1

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

For sale - Vente libre

Outlets - Points de vente

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

On Parallel Data Structures and Parallel Geometric Applications for Multicomputers

by

Andrew Rau-Chaplin, B.A., M.C.S.

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

**Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario
November 1 1992**

**© Copyright
1992, Andrew Rau-Chaplin, B.A., M.C.S.**

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

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

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

ISBN 0-315-84037-4

Canada

Name

Andrew Rawn-Christie

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Computer Science

0984

U-M-I

SUBJECT TERM

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
General 0289
Ancient 0290
Linguistics 0291
Modern 0291
Literature 0401
General 0294
Classical 0295
Comparative 0297
Medieval 0298
Modern 0316
African 0591
American 0305
Asian 0352
Canadian (English) 0355
Canadian (French) 0593
English 0311
Germanic 0312
Latin American 0315
Middle Eastern 0313
Romance 0314
Slavic and East European 0370

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion 0318
General 0321
Biblical Studies 0319
Clergy 0320
History of 0322
Philosophy of 0469
Theology 0323

SOCIAL SCIENCES

American Studies 0323
Anthropology 0324
Archaeology 0326
Cultural 0327
Physical 0310
Business Administration 0272
General 0770
Accounting 0454
Banking 0338
Management 0385
Marketing 0501
Canadian Studies 0503
Economics 0505
General 0508
Agricultural 0509
Commerce Business 0510
Finance 0511
History 0358
Labor 0366
Theory 0351
Folklore 0578
Geography 0366
Gerontology 0351
History 0578
General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science 0615
General 0616
International Law and 0617
Relations 0614
Public Administration 0452
Recreation 0626
Social Work 0627
Sociology 0936
General 0631
Criminology and Penology 0628
Demography 0629
Ethnic and Racial Studies 0630
Individual and Family 0700
Studies 0344
Industrial and Labor 0629
Relations 0630
Public and Social Welfare 0709
Social Structure and 0344
Development 0709
Theory and Methods 0999
Transportation 0453
Urban and Regional Planning 0453
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
General 0285
Agronomy 0475
Animal Culture and 0476
Nutrition 0359
Animal Pathology 0478
Food Science and 0479
Technology 0480
Forestry and Wildlife 0817
Plant Culture 0777
Plant Pathology 0746
Plant Physiology 0306
Range Management 0287
Wood Technology 0308
Biology 0309
General 0379
Anatomy 0329
Biostatistics 0353
Botany 0369
Cell 0793
Ecology 0410
Entomology 0307
Genetics 0317
Limnology 0416
Microbiology 0433
Molecular 0821
Neuroscience 0778
Oceanography 0472
Physiology 0786
Radiation 0760
Veterinary Science 0425
Zoology 0998
Biophysics 0425
General 0998
Medical 0425

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0998

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences 0566
General 0300
Audiology 0992
Chemotherapy 0567
Dentistry 0350
Education 0769
Hospital Management 0758
Human Development 0982
Immunology 0564
Medicine and Surgery 0347
Mental Health 0569
Nursing 0570
Nutrition 0380
Obstetrics and Gynecology 0354
Occupational Health and 0381
Therapy 0571
Ophthalmology 0419
Pathology 0572
Pharmacology 0382
Pharmacy 0573
Physical Therapy 0574
Public Health 0575
Radiology 0575
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences 0485
Chemistry 0749
General 0486
Agricultural 0487
Analytical 0488
Biochemistry 0738
Inorganic 0490
Nuclear 0491
Organic 0494
Pharmaceutical 0495
Physical 0754
Polymer 0405
Radiation 0605
Mathematics 0986
Physics 0606
General 0608
Acoustics 0748
Astronomy and 0607
Astrophysics 0798
Atmospheric Science 0759
Atomic 0609
Electronics and Electricity 0610
Elementary Particles and 0752
High Energy 0756
Fluid and Plasma 0611
Molecular 0463
Nuclear 0346
Optics 0984
Radiation 0346
Solid State 0984
Statistics 0346
Applied Sciences 0984
Applied Mechanics 0346
Computer Science 0984

Engineering 0537
General 0538
Aerospace 0539
Agricultural 0540
Automotive 0541
Biomedical 0542
Chemical 0543
Civil 0544
Electronics and Electrical 0545
Heat and Thermodynamics 0546
Hydraulic 0547
Industrial 0794
Marine 0548
Materials Science 0743
Mechanical 0551
Metallurgy 0552
Mining 0549
Nuclear 0765
Packaging 0554
Petroleum 0790
Sanitary and Municipal 0428
System Science 0796
Geotechnology 0795
Operations Research 0795
Plastics Technology 0994
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

Carleton University
Ottawa, Canada K1S 5J7

50 1912 1992

Thesis contains black & white
illustrations which when microfilmed
may lose their significance. The
hardcopy of the thesis is available
upon request from Carleton
University Library.

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis,
**On Parallel Data Structures
and Parallel Geometric Applications
for Multicomputers**
submitted by
Andrew Rau-Chaplin
in partial fulfilment of the requirements
for the degree of Doctor of Philosophy

Director, School of Computer Science

Thesis Supervisor

External Examiner

Carleton University

April 15 1993

To Michelle

Abstract

This thesis focuses on the application of parallelism to data structuring and geometric problems.

Data structure abstractions have played a vital role in the development of efficient sequential algorithms. We show that for a wide range of problems, parallel versions of standard sequential data structures can be constructed to yield highly efficient parallel algorithms for distributed memory multicomputers. In the context of parallel data structures, we study problems from computational geometry, artificial intelligence, image processing and solid modeling.

Geometric problems are of considerable importance in robotics, solid modeling, vision, image processing, and pattern recognition. In many cases, due to the computationally intensive nature of these problems the application of parallelism is of both practical and theoretical interest. In this thesis we study geometric problems including planar visibility, dominance, nearest neighbor and convex hull problems and both uni-directional and multi-dimensional geometric separability problems. We also study optical clustering and the construction and manipulation of multidimensional binary images.

Our focus is on algorithms for distributed memory multicomputers. In particular we give algorithms for mesh, hypercube and coarse-grained architectures. PRAM algorithms are also given when they arise from distributed memory algorithms. In addition to standard asymptotic analysis, this thesis also reports on implementations of several of the proposed algorithms.

Acknowledgments

I would like to thank all those individuals who have contributed, both personally and professionally, to the completion of this thesis. In particular I would like to thank Professor Frank Dehne for his support and guidance. He has been the best of supervisors, both patient and demanding. I would also like to thank my Ph.D. committee, Professors Nicola Santoro and Lou Birta for their support. Professor Selim Akl acted as my external examiner. His clear thinking and eye for detail greatly improved the presentation of this thesis.

Many colleagues and friends have helped me make it through these last four years. In particular I would like to thank, Andreas Fabri for his insightful conversation, Dr. Afonso Ferreira for his counter-examples, Alan Knight for listening, Dr. Russ Miller for his encouragement, and Una-May O'Reilly and Dwight Duego for their comradeship and sense of competition.

Implementing algorithms on parallel machines can be a time consuming and sometimes frustrating experience. I am grateful to the following people, all of whom have worked with me on implementation projects, both directly and indirectly, related to this thesis: Alain Bonopera, Kevin Bourgault, Ralph Cheung, Vincent Ho, Andrew Leung, Henry Lin, Joel Lucuik, Lin Nguyen, Anton Salimaki, Curtis Spencer, Dan Villeneuve, Desmond Yeong, and Mark Wineberg.

This thesis was written using that marvelous and somewhat mysterious typesetting system called $\LaTeX$. This is a feat I would not have attempted if it were not for the support and continual assistance of our local $\TeX$ gurus, Radha and Natana

Valiveti. If any part of this document looks beautiful it should be attributed to their efforts.

I've been very lucky to have many fine teachers. I'd like to thank the following people who taught me computer science and mathematics in both formal and informal settings: Mike Atkinson, Frank Dehne, Frank Fiala, Hugo, Margaret Kerr-Grun, Franz Oppacher, Randy Osbourne, Peter Rossen-Runge, Jorg-Rudiger Sack, Nicola Santoro and Dave Thomas.

Carleton University's School of Computer Science has been a wonderful environment in which to work and study. It is the least bureaucratic and most supportive setting I have ever had the pleasure of working in. Much of the credit must go to its director, Professor John Pugh, its graduate supervisor, Professor Dandamudi Sivarama and its administrator, Rosemary Carter. I would also like to thank the administrative and technical support staff for putting up with my near constant stream of requests with great humor and competence.

I gratefully acknowledge the financial support of scholarships from Bell-Northern Research, the National Science and Engineering Research Council, Carleton University's Faculty of Graduate Studies and the Ontario Government.

Lastly, I would like to thank my wife Michelle for her unwavering love and support. This thesis would not have been possible without her.

