{N}{v}) + O(\bar{\lambda} L)$ and the communication time to be $O(g \cdot \bar{\lambda} v^\epsilon b + \bar{\lambda} L)$ for $\bar{\lambda} = \lceil \frac{1}{\epsilon} \rceil^2 + 2 \cdot \lceil \frac{1}{\epsilon} \rceil - 1$. □

6.6 The EM Algorithm *EMLowSlackRouting*

In this section we describe algorithm *EMLowSlackRouting*, which is based on *LowSlackRouting*. *EMLowSlackRouting* performs the routing of messages among virtual processors during the simulation of a BSP algorithm on an EM-BSP machine.

In the general case of multiple disks and multiple real processors, a key step is the calculation of a fixed ordering of the communication blocks. This allows us to calculate the destination processor number (both virtual and real), and the disk address to which the block is to be written at its destination. We first describe this general case, and later we examine special cases such as a single disk and single real processor, which permit simpler algorithms.

First, we describe the simulation of *LowSlackRouting* as an external memory algorithm *EMLowSlackRouting*, then we describe an adaptation of *NumberBlocks* as external memory algorithm *EMNumberBlocks*. We do not simulate the parallel algorithm, but instead we use it directly on the real processors of the EM-BSP.

We assume the existence of p real processors on the EM machine. Each real processor i , where $0 \leq i < p$ simulates a fixed set of $\frac{v}{p}$ virtual processors $i, p + i, 2 \cdot p + i, \dots, (\frac{v}{p} - 1) \cdot p + i$. In other words, we assign virtual processors to physical processors in a *round-robin* fashion. As in Chapter 5 we introduce the concept of a *messaging matrix* $\mathcal{M}$. Let $\mathcal{M}$ be a matrix with v columns and $\frac{N}{vb}$ rows, that is stored in a distributed fashion, $\frac{v}{p}$ columns of $\mathcal{M}$ per real processor. Each cell of $\mathcal{M}$ contains a block of b items. Each column of $\mathcal{M}$ will be used to store the message blocks destined for a particular virtual processor. Each real processor will maintain the columns of $\mathcal{M}$ corresponding to the virtual processors which it is responsible for simulating.

We have v^ϵ buckets, and in the first round, $v^{1-\epsilon}$ processors per bucket. We deliver the messages within each bucket to their final destinations by delivering the component blocks of each message to a series of intermediate destinations. The immediate destination virtual processors for a fixed bucket j , for $0 \leq j \leq (v^\epsilon - 1)$ are those with labels $jv^{1-\epsilon}, jv^{1-\epsilon} + 1, \dots, (j + 1)v^{1-\epsilon} - 1$.

We simulate the routing of communication blocks in external memory by executing

algorithm *EMLowSlackRouting* below, on each real processor. *EMLowSlackRouting* is derived from the parallel algorithm *LowSlackRouting*. As before, S is a two-element vector consisting of the first and last labels of the processors participating in the routing operation. Initially, $S = (0, v - 1)$. We use $|S|$ to represent the number of processors in the range indicated by S . Initially, $|S| = v$ and round $r = 1$.

Algorithm 6.3 *EMLowSlackRouting* (S, ϵ)

Input: Each of the $|S|$ virtual processors has $\frac{N}{v} \geq bv^\epsilon$ elements, partitioned into $|S|$ messages. Each message may be of arbitrary length $\leq \frac{N}{v}$. The messages for each virtual processor are stored on the local disks of the real processor which simulates them.

Output: The $O(|S|)$ messages in each processor are delivered to their final destinations. Each processor receives $\frac{N}{v}$ elements.

0. If $|S| \leq v^\epsilon$, the processors in S route their messages to their destination by means of algorithm *BalancedRouting*.
1. The virtual processors in S organize themselves into v^ϵ groups, each of size $|S|/v^\epsilon$. See Figure 6.1. Each virtual processor divides its messages into v^ϵ buckets. The j^{th} bucket contains those messages destined to the processors in group j . Each message is subdivided into communication blocks of size b .
2. Each virtual processor in S determines a rank for each of its blocks as follows:
 - (a) Assign unit weight to each locally-held block.
 - (b) Create a vector T of the local partial sums of the weights in each bucket as follows:

$$T \leftarrow \text{the number of blocks in each bucket } j \text{ with size } \ell, \text{ for } 1 \leq \ell \leq b \text{ and } 1 \leq j \leq v^\epsilon$$
 - (c) Call *EMNumberBlocks* (S, T, ϵ), below. This determines the rank of the first full block and any partial block in each bucket.
 - (d) Label all of the locally-held blocks with their rank. The full blocks in each bucket receive consecutive ranks.
3. Call *EMSend* (S, ϵ), below: Each virtual processor addresses the i^{th} block of local bucket j to the $(i \bmod v^{1-r\epsilon})^{\text{th}}$ processor of group j . Blocks to a common virtual processor are then concatenated to form a single message and sent.

———— Superstep boundary: $r \leftarrow r + 1$ ————
4. Call *EMReceive* (S, ϵ), below: Each virtual processor receives at most one message from each of the $v^{1-(r-1)\epsilon}$ processors which were part of its group in the previous round (Step 3 above).

5. Let S_j be the range of virtual processors in group j , $1 \leq j \leq v^\epsilon$. Each virtual processor of S_j routes the blocks it received in Step 4 to their destination processor by executing $\text{EMLowSlackRouting}(S_j, \epsilon)$, for all $1 \leq j \leq v^\epsilon$.

— End of Algorithm —

Algorithm 6.4 $\text{EMSend}(S, \epsilon)$

Input: Each communication block held by the current virtual processor is associated with one of v^ϵ buckets, and labelled with a sequence number i , as calculated by NumberBlocks , that is unique within that bucket.

Output: Each communication block held by the current virtual processor is sent to the real processor which is responsible for simulating its immediate destination virtual processor. Each block is labelled with the disk track to which it should be written when it reaches its destination.

1. Let j represent the bucket number of a block and let i be its sequence number. Calculate for each block a column number and row number in the messaging matrix as follows:
 - a. Column number $C = jv^{1-\epsilon} + i \bmod v^{1-\epsilon}$. (Note: C is also the number of the virtual processor which is the immediate destination for the current block.)
 - b. Row number $R = \lfloor \frac{i}{v^{1-\epsilon}} \rfloor$. (Note: R can also be thought of as a sequence number for the current block within virtual processor C .)
2. Send block i to its destination real processor $C \bmod p$, for all i .

— End of Algorithm —

Algorithm 6.5 $\text{EMReceive}(S, \epsilon)$

Input: The current virtual processor has sent its data to the router and is waiting for data addressed to it to arrive. Each arriving block is labelled with the local disk track to which it should be written. Let i be the sequence number of a block, and C is the number of the virtual processor to which it is addressed.

Output: The blocks of data addressed to the current virtual processor in the current superstep have been written to the local disks (of the appropriate real processor).

1. While messages remain to be received
 - a. Receive next message from router
 - b. Write each block of the received message to parallel disk track $\frac{2N}{vb}(C \bmod \frac{v}{p}) + \lfloor \frac{i}{v^{1-\epsilon}} \rfloor$.

— End of Algorithm —

Details of Algorithms EMSend and EMReceive : For ease of exposition, we assume that p divides v . We also assume that $b = \Omega(BD)$, e.g. $b \geq BD$, where b is the communication block size, D is the number of local disks on each real processor, and B is the disk block size. We can therefore write each communication block on one or more tracks of the parallel disks of a real processor. Each communication block is labelled with its messaging matrix indices C , and R , where “column number” $C = jv^{1-\epsilon} + (i \bmod v^{1-\epsilon})$, and “row number” $R = \lfloor \frac{i}{v^{1-\epsilon}} \rfloor$. The blocks for a fixed virtual destination processor are stored on the real destination processor in a fixed band of tracks on the local disks. The band number is calculated as $C \bmod \frac{v}{p}$. Within a band, R determines the relative track number on which the message is stored. The message block with sequence number i in bucket j is sent to real processor $C \bmod p$ and written to parallel disk track $\frac{2N}{vb}(C \bmod \frac{v}{p}) + R$ on its local disks². Messages arriving at a real processor can therefore be written to its disks on the correct parallel track independent of other the arrival times of other messages. Figure 6.6 shows the assignment of messages on the local disks of a real destination processor.

6.7 The EM Algorithm *EMNumberBlocks*

It remains to show how *NumberBlocks* can be executed as an external memory algorithm. *NumberBlocks* involves two phases, an “upward” phase, where data propagates from leaves to root, and a “downward” phase, where data propagates from root to leaves. Each processor which is active in a round of an upward phase sends a block of b items to a parent processor which is specific to a given bucket. There are v^ϵ such processors sending to each parent, and so communication is fully blocked, and $v^\epsilon b$ data is received by a parent in any round.

Algorithm *NumberBlocks* forms v^ϵ v^ϵ -ary trees of processors, each one computing an ordering of the blocks for a single bucket. In the parallel external memory version

²The maximum number of blocks per virtual processor is $\frac{2N}{vb}$ from Lemma 6.1.

Figure 6.6: *The assignment of messages to local disks on a real destination processor.* A block with indices (C, R) in the messaging matrix $\mathcal{M}$ is assigned to real processor $\lfloor \frac{C}{v/p} \rfloor$, and a band of its local parallel disk tracks which corresponds to the block's destination virtual processor. The parallel disk track within that band is R .

EMNumberBlocks, the p processors compute the same v^ϵ prefix sums by forming a v^ϵ -ary tree over p processors for each bucket. We therefore do some local processing on each real processor to reduce the number of leaves in each tree from v to p , then apply *NumberBlocks* to the reduced problem. Finally, each real processor does some local processing to deliver the prefix sum information for each bucket to each of its v processors.

Each real processor maintains $\frac{v}{p}$ columns of a $v^\epsilon \times v$ matrix $\mathcal{M}_{NB}$. Each cell of $\mathcal{M}_{NB}$ is large enough to store a message of size b . Each virtual processor i can independently calculate its own rank i' within the collection of processors sending to a given destination j . Communication from virtual processor i to virtual processor j is implemented by the message from i being written to, and subsequently being read from $\mathcal{M}_{NB}(i', j)$.

Algorithm 6.6 *EMNumberBlocks* ($\mathcal{S}, T, \epsilon$)

Input: Each of the $|\mathcal{S}|$ virtual processors has v^ϵ local buckets, which contain blocks of size $\leq b$ items. T is a $b \times v^\epsilon$ array containing partial sums for the weights of each block size within each bucket. Each of p real processors has v/p virtual processors which it simulates. Each virtual processor is prepared to execute the algorithm *NumberBlocks*.

Output: The blocks in each global bucket of the virtual processors are each given a rank which is unique within that bucket.

1. Each real processor sums the values of T held by each of each virtual processors it simulates. It calculates b sums for each bucket, one sum for each blocksize $1, \dots, b$.
2. The p real processors execute the parallel algorithm *NumberBlocks* ($\mathcal{S}, T, \epsilon$), for $\mathcal{S} = (0, p-1)$ (all real processors).
3. Each real processor scans the virtual processors it simulates and delivers its portion of the prefix sum for each of its buckets.

— End of Algorithm —

Details of EMNumberBlocks: We arrange that the virtual processors are suspended as in a barrier synchronization operation, when they call *NumberBlocks*. In the worst case, steps (1) and (3) of *EMNumberBlocks* can be implemented by a scan over the contexts of the virtual processors held by each real processor.³ Step (2) now requires only a parallel algorithm, as no I/O is required. The virtual processors simply return from their calls to *NumberBlocks* with the required results, as calculated by *EMNumberBlocks*.

6.8 Analysis of *EMLowSlackRouting*

Lemma 6.8 *Algorithm EMNumberBlocks uses $O(\frac{N}{p} + L)$ computation time, $O(g \frac{N}{p} + L)$ communication time, and $O(G \frac{N}{pBD} + L)$ I/O time.*

³In practice, no extra I/O operations may be required for steps (1) and (3). The size of each bucket is contained in the variable T in the stack frame of each virtual processor, and so step (1) does not require extra I/O operations to accomplish. Similarly, when the results of the prefix sum are to be delivered in step (3), the real processors simply deliver the results to the variable T , already in the current stack frame of each process, prior to relinquishing control to each virtual processor.

Proof. In Step (1) of *EMNumberBlocks*, each real processor calculates b sums from $\frac{v}{p} \cdot v^\epsilon b = O(\frac{N}{p})$ items. In each round of *NumberBlocks* (Step (2) of *EMNumberBlocks*) any real processor computes at most $\frac{vb}{p}$ sums from at most $\frac{v}{p} v^\epsilon b \leq \frac{v}{p} \cdot \frac{N}{v}$ inputs. The maximum number of items received by a real processor is $\frac{v}{p} v^\epsilon b \leq \frac{v}{p} \cdot \frac{N}{v}$. From Lemma 6.4, the number of rounds used over $p \leq v$ processors by *NumberBlocks* is at most $2 \times \lceil \frac{1}{\epsilon} - r \rceil$, where $1 \leq r \leq \lceil \frac{1}{\epsilon} \rceil - 1$ is the recursion depth in *EMLowSlackRouting*. $\square$

The last round of ϵ -routing consists of $v^{1-\epsilon}$ subproblems, consisting of v^ϵ processors in each case. The data for each of these processors is spread over v^ϵ processors and is to be delivered to its final destination in this round.

Since the size of the groups in the final round is only $v^{1-\frac{1}{\epsilon}\epsilon} = 1$, we no longer can rely on successive blocks being sent to successive physical processors, and so the *round-robin* allocation approach can result in large imbalance between physical processors in the data they receive (the largest one can receive $O(p \cdot \frac{N}{vb})$ items). We do, however, have the necessary conditions to apply algorithm *BalancedRouting* in the final round instead, as Lemma 6.9 shows.

Lemma 6.9 *Let v' be the number of processor involved and let N' be the size of each routing subproblem, in the final round of *EMLowSlackRouting*. Then $N' = (v')^2 b$ and so algorithm *BalancedRouting* can be used, for $b \geq \frac{v'-1}{2}$.*

Proof. Since for each subproblem there are v^ϵ processors, each with at least $v^\epsilon b$ data items, each subproblem in the final round contains $N' \geq v^{2\epsilon} b$ data. Therefore, we have a problem of size N' with $v' = v^\epsilon$ processors and $N' = (v')^2 b$. From Theorem 5.4, therefore, we have the necessary conditions for algorithm *BalancedRouting* when $b \geq \frac{v'-1}{2}$. $\square$

We now turn to the issue of whether the communication between real processors in algorithm *EMLowSlackRouting* is balanced in rounds $1, \dots, \lceil \frac{1}{\epsilon} \rceil - 1$. If communication is unbalanced, not only may an arbitrary processor do more than its share of the work, but it may be asked to receive more data than can fit into its internal memory in a single round.

In general, the kp virtual processors may be participants of say, z groups in this, the r^{th} round.

First we consider the case when $k = 1$. In general, the first few real processors contain some of the virtual processors of group i , the last few processors contain some of the virtual processors of group $i + z - 1$, and the processors in between contain all of the virtual processors of groups $i + 1$ to $i + z - 2$.

We call the groups $i + 1$ to $i + z - 2$ whose virtual processors are completely contained within the p real processors *full* groups. A full group produces an imbalance in communication of at most one block between its real processors. The partial groups (defined in next paragraph) may also have an effect on these real processors, however.

We refer to the (at most) two groups, which are only partly contained within the p real processors, as *partial* groups. Within each bucket, every block has a label between 0 and $\frac{N}{v^\epsilon b}$, but only part of this range will be represented in such a group. We refer to these buckets as *partial* buckets.

Lemmas 6.10 and 6.11 show that although we simulate only p virtual processors at a time in each superstep of *EMLowSlackRouting*, the communication and memory usage of the real processors remains balanced when $p < v^\epsilon$.

Lemma 6.10 *Let $p \leq v^\epsilon$, and let j be an arbitrary bucket in a fixed round of *EMLowSlackRouting*. In Step 4 of *EMLowSlackRouting*, each real processor receives at most one more block of bucket j than any other real processor.*

Proof. There are $v^{1-r\epsilon}$ processors in any group during the r^{th} round of *EMLowSlackRouting*, for $1 \leq r \leq \frac{1}{\epsilon} - 1$, and during these rounds, $v^{1-r\epsilon} \geq v^\epsilon$. We therefore have at least as many virtual processor in a group as there are real processors. Since virtual processors are allocated to real processors in round-robin order, any real processor hosts at most one more virtual processor of group j than any other.

Since the blocks in bucket j are allocated to the virtual processors of group j in round-robin order, each real processor receives at most one more block of bucket j than any other real processor in Step 4 of *EMLowSlackRouting*. $\square$

Lemma 6.11 *Let $p < v^\epsilon$, and let $k = 1$. No real processor receives more than $\frac{3N}{vb}$ blocks in Step 4 of *EMLowSlackRouting* in rounds $1, \dots, \lceil \frac{1}{\epsilon} \rceil - 1$.*

Proof. Consider the memory of p consecutive virtual processors in the r^{th} round, for $1 \leq r \leq \lceil \frac{1}{\epsilon} \rceil - 1$, where only a single virtual processor is simulated concurrently in each of p real processors. Their local memories are divided into v^ϵ buckets.

We send blocks with consecutive numbers to different virtual processors with consecutive labels. Because of the *round-robin* allocation of virtual processors to physical processors, blocks with consecutive numbers are guaranteed to be sent to different real processors as well, in rounds $1 \leq r \leq \lceil \frac{1}{\epsilon} \rceil - 1$.⁴ The worrisome issue, however, is whether one real processor receives significantly more data than any other. From Lemma 6.10, this cannot be as a result of distributing a single bucket. We now examine whether it can still happen due to the fact that we distribute the v^ϵ different buckets independently.

Since $p < v^\epsilon$, the p virtual processors may be participants of at most two groups. Let us assume that p' virtual processors participate in the first of these groups, p'' participate in the second, and $p' + p'' = p$. Let us concentrate on the first of the two groups. In this case, the data from p' virtual processors is sent to $p \geq p'$ real processors.

The difference in the number of blocks received by a real processor for a single bucket of a single group is at most one, no matter how much data is in that bucket (i.e. partial or not).

The worst case of imbalance occurs when most of the buckets (“thin” buckets) contain only $w = o(p)$ blocks, which get allocated to the same set of w real processors, $P_0, \dots, P_{w-1}$ for every thin bucket.⁵ Figure 6.7 illustrates this case for $w = 1$. Since we have an $\frac{N}{v}$ -relation, each virtual processor distributes $O(\frac{N}{vb})$ blocks in total. Each of the w processors can receive at most $v^\epsilon - 1$ blocks from thin buckets, since we have only v^ϵ buckets in total.

Since every virtual processor must send v^ϵ blocks in total, and there are p of them participating, we will have $p \cdot v^\epsilon - w(v^\epsilon - 1)$ blocks in the last partial bucket (“fat” bucket).

⁴In round $\frac{1}{\epsilon}$, however, since the groups are only $v^0 = 1$ processor in size, successive blocks would not necessarily be sent to successive real processors if we used the same algorithm.

⁵If, say, two blocks are to be routed to P_0 in a bucket, this would imply that all of the other real processors receive at least one block as well.

Figure 6.7: *Imbalance in Data Received by the Real Processors* The first $v^\epsilon - 1$ buckets contain only one block, which coincidentally get routed to the same real processor. The last bucket must then contain $p \cdot v^\epsilon - (v^\epsilon - 1)$ blocks, which get evenly distributed over the p processors.

We know, however, that the data from the fat bucket will be distributed evenly among those real processors which contain the virtual processors of this group. Therefore, $P_0, \dots, P_{w-1}$ may each receive another $v^\epsilon - \frac{w(v^\epsilon - 1)}{p}$ blocks this way, for a total of $v^\epsilon - 1 + v^\epsilon - \frac{w(v^\epsilon - 1)}{p}$, or $2v^\epsilon - (1 + \frac{w(v^\epsilon - 1)}{p})$, and this is no more than $\frac{2N}{vb}$.

This process can happen at both ends of the range of p virtual processors. If we assume that some of $P_0, \dots, P_{w-1}$ can receive both sets of "extra" blocks, we obtain the bound of $\frac{3N}{vb}$. $\square$

We now consider $k \geq 1$ processors simulated concurrently by each real processor.

Lemma 6.12 *Let $k \geq 1$. No real processor receives more than $\frac{3kN}{vb}$ blocks in Step 4 of EMLowSlackRouting in rounds $1, \dots, \lceil \frac{1}{\epsilon} \rceil - 1$.*

Proof. Each additional p virtual processors added to the memories of the real processors in a simulation round can do no worse than independently produce the

scenario of Figure 6.7 (Lemmas 6.11. Since for $k = 1$ the maximum communication received by a processor was $\frac{3N}{vb}$ blocks, the worst case is $\frac{3kN}{vb}$ blocks for general k . $\square$

Lemma 6.13 *For $p = O(v^\epsilon)$, $EMLowSlackRouting$ uses $O(\bar{\lambda}\frac{N}{p}) + O(\bar{\lambda}L)$ computation time, $O(g \cdot \bar{\lambda}\frac{N}{p} + \bar{\lambda}L)$ communication time, and $O(G\frac{N}{pBD} + \bar{\lambda}L)$ I/O time, where $\bar{\lambda} = \lceil \frac{1}{\epsilon} \rceil^2 + 2 \cdot \lceil \frac{1}{\epsilon} \rceil - 1$ for constant ϵ , $0 < \epsilon \leq 1$.*

Proof. From Lemma 6.3 we can conclude that each virtual processor has $O(\frac{N}{v})$ data in each round, for $b = O(\frac{N}{v})$. Exclusive of $EMNumberBlocks$, therefore, the computation time and communication time of $EMLowSlackRouting$ are both $\frac{N}{v} + L$ per round. From Lemma 6.6, we get the overall computation time, including $EMNumberBlocks$, to be $O(\bar{\lambda}\frac{N}{v}) + O(L\bar{\lambda})$ and the communication time to be $O(g \cdot \bar{\lambda}v^\epsilon b + \bar{\lambda}L)$, for $\bar{\lambda} = \lceil \frac{1}{\epsilon} \rceil^2 + 2 \cdot \lceil \frac{1}{\epsilon} \rceil - 1$. $\square$

6.9 Single Processor Target Machine

In each round of $EMLowSlackRouting$, the EM processor first allocates v^ϵ buckets, each of size $\lceil \frac{b}{BD} \rceil v^{1-r^\epsilon}$ disk tracks, where r is the current recursion depth in algorithm $LowSlackRouting$ (the initial call to $LowSlackRouting$ corresponds to $r = 1$).

The execution of each virtual processor as it executes algorithm $LowSlackRouting$ is suspended after Step 3, which is a communication superstep. The simulation of Step 3 is performed by simply writing the communication blocks intended for bucket j to the end of a list of blocks stored in the area reserved for bucket j on the disk. When every virtual processor has completed recursion depth $r = r'$ of $LowSlackRouting$, the first processor's execution at level $r = r' + 1$ begins. For purposes of our EM routing algorithm we need only ensure that each processor of a given group receives the same number of blocks of the messages sent to that group in the previous superstep. As it begins execution of a superstep, each virtual processor takes the next $\frac{N}{v} \geq \lceil \frac{b}{BD} \rceil v^\epsilon$ undelivered disk tracks for its group.

Both reading and writing of message data is done BD items at a time. As we know beforehand the total number of blocks destined for each group, the sequential

simulation for this case need not include the ranking step (*NumberBlocks*) required by the parallel algorithm *LowSlackRouting*, and the final routing step need not be done by *BalancedRouting* as in the multiple real processor case.

6.10 Multiple Processor Target Machine

We now describe a simulation technique for the multiple real processor case.

Each real processor i , $0 \leq i \leq p - 1$, executes algorithm *ParCompoundSuperstep-6.7* in parallel. For ease of exposition, we assume that pk divides v . The v virtual processors are assigned to the p real processors in *round-robin* order, i.e. real processor i hosts virtual processors $i, i + p, i + 2p, \dots, i + v - p$.

Algorithm 6.7 *ParCompoundSuperstep-6.7*

Objective: Simulation of a compound superstep of a v -processor CGM on a p -processor EM-CGM.

Input: The message and context blocks of the virtual processors are divided among the real processors and their local disks. Each real processor i , $0 \leq i \leq (p - 1)$ holds $O(\frac{N}{pB})$ blocks of messages and $\frac{v\mu}{pB}$ blocks of context, and each local disk contains $O(\frac{N}{pBD})$ blocks of messages and $O(\frac{v\mu}{pBD})$ blocks of context.

Output: The changed contexts and generated messages distributed as required for the next compound superstep.

For $j = 0$ to $\frac{v}{pk} - 1$ do

- (a) Read the contexts of virtual processors $jkp + i, (jk + 1)p + i, \dots, ((j + 1)k - 1)p + i$ from the local disks.
- (b) Read any message blocks addressed to virtual processors $jkp + i, (jk + 1)p + i, \dots, ((j + 1)k - 1)p + i$ from the local disks.
- (c) Simulate the computation supersteps of virtual processors $jkp + i, (jk + 1)p + i, \dots, ((j + 1)k - 1)p + i$, collecting all generated messages in the local internal memory.
- (d) Send all generated messages to the required (real) destination processors. Upon arrival, the messages are arranged within the internal memory of the real destination processor and then written to its disks.
- (e) Write the contexts for virtual processors $jkp + i, (jk + 1)p + i, \dots, ((j + 1)k - 1)p + i$ back to the local disks.

— End of Algorithm —

Details of algorithm ParCompoundSuperstep-6.7: Steps (a), (b), (c), and (e) are identical to the corresponding steps in the simulation of Chapter 5 (see algorithm ParCompoundSuperstep-5.3). Step (d) is accomplished by algorithm *EMLowSlackRouting* for $\frac{1}{\epsilon} - 1$ routing steps, followed by algorithm *BalancedRouting* when the routing subproblems are reduced to delivering $\frac{N}{v^\epsilon}$ messages among v^ϵ processors. We choose $b \geq BD$, and so for each interim destination during the routing steps of *EMLowSlackRouting*, every message block can be written to the D local disks in parallel. In the final routing step, involving algorithm *BalancedRouting*, a distributed messaging matrix is created as in algorithm ParCompoundSuperstep-5.3, but for each subproblem of v^ϵ processors.

We can now restate Theorem 5.3 with a more general slackness constraint for the case $p = O(v^\epsilon)$.

Theorem 6.1 *A v processor CGM algorithm $\mathcal{A}$ with λ supersteps, computation time $\beta + \lambda L$, communication time $g\lambda\frac{N}{v} + \lambda L$, and local memory size μ can be simulated as a p -processor EM-CGM algorithm $\mathcal{A}'$ with computation time $\frac{v}{p}\beta + \frac{v}{p}O(\lambda\mu) + \frac{v}{p}O(\lambda)L$, communication time $g \cdot O(\lambda\frac{N}{p}) + \frac{v}{p}O(\lambda)L$, and I/O time $G \cdot \frac{v}{p}O(\lambda\frac{\mu}{BD}) + \frac{v}{p}O(\lambda)L$ for $p = O(v^\epsilon)$, $M \geq k\mu + BD$, and $N = \Omega(\bar{v}^{1+\epsilon}BD)$ for arbitrary integer $1 \leq k \leq \frac{v}{p}$, and $\bar{v} = \frac{v}{k}$.*

Let $g(N)$, $L(N)$, and $v(N)$ be increasing functions of N . If $\mathcal{A}$ is c -optimal on the CGM for $g \leq g(N)$, $L \leq L(N)$ and $v \leq v(N)$, then $\mathcal{A}'$ is a c -optimal EM-CGM algorithm for $\beta = \omega(\lambda\mu)$, $g \leq g(N)$, $G = o(\frac{\beta BD}{\mu\lambda})$ and $L \leq L(N) \cdot \frac{v}{\bar{v}}$. $\mathcal{A}'$ is work-optimal, communication-efficient, and I/O-efficient if $\mathcal{A}$ is work-optimal and communication-efficient, $\beta = \Omega(\lambda\mu)$, $g \leq g(N)$, $G = O(\frac{\beta BD}{\mu\lambda})$, and $L \leq L(N) \cdot \frac{v}{\bar{v}}$.

Proof. The changes between algorithms ParCompoundSuperstep-5.3 and ParCompoundSuperstep-6.7 are restricted to the first $\lceil \frac{1}{\epsilon} \rceil - 1$ routing steps, and the constraint that $b = BD$. *EMLowSlackRouting* takes a constant number of supersteps for constant $0 \leq \epsilon \leq 1$. Therefore, asymptotically, the computation and communication time, as well as the total number of supersteps do not change from Theorem 5.4.

However, the slackness requirement now becomes $N = \Omega(\bar{v}^{1+\epsilon}BD)$, absorbing a previous requirement that $N = \Omega(\bar{v}BD)$, and therefore eliminating the constraint $v = \Omega(D)$. $\square$

Chapter 7

Problems of Geometrically Changing Size

7.1 Introduction

The techniques described in Sections 4 and 5 may require asymptotically more I/O than is necessary for certain problems, specifically those whose active set of data changes in size by a constant factor in each round. Examples include certain algorithms for parallel prefix computations and list ranking, where during a series of rounds, the bridging of nodes eliminates a constant fraction of the input from the computation in each round (see [33] for example). Our previous approach is inefficient for this sort of problem, because over $O(\log \mu)$ rounds of geometrically decreasing problem size, the computation time is perhaps only $O(\mu)$ (as in the list ranking example), but we would swap a total volume of $O(\mu \log \mu)$ context. In this section we discuss modifications to the simulation which allow it to take advantage of the knowledge that the problem size is changing in this way. The modifications we will discuss are relevant to the asymptotic complexity of the simulation of a client algorithm with a non-constant number of rounds of such geometrically decreasing problem sizes, but may also represent a useful saving of elapsed time in practice.

The main idea of this improvement to the simulation is to swap only the active portion of the context of a virtual processor, and in order to do this, we require that

the client algorithm keep track of which portions of its data space are active and which are not. In practice, we anticipate that the client algorithm will also need to defragment its data space periodically (say every round or after a constant number of rounds) to make the active and inactive portions individually contiguous. We will also require the client algorithm to provide the sizes and locations of these portions of its data space to the simulation. Aside from these general implementation remarks, however, we will restrict our attention here to the theoretical issues arising from this modification.

We consider the messaging and the context swapping, but do not specifically consider the executable program code itself. It is not relevant to our asymptotic analysis because it can be considered to be of constant size. Its effects on the analysis are captured in the expressions for the context.

7.2 Reducing I/O Overheads

We use the term *phase* to mean a series of consecutive rounds of a BSP-like algorithm. A *decreasing phase* will refer to a phase in which the active context size is decreasing, by a constant factor $f > 1$ in each round, i.e. the context size in round $(i + 1)$ is $\frac{\mu_i}{f}$, where μ_i is the context size in round i . An *increasing phase* will refer to a phase in which the active context size is increasing by a constant factor $f > 1$ in each round, i.e. the context size in round i is μ_i , and the context size in round $(i + 1)$ is $f \cdot \mu_i$. A *monotone phase* is a phase which is strictly increasing or decreasing.

Let μ_{MIN} be the minimum size of active context that is swapped during a monotone phase. Clearly, in order to ensure that the I/O remains blocked and parallel to all D disks, we require that $k \cdot \mu_{MIN} \geq BD$, where k is the number of virtual processors simulated in the memory of a real processor concurrently. While the actual active context size may become smaller, we will require that the simulation track the actual active context size only as small as μ_{MIN} . Lemma 7.1 shows that if the simulation can track the context for $\log_f \ell$ rounds the total volume of contexts swapped into internal memory during the phase is only $O(\mu_{MAX})$.

Lemma 7.1 *Let μ_{MAX} be the context size at the beginning of a decreasing phase ϕ with ℓ rounds. For constant $f > 1$ and $\mu_{MIN} = \frac{\mu_{MAX}}{\ell}$, the sum μ_{Total} of the sizes of the active contexts which must be swapped during ϕ is $O(\mu_{MAX})$, and this can be achieved in a fully blocked fashion on a single processor EM-BSP with D disks provided that $\ell \leq \frac{k\mu_{MAX}}{BD}$.*

Proof. μ_{Total} is defined as follows:

$$\mu_{Total} = \mu_{MAX} + \frac{\mu_{MAX}}{f} + \frac{\mu_{MAX}}{f^2} + \dots + \frac{\mu_{MAX}}{f^{\ell-1}} \quad (7.1)$$

A useful bound on μ_{Total} for our purposes is

$$\mu_{Total} \leq \mu_{MAX} + \frac{\mu_{MAX}}{f} + \frac{\mu_{MAX}}{f^2} + \dots + \frac{\mu_{MAX}}{f^{\log_f(\mu_{MAX}/\mu_{MIN})}} + \sum_{i=1+\log_f \frac{\mu_{MAX}}{\mu_{MIN}}}^{\ell-1} \mu_{MIN} \quad (7.2)$$

Now, if we choose $\mu_{MIN} = \frac{\mu_{MAX}}{\ell}$, this becomes

$$\mu_{Total} = \mu_{MAX} + \frac{\mu_{MAX}}{f} + \frac{\mu_{MAX}}{f^2} + \dots + \frac{\mu_{MAX}}{f^{\log_f(\mu_{MAX}/\mu_{MIN})}} + \sum_{i=1+\log_f \frac{\mu_{MAX}}{\mu_{MIN}}}^{\ell-1} \frac{\mu_{MAX}}{\ell} \quad (7.3)$$

Now, using the identity

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}, \text{ where } x < 1,$$

we obtain, for $x = \frac{1}{f}$,

$$\begin{aligned} \mu_{Total} &\leq \frac{f}{f-1} \cdot \mu_{MAX} + \ell \cdot \frac{\mu_{MAX}}{\ell} \\ &= O(\mu_{MAX}) \end{aligned} \quad (7.4)$$

So for $\mu_{MIN} = \frac{\mu_{MAX}}{f^{\log_f \ell}} = \frac{\mu_{MAX}}{\ell}$, we have $\mu_{Total} = O(\mu_{MAX})$.

In order for the I/O to be blocked and in parallel to all D disks, we also require that at the contexts of k processors fill a parallel disk track, in other words,

$$k\mu_{MIN} \geq BD$$

Since we chose $\mu_{MIN} = \frac{\mu_{MAX}}{\ell}$, we get

$$\begin{aligned} \frac{k\mu_{MAX}}{\ell} &\geq BD \\ \iff \ell &\leq \frac{k\mu_{MAX}}{BD} \end{aligned} \tag{7.5}$$

□

The case of an increasing phase is symmetric. During the last(first) $\ell - \log_f \ell - 1$ rounds of a decreasing(increasing) phase, the actual context size may be less than μ_{MAX}/ℓ , but for $\ell \leq \frac{k\mu_{MAX}}{BD}$, we can transfer μ_{MAX}/ℓ data in each of these rounds without exceeding $O(\mu_{MAX})$ data transfer overall for the phase, or violating our blocking requirements for the disks.

Lemma 7.2 *A monotone phase ϕ with ℓ rounds, computation time $\alpha + \ell L$, and parameter f can be simulated on a single processor EM-CGM machine with $O(k\mu)$ internal memory in $O(v\alpha + \frac{v\mu_{MAX}}{B})$ computation operations and $O(\frac{v\mu_{MAX}}{BD})$ I/O operations provided that $\ell \leq \frac{k\mu_{MAX}}{BD}$, $N \geq \ell \bar{v}^2 B$, $B \geq \frac{v-1}{2}$, $D \leq v$, for integer constant $k > 0$, $\bar{v} = \frac{v}{k}$*

Proof. Algorithm *SeqCompoundSuperstep* can be used for the simulation with changes discussed below. We discuss the case of a decreasing phase. An increasing phase is symmetric to a decreasing phase, and so the arguments remain the same.

I/O Due to Contexts: First, we consider how the simulation should deal with the contexts. We can handle the swapping of contexts by allocating a fixed area on the D disks exactly as in the fixed context size case, but with the space for each group of k virtual processors divided into two parts: i) the active portion of the contexts, and ii) the inactive portion of the contexts. The inactive portion is not swapped into the real memory of the simulating processor until needed in a future round.

We first consider the case when the active context size is decreasing during a phase. In each round, the active portion decreases in size by the factor f . At the end of a round, the simulation concatenates the newly deactivated part of the context to the inactive portion on the disks, and writes the active portion into the remainder of

the disk space reserved for the context. By Lemma 7.1, we need only $O(\frac{\mu_{MAX}}{BD})$ I/O operations per virtual processor for the swapping of contexts during such a phase.

I/O Due to Messages: We assume that the total message data volume in each round is also limited by the volume of the active contexts, and that the message volume during a phase varies by the same factor f between rounds as does the context size.

As in Section 5, we use a fixed size messaging matrix throughout the simulation. As the message volume drops during a decreasing phase, less of the space allocated to the messages for each virtual processor will be required, but no change in the address calculation is required.

It remains to ensure that the message volume is sufficient for algorithm BalancedRouting (page 72) to work, and that I/O operations performed by algorithms SeqCompoundSuperstep (page 78) and ParCompoundSuperstep (page 83) to simulate this communication remain blocked and in parallel to the D disks.

Let N^{MIN}, N^{MAX} represent the minimum and maximum message volumes, respectively, of a round during phase ϕ . We apply the same analysis to the maximum message volume in a round as we used to derive equation (7.4). This gives us a minimum data volume of $N^{MIN} = N^{MAX}/\ell$ over a monotone phase of ℓ rounds.

First, we examine the constraints required by algorithm BalancedRouting. Lemma 5.2 requires a lower bound on the problem size N , i.e. $N \geq v^2 B$, $B \geq \frac{v-1}{2}$. If we replace N by the minimum data size $N^{MIN} = N/\ell$, this constraint becomes $N \geq \ell v^2 B$, $B \geq \frac{v-1}{2}$. When k virtual processors are simulated concurrently in each real processor, there are only $\bar{v} = \frac{v}{k}$ distinct destinations for the messages, and so this constraint becomes $N \geq \ell \bar{v}^2 B$, $B \geq \frac{\bar{v}-1}{2}$.

For algorithms SeqCompoundSuperstep and ParCompoundSuperstep, we have the requirement $N = \Omega(\bar{v}BD)$, which ensures that the messages sent or received by a group of k virtual processors in one round was enough to justify doing a parallel I/O to the D disks. In our current scenario, however, we must adapt this condition to the minimum data size N/ℓ . If we replace N by the minimum data size $N^{MIN} = N/\ell$, this constraint becomes $N = \Omega(\ell \bar{v}BD)$. This constraint is absorbed by the constraints $N \geq \ell \bar{v}^2 B$, $B \geq \frac{\bar{v}-1}{2}$.

Computation Time: As v virtual processors are simulated on a single real processor, the total time for Step 1(c) is $v\alpha$. I/O contributes computational overhead proportional to the number of blocks input or output, or $O(\frac{v\mu_{MAX}}{B})$.¹ $\square$

Lemma 7.3 *A monotone phase ϕ of a CGM with ℓ rounds, computation time $\alpha + \ell L$, communication time $g\beta + \ell L$, and parameter f can be simulated on a p -processor EM-CGM machine in $O(\frac{v}{p}\alpha + \frac{v\mu_{MAX}}{pB})$ parallel computation operations, $O(\frac{v}{pk}\beta)$ communication operations, $O(\frac{v}{p}\frac{\mu_{MAX}}{BD})$ I/O operations, and with synchronization time $\ell\frac{v}{pk}L$, provided that $\ell \leq \frac{k\mu_{MAX}}{BD}$, $N \geq \ell\bar{v}^2B$, $B \geq \frac{v-1}{2}$, $D \leq v$, for integer constant $k > 0$, $p \leq v$, and $\bar{v} = \frac{v}{k}$.*

Proof. We use the results of Lemma 7.2. Each of the p real processors simulates $\frac{v}{p}$ virtual processors, k at a time. The synchronization time is therefore $\ell\frac{v}{pk}L$. Because there are now p processors instead of only one, as in Lemma 7.2, the I/O operations decrease from $O(\frac{v\mu_{MAX}}{BD})$ to $O(\frac{v}{p}\frac{\mu_{MAX}}{BD})$, giving computational overhead of $O(\frac{v}{p}\frac{\mu_{MAX}}{BD})$. The basic computation time decreases from $O(v\alpha)$ to $O(\frac{v}{p}\alpha)$, however. Since real communication is not required between virtual processors simulated on the same real machine, and since k virtual processors are simulated concurrently on each real machine, the number of non-local processor groups is $\frac{v}{pk}$ and the number of communication operations is reduced, compared to the CGM, to $O(\frac{v\beta}{pk})$. The constraints carry over from the single processor case. $\square$

Theorem 7.1 *Let $\mathcal{A}$ be a v -processor BSP-like algorithm with λ supersteps, and Φ monotone phases, where during each phase ϕ_i , $1 \leq i \leq \Phi$, the active problem size changes by a constant factor f_i in each of ℓ_i rounds. Let μ be the maximum context size*

¹When we assume that computational overhead for I/O is proportional to the number of blocks read/written, it is equivalent to assuming that the CPU is involved only in posting the request for each block, and the I/O is performed by an I/O channel or disk controller directly from/to a buffer in our program's data space. While this is possible, and often true on larger systems, many older versions of the C I/O library have performed the I/O from/to a separate, system buffer. This approach requires that the data be copied to/from the user buffer after/before the input/output operation. The copy operation requires the involvement of the CPU for every item of the buffer, and so this approach implies computational overhead which is B times larger.

for each phase, and let $\alpha_i + L$ and $g\beta_i + L$ be the computation time and communication time, respectively, of each phase ϕ_i of $\mathcal{A}$. Let $\ell = \max_{i=1}^{\Phi} \ell_i$.