Contents

Abstract	iv
Acknowledgments	v
List Of Figures	xii
List Of Tables	xv
0 Introduction	1
0.1 Parallel Data Structures	2
0.2 Parallel Computational Geometry	5
0.3 Overview of Results	6
I Algorithms for Fine-Grained Mesh Multicomputers	12
1 The Mesh Multicomputer: Definitions and Basic Operations	13
1.0.1 Standard Mesh Operations	14
2 Multisearch Techniques: Parallel Data Structures on a Mesh	15
2.1 Definitions	18
2.1.1 The Multisearch Problem	18
2.1.2 Partitionable Graphs	20

2.2	Mesh Solutions to the Multisearch Problem	22
2.2.1	Constrained Multisearch	23
2.2.2	The Multisearch Problem for Directed α -Partitionable Graphs	26
2.2.3	The Multisearch Problem for Undirected α - β -Partitionable Graphs	29
2.3	Implementing Parallel Data Structures on a Mesh-Connected Computer	31
2.4	Conclusion	33
3	Optical Clustering on a Mesh	34
3.1	Optical Clustering	36
3.2	Parallel Optical Clustering of Points Sets in E^2	39
3.2.1	Computing Optical Clusters with Respect to a Given Separation Parameter r	40
3.2.2	Computing Optical Clusters with Respect to a Given Range in the Number of Clusters Desired	41
3.3	Optical Clustering for Chain Code Representations	45
3.4	Conclusion	46
II	Algorithms for Fine-Grained Hypercube Multicomputers	47
4	The Hypercube Multicomputer: Definitions and Basic Operations	48
4.1	The Hypercube Multicomputer	48
4.1.1	Standard Hypercube Operations	49
4.1.2	Sorting on the Hypercube	51
4.1.3	Multi-Way Search on Trees and General Graphs	51

5	Parallel Construction and Manipulation of Quadrees on a Hypercube	55
5.1	Storing Pointer Based Quadrees on a Hypercube Multicomputer . . .	59
5.2	Constructing Quadrees from Binary Images and Boundary Codes . .	60
5.2.1	Quadtree from Binary Image	60
5.2.2	Quadtree from Boundary Code	62
5.3	Operations on Quadrees	67
5.3.1	Finding Neighbors in Quadrees and Computing Region Properties	67
5.3.2	Rotating Quadrees By 90°	68
5.3.3	Forming the Union, Intersection, and Complement of Quadrees	69
5.4	Conclusion	74
6	Constructing d-dimensional Hyperoctrees from Slices on a Hypercube	75
6.1	Hyperoctrees and Hybrid Trees	76
6.2	Sequential Algorithm	78
6.3	Merging Hybrid Trees	78
6.4	Parallel Algorithm for Linear Hyperoctrees	80
6.5	Parallel Algorithm for Pointer-Based Hyperoctrees	85
6.6	Conclusion	86
7	A Massively Parallel Knowledge-Base Server on a Hypercube	88
7.1	Frame-Based Knowledge Representation	90
7.2	A Parallel Frame Based Knowledge Server Supporting Single Inheritance Hierarchies	91
7.2.1	Storing an Inheritance Hierarchy on a Hypercube Multicomputer	92
7.2.2	Answering Multiple bottom-up Inference Queries	92
7.2.3	Answering Multiple Top-Down Inference Queries	95

7.2.4	Analysis and Experimental Results	98
7.2.5	Dynamic Knowledge Representation: Supporting Assert and Retract Queries	101
7.3	Conclusion	102

III Algorithms for Coarse-Grained Multicomputers 103

8 Coarse-Grained Multicomputers: Definition and Basic Operations 104

8.1	Primitive (CGM) Operations	106
8.2	Algorithm Design for CGM: Asymptotic Optimality	109
8.3	Algorithm Design for CGM: Practical Considerations	111

9 Scalable Parallel Geometric Algorithms for Coarse-Grained Multicomputers 113

9.1	Preliminaries	114
9.2	2D-Visibility from a Point	116
9.3	2D-Weighted Dominance Counting	119
9.4	3D-Maxima	122
9.5	All Nearest Neighbors	125
9.6	Convex Hull in the plane	128
9.7	Experimental Results	134
9.8	Conclusion	136

10 Data Structures on Coarse-Grained Multicomputer with Applications to Geometric Separability Problems 137

10.1	CGM Multisearch: Parallel Data Structures on a CGM	138
10.1.1	The Multisearch Problem for balanced k -ary trees	138
10.1.2	Solving the Multisearch Problem on a CGM	139
10.1.3	CGM Multisearch with Dynamic Node Values	143

10.2	Segment Trees using CGM Multisearch	144
10.2.1	Parallel Segment Trees	145
10.2.2	The Multiple Stabbing Problem	146
10.3	The Uni-Directional Separability Problem	147
10.3.1	Definitions and Notation	148
10.3.2	Review of Sequential Literature	149
10.3.3	Solving the Uni-Directional Separability Problem on a CGM .	151
10.4	The Multi-Directional Separability Problem	153
10.4.1	Definitions and Notation	153
10.4.2	Solving the Multi-Directional Separability Problem on a CGM	154
10.5	Conclusion	157
IV	Conclusion	158
11	Conclusion	159
11.1	Multisearch	159
11.2	Directions for Future Research	163
	Bibliography	164

List of Figures

0.1	Standard processor interconnection networks used in multicomputer design.	2
0.2	A PRAM storing a shared data structure in shared memory.	4
1.1	A mesh-connected computer (mesh) of size $n = 64$	13
2.1	A directed balanced binary tree and its α -splitter ($\alpha = \frac{1}{2}$).	21
2.2	An undirected balanced binary tree with its α -splitter S_1 and β -splitter S_2	22
2.3	Next element search in a slab.	32
3.1	Sample images.	35
3.2	Illustration of the relation <i>r-connected</i>	37
3.3	Illustration of the relation <i>Delaunay connected</i>	38
3.4	A Voronoi diagram and Delaunay triangulation.	39
3.5	The space in which the binary search algorithm for determining $R(x)$ is conducted.	42
4.1	Hypercubes of dimensions $d = 1, 2, 3$ and 4	49
4.2	Level numbering of the nodes of a tree.	53
5.1	Row-Major and Shuffled Row-Major numbering.	60
5.2	Boundary code and boundary pixels of an image.	63

5.3	Combined level order numbering scheme.	70
6.1	A hybrid tree.	78
6.2	Case 1 for $d = 2$	79
6.3	Case 2 for $d = 2$	79
6.4	Merging linear trees T_1 and T_2	81
7.1	Many users sharing a single large knowledge-base server.	89
7.2	A layered view of a knowledge-base application.	89
7.3	An example of an inheritance hierarchy.	91
7.4	A frame record representing the class Oil-Tankers.	93
7.5	A sample query record.	93
7.6	Experimental results for Top-Down inference.	100
8.1	A Coarse-Grained Multicomputer (CGM).	105
9.1	Partitioning of the plane given unique and non-unique x -coordinates.	115
9.2	2D-Visibility from a point at $P = (0, -\infty)$	116
9.3	Weighted dominance for point u	119
9.4	The region of the plane dominated by the maximal elements of a point set.	122
9.5	Illustration of Lemma 9.1.	126
9.6	Convex hulls within each vertical slab.	129
9.7	Finding the tangent lines between two convex hulls.	130
9.8	Characterization of left endpoint of upper tangent line.	131
9.9	Graph of experimental results for 2D-Visibility.	135
10.1	The partitioning of T into $p + 1$ subtrees of size $O(\frac{n}{p})$	140
10.2	Polygon 1 is uni-directionally separable in direction d	148
10.3	The moveability wedge, $W(P, Q)$ for polygons P and Q	149
10.4	A set of multi-directionally separable polygons.	154

11.1 Deciding if multiseach is applicable.	162
---	------------

List of Tables

5.1	Overview of parameters used in quadtree algorithms.	56
5.2	Parallel pointer based quadtree construction algorithms.	56
5.3	Parallel linear quadtree construction algorithms.	57
5.4	Parallel pointer based quadtree manipulation algorithms.	57
5.5	Parallel linear quadtree manipulation algorithms.	58
8.1	Time required for each elementary operation on CGM.	107
8.2	Time required for each basic operation on CGM.	109
8.3	Minimum values of $\frac{n}{p}$, given p , for which our algorithms are optimal on a mesh-connected CGM.	110
9.1	Experimental results for 2D-Visibility.	135
11.1	Summary of multiseach algorithms.	160

Chapter 0

Introduction

Since the first computers were built in the early 1940's, computer users have thirsted for ever more speed to solve ever larger problems. In 1975 Seymour Cray introduced the Cray 1, the first true "supercomputer". By supercomputer we refer to a machine that is built for speed, a machine that is as fast as the technology of the day allows.

Many of the supercomputers of today, and perhaps all of the supercomputers of tomorrow are based on parallel architectures. Such computers consist of many individual processors that work in concert to solve a shared problem. These machines are a radical departure from their sequential (i.e. single processor) predecessors and there are many interesting open questions concerning their efficient use.