$\mathcal{A}$ can be simulated on a single processor EM-CGM machine in $O(\frac{v\mu}{BD} \cdot (\lambda - \sum_{i=1}^{\Phi} \ell_i) + \Phi)$ parallel I/O operations (each of BD items) provided that $\ell \leq \frac{k\mu}{BD}$, $N \geq \ell\bar{v}^2 B$, $B \geq \frac{v-1}{2}$, $D \leq v$, for integer constant $k > 0$, $\bar{v} = \frac{v}{k}$.

Proof. We use Lemma 7.2 for each monotone phase, plus the arguments of Theorem 5.2. $\square$

Theorem 7.2 Let $\mathcal{A}$ be a v -processor BSP-like algorithm with λ supersteps, and Φ monotone phases, where during each phase ϕ_i , $1 \leq i \leq \Phi$, the active problem size changes by a constant factor f_i in each of ℓ_i rounds. Let μ be the maximum context size, and $\alpha_i + L$ and $g\beta_i + L$ be the computation time and communication time, respectively, of each phase ϕ_i of $\mathcal{A}$.

$\mathcal{A}$ can be simulated on a p -processor EM-CGM machine in $O(\frac{v\mu}{pBD} \cdot (\lambda - \sum_{i=1}^{\Phi} \ell_i) + \Phi)$ parallel I/O operations (each of pBD items) provided that $\ell \leq \frac{k\mu}{BD}$, $N \geq \ell\bar{v}^2 B$, $B \geq \frac{v-1}{2}$, $D \leq v$, $p \leq v$, for integer constant $k > 0$, $\bar{v} = \frac{v}{k}$.

Proof. We use Lemma 7.3 for each monotone phase, plus the arguments of Theorem 5.3. $\square$

We therefore can simulate an algorithm containing monotone phases with less I/O cost than the simplest approach would suggest. Under the conditions stated in the theorems, it costs only a constant times $\frac{N}{pBD}$ parallel I/O operations for each of the monotone phases, despite the fact that such phases may involve more than a constant number of supersteps.

Chapter 8

New EM Algorithms

8.1 Overview

A number of basic algorithms with applications in data structuring, computational geometry, and graph theory have been developed for BSP models by various authors. In this section, we present adaptations of a selection of these algorithms for the EM-BSP models.

In many cases the I/O complexities of these EM-BSP algorithms appear at first glance to contradict known lower bounds for their problems. In Section 8.2, we explain this contradiction by discussing the effect of our constraints on the lower bounds.

In Section 8.3 we present simple BSP algorithms for the fundamental problems of sorting, permutation and matrix transpose, and describe their adaptations to our EM-BSP models. In Section 8.5, we present several new multisearch algorithms for EM-BSP* and EM-CGM. In Section 8.5, we present a number of other new EM-CGM algorithms which we obtained from known CGM algorithms.

8.2 Apparent Reductions in I/O Complexity

Several known lower bounds on I/O complexity (see Section 1.3.3) contain a multiplicative factor of $\log_{\frac{M}{B}} \frac{N}{B}$. We obtain a number of results where this factor does not appear. This can be explained by the fact that the term $\log_{\frac{M}{B}} \frac{N}{B}$ is a constant when

$N \geq v^{1+\epsilon}b$, for $0 < \epsilon \leq 1$, as shown by Lemma 8.1.

For instance, Lemma 8.1, together with Aggarwal and Vitter's lower bound of $\Omega\left(\frac{N}{BD} \log_{\frac{M}{B}} \frac{N}{B}\right)$ I/Os for sorting [4, 84] imply lower bounds of $\Omega(\frac{N}{pBD})$ I/O operations for a number of problems such as sorting, general permutation, matrix transpose, computing FFTs, etc. when the slackness constraint $N \geq v^{1+\epsilon}b$ is valid.

Lemma 8.1 *For slackness $N \geq v^{1+\epsilon}b$, where ϵ is a constant, $0 < \epsilon \leq 1$. the value of $\log_{\frac{M}{B}} \frac{N}{B}$ is a constant whose value depends on ϵ , but not on N .*

Proof. For constant $0 < \epsilon \leq 1$,

$$N \geq v^{1+\epsilon}b \quad (8.1)$$

$$\implies N^{\frac{\epsilon+1}{\epsilon}-1} \geq v^{\frac{\epsilon+1}{\epsilon}} b^{\frac{\epsilon+1}{\epsilon}-1} \quad (8.2)$$

Now, choosing $c = \frac{\epsilon+1}{\epsilon} \geq 2$, equation (8.2) becomes

$$N^{c-1} \geq v^c b^{c-1} \quad (8.3)$$

Substituting $v = \frac{N}{M}$ and $b = B$ into equation (8.3) gives

$$\begin{aligned} \left(\frac{M}{B}\right)^c &\geq \frac{N}{B} \\ \implies \log_{\frac{M}{B}} \frac{N}{B} &\leq c \end{aligned}$$

We can conclude that $\log_{\frac{M}{B}} \frac{N}{B}$ is a constant whose value depends on ϵ , but not on N when $N \geq v^{1+\epsilon}b$. $\square$

So for constant ϵ , where $0 < \epsilon \leq 1$, the logarithmic term is a constant of size at most $\frac{\epsilon+1}{\epsilon}$ when $N \geq v^{1+\epsilon}b$.

8.3 Fundamental Problems

We first present new EM-CGM algorithms, obtained from CGM algorithms via Lemma 5.2 and Theorem 5.3, for the fundamental problems of sorting, permutation and matrix transpose. In each case the CGM algorithm uses $\lambda = O(1)$ communication

rounds, and $O(\frac{N}{v})$ internal memory per processor. Since our simulation techniques require that $N \geq v^{1+\epsilon}b$, the I/O complexities of the resulting parallel, external memory algorithms do not exhibit the logarithmic factor known to be present in the general case for these problems.

8.3.1 Sorting

The time complexity of sorting N items is $\Theta(N \lg N)$ for a comparison-based model of computation. On the PDM, sorting has been shown to have I/O complexity $O(\frac{N}{BD} \log_{\frac{M}{B}} \frac{N}{B})$ for general values of N , M , D , and B [4, 84].

Goodrich [53] has described a deterministic BSP algorithm for sorting which has a constant number of supersteps for $N \geq v^{1+\epsilon}$, $\epsilon > 0$ a fixed constant. We can achieve I/O complexity $\kappa = O(\frac{N}{pBD})$ by simulating this algorithm using the deterministic simulation technique of Chapter 6, for $p = O(v^\epsilon)$, and $N \geq v^{1+\epsilon}BD$, $\epsilon > 0$ a fixed constant.

Alternatively, we can simulate the algorithm CGMSampleSort, due to Schi and Schaeffer[76], which is described and analyzed in Section 9.2.3, using the deterministic simulation techniques of Chapter 5. Lemmas 9.2 and 9.3 give the performance of CGMSampleSort and its external memory version EM-CGMSampleSort respectively, analyzed under the EM-CGM model presented in Chapter 2. The proofs of Lemmas 9.2 and 9.3 are given in Section 9.2.3.

Lemma 9.2 *CGMSampleSort uses $O(\frac{N}{v} \log \frac{N}{v} + L)$ computation time, $O(g \cdot \frac{N}{v} + L)$ communication time, $O(\frac{N}{v})$ local memory per processor, and $O(1)$ communication supersteps, provided that $\frac{N}{v} \geq v^2$.*

Lemma 9.3 *CGMSampleSort can be simulated on a p processor EM-CGM machine in $O(\frac{N \log(N/v)}{p} + \frac{\bar{v}}{p}L)$ computation time, $O(g \cdot \frac{N}{p} + \frac{\bar{v}}{p}L)$ communication time, and $O(G \cdot \frac{N}{pBD} + \frac{\bar{v}}{p}L)$ I/O time, provided that $p \leq v$, $N \geq v^2B$, $D \leq B$, $B \geq v$.*

Since CGMSamplesort requires $\frac{N}{v} \geq v^2$, we obtain I/O complexity of $\kappa = O(\frac{N}{pBD})$ for EM-CGMSampleSort for $N \geq v^2B$, $v \geq D$, $B \geq v$.

8.3.2 Permutation

Permutation of N items on a RAM has time complexity $\Theta(N)$. On the PDM, this problem has I/O complexity $\Theta(\min(\frac{N}{D}, \frac{N}{BD} \log_{M/B} \frac{N}{B}))$ (see [4, 84]). However, we can achieve I/O complexity $\kappa = O(\frac{N}{pBD})$ by simulating algorithm CGMPermute, for $p = O(v^\epsilon)$, $N \geq v^{1+\epsilon}BD$, and fixed constant $\epsilon > 0$. The slackness constraint $N \geq v^{1+\epsilon}BD$ comes from the routing algorithm *LowSlackRouting*.

Algorithm 8.1 CGMPermute

$\mathcal{V}$ is an N element vector containing items to be permuted. $\mathcal{P}$ is a corresponding N element vector containing new indices for each element of $\mathcal{V}$.

Input: Each processor i , $0 \leq i \leq (v-1)$ holds an $\frac{N}{v}$ element vector $\mathcal{V}_i$, containing elements $i \cdot \frac{N}{v}$ to $(i+1) \cdot \frac{N}{v} - 1$ of $\mathcal{V}$, and an $\frac{N}{v}$ element vector $\mathcal{P}_i$, containing elements $i \cdot \frac{N}{v}$ to $(i+1) \cdot \frac{N}{v} - 1$ of $\mathcal{P}$.

Output: Each processor i contains items $i \cdot \frac{N}{v}$ to $(i+1) \cdot \frac{N}{v} - 1$ of the permuted vector $\mathcal{V}'$.

Assumption: v divides N evenly.

1. Each processor i , $0 \leq i \leq (v-1)$ sends the items of $\mathcal{V}_i$ to the processors holding the items indicated by $\mathcal{P}_i$.
2. Each processor performs the necessary rearrangements in its local memory to complete the calculation of $\mathcal{P}$.

— End of Algorithm —

CGMPermute performs the indicated permutation in a single communication round consisting of an $\frac{N}{v}$ -relation. The internal computation time is $O(\frac{N}{v})$, and the memory used is $O(\frac{N}{v})$ per processor.

8.3.3 Matrix Transpose

Transposing a $n \times m$ matrix, where $N = n \times m$ takes $\Theta(N)$ time on a RAM. On the Parallel Disk Model, this problem has I/O complexity $\Theta(\frac{N}{BD} \frac{\log \min(M, n, m, N/B)}{\log(M/B)})$ [4, 84]. However, we can achieve I/O complexity $\kappa = O(\frac{N}{pBD})$ by simulating algorithm CGMTranspose, below, for $N \geq v^{1+\epsilon}BD$. For ease of exposition, we assume that v divides N .

Algorithm 8.2 CGMTranspose

Let $\mathcal{M}$ be an $n \times m$ matrix, and let a_{ij} be the element of $\mathcal{M}$ in row i and column j . Let $\mathcal{M}_0$ be a one dimensional array containing the elements of $\mathcal{M}$ row by row, i.e., $a_{11}, a_{12}, \dots, a_{1m}, a_{21}, a_{22}, \dots, a_{nm}$. Let $\mathcal{M}'$ be the transpose of $\mathcal{M}$, and let a'_{ji} be the element of $\mathcal{M}'$ in row j and column i . Let $\mathcal{M}'_0$ be a one dimensional array containing the elements of $\mathcal{M}'$ row by row, i.e., $a'_{11}, a'_{12}, \dots, a'_{1m}, a'_{21}, a'_{22}, \dots, a'_{nm}$.

Input: Processor i , $0 \leq i \leq (v-1)$ holds items $i\frac{N}{v}$ to $(i+1)\frac{N}{v} - 1$ of $\mathcal{M}_0$.

Output: Processor i holds items $i\frac{N}{v}$ to $(i+1)\frac{N}{v} - 1$ of $\mathcal{M}'_0$.

1. Each processor determines the destination processor for each of its items and sends them in a single superstep to their destination.
2. Each processor inserts the received items into the appropriate positions in its memory.

— End of Algorithm —

CGMTranspose transposes the matrix $\mathcal{M}$ in a single communication round consisting of an $\frac{N}{v}$ -relation. The internal computation time is $O(\frac{N}{v})$, and the memory used is $O(\frac{N}{v})$ per processor.

8.3.4 Summary

Theorem 8.1 and Figure 8.1 summarize the performance of our EM-BSP algorithms for sorting, permutation, and matrix transpose.

Theorem 8.1 *Sorting, permutation, and matrix transpose can be performed on a p -processor EM-CGM with M local memory in each processor in $\kappa = O(\frac{N}{pBD})$ I/O operations, provided $N \geq v^{1+\epsilon}BD$ for $p = O(v^\epsilon)$, and constant $\epsilon > 0$.*

Proof. The proof follows from Theorem 5.4, and the performance of the CGM algorithms for sorting, permutation, and matrix transpose described in Sections 8.3.1, 8.3.2, and 8.3.3. □

	Problem Description	PDM I/O Complexity (See notes ^{a b c})	Complexity in CGM Model	EM-BSP Complexity ^{d e}
Group A: Fundamental Algorithms				
1.	Sorting	$\Theta(\frac{N}{BD} \log_{\frac{M}{B}} \frac{N}{B})$ [4, 84]	$O(\frac{N \log N}{v})$ $\lambda = O(1), M = O(\frac{N}{B})$ [see section 8.3.1]	$\alpha = O(\frac{N}{p})$ $\beta = O(\frac{N}{p} \log N)$ $\kappa = O(\frac{N}{pBD})$
2.	Permutation	$\Theta(\min(\frac{N}{D}, \frac{N}{BD} \log_{\frac{M}{B}} \frac{N}{B}))$ [4, 84]	$O(\frac{N \log N}{v})$ $\lambda = O(1), M = O(\frac{N}{B})$ [see section 8.3.2]	$\alpha = O(\frac{N}{p})$ $\beta = O(\frac{N}{p})$ $\kappa = O(\frac{N}{pBD})$
3.	Matrix transpose ^f	$\Theta(\frac{N}{BD} \frac{\log \min(M, k, \ell, N/B)}{\log(M/B)})$ [4, 84]	$O(\frac{N \log N}{v})$ $\lambda = O(1), M = O(\frac{N}{B})$ [see section 8.3.3]	$\alpha = O(\frac{N}{p})$ $\beta = O(\frac{N}{p})$ $\kappa = O(\frac{N}{pBD})$

Figure 8.1: Overview of New EM Algorithms

^aPDM I/O complexities as listed apply to general values for N, M, D , and B . If the constraints required by our techniques are applied to these results, the term $\log_{\frac{M}{B}} \frac{N}{B}$ becomes a constant.

^bWhen the parameter D is quoted in the PDM I/O Complexity column, this parameter refers to the number of disks overall, whereas the parameter D used in the EM-BSP I/O Complexity column refers to the number of disks on each of the p processors of the machine.

^cThe PDM I/O complexities listed for this group apply also to multiple processors when the interconnection method is a shared RAM, hyper-cubic network, or cube-connected cycles.

^dThese results are subject to the conditions $N \geq v^2 B$, $B \geq \frac{v}{2}$, and $v \geq D$.

^eWe use the following notation: α is the amount of data communicated, β is the parallel computation time, and κ is the number of parallel I/O operations.

^f k, ℓ are the number of columns and rows, respectively, where $N = k\ell$.

8.4 The Multisearch Problem

8.4.1 Overview

Chapters 4 and 5 described two simulation methods, one randomized, and one deterministic, for obtaining EM-BSP algorithms from parallel algorithms for BSP models. In Chapter 2, our new optimality criteria were described. In the current section, we describe research into multisearch algorithms for the EM-BSP models in which we use the results of all three chapters. Thus the primary purpose of this chapter is as a “case study” of the use and limitations of the simulation techniques of Chapters 4 and 5. We omit many of the details of the proofs in this chapter, as they can be found in [46, 47].

In *multisearch* problems [44, 12, 71] a large number of queries are simultaneously processed and answered by navigating through a large data structure on a parallel computer. A closely related application is an Internet search engine. As input for the multisearch problem, we consider a set of n queries and a balanced binary search tree T of size m .¹ The search tree is distributed among the processors and stored in their local memories. For clarity and ease of exposition we restrict our discussion to the case $M = \Theta(\frac{n}{p})$, i.e., each processor has enough internal memory to keep its share of the queries completely in its internal memory. However, with minor changes larger sets of queries can be accommodated in an I/O-optimal manner. Figure 8.2 illustrates an example application of multisearch. In the example, a two dimensional universe is divided into regions by lines in the plane. The queries are points and the result of a query is the label of the region containing the point. Figure 8.2 shows a tree built over the line segments which separate one region from another. The dots represent point queries. The queries navigate from the root of the tree to a particular leaf in order to determine a result. Examining the leaf level of the tree with the point queries superimposed, it is clear that sorting is insufficient for this problem (see queries labelled $q1$ and $q2$, for instance).

Multisearch has been shown to be useful for solving a number of data structuring

¹The reader is cautioned that our use of the variables m and n is different than the meanings often used in the external memory literature, where $n = \frac{N}{B}$ and $m = \frac{M}{B}$.

Figure 8.2: *Multisearch example.* Point queries select a region of a two-dimensional area divided by line segments.

problems on a parallel machine, e.g., trapezoidal decomposition, next element search on hypercubes [44], multiple planar point location, interval trees, hierarchical representations of polyhedras on meshes [12], dictionaries on 2-3 trees on a EREW-PRAM [71], maintaining balanced binary search trees on a BSP* [17, 16], multiple point location in a class of hierarchical DAGs on a BSP* [46] and a BSP [51]. The BSP* multisearch algorithm due to Bäumer, Dittrich, and Meyer auf der Heide [17] (with improvements due to Dittrich [46]) was the first 1-optimal algorithm for multisearch on a BSP model for both small and large trees.

In external memory, multisearch has been studied in [46, 47] for balanced binary trees and optimal trees [59]. A major issue in external memory multisearch is deciding how to block the graph. Blocking of graphs for external memory searching is examined in [65, 1, 88].

The batch filtering technique, due to [54] and described in Section 1.3.2.1, can be viewed as a multisearch technique that is I/O-optimal on a single processor machine, when $n \geq m$. For larger trees, however, batch filtering does more I/O than necessary, and it does not accommodate multiple processors.

Our study of multisearch was prompted by its importance as a tool for various problems, but also because it highlights some of the issues regarding the optimality

of algorithms under our EM-BSP models.

Section 8.4.2 discusses work due to Rau-Chaplin [73] and Dehne et al. [43] on multisearch for the CGM model, and sketches our EM-CGM multisearch algorithm, obtained by simulating the CGM algorithm.

Simulation of BSP* multisearch gives us an efficient multisearch algorithm on the EM-BSP* for small trees, discussed in Section 8.4.3.1.

While we are able to give a multisearch algorithm that is 1-optimal and I/O-efficient for small trees, I/O-efficiency is not possible for large trees. There is a lower bound on the I/O complexity of multisearch on large trees, $n < m$, which exceeds the computational and communication costs [46, 47]. In Section 8.4.3.2 an I/O-optimal EM-BSP* algorithm for large trees is sketched. While this algorithm is based on the same BSP* multisearch algorithm as in Section 8.4.3.1, it is not derived via simulation, but rather by “hand-coding” the I/O management at the points in the algorithm where it is necessary. The simulation approach involves swapping the contexts of the virtual processors of a BSP algorithm, and the contexts in this case would include space for the large search tree. This approach would therefore require the entire search tree to be read once for each superstep of the BSP algorithm. This is inefficient for large trees, and so the hand-coded approach is necessary for efficiency reasons. Details are available in [46, 47].

8.4.2 Multisearch on the EM-CGM model

Rau-Chaplin [73] and Dehne et al. [43] described the first efficient multisearch algorithm for a BSP model. Theorem 8.2 describes the performance of their algorithm on a CGM machine for $n \geq m$. This CGM multisearch algorithm does not accommodate larger trees efficiently.

Theorem 8.2 (adapted² from [43])

Let Q be a set of n queries and let T be a search tree of size m . For $n \geq m$, multisearch of Q against T can be done in time $O(\frac{n}{v} \log \frac{n}{v} + g \cdot \frac{n}{v} + L)$ on a v processor

²Dehne et al. use different terminology. For instance, they reverse our meanings of m and n , and they do not use the usual BSP-style expressions for stating time complexity.

CGM machine, with $\frac{n}{v} \geq v$.

As mentioned earlier, multisearch is used as a subroutine for a number of computational geometry problems. In such a situation, the tree and the queries are often generated by a previous step of the higher level algorithm. Such an example illustrates the usefulness of the CGM technique and our external memory algorithms for the case $n \geq m$. Using the results of Chapter 5, in particular Theorem 5.4, we obtain an EM-CGM algorithm with the following performance:

Corollary 8.1 *Let Q be a set of n queries and let T be a search tree of size m . Multisearch of Q against T can be done in time $O(\frac{n \log n}{p} + O(\frac{m}{p}) + g \cdot O(\frac{n}{p}) + G \cdot O(\frac{n}{pBD}) + \frac{\bar{v}}{p}L)$ on a v processor EM-CGM machine, for $n \geq m$, $M \geq k\mu + BD$, $p \leq v$, $N = \Omega(\bar{v}^2 B)$, and $v = \Omega(D)$, for arbitrary integer k , $1 \leq k \leq \frac{v}{p}$, and $\bar{v} = \frac{v}{k}$.*

8.4.3 EM-BSP* Multisearch

We now turn to multisearch on the EM-BSP* model. We present two different kinds of results in the following. The first results (see Section 8.4.3.1) are applicable to smaller trees, as in Section 8.1 above, and are obtained by applying the simulation techniques of Chapters 4 to the BSP* multisearch algorithm of [17, 46].³ Section 8.4.3.2 sketches a new multisearch algorithm for large trees, tailored to the EM-BSP*. This does not rely on simulation, but is based on the same underlying BSP* algorithm as in Section 8.4.3.1.

Theorem 8.3 summarizes the performance of the BSP* multisearch algorithm *Tree-Search* [46].

Theorem 8.3 [46] *Let $\hat{T}$ be a d -ary balanced search tree of size m , distributed among v processors by a preprocessing step. Let Q be a set of n queries distributed evenly among the processors. Let $\alpha, \epsilon > 0$ and $c \geq 1$ be constants, and b is the BSP* block size parameter. Let $\frac{n}{v} = \Omega(b \log^{1+\epsilon} n)$, $d = (\frac{n}{vb})^\alpha$, and $\alpha \leq \frac{\epsilon}{2+2\epsilon}$. Multisearch of Q against $\hat{T}$ using BSP* algorithm *Tree-Search* uses $O(\frac{n+m}{v})$ memory per processor and*

³Refined versions of the BSP* multisearch algorithms presented in [17] can be found in [46], and so we base the current work on the latter descriptions.

$(1 + o(1)) \cdot \frac{n}{v} \log m + O(\hat{g} \cdot \frac{n}{vb} \log_d m + L \cdot \log_d v \log_d m)$ running time with probability $1 - n^{-c}$. In particular, Tree-Search is 1-optimal for $L = o(\frac{n \log^2 d}{v \log v})$ and $\hat{g} = o(b \log d)$.

8.4.3.1 EM-BSP* Multisearch on a Small Tree

When the search tree is small, i.e. about the size of the query set, we can obtain a work-optimal and I/O efficient algorithm by applying the simulation techniques of Chapter 4 (Theorem 4.2) to the BSP* multisearch algorithm of Theorem 8.3.

Theorem 8.4 *Let $\alpha, \beta, \psi, \gamma > 0$ be constants. Given a d -ary tree $\mathcal{D}_T$ of size $m \leq n^{O(1)}$ distributed by suitable preprocessing. Let $d = (n^{1-\beta}/b)^\psi$, $\psi \leq \frac{1}{4}$, $D = O(n^{\beta-\gamma}/\log^2 n)$, $B \leq b = O(n^{1-\alpha})$, and $1 > \alpha > \beta > \gamma > 0$.*

Multi-search for n queries against T can be done w.h.p. in time $(1 + o(1)) \frac{n \log m}{p} + O(\frac{m}{p}) + \hat{g} \cdot \frac{n}{pb} \log_d m + G \cdot \frac{m}{pBD} + L \cdot D \log(pD)$ on a p -processor EM-BSP with internal memory of size $O(\frac{n+m}{p} + BD)$ per processor and external memory of size $O(\frac{m}{pD})$ per disk.*

In particular, for $m = O(n \log n)$, $G = O(BD \frac{n \log m}{m})$, $\hat{g} = O(b \log n)$ and $L = O(\frac{n \log m}{D \log(pD)})$ the algorithm is work-optimal, communication and I/O efficient.

Proof. We combine Theorems 4.2 and 8.3, noting that $\mu = \frac{n+m}{v}$ and so $v\lambda\mu = O(m)$.

The expression for running time in Theorem 8.4 can be decomposed into four components as follows:

- (1) The synchronization time $L \cdot D \log(pD)$ must be accounted to each of computation, I/O and communication times. To be work-optimal, communication-efficient, and I/O-efficient, we require that $L \cdot D \log(pD) = O(\frac{n \log m}{p})$.
- (2) Exclusive of synchronization time, the computation time is $(1 + o(1)) \frac{n \log m}{p} + O(\frac{m}{p})$. For work-optimality, we require that $\frac{m}{p} = O(\frac{n \log m}{p})$.
- (3) The communication time is $\hat{g} \cdot \frac{n}{pb} \log_d m$. To be communication-efficient we require that $\hat{g} \cdot \frac{n}{pb} = O(\frac{n \log m}{p} \log_d m)$.
- (4) The I/O time is $G \cdot \frac{m}{pBD}$. To be I/O-efficient, we require that $G \cdot \frac{m}{pBD} = O(\frac{n \log m}{p})$.

□

The overhead of the simulation with regard to work does not affect the asymptotic running time. EM-BSP* multisearch is work-optimal for sufficiently small trees,

i.e. $m = O(n \log n)$. As in BSP* multisearch, the communication time scales according to the number of EM-BSP* processors, and is optimal for trees of size $n^{O(1)}$. The number of supersteps increases slightly compared to BSP* multisearch. However, the size of the internal memory is $O(m/n^\beta)$, where $0 < \beta \leq 1$, for a suitably large number of disks which allows very large problem instances. Moreover, the external storage used per disk is optimal. The constraints on the blocksize B and number of disks D are functions of n which pose no practical limitations for suitably large problem instances.

8.4.3.2 EM-BSP* Multisearch on a Large Tree

For trees of size $\omega(n \log n)$ the I/O time of the above algorithms becomes the dominating term of their running time complexity. I/O-efficiency is not possible for large ratios m/n .

For large trees, the binary tree T is partitioned into two parts which are referred to as the *upper part* and the *lower part*, respectively. The upper part contains nodes of all levels L_i for all $0 \leq i \leq \log \frac{m}{n}$ and the lower part contains the nodes of levels L_i for $\log \frac{m}{n} < i < \log m$. The root is on level L_0 and at height $\log m$, while the leaves are on level $L_{\log m}$ and at height 0. Thus, the upper part contains $O(n)$ nodes and the lower part contains $O(m - n)$ nodes.

In a preprocessing phase, a search tree $\hat{T}$ with node degree d is constructed from T and is distributed using the layered block mapping among the processors.

Within a processor each supernode is broken into subtrees of size B , which are distributed among the disk drives. The tree is much too large to be stored completely in the internal memory. We assume that each processor has enough internal memory to keep its share of the queries completely in its internal memory. However, with straightforward changes, larger sets of queries can be accommodated in an I/O-optimal manner. Theorem 8.5 gives the performance of the search phase of EM-BSP* multisearch on a large tree, $m \geq n$.

Theorem 8.5 *Let $\alpha, \epsilon > 0$ and $c > 0$ be constants. Given n queries and a d -ary tree $\hat{T}$ of size m distributed among p processors by a preprocessing step. Let*

$n = \Omega(Dbp \log^{1+\epsilon} n)$, $B < d = (\frac{n}{pb})^\alpha$ and $\alpha \leq \frac{\epsilon}{2+2\epsilon}$. Multisearch for n queries against T can be done on an EM-BSP* with p processors and D disks, in time

$$(1 + o(1)) \frac{n}{p} \log m + O\left(\frac{n \log(m/n)}{p \log B} + \hat{g} \cdot \frac{n \log m}{pb \log d}\right) \\ + \left(G \cdot \left(\frac{n}{pBD} + \frac{n \log(m/n)}{pD \log B}\right) + L \cdot \frac{\log p}{\log(n/(pb))} \frac{\log m}{\log d}\right)$$

using $O(\frac{n}{p})$ internal memory per processor and $O(\frac{m}{pD})$ external memory per disk, with probability $1 - 1/n^c$, for constant $c > 0$.

Proof Sketch. EM-BSP* multisearch is a variant of BSP* multisearch, and inherits its properties regarding communication time, computation time and number of supersteps. External memory space consumption is optimal, as it requires $O(m/(pD))$ space on each of the pD disks. Unfortunately, the I/O costs dominate the overall running time for large trees, so EM-BSP* multisearch is not I/O-efficient for large trees. However, [46, 47] show a lower bound that proves that the I/O-time of EM-BSP* multisearch is asymptotically optimal. In other words, this lower bound shows that I/O-efficiency is not feasible for large trees. In the light of this lower bound EM-BSP* multisearch is work-optimal, communication efficient and I/O-optimal for suitable parameter constellations. $\square$

8.4.4 Summary

We have derived new parallel EM-BSP multisearch algorithms for static balanced binary trees. Our multisearch algorithms involve a single preprocessing phase, after which multiple search phases can be performed.

1. For search trees up to size $O(n \log n)$ where n is the number of queries, we obtained EM-BSP* and EM-CGM multisearch algorithms which are work-optimal, I/O-efficient, and in the case of multiple processors, communication-efficient. These algorithms are obtained via the simulation techniques described in Chapters 4 and 5, respectively.

2. For larger trees we obtained an EM-BSP* algorithm which is simultaneously work-optimal and I/O-optimal. When multiple processors are present, the algorithm is also communication-efficient.

Batch filtering (described in Section 1.3.2.1) was introduced in [54] as a sequential EM technique for executing a collection of simultaneous queries in search structures modeled as planar layered directed acyclic graphs (DAGs). Batch filtering is therefore an EM multisearch technique. In this chapter we restricted our attention to multisearch in balanced binary trees. While batch filtering is I/O-optimal for sequential versions of these problems when $m = O(n \log n)$ where n is the number of queries and m is the size of the data structure to be queried, we consider data structures larger than $O(n \log n)$ and also parallel processing models of computation. Our EM multisearch techniques can therefore be viewed as extending the EM paradigm of batch filtering[54] for balanced trees from the $m = O(n)$ case to include also $m = \omega(n)$, and from the RAM model to BSP-like parallel computation models. We will refer to the extended technique as *multi-filtering*, while recognizing that it may itself invite extensions to other data structures such as hierarchical DAGs.

We also consider larger data structures than can be accommodated optimally by the techniques of [40]. The work of the algorithm remains an important aspect of our complexity measure for external memory algorithms, but achieving c -optimality in I/O may not be possible due to the blocking factor of the disks. Our algorithms for EM-BSP* multisearch are 1-optimal in the sense of [52]. Depending on the size of the data structure, they are also either I/O-efficient, or I/O-optimal.

We note that the simulation techniques introduced in Chapters 3 to 6 can be used to create efficient parallel EM algorithms from a large class of BSP algorithms. In the current chapter, we used these results in the application of multisearch, but we also considered larger data structures than can be accommodated optimally by simulation.

The results obtained by the BSP* simulation are the first (parallel) work-optimal and I/O-efficient external memory algorithms for multisearch. The EM-BSP* multisearch algorithm for large trees is the first multisearch algorithm for a realistic parallel external memory model that is efficient for large trees in terms of work, I/O

and communication.

While batch filtering is I/O-optimal for sequential versions of these multisearch problems when $m = O(n \log n)$ where n is the number of queries and m is the size of the data structure to be queried, we consider data structures larger than $n \log n$ and also parallel processing models of computation.

8.5 Other New External Memory Algorithms

Figures 8.3 and 8.4 list a number of other important problems arising in computational geometry, GIS, and graph algorithms, for which we report EM-CGM algorithms created by our technique, together with their I/O complexity and that of the previously best known algorithm for the problem. In some cases, we obtain a better I/O complexity, and in other cases we obtain the same I/O complexities as previously known. In each case, however, our algorithm is scalable not only in terms of the number of disks per processor but also in terms of the number of processors used. Previous algorithms were often not efficient in a multiprocessor environment (particularly in a distributed memory environment), and in many cases it is not clear how they could be adapted to parallel disks.

The problems listed in Table 8.3 are stated briefly below:

- The *endpoint dominance problem* is a special case of the *next element search problem* (see below), where the query set is composed of the endpoints of the line segments themselves.
- Given a set R of isothetic rectangles, the *measure problem* is to compute the area covered by the union of R .
- Consider a set S of n points in 3-space. For a point v let $x(v)$, $y(v)$ and $z(v)$ denote the x -coordinate, y -coordinate, and z -coordinate, respectively, of v . Point v *dominates* a point w iff $x(v) > x(w)$, $y(v) > y(w)$, and $z(v) > z(w)$. A point is *maximal* in S if it is not dominated by any other point of S . The *3D-maxima problem* consists of determining the set of all maximal points in S .

- Given a set S of n points in the Euclidean plane, the *all nearest neighbors problem* is to determine for each point $v \in S$ its nearest neighbor $NN_S(v)$ in S , where $NN_S(v)$ is a point $w \in S \setminus \{v\}$ such that $dist(v, w) \leq dist(v, u) \forall u \in S \setminus \{v\}$.
- Given a set S of n non-intersecting line segments in the plane, the *lower envelope problem* consists of computing the set of segment portions visible from the point $(0, -\infty)$.
- Let S be a set of n points in the plane with some *weight* $w(v)$ assigned to each $v \in S$. The *2D-weighted dominance counting problem* consists of determining for each $v \in S$, the total weight of all points which are dominated by v .
- Let S be a set of r pairwise disjoint m -vertex polygons. The *uni-directional separability problem* consists of determining all directions d such that S is separable by a sequence of r translations in direction d , one for each polygon.
- Let S be a set of r pairwise disjoint m -vertex polygons. The *multi-directional separability problem* consists of determining if S is separable by a sequence of r translations in different directions.
- Let S be a set of n non-intersecting line segments in the plane, and let Q be a set of m query points. The *next element search problem* is to find, for each query point $q_i \in Q$, the line segment directly above q_i .
- Let G be an embedding of a graph in the plane. The *planar point location problem* is to find, for each of a set of N query points q_i , $1 \leq i \leq N$, the face of G which contains q_i .
- Let S be a set of line segments in the plane. The *trapezoidation problem* is to decompose the plane into a set of trapezoids, based on the arrangement of the line segments.
- Given a simple polygon S , the *triangulation problem* is to partition the interior of S into a set of triangles.

The problems listed in Table 8.4 are stated briefly below:

- Given a linked list ℓ of N objects, the *list ranking* problem is to compute, for each object in ℓ , its distance from the beginning of the list.
- Given a connected graph $G = (V, E)$, the *Euler tour* problem is to find a cycle that traverses each edge of G exactly once.
- Given a rooted tree $G = (V, E)$, and a pair of vertices (u, v) of G , the *lowest common ancestor* problem is to find the vertex w of G that is an ancestor to both u and v and is farthest from the root.
- Given an undirected graph $G = (V, E)$, a connected subset of vertices is a subset of vertices in which there is a path in G between each pair of vertices. The *connected components* problem is to find the maximal connected subsets of vertices in G .
- Given a connected undirected graph $G = (V, E)$, the *bi-connected components* problem is to find the maximal connected subsets of vertices of G which remain connected when any single edge is deleted.
- Given a connected undirected graph $G = (V, E)$, the *ear decomposition* problem [21] is to find an ordered partition of E into r simple paths $P_1, \dots, P_r$ such that P_1 is a cycle, and for each i , $2 \leq i \leq r$, P_i is a simple path whose endpoints belong to $P_1 \cup \dots \cup P_{i-1}$, but with none of its internal vertices belonging to P_j , for $j < i$. The *open ear decomposition* problem is similar, but none of the P_i , for $i > 1$ is a cycle.

	Problem Description	PDM I/O Complexity (See notes ^{a,b})	Complexity in CGM Model ^c	EM-BSP I/O Complexity ^d
Group B: GIS and Computational Geometry Algorithms				
1.	Polygon triangulation ^e	$O(\frac{N}{B} \log \frac{M}{B} \frac{N}{B})$ [11]	$\tau = O(\frac{N \log N}{v})$, [21]	$O(\frac{N \log N}{pBD})$
2.	Trapezoidal decomposition		$\lambda = O(1), M = O(\frac{N \log N}{v})$	
3.	Segment tree construction			
4.	Next Element Search on line segments			
5.	Batched planar point location	$O((\frac{N}{B} + k) \log \frac{M}{B} \frac{N}{B})$ [11]	$\tau = O(\frac{N \log N}{v})$ [21] $\lambda = O(1), M = O(\frac{N \log N}{v})$	$O(\frac{N \log N}{pBD})$
6.	3D convex hull ^f	$O(\frac{N}{B} \log \frac{M}{B} \frac{N}{B})$ [54]	$\tau = \bar{O}(\frac{N \log N}{v})$ [39]	$\bar{O}(\frac{N}{pBD})$
7.	2D Voronoi diagram		$\lambda = \bar{O}(1), M = \bar{O}(\frac{N}{v})$	
8.	Delaunay triang.			
9.	Lower envelope of non-intersecting line segments		$\tau = O(\frac{N \log N}{v})$ [43] $\lambda = O(1), M = O(\frac{N}{B})$	$O(\frac{N}{pBD})$
10.	Generalized lower envelope		$\tau = O(\frac{N \log N}{v})$ [43] $\lambda = O(1), M = O(\frac{N \alpha(N)}{B})$	$O(\frac{N \alpha(N)}{pBD})$
11.	Area of Union of Rectangles	$O(\frac{N}{B} \log \frac{M}{B} \frac{N}{B})$ [54]	$O(\frac{N \log N}{v})$ [73, 43]	$O(\frac{N}{pBD})$
12.	3D-maxima		$\lambda = O(1), M = O(\frac{N}{B})$	
13.	2D-nearest neighbors			
14.	2D-weighted dominance counting		$\tau = O(\frac{N \log N}{v})$, [73, 43]	$O(\frac{N}{pBD})$
15.	Uni- and multi-directional separability		$\lambda = O(1), M = O(\frac{N}{B})$	

Figure 8.3: Overview of New EM Algorithms (continued)

^aPDM I/O complexities as listed apply to general values for N, M, D , and B . If the constraints required by our techniques are applied to these results, the term $\log \frac{M}{B} \frac{N}{B}$ becomes a constant.

^bWhen the parameter D is quoted in the PDM I/O Complexity column, this parameter refers to the number of disks overall, whereas the parameter D used in the EM-BSP I/O Complexity column refers to the number of disks on each of the p processors of the machine.

^c τ represents the computation time of the BSP algorithm

^dThese results are subject to the conditions $N \geq v^2 B$, $B \geq \frac{v}{2}$, and $v \geq D$.

^eThe PDM I/O complexities for this group are documented for the single processor, single disk case. We are not aware of multi-disk or multi-processor extensions.

^fThe PDM I/O complexities for these problems apply to a PRAM-like interconnection only.