Throughout this thesis we will be concerned with a class of parallel machines called multicomputers. Multicomputers are comprised of a number of identical processors, each equipped with local memory. Each processor is connected via a communication link to a small number of other processors in a regular interconnection pattern. Figure 0.1 illustrates several interconnection patterns commonly used in designing multicomputers. In this thesis, a multicomputer is defined as "fine-grained" if each processor's local memory is of a small constant size (i.e., $O(1)$), otherwise we refer to it as "coarse-grained". Note that there is no shared memory,

Figure 0.1: Standard processor interconnection networks used in multicomputer design.

a processor requiring data from the local memory of another processor must route a message to that processor through the interconnection network.

This thesis has two different, yet often intertwined themes. We study techniques for efficiently constructing and manipulating parallel versions of standard sequential data structures on multicomputers. We then show how such parallel data structures can be used to efficiently solve problems from such diverse application domains as artificial intelligence and solid modeling. We also study how multicomputers can be used to solve geometric problems ranging from clustering to separability problems. In some cases we use parallel data structuring techniques to solve these geometric problems, while in other cases different parallel techniques proved more effective.

The following two sections introduce our themes, parallel data structures and parallel geometric applications, in more detail. A more detailed overview of the results presented in this thesis can be found in Section 0.3.

0.1 Parallel Data Structures

The use of efficient data structures is at the heart of algorithm design for the standard sequential model of computing. The literature on the design of efficient data structures is extensive and a significant amount of experience with data structures has been accumulated in both the academic and industrial settings. Data structure

abstractions are central to the design of efficient algorithms, be they in a sequential or parallel setting. For parallel algorithm design, one reason why many researchers prefer to use the CREW-PRAM model of computation over other models, is that this model allows the implementation of data structures in a straightforward manner. In this model an arbitrary number of processors, n , are connected to an infinitely large shared memory; each processor having concurrent read access to every memory location. Note that the structure of the PRAM memory resembles that of the standard sequential machine memory and can therefore be used to implement data structures in a similar manner. The usual approach to implementing data structures on the PRAM is to have all processors collectively build the data structure in shared memory. All individual processors can then search the data structure independently of each other (using the concurrent read operation), each one responsible for one of $m = O(n)$ given search queries; see Figure 0.2.

Since these query processes need only read access to the data structure, each processor can execute (in parallel) its query independent of the others, in the standard sequential manner. Each processor can be considered to have its own copy of the entire data structure solely for the purposes of answering its query. We will refer to the simultaneous processing of n queries on a shared data structure as the *multisearch* paradigm.

This paradigm allows researchers to apply numerous results about sequential data structures to the design of efficient PRAM-algorithms. It has been extensively used in the implementation of well established data structures; see for example [1, 8, 50].

In practice most parallel computers are not PRAMs. The PRAM is an idealized theoretical model of parallel computing, and is not (currently) a practical parallel architecture. The vast majority of existing parallel machines are multicomputers. For multicomputers the parallel execution of independent queries on one joint data structure is not as straightforward as in the PRAM case. The problem lies with the distributed nature of memory in such machines. Lacking the shared memory of the PRAM, any joint data structure on a multicomputer must be subdivided into pieces;

Figure 0.2: A PRAM storing a shared data structure in shared memory.

each piece, or even individual node, being stored on a different processor. Having distributed the nodes of a data structure across the memories of a multicomputer, one is left with a challenging task: How can the now distributed data structure be efficiently accessed by all processors in parallel? If a large number of processors require the same piece of the data structure at the same time, how can these requests be handled efficiently; that is without queuing requests for the required piece at the processor that stores it? If the multisearch paradigm is to be supported on a multicomputer, an access scheme must be developed that allows an arbitrary number of processors to access the same piece of a shared data structure at the same time. For each of the different architectures considered in this thesis, a different instance of multisearch will be studied.

The multisearch problem for fine-grained hypercube multicomputers was previously addressed by the author in [87]. In this thesis we use "hypercube multisearch" to solve problems on quadrees (Chapters 5 and 6) and inheritance hierarchies (Chapter 7). Section 4.1.3 of Chapter 4 briefly recalls the results given in [38, 87]. The hypercube technique presented in [38, 87] was based on the idea of moving the search queries synchronously through the partitioned data structure, and required

time proportional to the diameter of the hypercube interconnection network to move all queries to the next nodes in their search paths.

In Chapter 2 of this thesis we study the multisearch problem for fine-grained mesh connected computers. Unfortunately, the approach used on the hypercube was not viable on the *mesh*, since in order to obtain an *optimal* mesh algorithm to solve the multisearch problem, the time per advancement of all queries by one step needed to be $O(\frac{\sqrt{n}}{\log_2 n})$, which is *less* than the diameter of the network. The techniques used in Chapter 2 to solve the multisearch problem for the mesh are very different from those used for the hypercube in [38, 87].

0.2 Parallel Computational Geometry

The modern approach to computational geometry was pioneered by Shamos in his 1978 Ph.D. thesis [97]. Since then there has been enormous activity in the field. Preparata and Shamos's 1985 book [85] gives an excellent, if somewhat dated, overview. The sheer volume of research in this area can perhaps be attributed to two primary factors. Firstly, there is something fundamental about geometric problems. They have been a focus of mathematical research for 2000 years. Secondly, geometry abounds with practical uses. Computational geometry has found application in such diverse fields as VLSI design, robotics, solid modeling, vision, image processing, pattern recognition, and geographic information systems. In many cases, due to the computationally intensive nature of these problems the application of parallelism is of both theoretical and practical interest. The study of parallel computation geometry began with Anita Chow's 1980 Ph.D. thesis. See Akl and Lyons's 1992 book [6] for an up-to-date overview.

In this thesis we study geometric problems for a variety of parallel computational models. For the fine-grained mesh multicomputer we study optical clustering: a clustering technique used in image processing and pattern recognition. For the fine-grained hypercube multicomputer we study parallel quadrees and their higher dimensional extension. These structures have been found to be of importance in

image processing and solid modeling. In the concluding chapters of this thesis, we study geometric problems on coarse-grained multicomputers. We give new efficient scalable algorithms for the three dimensional dominance problem and planar visibility, weighted dominance, nearest neighbor and convex hull problems. In addition, using parallel data structures, we give solutions to both uni-directional and multi-dimensional geometric separability problems. Our algorithms for coarse-grained multicomputers have good asymptotic performance and are efficient in practice, as implementations have indicated.

0.3 Overview of Results

The thesis is organized in three parts. Each of the parts introduces a parallel machine model and then describes algorithms and implementations for this model. The following is a brief overview of the results.

Part I: Algorithms for Fine-Grained Mesh Multicomputers

Multisearch Techniques: Parallel Data Structures on a Mesh.

The *multisearch problem* is formally defined as follows. Given a data structure D modeled as a graph with n constant-degree nodes, perform $O(n)$ searches on D . Let r be the length of the longest search path associated with a search process, and assume that the paths are determined "on-line". Note that the search paths may overlap arbitrarily.

In Chapter 2, we solve the multisearch problem for certain classes of graphs in $O(\sqrt{n} + r \frac{\sqrt{n}}{\log_2 n})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh-connected computer. For many data structures, the search path traversed when answering one search query has length $r = O(\log_2 n)$. For these cases, our algorithm processes $O(n)$ such queries in asymptotically optimal $\Theta(\sqrt{n})$ time. The classes of graphs we consider contain many of the important data structures that arise in practice, including k -ary trees. Mesh Multisearch is a useful data structure abstraction which can be used to implement

parallel versions of standard sequential data structures on a mesh. These results have been published in [9, 10].

Optical Clustering on a Mesh.

In Chapter 3, we present optimal parallel algorithms for optical clustering on a mesh-connected multicomputer. *Optical clustering* is a clustering technique based on the principle of optical resolution, and is of particular interest in picture analysis. The algorithms we present are based on the application of parallel algorithms from computational geometry and graph theory. In particular, we show that given a set S of N points in the Euclidean plane, the following problems can be solved in optimal $O(\sqrt{N})$ time on a mesh-connected computer of size N .

1. Determine the optical clusters of S with respect to a given separation parameter.
2. Given an interval $[a, b]$ representing the number of optical clusters desired in the clustering of S , determine the range of the separation parameter that will result in such an optical clustering.

These results have been published in [37].

Part II: Algorithms for Fine-Grained Hypercube Multicomputers

Parallel Construction and Manipulation of Quadrees on a Hypercube.

In Chapter 5, we study the parallel construction and manipulation of pointer based quadrees. While parallel algorithms for the manipulation of a variant of linear quadrees have been previously studied in the literature, no parallel pointer based quadtree construction algorithms have been presented. In this chapter, we solve the problem of efficiently constructing pointer based quadrees on the hypercube, from images represented by either binary matrices or boundary codes. In addition we

show how these algorithms can be efficiently implemented on the PRAM providing new construction algorithms for both pointer based and linear quadtrees.