	Problem Description	PDM I/O Complexity (see notes ^{a,b})	Complexity in CGM Model	EM-BSP I/O Complexity ^c
Group C: Graph Algorithms				
1.	List ranking	$O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ [25]	$\lambda = O(\log v), \tau = O(\frac{N}{v})$ $M = O(\frac{N}{v})$ [20]	$O(\frac{N \log v}{pBD})$
2.	Euler tour of tree			
3.	Lowest common ancestor			
4.	Tree contraction			
5.	Expression tree evaluation			
6.	Connected components ^d	$O(\frac{E}{V} \frac{N}{DB} \log_{\frac{M}{B}} \frac{N}{B} \cdot \max\{1, \log \log \frac{VBD}{E}\})$ [64]	$\lambda = O(\log v)$ $\tau = O(\frac{V+E}{v})$ $M = O(\frac{V+E}{v})$ [20]	$O(\frac{(V+E) \log v}{pBD})$
7.	Spanning forest			
8.	Ear and open ear decomposition			
9.	Biconnected components			

Figure 8.4: Overview of New EM Algorithms (continued)

^aPDM I/O complexities as listed apply to general values for N, M, D , and B . If the constraints required by our techniques are applied to these results, the term $\log_{\frac{M}{B}} \frac{N}{B}$ becomes a constant.

^bWhen the parameter D is quoted in the PDM I/O Complexity column, this parameter refers to the number of disks overall, whereas the parameter D used in the EM-BSP I/O Complexity column refers to the number of disks on each of the p processors of the machine.

^cThese results are subject to the conditions $N \geq v^2 B$, $B \geq \frac{v}{2}$, and $v \geq D$.

^dfor a graph of V vertices and E edges

Chapter 9

Experiments

The growing theory of external memory algorithms is less interesting to practitioners than a theoretician might expect. The main reasons lie in the difficulties which surround implementation of these algorithms on real machines. Often it is extremely time consuming to verify whether a theoretically attractive algorithm is also capable of being attractive in practice. A new algorithm may apparently perform poorly for many reasons, which may include:

1. resource contention between the program and other tasks running concurrently,
2. incorrect implementation of the algorithm,
3. large constants in the algorithm's time complexity,
4. inappropriate choice of algorithm parameters,
5. conflicts between the assumptions of the model under which the algorithm was conceived, and the realities of the machine on which it is implemented.

Some of these situations can themselves have a number of cases. For instance, Item (1) above may be caused by other application programs (which may be avoidable), or by tasks spawned by the operating system. The resources in contention can be internal memory, I/O buffers and devices, or a processing unit. Resolution, or even

identification of such a conflict may require insight into the internal workings of the particular operating system and the particular hardware devices used.

Recently there has been an increasing interest in implementation and experimental research work targeted to I/O efficient computation. Research work in this area includes:

- The TPIE (Transparent Parallel I/O Environment) project of Vengroff and Vitter [82, 80] which aims to collect implementations of existing algorithms within a common framework, and to make development and validation of new implementations easier.
- Experiments by Chiang [24, 23] with four algorithms for the orthogonal segment intersection problem.
- Cormen et al. [32, 31, 34, 36] have reported on a number of implementation issues and results relating to I/O efficient algorithms, including FFT computations using parallel processors, and FFT, permutations, and sorting using the Parallel Disk Model.
- LEDA-SM [37] is a project which aims to make the LEDA library [62] of computational geometry algorithms and data structures compatible with external memory applications.

As part of the research for this dissertation, two implementation studies were undertaken. The first study was to the author's knowledge the first full implementation of the *buffer tree* external memory data structure due to Arge [8, 9]. This study is described in Section 9.1, and is also presented in [58].

The second implementation project, discussed in Section 9.2, involved implementation of the deterministic simulation described in Chapter 5 and [42]. The implementation design, and the experiments described, are joint work with Kumanan Yogaratnam, and are also mentioned in [41].

9.1 Buffer Tree Case Study

9.1.1 Introduction

Motivated by the goal of constructing I/O efficient versions of commonly used internal memory data structures, Arge [8, 9] proposed the data structuring paradigm, and in particular the *buffer tree*. A buffer tree is an external memory search tree. It supports operations such as insert, delete, search, deletemin, and it enables the transformation of a class of internal-memory algorithms to external memory algorithms by exchanging the data structures used. A large number of external memory algorithms have been proposed [8, 9] using the buffer tree data structure, including sorting, priority queues, range trees, segment trees, and time forward processing. These in turn are subroutines for many external memory graph algorithms, such as expression tree evaluation, centroid decomposition, least common ancestor, minimum spanning trees, ear decomposition. There are a number of major advantages of the buffer tree approach. It applies to a large class of problems whose solutions use search trees as the underlying data structure. This enables the use of many normal internal memory algorithms, and “hides” the I/O specific parts of the technique in the data structures. Several techniques based on the buffer tree, e.g. time forward processing [9], are simpler than competitive EM techniques, and are of the same I/O complexity, or better, with respect to their counterparts.

In this section, we present an implementation of the buffer tree, and show the flexibility and generality of the structure by implementing EM sorting and an EM priority queue. To test the efficiency of our implementation we use sorting as an example. For data sets larger than the available main memory, our implementation of buffer tree sort outperforms internal memory sort (e.g. qsort) by a large and increasing margin. We use an EM merge sort algorithm from TPIE to provide comparative performance results for the larger problem sizes. We observe that certain parameters suggested in [8, 9] may not provide the best results in practice. By tuning these parameters we obtained improved results while maintaining the same asymptotic worst case I/O complexity. The buffer tree is a conceptually simple data structure, and it turned out that implementation of these applications based on the buffer tree

Figure 9.1: *The buffer tree.*

was straightforward. Therefore we can support the claim that the buffer tree is a generic EM data structure that performs well in theory and practice.

9.1.2 Description of the Buffer Tree

We first describe the buffer tree data structure of Arge [8, 9], with two update operations, namely insert and delete. Subsequently, we will discuss how we can perform sorting and maintain a priority queue using a buffer tree.

Let N be the total number of update operations, M be the size of the internal memory and B be the block size and set $m = M/B$ and $n = N/B$. The buffer tree is an (a, b) -tree [57], where $a = m/4$ and $b = m$, augmented with a buffer in each node of size $\Theta(m)$ blocks. Each node (with the exception of the root) has a fan out (number of children) between $m/4$ and m . Each node also contains partitioning elements, or “splitters” which delimit the range of keys that will be routed to each child. The number of splitters is one less than the fanout of the node. The height of the buffer tree is $O(\log_m n)$ (see Figure 9.1). Since the buffer tree is an extension of the (a, b) -tree, the computational complexity analyses of the various (a, b) -tree operations still apply. The buffers are used to defer operations, to allow their execution in a “lazy manner”, thus achieving the necessary blocking for performing operations efficiently in external memory. A buffer is *full* if it has more than $m/2$ blocks.

For any update operation, a *request element* is created, consisting of the record to be inserted or deleted, a flag denoting the type of the operation, and an automatically

generated time stamp. Such request elements are collected in the internal memory until a block of B requests has been formed. The request elements, as a block, are inserted into the buffer of the root using one I/O. If the buffer of the root contains less than $m/2$ blocks there is nothing else to be done in this step. Otherwise the buffer is emptied by a *buffer-emptying* process.

The buffer-emptying process at an internal node requires $O(m)$ I/Os, since we load $m/2$ blocks into the internal memory and distribute the elements among the $\Theta(m)$ children of that node. A buffer-emptying process at a leaf may require rebalancing the underlying (a, b) -tree. An (a, b) -tree is rebalanced by performing a series of “splits” in the case of an insertion or a series of “fuse” and “share” operations in the case of a delete [57]. Before performing a rebalance operation, we ensure that the buffers for the corresponding nodes are empty. This is achieved by first doing the buffer-emptying process at the node involved. The deletion of a block may involve the initiation of several buffer-emptying processes. By using dummy blocks during the deletion process, a buffer emptying process can be protected from interference by other processes. (See [8] for details.)

The analysis (i.e., the I/O complexity) of operations on the buffer tree is obtained by adapting the amortization arguments for (a, b) trees [57]. Each update element, on insertion into the root buffer, is given $O(\frac{\log_m n}{B})$ credits. Each block in the buffer of node v holds $O(\text{the height of the tree rooted at } v)$ credits. For an internal node, its buffer is emptied only if it gets full and moreover this requires $O(m)$ I/O's. Therefore, ignoring the cost of rebalancing, the total cost of all buffer emptying on internal nodes is bounded by $O(n \log_m n)$ I/Os. The total number of rebalance operations required in an (a, b) -tree, where $b > 2a$, over K update operations on an initially empty (a, b) -tree, is bounded by $K/(b/2 - a)$. Therefore, for N update operations, on an $(m/4, m)$ -tree, the total number of rebalance operations is bounded by $O(n/m)$. Moreover, each rebalance operation may require a buffer-emptying process as well as updating the partitioning elements, and therefore may require up to $O(m)$ I/Os. Thus the total cost of rebalancing is $O(n)$ I/Os. The cost of emptying leaf nodes is bounded by the sorting operation. We summarize.

Theorem 9.1 (*Arge [8, 9]*) *The total cost of an arbitrary sequence of N intermixed*

insert and delete operations on an initially empty buffer tree is $O(n \log_m n)$ I/O operations.

Sorting : A buffer tree can be used to sort N items as follows. First insert N items into the buffer tree followed by an *empty/write operation*. This is accomplished by performing a buffer-emptying process on every node starting at the root, followed by reporting the elements in all the leaves in the sorted order. This can be done within the complexity of computing the buffer tree data-structure.

Corollary 9.1 (*Arge [9]*) *N elements can be sorted in $O(n \log_m n)$ I/O operations using the buffer tree.*

The PDM compares competing EM algorithms according to the asymptotic number of I/O operations they require to solve a given problem of size N . By this model, the buffer tree sorting algorithm [9] is optimal, as the number of I/O operations matches the lower bound $\Omega(n \log_m n)$ for the sorting problem [4].¹ In practice, however, other factors can also affect the running time. Cormen and Hirschl [31] observe that many PDM applications are not I/O bound, which suggests that CPU time is an important factor to be considered. The EM-BSP, EM-BSP* and EM-CGM models incorporate both I/O and CPU time (see Chapter 2).

Priority Queues : A dynamic search tree can be used as a priority queue, since in general, the leftmost leaf of the search tree contains the smallest element. We can use the buffer tree for maintaining a priority queue in external memory by permitting the update operation described previously for insertion into the priority queue and adding a *deletemin* operation. It is not necessarily true that the smallest element is in the leftmost leaf in the buffer tree, as it could be in the buffer of any node on the path from the root to the leftmost leaf. In order to extract the minimum element, i.e., execute the *deletemin* operation, a buffer-emptying process must first

¹For a single disk. n is the number of disk blocks in the problem. m is the number of disk blocks that fit into the memory size M .

be performed on all nodes on the path from the root to the leftmost leaf. After the buffer emptying the leftmost leaf consists of the B smallest elements, and the children of the leftmost node in the buffer tree consists of at least the $\frac{m}{4}B$ smallest elements. These elements can be kept in the internal memory, and at least $\frac{m}{4}B$ deletemins can be answered without doing any additional I/O. In order to obtain correct results for future deletemins, any new insertion/deletion must be checked first with these elements in the internal memory. This realization of a priority queue does not support the changing of priorities on elements already in the queue.

Theorem 9.2 (*Arge [9]*) *The total cost of an arbitrary sequence of N insert, delete and deletemin operations on an initially empty buffer tree is $O(n \log_m n)$ I/O operations.*

9.1.3 Implementation of the Buffer Tree

The (a,b)-Tree : The buffer tree is an (a,b)-tree with buffers added to each tree node. One source of code for an (a,b)-tree is LEDA [62]. It quickly became clear, however, that this code was designed specifically for internal memory usage, as nodes were linked by many pointers to support a wide range of higher level operations. Converting the various pointers of the (a,b)-tree implementation to external memory representations turned out to be time consuming and ineffective for a data structure that was required to be I/O efficient. Too many I/O operations were required to update a single field in a child or parent of the node in memory to give attractive EM performance.

The Buffers : Each node of the buffer tree has an associated buffer, which may contain between 0 and $m/2$ blocks of data² which have not yet been inserted into the (a,b)-tree part of the buffer tree. The fanout of an internal node is at most m . Therefore, for a buffer tree consisting of ℓ levels, there may exist up to $m^{\ell-1}$ buffers, each $m/2$ bytes in size.

²This may increase temporarily during a buffer emptying process.

Our implementation currently models each buffer as a Unix file. Due to restrictions on the number of Unix files that could be open at a time, each buffer file is closed after use and reopened when necessary. The time required by file open and close operations, as measured in our tests, was small. However, preliminary experiments suggest that this scheme may limit the effectiveness of asynchronous I/O, since the file *close* operation must wait for any outstanding I/O to complete. In this thesis we report primarily on our experiences using synchronous I/O.

Compatibility, Usability and Accessibility : We wanted our implementation to be compatible with the use of LEDA [62] and with TPIE [82, 81]. The large number of algorithms and data structures available in LEDA forms an attractive context for implementation of internal memory algorithms, which are often components of a larger external memory system. For example, the external memory priority queue uses an internal memory priority queue as a component. The collection of efficient external memory techniques provided by TPIE forms an attractive workbench for building and testing new EM implementations. We chose C++ as our implementation language to preserve potential compatibility with these code libraries. In addition, we adopted the automated documentation tools from LEDA.

EM Sorting Using the Buffer Tree : A buffer tree can be made to sort a data set simply by inserting the data into the tree, and then force-emptying the buffers. Our implementation allows the leaves to be read sequentially from left to right, thus the implementation of sorting was straightforward.

An EM Priority Queue using the Buffer Tree : The buffer tree can be modified to construct an I/O-optimal priority queue with *insert* and *deletemin* operations in EM [9]. Our implementation is obtained as follows:

- The leftmost leaf node of the buffer tree, together with its associated $\frac{m}{4}$ to m leaf blocks are kept in internal memory, instead of on disk. The (a,b)-tree nodes on the path from the root to the leftmost leaf are also kept in memory. For

typical values of M , N , B , a , and b these (a,b)-tree nodes make a negligible impact on internal memory consumption.

- The data records in memory are organized using an appropriate internal memory priority queue. In our initial implementation, this is a conventional heap, originally obtained from LEDA.
- Any requests that would normally be inserted into the root buffer of the buffer tree are first compared to the leftmost splitters of the (a,b)-tree nodes on the path from the root to the leftmost leaf. As argued above, this gives a small constant number of comparisons in practice. If a request would be routed to the cached data blocks, it is inserted directly into the internal heap. Otherwise, it is inserted into the buffer tree in a “normal” fashion.
- The balance of the underlying (a,b)-tree is maintained in a normal fashion. If the leftmost leaf node underflows, sharing or fusing with siblings will occur. If it overflows, splitting may occur as it would in an (a,b)-tree.

Testing Platform and Parameters : We performed most of our development and experimental work on a network of sixteen 166 MHz Pentiums, each with 32 MB of internal memory and a pair of 2 GB hard disks used exclusively for data storage. A central 1 GB hard disk is available via NFS for program storage. The processors each run the Linux operating system. Although we did not use it except for NFS access to the central program storage, the processors are interconnected by a fast ethernet switch. We found that our experimental timings were reproducible between processors, and so we were able to run multiple timing tests independently, simultaneously, and reliably on this platform.

We chose $k = 16$ bytes as the record size for our tests. A record (sometimes we will call it an element) consisted of four integer (4 byte) fields: a key, an associated “data field”, a time stamp, and an operation type (insert/delete/query) field. We chose $kB = 4096$ bytes, since this was the system page size. We chose $kM = 500KB$

($m=125$), which is small, but allowed us to choose manageable problem sizes (from a disk space point of view) yet still apply stress to our algorithms.

9.1.4 Evaluating the Implementation

In order to obtain meaningful performance results, we attempted to control the following factors:

Contention with other processes or users of the machine for machine resources such as memory, CPU and disks: We perform the testing on dedicated machines, and so the results were not affected by other users. The Unix operating system spontaneously initiates various system tasks to perform routine maintenance and monitoring functions, thus it is not easy to avoid contending with these tasks for system resources. Smaller test runs may vary significantly due to these effects. However, on the larger test runs, the influence of system tasks on the run time can generally be ignored.

Virtual memory effects such as the ‘transparent’ behaviour of the operating system to swap parts of the program image between main memory and secondary storage: The swapping of portions of a task to disk to make room for another activity can occur without warning or notification. In our tests we attempted to minimize the likelihood of this occurring by choosing kM , the problem size in bytes, to be much smaller than the physical memory size. For instance, on our Linux machines with 32MB of physical memory, we performed the majority of our tests using $kM = 500KB$. Another tactic is to use the Linux *mlockall* service call to lock the application into memory. This seems to work reasonably well in some cases, and does allow the application to use up to 50% of the physical memory without fear of being affected by virtual memory effects. However, the requesting program must be running with “root” privileges for the request to be honoured.

Comparison to Quicksort : We found that the buffer tree easily outperformed the “built-in” internal memory quicksort technique. A simple quicksort program was

Figure 9.2: *Timings for Internal Quicksort and Buffer Tree Sort on Random Inputs.* For problem sizes larger than 2.8 million items the internal quicksort failed due to lack of internal memory.

written using the built-in C function “qsort”. Figure 9.2 shows the results for a range of problem sizes. For larger input sets, the (recursive) quicksort program ran out of stack space on our system, but by that time the internal sort was slowing due to virtual memory effects and the buffer tree was already outperforming it.

Tuning the Buffer Tree : We discovered that the value of b relative to m is important to the performance of buffer tree sort on random data. For buffers of size $\frac{m}{2}$ and $(a, b) = (\frac{m}{4}, m)$ as suggested in [9], we obtain (partial) block sizes of approximately $\frac{B}{2}$ keys pushed to the next level for each of (perhaps) m children of a node whenever the parent’s buffer is emptied. Reducing the fanout, while maintaining the buffer size increases the expected number of elements in each block. We found that $(a, b) = (\frac{m}{32}, \frac{m}{8})$ gave the best performance in our tests. Smaller or larger values of b resulted in longer run times. Figure 9.3 shows running time curves for Buffer Tree Sort (BTS) and for TPIE Merge Sort (TMS). Results for BTS are shown for several values of b , where $a = b/4$ in all cases. Both TMS and BTS are running

Figure 9.3: *Timings for Buffer Tree Sort and TPIE Merge Sort.* TMS ran out of disk space at about 35 million elements because it keeps its original data file. BTS does not require that all of the input data be available before it begins, and does not require a separate file for the original data.

with synchronous I/O and single buffering. The TPIE MMB stream option is used. The buffer tree is using the ‘iostream’ access method. BTS performance is best for about $b = \frac{m}{8}$ and gets worse if b differs much from this value. BTS with $b = m/8$ has running times that change nearly linearly with the problem size for the problem sizes shown.

We caution that our experiments focussed on finding support for the predictions of asymptotic behaviour of Buffer Tree Sort. The actual running times of BTS may be improved by further tuning, and the performance of TPIE Merge Sort may improve with other choices of parameters and options.

We found the increased speed with smaller b intriguing, and so we counted the number of *block pushes* performed by the algorithm. A block push occurs when data is pushed to a child buffer after the parent’s buffer becomes full. It may consist of a partial block, a full block, or more than a block of request elements. Reducing the fanout increases the expected size of the data in a block push, and therefore

Figure 9.4: *Number of Block Pushes for Buffer Tree Sort on Random Inputs.*

may reduce the number required, and the number of I/O operations as a result. Figure 9.4 shows the relationship between several fanout values b and the number of block pushes over a range of input sizes. The reduction in block pushes seems to be the major reason for the improvement in running time between $b = m$ and $b = \frac{m}{8}$.

Non-linearities in the Running Time : We observed that contrary to predictions of the I/O model, for random input data our buffer tree sort implementation tended to have non-linear run times as the problem size increased. (Actually, we expect the running time to increase more than linearly by a logarithmic factor. However, since the base of this logarithm is large, the predicted increase in running time is close to linear for the range of problem sizes considered.)

Figure 9.5 shows a graph of problem size versus runtime for random input data and $b = m$. Also shown in this graph are curves for the various activities of the buffer tree, i.e., a breakdown of where this time is spent. The total running time appears to be increasing super-linearly with the problem size. *Total Running Time* is the sum of running time for *Insertions* plus *Force Empty All Buffers*. Running time for *Insertions* is composed of the sum of *Insertion: Empty Internal Buffers* plus

Figure 9.5: *Timings for Buffer Tree Sort with Smaller Fanout.* $(a, b) = (\frac{m}{4}, m)$, $m = 125$, $B = 256$.

Insertion: Empty Leaf Buffers. Both of these seem to be more than linear with the problem size. However, referring to Figure 9.4, the number of block pushes is not increasing super-linearly.

Adjusting the parameter b in the buffer tree both reduced the number of I/O operations performed by BTS and apparently removed the non-linear behaviour in our tests. Figure 9.6 shows the same graph as Figure 9.5 for $b = m/8$. In contrast to Figure 9.5, the Total Running Time curve is quite linear after about 10 million elements. The component curves in the figure are equally well behaved.

While the constant represented by the slope of the running time curve is larger for BTS than for TMS, we note that BTS is an online sorting technique and therefore addresses a different situation than does TMS. (See Figure 9.7).

Experiments with Parallel Disks : We experimented briefly with storing the buffer tree on multiple disks, by striping the buffers and leaves across two disks. We obtained a multiple disk driver (the “PDM API”) from Tom Cormen at Dartmouth College, and ported it from the DEC Alpha environment to Linux without much

Figure 9.6: *Timings for Buffer Tree Sort with Larger Fanout.* $(a, b) = (\frac{m}{32}, \frac{m}{8})$, $m = 125$, $B = 256$.

difficulty. To manage concurrent disk access, the PDM API requires a Posix threads implementation, which we obtained from Florida State University. Perhaps due to the large data cache maintained by Linux, we found that large data volumes were required before two parallel disks outperformed a single disk for a simple “write-as-quick-as-you-can” application. For buffer tree sort this would require a single block push to be very large. We tried increasing m to allow this and did see marginally better performance with two disks for moderate values of n . Unfortunately, as n grew towards a more interesting size we began to see our performance degrade, apparently due to virtual memory effects. We concluded that we needed more real memory for this sort of experiment.

Figure 9.8 shows running times for a single disk under the PDM API and C++ iostream access methods. The PDM API could be expected to be slightly slower as it introduces some extra computation such as its use of threads. This seems to be true in the case of $b = \frac{m}{8}$, but its ability to overlap computation with I/O (asynchronous I/O) seems to allow it to outperform in the case $b = m$.

Figure 9.7: *Pipelining sort with generation of inputs.* If the generation of the inputs is sufficiently time consuming, Buffer Tree Sort can provide a speed advantage over offline methods by permitting the insertion time to be hidden by the time to generate its inputs. The time to Force Empty Buffers then may be the only time that remains “visible”.

9.1.5 Conclusions and Lessons Learned

In this thesis we describe an implementation of a buffer tree and two EM algorithms based on the buffer tree: an external memory treesort, and an external memory priority queue.

While the buffer tree gives us an I/O-optimal sort, our timing studies of the implementation indicate that its performance is sensitive to some nonlinearities in the environment or algorithm. Experimental results show that these non-linearities are reduced by an optimal choice of parameters.

Our tests on random input sets lead to an experimental determination of parameter values different from those originally suggested in the design of the data structure.

Although the running times of our treesort implementation (BTS) with parameter $b = m$ clearly show non-linearities, $b = \frac{m}{8}$ produced a running time curve which is

Figure 9.8: *Running Times of PDM and Iostream I/O Access Methods.* The PDM uses asynchronous I/O and this seems to give it an advantage for the larger fanouts.

for practical purposes a straight line when the problem size is more than 10 million elements. The application was also heavily I/O bound. This supports the prediction of the algorithm [9] and the model [85] that the asymptotic running time is $\Theta(n \log_m n)$ I/Os and the number of I/O operations is the dominant issue in the algorithm.

The non-linear behaviour of BTS with parameter $b = m$ was manifested to various degrees in some of the other fanouts which we tried. While we expected some effect on running time as this parameter was varied, the sensitivity to non-linearity is troubling and we do not rule out implementation decisions as a possible cause.

We conclude that (a) the buffer tree as a generic data structure appears to perform well in theory and practice, and (b) measuring I/O efficiency experimentally is an important topic that merits further attention.

We conclude by noting that this implementation work was intended primarily to test the validity of the asymptotic performance predictions of the buffer tree. Naturally, this was possible only to the extent that available disk space would allow. Nevertheless, we believe the results are useful, as we were able to perform experiments

Figure 9.9: *Stevens' measurements on the effects of varying the block size.*

where the internal memory size was as small as 0.001 of the problem size. We believe that the timing results therefore confirm the asymptotic performance predictions for the buffer tree.

In terms of the absolute performance of our implementation, we believe a number of efficiency improvements could be made, including a custom implementation of buffers (using the file system for this likely introduces unnecessary overhead), and improved physical I/O (copying buffers in the C I/O library).

Another area for further investigation arises from our desire to keep the effective internal memory size of the buffer tree small. In order to do this, we restricted the block size to only 512 kilobytes. While the work of Stevens [78] suggests that such a block size achieves most of the efficiencies one expects by blocking I/O to disk (see Figure 9.9), other researchers have suggested that larger block sizes are better. It has been suggested that the sensitivity to non linear increases in running time for a range of problem sizes is related to this small block size.

9.2 Simulation Case Study

9.2.1 Introduction and Motivation

In this section we describe implementation issues arising from a case study of the deterministic simulation approach of Section 5.

The main objectives of this implementation project were as follows:

1. explore the implementation issues raised in the simulation technique of Chapter 5 and determine if, in practice, there are any unforeseen difficulties,
2. perform performance comparisons to validate the expectations (Lemma 9.3) of linear I/O, and linear running time as the problem size grows,
3. investigate the issues involved in an automatic translation of CGM algorithms to EM-CGM algorithms.

In Section 9.2.2 we sketch a design of a run-time system that allows a parallel algorithm implemented in *MPI* (Message Passing Interface standard) [63] to be executed on a “target machine” with one or more processors, and one or more disks per processor. Utilizing the ideas and techniques of Chapter 5 (or, potentially, Chapters 4 or 6), such a run-time library promises to allow large problems to be executed efficiently on a machine with limited internal memory by ensuring that the disks are used effectively. Such a capability is interesting from at least two perspectives:

- from the perspective of obtaining good external memory algorithms, the practicality of the simulation techniques should be determined,
- from the perspective of parallel algorithms, the simulation idea promises efficient scalability, permitting larger problems to be executed efficiently on a given machine than would otherwise be the case. This idea is described more fully in [41].

In Sections 9.2.3 and 9.2.4 we describe some results of implementing an external memory version of a parallel algorithm for sorting.

We use the usual notation: N is the problem size, v is the number of (virtual) processors in the BSP algorithm, p is the number of real processors in the EM-BSP machine, B is the disk block size, b is the communication message size, each real processor has $M = O(\frac{kN}{v})$ internal memory, for $1 \leq k \leq \frac{v}{p}$, and each real processor has D local disks.

9.2.2 An EM Run-time System for Parallel Algorithms

We first give an overview of the design of a suitable run-time system to support execution of BSP algorithms on an EM-BSP. We describe the following in terms of the Linux operating system, although the issues and decisions taken should in most cases apply equally well to other operating systems. We implement each virtual processor as a separate process or task, and so we will refer to virtual processors as processes/tasks, and vice-versa. In this particular context, we use the terms *virtual processor*, *process*, and *task* interchangeably.

In many cases, each virtual processor executes the same program code, or at least contains the program code for all roles that a virtual processor may be required to play. It is common practice for the program code to make decisions based on a unique label associated with every virtual processor. In our implementation we therefore did not attempt to swap program code of the virtual processors between internal memory and the disks.

9.2.2.1 One Real Processor

In the following, we describe the operation of a single real processor, executing the operations required by a number of virtual processors. At various points in the discussion, however, we mention issues that arise when multiple real processors are present. More discussion of the case of multiple real processors is presented in Section 9.2.2.2.

The discussion is broken down into three parts: 1) implementation of virtual processors, 2) implementation of communication between virtual processors, and 3) implementation of multiple disks.

1. Implementation of virtual processors : We chose to represent the virtual processors of the CGM algorithm as *threads*, or *lightweight tasks*. A lightweight task is a task whose memory is shared with other tasks, and therefore its context does not have to be swapped individually by the operating system. We will use the terms *thread* and *lightweight task* interchangeably. In our implementation, the lightweight tasks representing virtual processors relinquish control of the CPU to the next virtual processor voluntarily, at predetermined points in their code. A virtual processor task might as well run to completion before any other virtual processor task begins execution. In fact, permitting more than k of them to compete for memory would violate a basic premise of our simulation technique, as it would lead the operating system to attempt to allocate more than $O(\frac{kN}{v})$ virtual memory, and the resulting swapping could cause fine grained accesses to disk. Accordingly, we must “manually” swap the context of each virtual processor in and out of the local internal memory of a real processor.

The virtual processor tasks are represented by sequentially scheduled sub-tasks of a single, preemptively scheduled “background” thread. By creating k such background threads, each permitting only a single virtual processor to be active at once, k virtual processes can be resident in the memory of a real processor concurrently, for $1 \leq k \leq \frac{v}{p}$.

It is easy to determine when a virtual processor should surrender control of the CPU to the next one. At the end of each computation superstep, each virtual processor issues a communication request (part of the next communication superstep). We discuss the implementation / simulation of these communication functions below. The yield function provides a means for passing control from one virtual processor to the next.

- **yield :** This function is called by each virtual processor at the end of every computation superstep. In particular, yield is called at the beginning of every communication function. It acts as a barrier synchronization operation. It ensures that several actions occur:

1. The swappable context of the processor is written to the local disks, and then the related internal memory is freed.
2. The memory areas required for the context of the next virtual processor are allocated, and its context is read from the local disks.
3. The messages sent to the new processor in the previous superstep are read from the local disks.
4. Control of the CPU is passed to the new virtual processor.

Thus, the round-robin scheduling of virtual processors within a background thread is accomplished via the `yield` function, which in turn, uses the C library `setjmp` and `longjmp` functions. Using `setjmp`, a task can store its program counter and other system context in a specified spot. Using `longjmp`, the task can transfer control to a task context which was previously saved by `setjmp`. The stack pointer of a running task points to a stack frame, stored on its system stack. A stack frame contains space for each local, dynamic variable allocated implicitly by the current function of the running task. It also contains a return address to the function that called the current function of the running task. Other stack frames, for each of the functions in which the current function of the running task is nested, are also on the stack. This imposed some constraints on our implementation. The stack of the background thread cannot be swapped out of memory by the simulation, as it must be available to the operating system at all times. We therefore must minimize the amount of data that is stored on this stack, not permitting dynamic, implicitly allocated local variables in our virtual processor tasks, for instance. In our implementation, the stack contents cannot be part of the virtual processor context that is managed/swapped by the simulation technique.

These considerations require us to maintain a minimal, unswappable, global context on the stack, plus larger, swappable, local context for each virtual processor. The latter are managed according to the requirements of the simulation technique. The swappable context can consist only of that memory which is allocated explicitly by the user BSP code at execution time. (i.e. via the C `malloc` call). In order to keep track of the size and components (memory areas) of a virtual processor, we require

the code for the virtual processors to use special library calls, `emmalloc/emfree`, for memory allocation/deallocation requests. This permits us to determine the regions of memory to be swapped at the end of each superstep.

- `emmalloc/emfree` : In order to control the swapping of the local context of a virtual processor, we require that its local variables be allocated dynamically via a call to the EM library function `emmalloc`, and deallocated if necessary using a call to `emfree`. The EM implementation keeps track of each allocation and deallocation request in a data structure local to each real processor. It ensures that any such memory contents belonging to a virtual processor is swapped into the real memory prior to its use, and out to the local disks after the respective virtual processor has finished its computation superstep.

Figure 9.10 shows the main services provided by the EM run-time library, organized into layers for explanation purposes. In the application layer, the program code originally written for execution in each processor of a BSP machine is run as a series of processes on an EM-BSP machine. Requests for communication between processes, and implicit barrier synchronization of processes are assumed to have been coded as MPI function calls. We also assume that memory allocation and deallocation requests have been coded as calls to our `emmalloc` and `emfree` library services.

2. Implementation of communication between virtual processors : The exchange of messages between virtual processors of a parallel algorithm is modelled in our implementation by the writing of the message data to disk by the sender, and the reading of message data from disk by the receiver. In general, it may also be necessary to move message data from one real processor to another. We discuss the latter issue in Section 9.2.2.2. Here we focus on the case of a single real processor, and the movement of message data between internal memory and disk.

We pattern our implementation after the suite of communication services offered by the MPI communication standard. The functions described below do not cover the entire spectrum of services available in MPI, but they address the major functionality requirements, and they are sufficient for our case study. We believe that the missing

Figure 9.10: *Organization of EM Library Services.* Services in a layer are available to layers above.

pieces can be designed without major difficulties. Each EM library function replaces an MPI communication function of the same name. We omit details of the arguments of each function for simplicity.

- **alltoall** : This function sends v different, specified portions of a buffer to v different destination processors. It also implies a reception of v such messages in each processor by the beginning of the next computation superstep. On a single real processor, the external memory version of alltoall involves (i) writing the v outgoing buffer portions to the local disks, in parallel, as each virtual processor is simulated in turn, (ii) calling yield to pass control to the next virtual processor in the current superstep, (iii) reading the appropriate data from the disks in parallel just prior to the next computation superstep of each destination processor. The methodology we use here is the topic of most of Chapter 5. On multiple real processors the EM implementation must use a real communication superstep after each simulation round to deliver messages to the correct real processor, where they must be received and stored on the local disks. This issue arises independently of the particular MPI function used, and is discussed further in Section 9.2.2.2.

- **bcast** : This function broadcasts the contents of a buffer on a designated “root” node to each of the v processors. On a single real processor, the external memory implementation writes the contents of the buffer to the message areas of the destination processors on the local disks. Since there can be only one broadcasting processor in this superstep, the EM implementation should arrange to simulate only the broadcasting processor and skip the others until the next superstep. For multiple real processors, the EM implementation should first conscript a virtual processor on each of the real nodes, and perform a partial broadcast from the root node to each of these. Then the broadcast can proceed locally, in parallel, according to the technique used for a single real processor.
- **gather** : This function causes each of the v processors to send a local buffer to a common, designated destination processor. On a single real processor, the external memory implementation writes the outgoing message from a processor to a designated ‘slot’ in the incoming message area of the destination processor on the local disks. The BSP algorithm being simulated must respect the fact that, overall, no more than $c \cdot \frac{N}{v}$ data should be sent to any single destination, for some known, fixed constant c . In the case of multiple real processors, the EM implementation should first conscript a virtual processor as an intermediary on each real node. Simultaneous local gathers should be done to these intermediaries, followed by a single MPI gather involving real communication between the intermediaries and the destination virtual processor.

Implementation of parallel disks : To implement parallel disk access we also use lightweight tasks, one per physical disk. These tasks, which we refer to as *disk drivers*, do not require the services of the CPU for a prolonged period at a time. They need prompt service when real events such as I/O completion or requests for I/O occur, but the amount of computation required before they become blocked should be small.

As in Section 9.1, the disk driver tasks were implemented as individual, preemptively scheduled threads, reflecting their role as high priority, foreground processes. This means that the access to the CPU is driven by events such as the completion of

an I/O request, or the expiration of a time slice. Unlike the virtual processor tasks, the disk drivers do not relinquish control of the CPU “voluntarily”, except when they wait for an event to occur, such as arrival of a request to read or write a block, or arrival of a completion event on a previously posted I/O operation.

The preemptively scheduled threads (background thread and disk drivers) were implemented by means of a Posix threads package, included with the Linux operating system.

Figure 9.11 illustrates the relationships between the contexts for the various tasks. In the figure, we differentiate between two types of context: 1. *Application context* (see item “G” in the figure): This consists of all of the application’s data that is required for it to execute. This is the usual meaning, as used in other chapters of this thesis. 2. *System context* (see item “J” in the figure): This is a small subset of the application context, consisting of the contents of the CPU registers of a running task, i.e. program counter, stack pointer, etc. This is the context saved / restored by the `setjmp` / `longjmp` calls.

9.2.2.2 Multiple Real Processors

When multiple real processors are present, the EM-MPI functions (e.g. `alltoall`, `bcast`, etc.) not only must write the indicated data to disk, but must also send data to the necessary real processors, using the corresponding real MPI function. Because each original communication superstep is also broken up into two communication substeps by the `BalancedRouting` procedure (Section 5.2), we can be assured that in each of these substeps a virtual processor sends the same amount of data to every other virtual processor (Theorem 5.1). Since each real processor simulates the same number of virtual processors, the real processors also receive equal amounts of data from each other.

The p real processors require one real communication step per computation round ($\frac{v}{pk}$ per computation round of the original CGM algorithm). Because of the perfect balance in communication between real processors, we can perform the corresponding real communication round at the end of each computation round of the real processors, before the current batch of $\frac{v}{pk}$ processors are swapped out.

Figure 9.11: *System Design of the External Memory Library.*

The steps in executing the EM library version of `alltoall`, for instance, is as follows (other EM-MPI communication functions are similar). Recall that `alltoall` sends v , not necessarily equal sized, portions of a buffer, one portion to each virtual processor. Let S represent the current set of k virtual processors in the memory of the current real processor.

1. execute `BalancedRouting` to route the v specified buffer portions using the real MPI `alltoall` function,
2. write the v buffer portions received in step (1) to the local disks,
3. call `yield` to pass control to the next set of k virtual processors in the current superstep,
4. read the appropriate data from the disks in parallel (i.e. just prior to the next computation superstep of the processors in S).

9.2.3 A Deterministic EM-CGM Samplesort

We now present a CGM sorting algorithm due to Shi and Schaeffer [76] which we call `CGMSampleSort`. This algorithm has the following features [76]:

1. It is asymptotically optimal in computation time for $N \geq v^3$,
2. The load balancing between processors is nearly perfect in practice, and within a factor of two in theory,
3. It causes little memory and network contention. We will use it as a basis for a parallel, external memory sort.

Algorithm 9.1 `CGMSampleSort` [76]

Input: The N items to be sorted are distributed evenly among the internal memories of the v processors of a CGM machine. Each processor has a unique label between 0 and $v - 1$.

Output: The N items are sorted in the memories of the CGM processors. The $\frac{N}{v}$ smallest items are located in the memory of processor 0, the next $\frac{N}{v}$ smallest items are located in the memory of processor 1, etc.

1. The v processors each sort the $\frac{N}{v}$ items in their local internal memories.
2. Each processor chooses v equally spaced samples from the items in its possession.
3. The processors each send the v samples to processor 0
4. Processor 0 sorts the v^2 samples and chooses from them v equally spaced splitter elements.
5. Processor 0 sends the v splitters to each of the other processors.
7. Each processor divides its local elements into v buckets determined by the splitter elements.
8. Each processor sends the contents of bucket i to processor i for all $0 \leq i < v$.
9. Each processor merges the v sets of items now in its possession to produce the final sorted order.

— End of Algorithm —

Lemma 9.1 [76] *No processor receives more than $2 \cdot \frac{N}{v}$ items as a result of Step (8.) of CGMSampleSort, provided that $\frac{N}{v} \geq v^2$.*

Lemma 9.2 *CGMSampleSort uses $O(\frac{N}{v} \log \frac{N}{v} + L)$ computation time, $O(g \cdot \frac{N}{v} + L)$ communication time, $O(\frac{N}{v})$ local memory per processor, and $O(1)$ communication supersteps, provided that $\frac{N}{v} \geq v^2$.*

Proof. Steps 3 and 4 require that $\frac{N}{v} \geq v^2$

Lemma 9.1 ensures that $O(\frac{N}{v})$ local memory is used by CGMSampleSort, and there are clearly a constant number of supersteps.

Computation time for Step 1 is $O(\frac{N}{v} \log \frac{N}{v})$. Processor 0 takes $O(v^2 \log v)$ time for Step 4. Computation times for steps 2,3,5 are $O(v)$, and for steps 7,8,9 the computation times are $O(\frac{N}{v})$ because of Lemma 9.1. Since $\frac{N}{v} \geq v^2$, the computation time overall is $O(\frac{N}{v} \log \frac{N}{v})$.