Furthermore, previous research in this area considered exclusively the parallel processing of a variant of linear quadtrees, namely linear quadtrees with path encoding. In Chapter 5, we demonstrate that, in the parallel setting, pointer based quadtrees are an attractive alternative to linear quadtrees with path encodings. We present new efficient and practical parallel algorithms for standard quadtree operations, (such as finding the neighbors of all leaves in a quadtree, and computing the union/intersection of two quadtrees) for the hypercube multicomputer. These results have been published in [35].

Constructing d -dimensional hyperoctrees from slices on a hypercube.

Hyperoctrees are the multidimensional extension of quadtrees. In many applications a description of a multidimensional object is given as a set of cross sections, e.g., in medical diagnosis planar cross sections are obtained via tomography systems. In Chapter 6, we consider the problem of computing the d -dimensional representation of an object given as a set of $(d - 1)$ -dimensional cross sections ("slices").

We present a parallel algorithm for the construction of a d -dimensional hyperoctree from a set of n $(d - 1)$ -dimensional hyperoctrees, representing adjacent "slices". On a p -processor hypercube, the time complexity of our algorithm is $O(\frac{m \log_2 n}{p} \log_2 p)$, where m is the maximum of input and output size. The input and output hyperoctrees may be based on either pointer or linear based representations. These results have been published in [31].

A Massively Parallel Knowledge-Base Server on a Hypercube.

In Chapter 7, we study the design and implementation of a traditional frame based knowledge representation system for a general purpose massively parallel hypercube architecture (such as the Connection Machine CM-2). We show that, using a widely available parallel system (instead of a special purpose architecture), it is possible to provide multiple users with efficient shared access to a large scale knowledge base.

Parallel algorithms are presented for answering multiple inference, assert and retract queries on both, single and multiple inheritance hierarchies. In addition to theoretical time complexity analysis, empirical results obtained from extensive testing of a prototype implementation are presented. These results have been published in [32].

Part III: Algorithms for Coarse-Grained Multicomputers

Scalable Parallel Geometric Algorithms for Coarse-Grained Multicomputers.

Parallel Computational Geometry is concerned with solving some given geometric problem of size n on a parallel computer with p processors (e.g., a PRAM, mesh, or hypercube multiprocessor) in time T_{parallel} . The parallel solution is *optimal*, if $T_{\text{parallel}} = O(\frac{T_{\text{sequential}}}{p})$ where $T_{\text{sequential}}$ is the sequential time complexity of the problem. Theoretical work for Parallel Computational Geometry has so far focused on the case $\frac{n}{p} = O(1)$, also referred to as the *fine grained* case. However, for parallel geometric algorithms to be relevant in practice, such algorithms must be *scalable*, that is, they must be applicable and efficient for a wide range of ratios $\frac{n}{p}$.

Yet, little theoretical work has been done for designing scalable parallel algorithms for Computational Geometry. The first and, to our knowledge, only previous theoretical study to address this problem was [11]. The model considered there was a host machine with $O(n)$ memory attached to a systolic array of size p with $O(1)$ memory per processors. This model suffers however from fact that data has to be frequently swapped between the host and the systolic array, and this "I/O bottleneck" is the main factor determining the computation time. The architectures of most existing multicomputers (e.g. the Intel Paragon, Intel iPSC/860, and CM-5) are quite different. They consist of a set of p *state-of-the-art* processors (e.g. SPARC proc.), each with considerable local memory, connected to some interconnection network (e.g. mesh, hypercube, fat tree). These machines are usually *coarse grained*, i.e. the size of each local memory is considerably larger than $O(1)$. In order to minimize the I/O bottleneck, the entire data set for a given problem is immediately loaded

into the local memories and remains there until the problem is solved. The problem studied in this chapter is the design of *scalable parallel geometric algorithms* for such Coarse Grained Multicomputers (CGM), which are optimal or at least efficient for a wide range of ratios $\frac{n}{p}$.

In Chapter 9, we present efficient *scalable* parallel CGM algorithms for the following geometric problems:

- 2D-Visibility from a point.
- Weighted Dominance counting of planar points.
- 3D-Maxima.
- All Nearest Neighbors in the plane.
- Convex Hull of planar points.

For the problems listed above the most efficient known sequential algorithms require time $O(n \log n)$. Our parallel algorithms require time $O(\frac{n \log n}{p} + T_{\text{sort}})$ for $n \geq p^2$. That is, even when our algorithms are not optimal they run in at most sort time. In addition to being efficient in the asymptotic sense these algorithms tend to be efficient and relatively easy to implement in practice. Experimental data is presented.

Data Structures on Coarse-Grained Multicomputers with Applications to Geometric Separability Problems.

The basic technique used in the algorithms described in Chapter 9 was *spatially* partitioning of geometric data. In contrast, the geometric algorithms described in Chapter 10 have as their basic technique the partitioning of a data structure abstraction that in some way represents geometric data.

In Chapter 10, we first describe a technique called CGM multisearch which allows the efficient CGM implementation of a large class of sequential data structures. We then show how CGM multisearch can be used to implement parallel segment

trees (with catalog counts) and describe a sample geometric application based on this parallel data structure. We also describe solutions to both the uni-directional and multi-directional separability problems, again based on data structure partitioning. Note that none of the geometric problems addressed in this chapter have been previously solved using the multiseach paradigm (for any parallel model).

Part I

Algorithms for Fine-Grained Mesh Multicomputers

Chapter 1

The Mesh Multicomputer: Definitions and Basic Operations

The *mesh-connected computer (mesh)* of size n is a multicomputer with n processors arranged in a square lattice. To simplify the exposition, it is assumed that $n = 4^c$, for some integer c . For all $i, j \in [0, \dots, n^{1/2} - 1]$, let $P_{i,j}$ represent the processor in row i and column j . Processor $P_{i,j}$ is connected via bidirectional unit-time communication links to its four neighbors, $P_{i-1,j}$, $P_{i+1,j}$, $P_{i,j-1}$, and $P_{i,j+1}$, assuming they exist. In the

Figure 1.1: A mesh-connected computer (mesh) of size $n = 64$.

following two chapters we present algorithms for *fine-grained* mesh multicomputers. Such machines are comprised of many very simple processors and typically operate in a SIMD fashion. Each processor has a fixed number of $\Theta(\log n)$ bit words of

memory (registers), and can perform standard arithmetic and Boolean operations on the contents of these registers in unit time. Each processor can also send or receive a word of data to or from one of its neighbors in unit time.

The communication diameter of a mesh of size n is $\Theta(\sqrt{n})$, as can be seen by examining the distance between processors in opposite corners of the mesh. This means that if a processor in one corner of the mesh needs data from a processor in another corner of the mesh at some time during an algorithm, then a lower bound on the running time of the algorithm is $\Omega(\sqrt{n})$. It is easy to see that, because of the communication diameter, the problems addressed in the following two chapters have time complexities $\Omega(\sqrt{n})$.

1.0.1 Standard Mesh Operations

In the following chapters, we will frequently use $\Theta(\sqrt{n})$ time *standard mesh operations* such as broadcast, parallel prefix [56], random access read [78], random access write [78] and sorting [4, 12, 103].

Broadcast: Broadcast a single value from one processor to all other processors.

Parallel Prefix: Initially each processor P_i , $0 \leq i \leq n$, stores a value a_i . In a Parallel Prefix operation each processor P_i determines the prefix $a_0 \oplus a_1 \oplus \dots \oplus a_i$, where $\oplus$ is a binary associative operation.

Random Access Read (RAR): Each processor reads the value of some designated register from another processor of its choice.

Random Access Write (RAW): Each processor writes a value to a designated register of some other processor of its choice. An associative combining rule must be specified, if more than one value is to be written to any processor.

Sorting: Sort n values of constant size distributed one per processor.

Chapter 2

Multisearch Techniques: Parallel Data Structures on a Mesh

Let D be a data structure modeled as a graph G with n constant-degree nodes. The *multisearch problem* consists of performing $O(n)$ searches on D , where the searches need not be processed in any particular order. Further, the searches may be simultaneously processed in parallel by using, for example, one processor per search. However, the path that an individual search will trace in G is *not* known ahead of time, and must instead be determined “on-line”. That is, only when a search query is at node v of G can it determine which node of G it should visit next. (This is accomplished by comparing the search key to the information stored at v . It should be noted that the nature of the information stored at the nodes, as well as the nature of the comparison that is performed at every node, depends on the specific problem being solved.) It is important to note that the paths of the search queries can overlap arbitrarily. That is, at any time, any node of G may be visited by an arbitrary number of search queries.