The only communication supersteps are steps 3, 5, and 8. Communication time is $O(g \cdot \frac{N}{v})$ (step 8) plus $O(g \cdot v)$ (steps 3,5), which is bounded by $O(g \cdot \frac{N}{v})$. $\square$

Lemma 9.3 *CGMSampleSort can be simulated on a p processor EM-CGM machine in $O(\frac{N \log(N/v)}{p} + \frac{\bar{v}}{p} L)$ computation time, $O(g \cdot \frac{N}{p} + \frac{\bar{v}}{p} L)$ communication time, and $O(G \cdot \frac{N}{pBD} + \frac{\bar{v}}{p} L)$ I/O time, provided that $p \leq v$, $N \geq v^2 B$, $D \leq B$, $B \geq v$.*

Proof. We use the results of Lemma 9.2 and Theorem 5.4. There are $O(1)$ supersteps in CGMSampleSort, so $\lambda = 1$ in Theorem 5.4. The context size μ for a CGM algorithm is $O(\frac{N}{v})$. We assume $k = 1$. The slackness constraint required by Lemma 9.1 is $\frac{N}{v} \geq v^2$, which is equivalent to $N \geq v^2 B$, for $B \geq v$. $\square$

9.2.4 Experimental Results

Preliminary implementation experiments with sorting support our expectations of linear running time. Figure 9.12 shows running times for a CGM sorting algorithm on a single processor a) using virtual memory and LAM MPI (see [60]), and b) converted to an EM-CGM algorithm by our deterministic simulation. Initially, the simulated algorithm runs slower than the LAM MPI algorithm. The main reason is that the simulated algorithm is performing I/O to swap contexts and deliver messages between the 8 virtual processors, while the LAM MPI version does little or no I/O until the 8 virtual processors begin to exceed the available virtual memory. Once the memory images of the virtual processors collectively get large enough, however, swapping begins to dominate the computation of the LAM MPI version. The simulated algorithm continues steadily, however, its running time apparently growing linearly with the problem size.

The anomalous point on the EM-CGM timing curve of Figure 9.12 can be eliminated, and was included for purposes of discussion in Section 9.2.5, below.

9.2.5 Discussion and Conclusions

The experiments done to date provide some confirmation of the value of our simulation technique in practice, but are clearly incomplete. More experimental work would be useful in this area, especially to determine the performance of the parallel EM cases.

Our results show that for problems sufficiently larger than the available real memory, the EM-CGM algorithm can outperform the CGM algorithm by a wide and increasing margin. We note that the experiments were performed by implementing algorithm SeqCompoundSuperstep (Chapter 5) without the message balancing step

Figure 9.12: *Running times for CGMSampleSort and EM-CGM SampleSort.* In each case, 8 virtual processors were simulated on a Pentium 133 with one disk. In the CGM case, LAM MPI was used.

of algorithm *BalancedRouting* (or equivalent). As a result, we were forced to accommodate messages up to M in size. This resulted in the messaging matrix being $v \times N$ items in size, rather than N items in size. We expect that the inclusion of the balancing step, while increasing the computation time slightly, should reduce the overall execution time by reducing the I/O requirements by nearly an order of magnitude when $v = 8$. The excessive size of the messaging matrix also limited the size of problem that we were able to accommodate due to limited disk size and we expect that experiments with larger problems would be instructive.

There appears to be an anomalous point on the EM-CGM timing curve of Figure 9.12. This point is both anomalous and reproducible! When the test runs are performed in a batch, one after the other, this observation can be reproduced. However, when the tests are run individually, separated in time, and with careful attention to the elimination of transient disk and memory congestion, we observe a running time more in line with our expectations. Such phenomena occur regularly in our tests, and can be attributed to the buffer cache management policies of the Linux operating

system. As mentioned in Section 9.1, Linux uses any available main memory as a “buffer cache”, caching I/O buffers for potential future use. We also observed that writes to disk were faster than reads in many cases, suggesting that applications are allowed to continue after a write request, when data had been written to the buffer cache, but had not yet physically been written to disk. Naturally, this complicates the measurement process.

As mentioned above, the first order effects of our techniques on performance were clear, and observable. However, as in Section 9.1, we found that in practice, it was difficult to quantify the performance effects of many of our attempts at “fine tuning” the code, due to the memory management and I/O buffering practices of the Linux operating system. While the Linux buffer cache management is very effective on programs that are not I/O-aware, we believe that the advent of I/O aware applications may justify a rethinking of the buffer cache management policies of Unix to enable further performance gains.

Chapter 10

Conclusions and Future Work

It has been said that “Beauty is in the eye of the beholder”. The same can be said about what is “interesting”, and what is not. To me, the relationship between parallel algorithms and external memory algorithms is interesting, and well worth the effort that went into this research. That a body of existing work in parallel algorithms can now be put to use in the field of external memory algorithms is a satisfying effect that has only been partially explored in this thesis.

This work has demonstrated several ways to transform BSP-like algorithm with appropriate attributes to an external memory algorithm that is efficient according to our model, and has identified a considerable number of example applications where suitable BSP-like algorithms are already known.

We believe that the ability of our simulations to accommodate a variable number k , for $1 \geq k \leq \frac{v}{p}$, of virtual processors in the memory of a real processor allows our techniques to conform to a new *memory-adaptive* EM model, recently proposed by Barve and Vitter [14]. In this model, the available memory changes periodically and unpredictable, perhaps due to the demands of unrelated applications. More investigation of this idea, as it applies to our techniques, seems warranted.

Our new EM-BSP algorithms have the advantage that they are essentially quite simple compared to some previously known external memory algorithms for the same problems, and they accommodate multiple processors on the target machine. They have the limitation that the ratio of internal memory size M to problem size N should

not be too small. This constraint comes rather directly from the slackness built into the parallel BSP algorithms from which our EM-BSP algorithms have generally been derived. However, it seems that in many cases this slackness constraint can be reduced at the cost of more communication and I/O rounds. This provides an opportunity for further research.

We argue, however, that even for very large problems, the slackness constraint is not an impediment in practice, as it seems reasonable that the internal memory size should grow as some function of the external memory size. Even using our deterministic simulation from Chapter 5, terabyte-sized problems can be solved with internal memory which is only 0.0001 of the memory size.

The fact that we use the term *simulation* to characterize our techniques raises the obvious question of their performance in practice. It should be emphasized that implementations of parallel algorithms can, in principle, be executed directly by substituting a different run-time library for the original communications library. This was illustrated in a preliminary way by the design and experiments described in Section 9.2. The most interesting issue seems to what extent the overheads incurred during the swapping of the contexts of virtual processors can be reduced. In a purely automatic adaptation of existing parallel codes, the prospects for achieving comparable performance to a hand-coded external memory algorithm seem limited. However, there may be possibilities for the programmer of a parallel program to provide additional information to the run-time library, or even the compiler, in order to make such automatic, efficient scalability achievable in practice. This is an area for future research.

Another interesting direction is to view parallel algorithms as a series of decompositions of a problem into subproblems which are independent, and can be solved in any order, including simultaneously. These are algorithms which we have argued can be executed efficiently as external memory algorithms under the appropriate conditions. The resulting algorithms have the distinct advantage that they continue to be efficient when multiple processors are present. While it may turn out that the individual instances of parallel external memory algorithms which we cite in Chapter 8 are efficient only in theory and not when compared to algorithms “hand-coded” for

the purpose, the paradigm of designing external memory algorithm from a parallel framework is compelling. This is an area which is receiving increasing attention.

The techniques presented here are only analyzed with respect to their asymptotic complexity, and we have not investigated the size of the constant factors. On the other hand, very few benchmark studies have been made to test the true practicality of some of the known external memory algorithms. Even without such experience, it seems clear that many of the existing external memory algorithms have not been designed for multiple processors, and may therefore perform poorly in such an environment. Further benchmark studies are indicated, both on the algorithms we propose here, and on many previously known algorithms.

Furthermore, if an existing external memory algorithm for a particular problem can be shown to be intrinsically better than one obtained from a parallel algorithm, then perhaps the external memory algorithm can be used to produce a better parallel algorithm. The cross-fertilization of the two areas seems to me to be one of the most exciting areas for future research.

If a “better” external memory algorithm already exists for a single processor machine, but is not efficient on any available parallel machine, perhaps it is not better. Large problems often demand the sort of computing power that is most economically available via parallel processing. Scalability to multiple processors is an intrinsic feature of the external memory algorithms derived from our simulation techniques.

We have concentrated on the interaction between the main memory and the external disk system. Many of the same issues also arise between the cache memory and main memory layers of the memory hierarchy. This is an area which is becoming increasingly important as increases in processor speed continue to outstrip the improvements in the speed of affordable main memory. The increasing discrepancy in speed seems likely to prompt the addition of more layers in the cache hierarchy. Perhaps, in particular, parallel algorithms can provide an algorithmic basis for using such caches more effectively.

Appendix A

Useful Probabilistic Results

In this Appendix we list a number of results that are used in Chapter 4.

A.1 Tail Estimates

We use the following tail estimates:

Lemma A.1 *If X is a non-negative random variable and $r \geq 0$, we have*

$$\Pr[X \geq u] \leq \frac{E[e^{rX}]}{e^{ru}}.$$

Proof. We use the following Markov inequality. Let X be any random variable. Then, for all $t \in \mathbb{R}^+$,

$$\Pr[X \geq t] \leq \frac{E[X]}{t}.$$

For any positive real r ,

$$\Pr[X \geq u] = \Pr[e^{rX} \geq e^{ru}].$$

Applying the above Markov inequality to the right hand side, we have

$$\Pr[X \geq u] \leq \frac{E[e^{rX}]}{e^{ru}}.$$

□

Lemma A.2 *Let $X_1, \dots, X_n$ be independent random variables with $X_i \in [0, \dots, k]$ and $m = E[\sum_{i=1}^n X_i]$. Then for $u \geq e^2$:*

$$\Pr \left[\sum_{i=1}^n X_i \geq u \cdot m \right] \leq \exp(-u \frac{m}{k})$$

Proof. Hoeffding [56] showed that for this situation we have

$$\Pr \left[\sum_{i=1}^n X_i \geq (\delta + 1) \cdot m \right] \leq \left(\frac{e^\delta}{(\delta + 1)^{(\delta+1)}} \right)^{\frac{m}{k}}$$

Let $u = \delta + 1$. We have

$$\begin{aligned} \left(\frac{e^\delta}{(\delta + 1)^{(\delta+1)}} \right)^{\frac{m}{k}} &= e^{(u-1)\frac{m}{k}} \cdot u^{-u\frac{m}{k}} \\ &= e^{(u-1)\frac{m}{k}} \cdot e^{\ln u \cdot (-u\frac{m}{k})} \\ &= e^{-\frac{m}{k}(u \ln u - u + 1)} \end{aligned}$$

Finally, we have $u \ln u - u + 1 \geq u$ if $\log u \geq 2$, or $u \geq e^2$. □

Lemma A.3 *Given x balls and y bins. If the balls are randomly and independently distributed to the bins, each bin contains more than lx/y balls with probability at least $1 - e^{-\Omega(l \cdot \frac{x}{y} - \ln y)}$.*

Proof. The probability of the event that a bin receives exactly i balls is

$$\binom{x}{i} \cdot \left(\frac{1}{y} \right)^i \cdot \left(1 - \frac{1}{y} \right)^{x-i} \leq \binom{x}{i} \cdot \left(\frac{1}{y} \right)^i$$

Let X be the event that a bin receives more than k balls, and let Y denote the event that at least one of the bins receives more than $k = l \frac{x}{y}$ balls.

Thus,

$$\Pr[X] = \sum_{i=k+1}^x \binom{x}{i} \cdot \left(\frac{1}{y} \right)^i$$

We can conclude that

$$\begin{aligned}
\Pr[X] &= \sum_{i=k+1}^x \binom{x}{i} \cdot \left(\frac{1}{y}\right)^i \\
&\leq \sum_{i=k}^x \frac{x!}{i!(x-i)!} \frac{1}{y^i} \\
&= \frac{x!}{k!(x-k)!y^k} + \frac{x!}{(k+1)!(x-(k+1))!y^{k+1}} + \frac{x!}{(k+2)!(x-(k+2))!y^{k+2}} + \dots \\
&= \frac{x!}{k!(x-k)!y^k} \cdot \left(1 + \frac{x-k}{(k+1)y} + \frac{(x-k)(x-(k+1))}{(k+1)(k+2)y^2} + \dots\right) \\
&\leq \binom{x}{k} \frac{1}{y^k} \cdot \left(1 + \frac{x-k}{(k+1)y} + \left(\frac{x-k}{(k+1)y}\right)^2 + \dots\right) \\
&\leq \left(\frac{xe}{ky}\right)^k \cdot \sum_{i \geq 0} \left(\frac{x-k}{(k+1)y}\right)^i
\end{aligned}$$

For $k = l\frac{x}{y}$, and $l \leq y$ we note that

$$\frac{x-k}{(k+1)y} = \frac{x - l\frac{x}{y}}{(l\frac{x}{y} + 1)y} = \frac{1 - \frac{l}{y}}{l + \frac{y}{x}} < 1 - \frac{l}{y} < 1$$

and so

$$\sum_{i \geq 0} \left(\frac{x-k}{(k+1)y}\right)^i = \frac{y}{l}$$

Thus, for $k = l\frac{x}{y}$, and $l > e$

$$\begin{aligned}
\Pr[X] &\leq \left(\frac{xe}{ky}\right)^k \cdot \sum_{i \geq 0} \left(\frac{x-k}{(k+1)y}\right)^i \\
&= \left(\frac{e}{l}\right)^{\frac{lx}{y}} \cdot \frac{y}{l} \\
&= \frac{y}{l} \cdot e^{l\frac{x}{y} - l \ln l \frac{x}{y}}
\end{aligned}$$

Recall that Y denotes the event that at least one of the bins receives more than $k = l\frac{x}{y}$ balls.

$$\begin{aligned}
\Pr[Y] = y \cdot \Pr[X] &\leq \frac{y^2}{l} \cdot e^{l\frac{x}{y} - l \ln l \frac{x}{y}} \\
&= e^{l\frac{x}{y} - l \ln l \frac{x}{y} - \ln l + 2 \ln y}
\end{aligned}$$

So for x and y functions of the problem size,

$$\Pr[Y] = e^{-\Omega(l \cdot \frac{x}{y} - \ln y)} \text{ for constant } l \geq e.$$

□

A.2 Balancing Lemmas

Lemma A.4 *Let R be the number of blocks in each bucket. Let $X_{j,k}$ be a random variable representing the number of tracks of disk k that belong to bucket j (a track belongs to bucket j , when it contains a record of bucket j). Then, for any fixed j, k we have the following:*

$$\Pr\left[X_{j,k} \geq l \frac{R}{D}\right] \leq e^{-\Omega(l \cdot \frac{R}{D})}$$

Proof. The proof is similar to one described in [85]. Let g_t denote the number of disks written to from bucket j during write cycle t , for $1 \leq t \leq C$, where C is the total number of write cycles used. We have

$$\sum_{1 \leq t \leq C} g_t \leq R \tag{A.1}$$

Let G_t be the number of tracks belonging to bucket j that are assigned to disk k in write cycle t . Since only one track can be written to any disk in a write cycle, G_t is restricted to the values 0 and 1. We have $\Pr[G_t = 1] = g_t/D$ and $\Pr[G_t = 0] = 1 - g_t/D$. Let $\mathcal{G}_{G_t}(z)$ be the probability generating function for G_t :

$$\begin{aligned} \mathcal{G}_{G_t}(z) &= \Pr[G_t = 0]z^0 + \Pr[G_t = 1]z^1 \\ &= 1 - \frac{g_t}{D} + \frac{g_t}{D}z \\ &= 1 + \frac{g_t}{D}(z - 1) \end{aligned} \tag{A.2}$$

Let $\mathcal{G}_{X_{j,k}}(z)$ be the probability generating function for $X_{j,k}$. We can bound $X_{j,k}$ by the sum of independent random variables: $X_{j,k} \leq G_1 + G_2 + \dots + G_C$. For

purpose of bounding, let us consider that $X_{j,k} = G_1 + G_2 + \dots + G_C$. Since the sum of independent random variables corresponds to the product of the corresponding probability generating functions and using (A.2), we have

$$\begin{aligned} \mathcal{G}_{X_{j,k}}(z) &= \mathcal{G}_{G_1+G_2+\dots+G_C}(z) \\ &= \mathcal{G}_{G_1}(z) \times \mathcal{G}_{G_2}(z) \times \dots \times \mathcal{G}_{G_C}(z) \\ &= \prod_{1 \leq t \leq C} (1 + \frac{g_t}{D}(z - 1)) \end{aligned} \quad (\text{A.3})$$

By the tail estimate Lemma A.1 we have

$$\Pr \left[X_{j,k} \geq l \frac{R}{D} \right] \leq \frac{E[\exp(r X_{j,k})]}{\exp(r l R/D)} \quad (\text{A.4})$$

for each $r \geq 0$. We can express the numerator in (A.4), using (A.3) and the definitions of expected value and probability generating function, as

$$\begin{aligned} E[e^{r X_{j,k}}] &= \sum_{t \geq 0} \Pr[e^{r X_{j,k}} = e^{rt}] e^{rt} \\ &= \sum_{t \geq 0} \Pr[X_{j,k} = t] e^{rt} \\ &= \mathcal{G}_{X_{j,k}}(e^r) \\ &= \prod_{1 \leq t \leq C} ((1 + \frac{g_t}{D}(e^r - 1))) \end{aligned} \quad (\text{A.5})$$

By (A.1) and convexity arguments, we can maximize (A.5) by setting $g_t = R/C$ for each t . Thus

$$\begin{aligned} E[e^{r X_{j,k}}] &\leq \prod_{1 \leq t \leq C} \left(1 + \frac{R \cdot (e^r - 1)}{DC} \right) \\ &= \left(1 + \frac{R \cdot (e^r - 1)}{DC} \right)^C \end{aligned}$$

Substituting this bound into (A.4), we get

$$\Pr \left[X_{j,k} \geq l \frac{R}{D} \right] \leq \frac{(1 + R(e^r - 1)/DC)^C}{\exp(r l R/D)} \quad (\text{A.6})$$

From the bound $(1 + a)^b \leq e^{ab}$, for $a > -1$, we can approximate the numerator in (A.6) and get for $r = \ln l$

$$\Pr[X_{j,k} \geq l \frac{R}{D}] \leq \frac{\exp(R \cdot (e^r - 1)/D)}{\exp(r l R/D)}$$

$$\begin{aligned}
&= \exp\left(\frac{R \cdot (e^r - 1) - rlR}{D}\right) \\
&= \exp\left(\frac{R}{D} \cdot (e^{lnl - l \cdot lnl - 1})\right)
\end{aligned} \tag{A.7}$$

Now, since

$$l \cdot lnl - l + 1 \geq l \tag{A.8}$$

for approximately $l \geq 6.4$, and since R and D are functions of the problem size,

$$\Pr[X_{j,k} \geq l \frac{R}{D}] \leq e^{-\Omega(l \cdot \frac{R}{D})}$$

for constant $l \geq 6.4$. □

Lemma A.5 allows us to exploit the independence of the random experiments performed during each compound superstep in order to prove that the success probability for the entire simulation is as large as for the simulation of a single compound superstep.

Lemma A.5 *Let $X_1, X_2, \dots, X_z$ be independent random variables such that $\Pr[X_i \leq lT] \geq 1 - \exp(-l \log l \cdot m)$, and $\Pr[X_i > lT] \leq \exp(-l \log l \cdot m)$, where $m \geq \ln x$ and $l \geq 2$, for $1 \leq i \leq z$. Let $T_{wc} \leq xT$ be the worst-case size of X_i for any i , that is $X_i \leq T_{wc}$, for $1 \leq i \leq z$. Then,*

$$\Pr\left[\sum_{i=1}^z X_i \leq e^2(l+1)zT\right] \geq 1 - e^{-\Omega(l \log l \cdot m)}$$

Proof. We identify two cases: (1) $z \geq xm^c$ and (2) $z \leq xm^c$ for $c = 1 + \frac{\log \log l}{\log m}$.