Multisearch is a useful abstraction that can be used to implement parallel versions of standard sequential data structures on a mesh. The Multisearch problem is a challenging problem for distributed memory multicomputers primarily because many search queries might want to visit a single node of G , creating a “congestion” problem. In fact, this problem of congestion can be complicated by the fact that we cannot even tally ahead of time the amount of congestion that will occur at a

node, since we do not know ahead of time the full paths of the search queries, only the nodes of G at which the queries start. On the PRAM, the graph G is stored in the shared memory in the standard way. When the parallel model used to solve the problem is a fine-grained multicomputer, the graph G is initially stored in the network such that each processor contains one node of G , as well as that node's adjacency list. It is important to keep in mind that the multicomputer's topology is *not* the same as the search structure G , so that a neighbor of node v in G need not be stored in a processor adjacent to the one containing v . Note that, due to the congestion problem, even an efficient embedding of the graph G into the network will not lead to an efficient multisearch algorithm as it does not solve the congestion problem. Initially, the $O(n)$ search queries are arbitrarily distributed one per processor.

In this chapter we solve the multisearch problem for certain classes of graphs in $\Theta(\sqrt{n} + r \frac{\sqrt{n}}{\log n})$ time on a $\sqrt{n} \times \sqrt{n}$ mesh, where r is the length of the longest search path associated with a query. Note that for many data structures the search path traversed when answering a query has length $r = O(\log n)$. For this situation, our algorithm processes $O(n)$ search queries in asymptotically optimal $\Theta(\sqrt{n})$ time. The classes of graphs considered include many important data structures that arise in practice, including balanced k -ary trees.

The multisearch problem for fine-grained hypercube multicomputers was previously studied in [38, 87]. The hypercube technique presented in [87] was based on the idea of moving the search queries synchronously through G , and required time proportional to the diameter of the network to move all queries to the next nodes in their search paths. Unfortunately, such an approach is not viable on the *mesh*, since in order to obtain an *optimal* mesh algorithm to solve the multisearch problem, the time per advancement of all queries by one step needs to be $O(\frac{\sqrt{n}}{\log n})$, which is *less* than the diameter of the network. The techniques we use to solve the multisearch problem for the mesh are very different from those used for the hypercube [38, 87].

In broad terms, our techniques for solving the multisearch problem are a combination of the following ideas.

- Partition G into pieces, some of which are processed sequentially, while others are processed in parallel.
- Create multiple copies of those pieces of G for which too many searches need access, and distribute the copies to disjoint submeshes, each of which is responsible for advancing a manageable subset of the “congested” searches. It should be noted that the straightforward strategy of making multiple copies of G , and using one copy for each search, does not work. This is due to the fact that it would not only take too much time to create the $O(n)$ copies, but there is not enough space to store all of these copies of G . In fact, there is only enough space to store $\Theta(1)$ copies of G , since G has n nodes.

The parameters needed to efficiently perform these partitioning, duplication, and mapping strategies cannot be pre-computed, since the full search paths are computed on-line. Therefore, these parameters must also be determined on-line, as the searches advance through G .

The classes of graphs considered in this chapter are called *Partitionable graphs*. We will be defining this class in detail later, but it should be noted that an important member of this class of graphs is the balanced k -ary tree. For partitionable graphs, we consider the multisearch problem for both the *undirected* and the *directed* case. For tree data structures, the directed partitionable graphs model tree algorithms in which search queries move along tree edges only in one direction, either from the root towards the leaves, or from the leaves towards the root. Many standard tree searches are of this type. Undirected partitionable graphs model tree algorithms in which search queries are permitted to move within the tree in an arbitrary manner. Such cases arise when queries are traversing parts of a tree, for example, in inorder.

Note that, after the publication of the work described in this chapter [9, 10], other researchers have begun to study other instances of the mesh multisearch problem for search trees and hierarchical acyclic directed graphs; see for example [7, 106].

The next section contains a more formal definition of the multisearch problem, and of the various terms used in this chapter. Section 2.2 contains the main result:

our solutions to the multisearch problem for partitionable graphs. Section 2.3 describes the use of multisearch to solve a variety of problems efficiently on the mesh. For other applications see [9].

2.1 Definitions

In this section we will define the multisearch problem, and the classes of graphs for which we will present efficient multisearch algorithms in Section 2.2.

2.1.1 The Multisearch Problem

Let $G = (V, E)$ be a directed or undirected graph of size $n = |V| + |E|$, where the out-degree or degree, respectively, of any vertex is bounded by some constant. Let U be a universe of possible *search queries* on G . Define the *search path* of a query $q \in U$, denoted $path(q)$, to be a sequence of h vertices $(v_1, \dots, v_h)$ of G defined by a successor function $f : (V \cup start) \times U \rightarrow V$ as

- $f(start, q) = v_1$, and
- $f(v_i, q) = v_{i+1}$ for $i = 1, \dots, h - 1$.

The function f has the following properties

- If G is directed, then for every vertex $v \in V$ and query $q \in U$, $(v, f(v, q)) \in E$.
- If G is undirected, then for every vertex $v \in V$ and query $q \in U$, $\{v, f(v, q)\} \in E$.
- $f(v, q)$ can be computed in $\Theta(1)$ time by a single processor that contains the information pertinent to q and v .

We say that a query $q \in U$ *visits* a node $v \in V$ at time t if and only if, at time t , the mesh is in a state where there exists a processor which contains a description of both the query q and the node v . (Note that this definition implies that many queries can simultaneously visit node v , if each such query uses a different copy of

v 's information.) The *search process* for a search query q with search path $path(q) = (v_1, \dots, v_h)$ is a process divided into h time steps, $t_1 < t_2 < \dots < t_h$, such that at time t_i , $1 \leq i \leq h$, query q visits node v_i . We will refer to the change of state between t_i and t_{i+1} , $1 \leq i < h$, as *advancing query q one step in its search path*. It is important to note that we do not assume the search path to be given in advance. In fact, we assume that the search path for each query is constructed *online* during the search by successive applications of the function f .

Note that for a *directed* graph, a query can be advanced along an edge only in the indicated direction, whereas for *undirected* graphs a query can advance along an edge in both directions.

Given a set $Q = \{q_1, \dots, q_m\} \subseteq U$ of m search queries, where $m = O(n)$, then the *multisearch problem* for Q on G consists of executing (in parallel) all m search processes induced by the m search queries. It is important to note that the m search processes can overlap arbitrarily. That is, at any time t , any node of G may be visited by an arbitrary number of queries, which may, in fact, be at very different time steps in their respective search paths (of course each such query would be using a different copy of v 's information).

We will refer to the process of advancing, in parallel, a subset of the m search queries by one step in their respective search paths as a *multistep*. Notice that we do not require all queries to be advanced synchronously. We will refer to a sequence of multisteps which has the property that every search query is advanced $\Omega(\log n)$ steps in its respective search path, as a *log-phase*.

A convenient way of visualizing the multisearch process is by associating a *pebble* with each query. Initially, the pebble associated with query q is placed on the first node in $path(q)$. During the multisearch process, the m pebbles move in parallel along edges of G , each pebble according to its respective search path. Each node of the graph may be visited, at any time, by an arbitrary number of pebbles. Notice that if G is undirected, then pebbles move freely along edges of the graph, while if G is directed, then pebbles can only move in the proper direction of an edge. Note that, pebbles may move with different and possibly changing speeds.

For the remainder of this chapter, we will assume that G is connected (by a "connected" directed graph we mean that the undirected version of that graph is connected). For graphs with several connected components, the multisearch algorithms described in Section 2.2 can be easily applied independently and in parallel to each connected component, such that the overall time complexity remains unchanged.

2.1.2 Partitionable Graphs

δ -Splitters

Let $G = (V, E)$ be a (directed or undirected) graph with vertex set V , edge set E , and size $n = |V| + |E|$. Let $S \subset E$. Then $(V, E - S)$ is a graph with vertex set V and edge set $E - S$ that consists of a set of $k \leq n$ connected components, denoted $\{G_1, \dots, G_k\}$.

We define S to be a δ -splitter of G , $0 < \delta < 1$, if and only if $|G_i| = |V_i| + |E_i| = O(n^\delta)$, for all $1 \leq i \leq k$. Given a δ -splitter S , we will refer to $G(S) = \{G_1, \dots, G_k\}$ as a δ -splitting of G .

A vertex $v \in V$ is defined to be *at the border* of a δ -splitter S if and only if v is a vertex of an edge $e \in S$. A δ -splitting $G(S) = \{G_1, \dots, G_k\}$ is called *normalized*, if $k = \Theta(n^{1-\delta})$ (which implies that $|G_i| = \Theta(n^\delta)$, for all i).

α -Partitionable (Directed) Graphs

Let $G = (V, E)$ be a directed graph with vertex set V , edge set E , and size $n = |V| + |E|$, where the out-degree of any vertex is bounded by some constant. We define G to be α -partitionable if and only if G has an α -splitter S , $0 < \alpha < 1$, such that $G(S) = \{G_1, \dots, G_k\}$ can be partitioned into two sets of graphs, $\{H_1, \dots, H_{k_1}\}$ and $\{T_1, \dots, T_{k_2}\}$, such that for every directed edge $(v_1, v_2) \in S$, $v_1 \in H_i$ and $v_2 \in T_j$, for some i, j .

Note that, for example, every balanced k -ary search tree with all edges either directed towards the leaves or directed towards the root (i.e., all search queries can

Figure 2.1: A directed balanced binary tree and its α -splitter ($\alpha = \frac{1}{2}$).

only move in one direction, either from the root towards the leaves, or from the leaves towards the root) is α -partitionable; see Figure 2.1.

α - β -Partitionable (Undirected) Graphs

Let $G = (V, E)$ be an undirected graph with vertex set V , edge set E , and size $n = |V| + |E|$, where the degree of any vertex is bounded by some constant. For two vertices $v_1, v_2 \in V$, let $\text{dist}_G(v_1, v_2)$ denote the length of a shortest (undirected) path in G connecting v_1 and v_2 .

Let S_1 and S_2 be an α -splitter and a β -splitter, respectively, of G . We define S_1 and S_2 to have distance k if and only if $k = \min\{\text{dist}_G(v_1, v_2) : v_1 \text{ is at the border of } S_1 \text{ and } v_2 \text{ is at the border of } S_2\}$.

G is called α - β -partitionable if and only if G has an α -splitter S_1 and a β -splitter S_2 , such that S_1 and S_2 have distance $\Omega(\log n)$.

Figure 2.2: An undirected balanced binary tree with its α -splitter S_1 ($\alpha = \frac{1}{2}$) and β -splitter S_2 ($\beta = \frac{1}{3}$), such that S_1 and S_2 have distance $\frac{h}{6} = \Omega(\log n)$.

Note that, for example, every undirected balanced k -ary search tree (i.e., search queries can move within the tree in arbitrary direction) is α - β -partitionable; see Figure 2.2.

2.2 Mesh Solutions to the Multisearch Problem

In this section, we present mesh solutions to the multisearch problem for both α -partitionable graphs, and α - β -partitionable graphs.

First, we define some notation that will be used throughout this section. Define $G = (V, E)$ to be a graph with vertex set V , edge set E , and size $n = |V| + |E|$. In each subsection, we will specify whether the graph is directed or undirected. For directed graphs, we assume that the out-degree of every vertex is bounded by some constant, and for undirected graphs, we assume that the degree of every vertex

is bounded by some constant. Finally, we define $Q = \{q_1, \dots, q_m\}$ to be a set of $m = O(n)$ search queries.

We now discuss the manner in which G and Q will be represented on the mesh. Every processor will initially store

- one arbitrary vertex $v \in V$,
- the addresses of all processors storing a vertex $w \in V$, such that $(v, w) \in E$ (recall that G has out-degree $\Theta(1)$), and
- one arbitrary query $q \in Q$.

During an algorithm, no processor will store information associated with more than $\Theta(1)$ items of V nor more than $\Theta(1)$ items of Q . Notice that the assignment of vertices and queries to processors may change during the course of the algorithms. In addition, we assume that every processor p has a register $visit(p)$, where at any stage of a multisearch algorithm, a query $q \in Q$ will be said to *visit* a node $v \in V$ if processor p is responsible for query q and stores a copy of v in $visit(p)$.

In the remainder of this section, we present mesh solutions to the multisearch problem for α -partitionable graphs and α - β -partitionable graphs. We will first introduce a tool referred to as *constrained multisearch*, which will be utilized in Sections 2.2.2 and 2.2.3.

2.2.1 Constrained Multisearch

Let $G = (V, E)$ be a directed or undirected graph. Consider a set $\Psi = \{G_1, \dots, G_k\}$ of k edge and vertex disjoint subgraphs of G such that $|G_i| = \Theta(n^\delta)$ and $k = \Theta(n^{1-\delta})$, for some $0 < \delta < 1$. It is important to note that we do not assume that the union of the subgraphs in Ψ contains all vertices of G .

Consider any stage of the multisearch for Q on G , and let $v(q) \in path(q)$ denote the node currently visited by query $q \in Q$.

The *constrained multisearch problem* with respect to Ψ consists of advancing, for every $G_i \in \Psi$, every search query q with $v(q) \in G_i$ by $\log_2 n$ steps in its search path,

unless the next node to be visited by q is not in G_i . Notice that the queries may be advanced by a nonuniform number of steps.

The remainder of this section focuses on procedure *Constrained-Multisearch*(Ψ, δ), which solves the constrained multisearch problem on a mesh of size n in $\Theta(\sqrt{n})$ time.

For every $G_i = (V_i, E_i) \in \Psi$, we define

$$\Gamma_{\Psi}^{\delta}(G_i) = \left\lceil \frac{|\{q \in Q : v(q) \in V_i\}|}{n^{\delta}} \right\rceil.$$

Property 2.1 $\sum_{G_i \in \Psi} \Gamma_{\Psi}^{\delta}(G_i) = O(n^{1-\delta})$

Proof: A trivial consequence of the fact that $|Q| = m = O(n)$. $\square$

We now present our mesh algorithm for solving the constrained multisearch problem with respect to Ψ . It assumes an initial configuration in which the calling program is at a stage of the multisearch for Q on G , where every $q \in Q$ currently visits node $v(q) \in \text{path}(q)$. Furthermore, every processor storing a vertex $v \in V$, also stores an index indicating to which $G_i \in \Psi$ the vertex v belongs, if any.

Procedure Constrained-Multisearch(Ψ, δ):

1. All queries $q \in Q$, such that $v(q)$ is in some subgraph $G_i \in \Psi$, are marked *active*; all other queries are marked *inactive*. (Queries whose search paths have already terminated are also marked inactive.)
2. For every $G_i \in \Psi$, the value of $\Gamma_{\Psi}^{\delta}(G_i)$ is computed.
3. If $\sum_{G_i \in \Psi} \Gamma_{\Psi}^{\delta}(G_i) = 0$ then EXIT.
4. For each $G_i \in \Psi$, $\Gamma_{\Psi}^{\delta}(G_i)$ copies of G_i are created. Each copy is placed in a $\sqrt{n^{\delta}} \times \sqrt{n^{\delta}}$ size subsquare of the mesh-connected computer. That is, a *submesh* of size $\sqrt{n^{\delta}} \times \sqrt{n^{\delta}}$.
5. Every active query $q \in Q$, with $v(q) \in G_i$, is moved to one of the submeshes storing a copy of G_i . This movement is coordinated so that each submesh containing a copy of G_i will receive $O(n^{\delta})$ queries.

6. Within every submesh storing a subgraph $G_i \in \Psi$, the following is executed $\log_2 n$ times.

- (a) For every active query $q \in Q$, the next node in its search path is determined (by applying the successor function f).
- (b) Every active query for which the next node in its search path is not in G_i , is marked inactive. (A query whose search path terminates is also marked inactive.)
- (c) Every active query visits the next node in its search path.

7. Discard the copies of the subgraphs $G_i \in \Psi$ created in Step 4.

End of Procedure Constrained-Multisearch(Ψ, δ).

Lemma 2.1 *The constrained multisearch problem with respect to Ψ can be solved on a mesh of size n in $\Theta(\sqrt{n})$ time.*

Proof: We first study the correctness of Constrained-Multisearch(Ψ, δ), then give some implementation details, and finally prove the time complexity. Obviously, every query q either

- visits the next $\log_2 n$ nodes in its search path,
- visits the next N nodes in its search path, where $N < \log_2 n$, until the next node to be visited is no longer in the same subgraph $G_i \in \Psi$ that contains $v(q)$, or
- does not advance any steps in its search path, for the case where $v(q)$ is not in any $G_i \in \Psi$.

The crucial step for proving the correctness of the procedure is to show that (1) the total size of the copies of subgraphs G_i created in Step 4 is $O(n)$, and (2) in Step 5, the sizes and total number of queries to be moved match the sizes and total number of submeshes available. Item (1) follows from Property 2.1, and Item (2) follows from the definition of $\Gamma_\Psi^\delta(G_i)$ and the fact that each submesh is of size $\Theta(n^\delta)$.