First, we consider Case (1). The mean of the quantity $\sum_{i=1}^z X_i$ can be bounded from above as follows for suitable constant $l \geq 2$ and $m \geq \ln x$:

$$\begin{aligned}
E\left[\sum_{i=1}^z X_i\right] &\leq \sum_{i=1}^z (lT \cdot \Pr[X_i \leq lT] + T_{wc} \cdot \Pr[X_i > lT]) \\
&\leq zlT(1 - \exp(-l \log l \cdot m)) + zxT \exp(-l \log l \cdot m) \\
&\leq zlT + zT \exp(-l \log l \cdot m + \ln x) \\
&\leq zlT((1 - e^{-l \log l \cdot m})) + zxT e^{-l \log l \cdot m} \\
&\leq zlT + zT e^{-l \log l \cdot m + \ln x} \\
&\leq (l+1)zT
\end{aligned} \tag{A.9}$$

We can bound the mean $E[\sum_{i=1}^z X_i]$ from below as follows:

$$\begin{aligned} E\left[\sum_{i=1}^z X_i\right] &\geq \sum_{i=1}^z (\Pr[X_i \leq lT] \cdot lT + \Pr[X_i > lT] \cdot T) \\ &\geq (l-1)zT \end{aligned} \tag{A.10}$$

From Lemma A.2 (Hoeffding), using $k = T_{wc} \leq xT$, $m = E[\sum_{i=1}^z X_i] \geq (l-1)zT$, and $z \geq xm^c$:

$$\begin{aligned} \Pr\left[\sum_{i=1}^z X_i \geq e^2 m\right] &\leq \exp\left(-e^2 \frac{(l-1)zT}{xT}\right) \\ \Pr\left[\sum_{i=1}^z X_i \geq e^2(l-1)zT\right] &\leq \exp(-(l-1)m^c) \end{aligned}$$

So,

$$\Pr\left[\sum_{i=1}^z X_i \geq e^2(l+1)zT\right] \leq e^{-(l-1)\log l \cdot m}$$

and,

$$\Pr\left[\sum_{i=1}^z X_i \leq e^2(l+1)zT\right] \geq 1 - e^{-(l-1)\log l \cdot m}$$

Now, we consider Case (2). We repeat the experiment only $z \leq xm^c$ times, where $c = 1 + \frac{\log \log l}{\log m}$. Thus we have with $m \geq \ln x$:

$$\begin{aligned} \Pr\left[\sum_{i=1}^z X_i \leq zlT\right] &\geq (1 - \exp(-l \log l \cdot m))^z \\ &\geq 1 - xm^c \exp(-l \log l \cdot m) \\ &\geq 1 - \exp(-l \log l \cdot m + c \ln m + \ln x) \\ &\geq 1 - e^{-l \log l \cdot m} \end{aligned}$$

So,

$$\Pr\left[\sum_{i=1}^z X_i \leq z(l+1)T\right] \geq 1 - e^{-l \log l \cdot m}$$

□

A.3 Summary of Notation and Index

Symbol	Meaning
b	the communication block size
B	the disk block size
D	the number of disk drives on a real processor
$\hat{g}$	the time required for the router to deliver a packet of size b (see BSP* Model)
g	the time required for the router to deliver a packet of unit size (see BSP model)
G	the time required for a processor to transfer D blocks between its local disks and its local memory
k	the number of virtual processors simulated concurrently by a single real processor.
L	the minimum time required for synchronizing the processors
M	the memory size of a (real) processor
N	the number of data items in the problem
p	the number of real processors
v	the number of virtual processors
$\bar{v}$	$\frac{v}{k}$
γ	the maximum communication volume sent or received by a virtual processor in a single superstep
λ	the number of <i>BSP</i> supersteps in an algorithm
μ	the maximum size of the context of a virtual processor

Index

γ , 48, 49

μ , 48, 49

c -optimal, 38

h -relation, 86, 89

Algorithms

BalancedRouting, 72, 72, 76,
77, 86, 88, 89, 91, 98, 107,
112, 113

CGMPermute, 125

CGMSampleSort, 124, 168

CGMTranspose, 77, 126

EMLowSLackRouting, 107

EMLowSlackRouting, 91, 92, 101,
102, 102, 103, 106–108, 110,
111, 113

EMNumberBlocks, 98, 101, 102,
104, 105, 105, 106, 107, 111

EMReceive, 103

EMSend, 103

LowSlackRouting, 89–92, 92, 93–
102, 112, 125

NumberBlocks, 93, 93, 95, 97–
101, 103–107, 112

ParCompoundSS-4.3, 63

ParCompoundSuperstep, 53

ParCompoundSuperstep-5.3, 83

ParCompoundSuperstep-6.7, 112

SeqCompoundSS-4.1, 56

SeqCompoundSuperstep, 50, 118

SeqCompoundSuperstep-5.2, 78

SimulateRouting, 58

batch, 50

batch filtering, 12, 14

block, 8

blocking, 23

blocks, 2

BSP, iii, 18, 22, 22, 23, 26–28, 30,
31, 36–44, 47, 48, 50–53, 55,
71, 74, 76, 79, 87, 101, 122,
124, 128–130, 135, 139, 160,
162, 163, 165, 174, 183

BSP*, 4, 22, 23, 26–28, 30, 31, 38–
41, 44, 47, 52, 53, 55–58,
61–63, 66, 68–71, 74, 76, 87,
91, 129–135

BSP-like, iii, 5, 22, 40, 41, 74, 77,
116, 120, 121, 135, 173

bucket, 58

buffer tree, 12, 12, 13, 14, 16, 17,
21, 24, 25, 28, 142, 143, 143,

- 144, 146–158
- CGM, 4, 21–24, 28, 30, 31, 40, 41, 44, 45, 47, 55, 71, 74, 76–78, 81, 83–87, 91, 112, 113, 120, 122, 123, 126, 127, 130, 131, 139, 140, 159, 161, 166, 168, 170, 171
- communication, 87
- communication-efficient, 46
- compound superstep, 48
- computation phase, 48
- computation supersteps, 36
- consecutive format, 8, 56, 78
- consecutive I/O, 8
- context, 48
- context size, 48
- Corollaries
 - 5.1, 74
 - 8.1, 131
- cylinder, 7
- declustering, 6
- disk storage block, 7, 8
- disk tracks, 7
- distribution sweeping, 12
- EM-BSP, iii, 21, 25–28, 30, 31, 42, 42, 43–47, 50, 52, 53, 55, 71, 91, 101, 117, 122, 126–128, 130, 134, 139, 140, 146, 160, 163, 173, 174
- parameters
 - B , 42
 - D , 42
 - G , 42
 - M , 42
 - p , 43
 - v , 43
- EM-BSP*, iii, 25, 27, 28, 30, 31, 42, 44, 55, 56, 61–63, 66, 68–71, 87, 122, 130–135, 146
- EM-CGM, iii, 25, 28, 30, 31, 42, 44, 45, 55, 71, 77, 78, 81, 83–87, 112, 113, 118, 120–124, 126, 130, 131, 134, 136, 146, 159, 168–171
- fanout, 90
- fetching phase, 48
- h-relation, 36
- I/O-aware, 5
- I/O-efficient, 46
- I/O-optimal, 32, 46
- I/O-optimality, 3
- independent format, 9
- independent I/O, 8, 20
- LEDA, 142, 147–149
- LEDA-SM, 16, 142
- Lemmas
 - 4.1, 59
 - 4.2, 60
 - 4.3, 61

- 4.4, 64
- 4.5, 66
- 5.1, 76
- 5.2, 76
- 5.3, 77
- 5.4, 84
- 6.11, 108
- 6.12, 110
- 6.13, 111
- 6.1, 96
- 6.2, 97
- 6.3, 97
- 6.4, 98
- 6.5, 99
- 6.6, 100
- 6.7, 100
- 6.8, 106
- 6.9, 107
- 7.1, 116
- 7.2, 118
- 7.3, 120
- 8.1, 123
- 9.1, 169
- 9.2, 169
- 9.3, 169
- A.5, 181
- linked format, 56, 58
- LogP model, 41
- memory-adaptive, 173
- MPI, 159
- MTBF, 20
- multisearch, 14
- Observations
 - 5.1, 73
- optimal
 - c-optimality of EM-BSP, 45
- parallel format, 56, 77
- parallel I/O, 8
- PDM, iii, 3, 5, 11, 12, 21, 26, 30–32, 34, 41, 124, 125, 127, 139, 140, 146, 155, 157
- PRAM simulation, 12
- RAID, 20
- receiving superstep, 48
- rotational delay, 7
- round-robin order, 91, 98, 101, 107, 109, 112
- sectors, 7
- seek time, 7
- sending superstep, 48
- slackness, 5, 87–89, 113, 114, 174
- staggered format, 9, 78
- staggered I/O, 8
- stripe, 8
- striped format, 9, 56
- striped I/O, 8, 20
- striping unit, 6
- supersteps, 36
- Theorems
 - 4.1, 62

- 4.2, 68
- 5.1, 73
- 5.2, 81
- 5.3, 85
- 5.4, 86
- 6.1, 113
- 7.1, 120
- 7.2, 121
- 8.1, 126
- 8.2, 130
- 8.3, 131
- 8.4, 132
- 8.5, 133
- time-forward processing, 12
- TPIE, 13, 142, 143, 148, 151, 152,
154
- virtual processors, 36, 48
- work-optimal, 46
- writing phase, 49

Bibliography

- [1] P. Agarwal, L. Arge, M. Murali, K. Varadarajan, and J. Vitter. I/O-efficient algorithms for contour-line extraction and planar graph blocking. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1998.
- [2] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proc. ACM Symp. on Theory of Computation*, pages 305–314, 1987.
- [3] A. Aggarwal, A. K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 204–216, 1987.
- [4] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
- [5] R. Ahulja, K. Mehlhorn, J. Orlin, and R. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 3(2):213–223, 1990.
- [6] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. BSP vs LogP. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 95–105, 1995.
- [7] B. Alpern, L. Carter, E. Feig, and Selker. The uniform memory hierarchy model of computation. *Algorithmica*, 12:72–109, 1994.
- [8] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995.

- A complete version appears as BRICS technical report RS-96-28, University of Aarhus.
- [9] L. Arge. *Efficient External-Memory Data Structures and Applications*. PhD thesis, University of Aarhus, February/August 1996.
 - [10] L. Arge, M. Knudsen, and K. Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 709*, pages 83–94, 1993.
 - [11] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Proc. Annual European Symposium on Algorithms, LNCS 979*, pages 295–310, 1995. A complete version (to appear in special issue of *Algorithmica*) appears as BRICS technical report RS-96-12, University of Aarhus.
 - [12] M. Atallah, F. Dehne, R. Miller, A. Rau-Chaplin, and J.-J. Tsay. Multisearch techniques for implementing data structures on a mesh-connected computer. *Journal of Parallel and Distributed Computing*, 20:1–13, 1994.
 - [13] D. Bader, D. Helman, and J. Jájá. Practical parallel algorithms for personalized communication and integer sorting. *Journal of Experimental Algorithmics*, 1, 1996. <http://www.jea.acm.org/1996/BaderPersonalized/>.
 - [14] R. Barve and J. S. Vitter. External memory algorithms with dynamically changing memory allocation. Technical Report CS199809, Duke University, June 1998.
 - [15] R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. *Parallel Computing*, 23(4):601–631, June 1997.
 - [16] A. Bäumer and W. Dittrich. Fully dynamic search trees for an extension of the bsp model. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 233–242, 1996.

- [17] A. Bäumker, W. Dittrich, and F. M. auf der Heide. Efficient parallel algorithms: c-optimal multisearch for an extension of the bsp model. In *Proc. Annual European Symposium on Algorithms*, pages 17–30, 1995.
- [18] G. Bilardi, K. T. Herley, A. Pietracprina, G. Pucci, and P. Spirakis. BSP vs LogP. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 25–32, 1996.
- [19] G. Brassard and P. Bratley. *Algorithmics: Theory and Practice*. Prentice Hall, Englewood Cliffs, New Jersey, 1988.
- [20] E. Cáceres, F. Dehne, A. Ferreira, P. Flocchini, I. Reiping, N. Santoro, and S. Song. Efficient parallel graph algorithms for coarse grained multicomputers and bsp. In *Proc. Int. Colloquium Algorithms, Languages and Programming, LNCS 1256*, pages 390–400, 1997.
- [21] A. Chan, F. Dehne, and A. Rau-Chaplin. Coarse grained parallel next element search. In *Proc. International Parallel Processing Symposium*, pages 320–325, 1997.
- [22] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.
- [23] Y.-J. Chiang. *Dynamic and I/O-Efficient Algorithms for Computational Geometry and Graph Problems: Theoretical and Experimental Results*. PhD thesis, Brown University, August 1995.
- [24] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 346–357, 1995.
- [25] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995.

- [26] A. Colvin and T. H. Cormen. ViC*: A Compiler for Virtual-Memory C*. Technical Report PCS-TR97-323, Dartmouth College, Computer Science, Hanover, NH, November 1997.
- [27] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
- [28] T. H. Cormen. *Virtual Memory for Data Parallel Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992.
- [29] T. H. Cormen. Fast permuting on disk arrays. *Journal of Parallel and Distributed Computing*, 17(1-2):41–57, 1993.
- [30] T. H. Cormen and M. T. Goodrich. Position Statement, ACM Workshop on Strategic Directions in Computing Research: Working Group on Storage I/O for Large-Scale Computing. *ACM Computing Surveys*, 28A(4), Dec. 1996.
- [31] T. H. Cormen and M. Hirschl. Early Experiences in Evaluating the Parallel Disk Model with the ViC* Implementation. *Parallel Computing*, 23(4-5):571–600, 1997.
- [32] T. H. Cormen and D. Kotz. Integrating theory and practice in parallel file systems. In *Proceedings of the 1993 DAGS/PC Symposium*, pages 64–74, Hanover, NH, June 1993. Dartmouth Institute for Advanced Graduate Studies. Revised as Dartmouth PCS-TR93-188 on 9/20/94.
- [33] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Mass., 1990.
- [34] T. H. Cormen and D. M. Nicol. Performing Out-of-Core FFTs on Parallel Disk Systems. *Parallel Computing*, 24(1):5–20, 1998.
- [35] T. H. Cormen, T. Sundquist, and L. F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. *SIAM Journal of Computing*, 28(1):105–136, 1999.

- [36] T. H. Cormen, J. Wegmann, and D. M. Nicol. Multiprocessor Out-of-Core FFTs with Distributed Memory and Parallel Disks. Technical Report PCS-TR97-303, Dartmouth College, Computer Science, Hanover, NH, January 1997.
- [37] A. Crauser, K. Mehlhorn, E. Althaus, K. Brengel, T. Buchheit, J. Keller, H. Krone, O. Lambert, R. Schulte, S. Theil, M. Westphal, and R. Wirth. On the performance of LEDA-SM. Technical Report MPII981028, MaxPlanckInstitut für Informatik, Nov. 1998.
- [38] D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauer, R. Subramonian, and T. von Eicken. Logp: A practical model of parallel computation. *Communications of the ACM*, 39(11):260–270, Nov. 1996.
- [39] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. Khokhar. A randomized parallel 3d convex hull algorithm for coarse grained multicomputers. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 27–33, 1995.
- [40] F. Dehne, W. Dittrich, and D. Hutchinson. Efficient external memory algorithms by simulating coarse-grained parallel algorithms. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 106–115, 1997.
- [41] F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari. Parallel virtual memory. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 889–890, 1998.
- [42] F. Dehne, W. Dittrich, D. Hutchinson, and A. Maheshwari. Reducing I/O complexity by simulating coarse grained parallel algorithms. In *Proc. International Parallel Processing Symposium*, pages 14–20, 1999.
- [43] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6:379–400, 1996.
- [44] F. Dehne and A. Rau-Chaplin. Implementing data structures on a hypercube multiprocessor, and applications in parallel computational geometry. *Journal of Parallel and Distributed Computing*, 8:367–375, 1990.

- [45] E. W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 1–100. Academic Press, 1968.
- [46] W. Dittrich. *Communication and I/O Efficient Parallel Data Structures*. PhD thesis, University of Paderborn, 1997.
- [47] W. Dittrich, D. Hutchinson, and A. Maheshwari. Blocking in parallel multisearch problems. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, 1998.
- [48] H. Edelsbrunner and M. Overmars. Batched dynamic solutions to decomposable searching problems. *Journal of Algorithms*, 6:515–542, 1985.
- [49] R. W. Floyd. Permuting information in idealized two-level storage. In *Complexity of Computer Calculations*, pages 105–109, 1972. R. Miller and J. Thatcher, Eds. Plenum, New York.
- [50] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. ACM Symp. on Theory of Computation*, pages 114–118, 1978.
- [51] A. Gerbessiotis and C. Siniolakis. Communication Efficient Data Structures on the BSP Model with Applications in Computational Geometry. In *Proceedings of EUROPAR'96*, August 1996.
- [52] A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.
- [53] M. Goodrich. Communication efficient parallel sorting. In *Proc. ACM Symp. on Theory of Computation*, 1996.
- [54] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 714–723, 1993.
- [55] H. Hellwagner. Design considerations for scalable parallel file systems. *The Computer Journal*, 36(8):741–755, 1993.

- [56] W. Hoeffding. Probability inequalities for sums of bounded random variables. *American Statistical Association Journal*, pages 13–30, 1963.
- [57] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.
- [58] D. Hutchinson, A. Maheshwari, J.-R. Sack, and R. Velicescu. Early experiences in implementing the buffer tree. In *Proceedings of the Workshop on Algorithm Engineering*, 1997. <http://www.dsi.unive.it/~wae97/proceedings>.
- [59] D. Knuth. *The Art of Computer Programming, Vol. 3 Sorting and Searching*. Addison-Wesley, 1973.
- [60] LAM MPI. <http://www.osc.edu/lam.html>.
- [61] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag, EATCS Monographs on Theoretical Computer Science, 1984.
- [62] K. Mehlhorn and S. Näher. LEDA: A platform for combinatorial and geometric computing. *Communications of the ACM*, 38:96–102, 1995.
- [63] Message Passing Interface Standards web site. <http://www.mpi-forum.org/>.
- [64] K. Munagala and A. Ranade. I/O complexity of graph algorithms. *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 687–694, 1999.
- [65] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.
- [66] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 120–129, 1993.
- [67] M. H. Nodine and J. S. Vitter. Paradigms for optimal sorting with multiple disks. In *Proc. of the 26th Hawaii Int. Conf. on Systems Sciences*, 1993.

- [68] M. H. Nodine and J. S. Vitter. Greed sort: Optimal deterministic sorting on parallel disks. *Journal of the ACM*, 42(4):919–933, 1995.
- [69] Y. N. Patt. The I/O subsystem—a candidate for improvement. *Guest Editor's Introduction in IEEE Computer*, 27(3):15–16, 1994.
- [70] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). *Proc. ACM Conf. on Management of Data*, pages 109–116, 1988.
- [71] W. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2-3 trees. In *Proc. Int. Colloquium Algorithms, Languages and Programming, LNCS 154*, pages 597–604, 1983.
- [72] J. T. Poole. Preliminary survey of I/O intensive applications. Technical Report CCSF-38, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.
- [73] A. Rau-Chaplin. *On Parallel Data Structures and Parallel Geometric Applications for Multicomputers*. PhD thesis, Carleton University, 1992.
- [74] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.
- [75] Scalable I/O Initiative, Concurrent Supercomputing Consortium. <http://www.cacr.caltech.edu/SIO/>.
- [76] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992.
- [77] J. Sibeyn and M. Kaufmann. Bsp-like external-memory computation. In *Proc. 3rd Italian Conference on Algorithms and Complexity, LNCS 1203*, pages 229–240, 1997.
- [78] R. W. Stevens. *Advanced Programming in the Unix Environment*. Addison-Wesley, Don Mills, Ont., Canada, 1995.

- [79] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.
- [80] D. E. Vengroff. A transparent parallel I/O environment. In *Proc. 1994 DAGS Symposium on Parallel Computation*, 1994.
- [81] D. E. Vengroff. *TPIE User Manual and Reference*. Duke University, 1995. Available via WWW at <http://www.cs.duke.edu/~dev>.
- [82] D. E. Vengroff and J. S. Vitter. Supporting I/O-efficient scientific computation in TPIE. In *Proc. IEEE Symp. on Parallel and Distributed Computing*, 1995. Appears also as Duke University Dept. of Computer Science technical report CS-1995-18.
- [83] D. E. Vengroff and J. S. Vitter. Efficient 3-d range searching in external memory. In *Proc. ACM Symp. on Theory of Computation*, pages 192–201, 1996.
- [84] J. S. Vitter. External memory algorithms. *Proc. ACM Symp. Principles of Database Systems*, pages 119–128, 1998.
- [85] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.
- [86] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, II: Hierarchical multilevel memories. *Algorithmica*, 12(2–3):148–169, 1994.
- [87] K. Yogaratnam. CGM Algorithms as EM-CGM Algorithms. Technical report, Carleton University, School of Computer Science, Ottawa, Ontario, Canada, 1998. Undergraduate Honours Project.
- [88] N. Zeh. An external-memory data structure for shortest path queries. Master’s thesis, Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena, Germany, <http://www.scs.carleton.ca/~nzeh/Publications/thesis.ps.gz>, November 1998.