We will now prove the claimed time complexity. Steps 1, 2, 3, and 7 can be easily implemented in time $O(\sqrt{n})$ by applying a constant number of standard mesh operations. For Step 4, the mesh is subdivided into a grid of $\sqrt{n^{1-\delta}} \times \sqrt{n^{1-\delta}}$ submeshes, each of size n^δ . The total number of copies created of every subgraph G_i is $\Theta(n^{1-\delta})$ (Property 2.1). Hence, every submesh needs to simulate only a constant number of “virtual” submeshes, where each “virtual” submesh stores just one copy of some subgraph $G_i \in \psi$. Creating the required copies of subgraphs and moving them to the “virtual” submeshes can be implemented by a constant number of standard mesh operations. Step 5 is implemented analogously. Finally, we discuss the time complexity of Step 6. Notice that each execution of the loop body is executed independently and in parallel on every submesh of size $O(n^\delta)$ created in Step 4. Therefore, by using standard random access read and write operations within every submesh, each iteration of the loop can be implemented in $O(\sqrt{n^\delta})$ time, which implies a total of $O(\log n \sqrt{n^\delta})$ time for Step 6 (since there are $\log_2 n$ iterations). Since $0 < \delta < 1$, the total time complexity of Step 6 is $O(\log n \sqrt{n^\delta}) = O(\sqrt{n})$. $\square$

2.2.2 The Multisearch Problem for Directed α -Partitionable Graphs

Let $G = (V, E)$ be a directed α -partitionable graph. Let $Q = \{q_1, \dots, q_m\}$ be a set of $m = O(n)$ search queries, and let r denote the length of the longest search path associated with a query $q \in Q$. In this section, we present an algorithm to solve the multisearch problem for Q on G in $O(\sqrt{n} + r \frac{\sqrt{n}}{\log n})$ time. Our strategy is to give an algorithm which executes one log-phase of multisearch in $(\sqrt{n})$ time. The entire multisearch algorithm consists of iterating the log-phase algorithm $O(\lceil \frac{r}{\log n} \rceil)$ times.

Let $G(S) = \{H_1, \dots, H_{k_1}, T_1, \dots, T_{k_2}\}$ be an α -splitting of G such that for every edge $(v_1, v_2) \in S$ (directed from v_1 to v_2), $v_1 \in H_i$ and $v_2 \in T_j$, for some $1 \leq i \leq k_1, 1 \leq j \leq k_2$. Recall that this implies $0 < \alpha < 1$, $|H_i| = O(n^\alpha)$, and $|T_j| = O(n^\alpha)$.

We assume that the α -splitter S is known a priori. That is, initially the processor that stores vertex $v \in V$ also stores an index indicating the graph in $G(S)$ to which

v belongs. We can also assume, without loss of generality, that $G(S)$ is normalized. That is, we can assume that $k = k_1 + k_2 = \Theta(n^{1-\alpha})$; see Section 2.1.2. Otherwise, we group the subgraphs H_i (T_i) such that each resulting subgraph has size $\Theta(n^\alpha)$. This operation is easily performed on a mesh of size n in $O(\sqrt{n})$ time. Furthermore, the algorithm described in this section does not require that every subgraph in $G(S)$ consist of only one connected component of the graph $(V, E - S)$.

Before presenting Algorithm α -partitionable log-phase which implements one log-phase of the multisearch problem for Q on G , we observe some properties of α -partitionable graphs.

Property 2.2 *Let $G(S) = \{H_1, \dots, H_{k_1}, T_1, \dots, T_{k_2}\}$ be an α -splitting of G . Then the following holds*

- *A query q that has a node of a subgraph H_i in its search path does not visit any node of another subgraph $H_j, i \neq j$.*
- *Once a query q has visited a node in a subgraph T_i , all subsequent nodes visited by q will be in the same subgraph T_i .*

Proof: The proof follows from the fact that edges of an α -partitionable graph are either directed from some H_i to some T_j , or have both endpoints in the same subgraph H_i or T_i . □

Algorithm α -partitionable log-phase:

1. If this is the first log-phase, then every query $q \in Q$ visits the first node in its search path; otherwise, every $q \in Q$ visits the next node in its search path.
2. Constrained-Multisearch $(\{H_1, \dots, H_{k_1}, T_1, \dots, T_{k_2}\}, \alpha)$.
3. Every $q \in Q$ visits the next node in its search path.
4. Constrained-Multisearch $(\{H_1, \dots, H_{k_1}, T_1, \dots, T_{k_2}\}, \alpha)$.

End of Algorithm α -partitionable log-phase.

Lemma 2.2 *One log-phase of multisearch on a directed α -partitionable graph of size n can be performed in $\Theta(\sqrt{n})$ time on a mesh of size n .*

Proof: We first consider the correctness of Algorithm α -partitionable log-phase. The algorithm is based on the following. Initially, every query starts at the first node in its search path, which is in some H_i , $1 \leq i \leq k_1$, or T_j , $1 \leq j \leq k_2$. Using Constrained-Multisearch, every query is advanced until it visits either its $\log_2 n$ successors, or needs to visit a node that is not in its initial subgraph, at which point it stops. Next, every query is advanced one node and then Constrained-Multisearch is performed again. So, by two applications of Constrained-Multisearch, every query will be advanced at least $\log_2 n$ nodes. (Note, if there are fewer than $\log_2 n$ nodes in a given search path, then that query will terminate at the appropriate time.) From Property 2.2, it follows that for every query $q \in Q$, one of the following cases must apply:

1. All nodes visited by q within the log-phase are in one subgraph H_i .
2. All nodes visited by q within the log-phase are in one subgraph T_j .
3. Within the log-phase, query q first visits only nodes within one subgraph H_i , and once it "leaves" H_i it will only visit nodes in one subgraph T_j .

For those queries to which either Case 1 or Case 2 applies, all nodes visited on the search path during the log-phase are visited during Steps 1 and 2; see Lemma 2.1. Let q be a query to which Case 3 applies, and let $(v_1, \dots, v_x, v_{x+1}, \dots, v_y)$ be the sequence of nodes to be visited within the log-phase, where $v_1, \dots, v_x$ are in some subgraph H_i , and $v_{x+1}, \dots, v_y$ are in some subgraph T_j . It follows from Lemma 2.1 that $v_1, \dots, v_x$ are visited during Steps 1 and 2, and that $v_{x+1}, \dots, v_y$ are visited during Steps 3 and 4.

From Lemma 2.1 it also follows that Algorithm α -partitionable log-phase has time complexity $\Theta(\sqrt{n})$ and requires only $\Theta(1)$ memory per processor. $\square$

Therefore, by iterating Algorithm α -partitionable log-phase $O(\lceil \frac{\tau}{\log n} \rceil)$ times, the multisearch problem can be solved for α -partitionable graphs.

Theorem 2.1 *Let G be a directed α -partitionable graph of size n , and let $Q = \{q_1, \dots, q_m\}$ be a set of $m = O(n)$ search queries. Then the multisearch problem for Q on G can be solved in $O(\sqrt{n} + r \frac{\sqrt{n}}{\log n})$ time on a mesh of size n , where r is the length of the longest search path associated with a query $q \in Q$.*

2.2.3 The Multisearch Problem for Undirected α - β -Partitionable Graphs

Let $G = (V, E)$ be an (undirected) α - β -partitionable graph. Let $Q = \{q_1, \dots, q_m\}$ be the set of $m = O(n)$ search queries, and let r denote the length of the longest search path associated with a query $q \in Q$. In this section, we present an algorithm to solve the multisearch problem for Q on G in $O(\sqrt{n} + r \frac{\sqrt{n}}{\log n})$ time. As in Section 2.2.2, we will again give an algorithm to execute one log-phase of the multisearch problem in $\Theta(\sqrt{n})$ time. The multisearch algorithm will consist of iterating this log-phase algorithm $O(\lceil \frac{r}{\log n} \rceil)$ times.

Let S_1 and S_2 be an α -splitter and a β -splitter, respectively, of G such that S_1 and S_2 have distance $\Omega(\log n)$. We assume that S_1 and S_2 are known a priori. That is, initially the processor that stores vertex $v \in V$ also stores an index indicating the graph $G(S_1)$ to which v belongs, and an index indicating the graph $G(S_2)$ to which v belongs.

With the same argument as in Section 2.2.2, we also assume that $G(S_1)$ and $G(S_2)$ are normalized. Let $G(S_1) = \{W_1^1, \dots, W_{k_1}^1\}$ and $G(S_2) = \{W_1^2, \dots, W_{k_2}^2\}$. Recall that $0 < \alpha < 1$, $0 < \beta < 1$, $|W_i^1| = \Theta(n^\alpha)$, $|W_i^2| = \Theta(n^\beta)$, $k_1 = \Theta(n^{1-\alpha})$, and $k_2 = \Theta(n^{1-\beta})$.

We first state a property of α - β -partitionable graphs that will be used in the algorithm.

Property 2.3 *Let S_1 and S_2 be an α -splitter and β -splitter, respectively, of G , such that S_1 and S_2 have distance $\Omega(\log n)$. Then, if at any stage of the multisearch, a query $q \in Q$ visits a node v at the border of S_1 , it can advance $\Omega(\log n)$ more steps in its search path without visiting a node v' at the border of S_2 .*

Proof: The proof follows immediately from the definition of α - β -partitionable graphs. $\square$

Algorithm α - β -partitionable log-phase:

1. If this is the first log-phase, then every query $q \in Q$ visits the first node in its search path; otherwise, every $q \in Q$ visits the next node in its search path.
2. Constrained-Multisearch ($\{W_1^1, \dots, W_{k_1}^1\}, \alpha$).
3. Every $q \in Q$ visits the next node in its search path.
4. Constrained-Multisearch ($\{W_1^2, \dots, W_{k_2}^2\}, \beta$).

End of Algorithm α - β -partitionable log-phase.

Lemma 2.3 *One log-phase of multisearch on an (undirected) α - β -partitionable graph of size n can be performed in $\Theta(\sqrt{n})$ time on a mesh of size n .*

Proof: We first consider the correctness of Algorithm α - β -partitionable log-phase. The algorithm is based on the following. Initially, every query starts at the first node in its search path. Using Constrained-Multisearch on $G(S_1)$, every query is advanced until it visits either its $\log_2 n$ successors, or needs to visit a node that is not in its initial subgraph, at which point it stops. Next, every query is advanced one node and then Constrained-Multisearch is performed again, but this time with respect to $G(S_2)$. Notice that by performing the second application of Constrained-Multisearch with respect to $G(S_2)$, every query that had reached a border of $G(S_2)$ will be able to advance $\Omega(\log n)$ more steps in its search path without visiting another node at the border of S_2 ; by this time, the log-phase is completed. Therefore, by 2 applications of Constrained-Multisearch, every query will be advanced at least $\log_2 n$ nodes. (Note, if there are fewer than $\log_2 n$ nodes in a given search path, then that query will terminate at the appropriate time.) That is, for every query $q \in Q$, one of the following cases applies:

1. All nodes visited by q within the log-phase are in one subgraph W_i^1 .

2. All nodes visited by q within the log-phase are in one subgraph W_i^2 .
3. Within the log-phase, query q first visits some nodes in one subgraph W_i^1 of $G(S_1)$. Once it "leaves" W_i^1 , it is sufficient (for the completion of a log-phase) to consider only the subgraph W_j^2 of $G(S_2)$ visited at that point in time, and let the query continue on its search path until it reaches a vertex at the border of S_2 .

The correctness of Algorithm α - β -partitionable log-phase, as well as the time and space complexity, follow immediately from Lemma 2.1. $\square$

Therefore, by iterating Algorithm α - β -partitionable log-phase $O(\lceil \frac{r}{\log n} \rceil)$ times, the multisearch problem can be solved for α - β -partitionable graphs.

Theorem 2.2 *Let G be an (undirected) α - β -partitionable graph of size n , and let $Q = \{q_1, \dots, q_m\}$ be a set of $m = O(n)$ search queries. Then the multisearch problem for Q on G can be solved in $O(\sqrt{n} + r \frac{\sqrt{n}}{\log n})$ time on a mesh of size n , where r is the length of the longest search path associated with a query.*

2.3 Implementing Parallel Data Structures on a Mesh-Connected Computer

In this section, we illustrate the use of the multisearch techniques presented in Section 2.2. The multisearch technique for partitionable graphs, described in Section 2.2, can be immediately applied to parallelize standard query processes on balanced search trees. When processing many such queries independently and in parallel, the query paths may overlap arbitrarily. Of particular interest are online processes where the paths taken by the queries can not be computed a priori. Such cases occur, for example, when no global order exists for the set of queries and data.

We give two simple illustrations of possible applications. Consider a set S of n non intersecting line segments spanning a vertical slab, as illustrated in Figure 2.3. Each query consists of a point within the slab, for which the two segments deter-

Figure 2.3: An example with $S = \{L1, \dots, L5\}$ and $Q = \{p1, \dots, p6\}$. Note that there exists no total ordering of $Q \cup S$.

mining the region containing that point must be computed. The obvious sequential solution is to build a balanced binary tree for the line segments and answer queries by a straight forward tree search. Using our multisearch technique, a set Q of n such queries can be processed in time $O(\sqrt{n})$ on a mesh of size n . Note that, there exists no total ordering on the set $Q \cup S$.

Now, consider the problem of determining the “best” common ancestor of a pair of nodes in a tree. Such a problem occurs, e.g., in clustering [64]: given a hierarchical agglomerative clustering scheme, determine for two data elements the “best” cluster (e.g., the cluster with closest cluster center) containing both elements. The obvious sequential solution to the general “best” common ancestor problem, in a tree of size n , is to visit the path of all common ancestors in the tree while maintaining the current best element. Using our multisearch technique, a set of n such queries can be processed in time $O(\sqrt{n})$ on a mesh of size n . Another application is parallel searching in segment trees (with catalog counts). Many efficient algorithms in computational geometry rely on this data structure, see for example the Multiple Line Segment problem described in Section 10.2.2.

The multisearch techniques for multiple online overlapping queries on partitionable graphs also supports cases where queries may change directions independently. For example, in a tree, queries may move both upwards and downwards during the search. Possible applications include cases where each query performs an inorder traversal of a certain subtree [32].

We believe that the multisearch problem is such a fundamental problem that we expect it to have many additional applications (e.g., in parallel databases and related areas).

2.4 Conclusion

In this chapter, we have considered the *multisearch problem* for $O(n)$ search queries on a data structure modeled as a graph G with n constant-degree nodes. We have presented a $\Theta(\sqrt{n} + r \frac{\sqrt{n}}{\log n})$ time algorithm for performing, in parallel, $O(n)$ searches on a shared data structure stored in a $\sqrt{n} \times \sqrt{n}$ mesh-connected computer. The main problem for the mesh, in comparison to other networks like the hypercube, is that in order to obtain optimal algorithms from multisearch, the time per advancement of all queries by one step in their search paths must be $O(\frac{\sqrt{n}}{\log n})$. That is, it must be less than the diameter of the network. The algorithms presented here show how to overcome this problem.

Chapter 3

Optical Clustering on a Mesh

This chapter is concerned with parallel solutions to two problems in the area of clustering on fine-grained mesh multicomputers. We study the parallelization of *optical clustering* [28], which is a clustering technique based on the principle of optical resolution and is of particular interest in picture analysis. The algorithms we present are asymptotically optimal for the mesh.

The input to both problems is a set of n points in the plane, which usually represent the pixels of an image (see Figure 3.1). For the mesh, each processor initially stores one of the input points, with an arbitrary assignment of points to processors. For *optical clustering* [28], a relation *connected* is defined as follows. Two points are *connected* if and only if there exists a circle with radius $\leq r$ containing the two points. Since the transitive closure of this relation is an equivalence relation, the *connected components* of a set of points are defined to be the equivalence classes generated by the transitive closure over the relation *connected*. The first clustering problem studied in this chapter consists of computing the equivalence classes (i.e., determining the connected components) for a given parameter r . The second clustering problem, which is more difficult, is the inverse problem. As input we specify a *range* over the number of connected components we expect. The problem is to determine a maximal interval for the parameter r such that the number of components of the respective clusterings is within the specified range. Sequential $O(n \log n)$ time solutions to both problems have been presented in [28]. In this chapter, we present

Figure 3.1: Sample images.

optimal $O(\sqrt{n})$ time parallel algorithms for the mesh. A direct parallelization of the algorithms presented in [28] would yield $O(\sqrt{n} \log n)$ time parallel solutions on the mesh. In this chapter we present a data compression technique which reduces the time of the mesh algorithm to $O(\sqrt{n})$, which is optimal. Our solutions to these clustering problems involve parallel techniques and algorithms from computational geometry and graph theory.

Our main motivation for studying parallel optical clustering on the mesh is that (1) the mesh is a very popular architecture for image processing and (2) optical clustering, as part of a picture recognition system, often has on-line response time requirements, which can only be met by using parallel architectures or by creating customized hardware. Next, we outline particularly interesting applications of our techniques which have such on-line requirements¹.

The second clustering problem outlined above has applications in pattern recognition systems where one knows a range in the number of objects that are anticipated and would like to perform a clustering operation that will yield the anticipated number of objects. For example, such knowledge can be exploited when considering the

¹These problems were brought to our attention by Ed Cohen and Jon Hull from the CEDAR project at the University of Buffalo.

digitization of a handwritten social insurance number, where one would anticipate approximately 11 distinct objects (9 digits and two hyphens). In fact, since some of the handwritten digits in a social insurance number may overlap each other, while other individual digits might not even be connected, a suitable range in the number of clusters expected for a handwritten social insurance number might be, say, between 7 and 15. Once the objects are identified there are numerous strategies, such as template matching or recognition based on feature sets, for attempting to recognize the digits. Another interesting application of optical clustering arises in the problem of determining the location of a handwritten address on an envelope [83].

The remainder of this chapter is organized as follows. In Section 3.1, the two problems considered in this chapter are defined. In Section 3.2, we present optimal mesh solutions to both clustering problems. Section 3.3 outlines how these results can also be applied to images represented by chain codes and Section 3.4 concludes the chapter.

3.1 Optical Clustering

This section gives a brief review of the definition and some basic properties of *optical clustering* as described in [28].

Let $S = \{s_1, \dots, s_n\}$ be a set of n disjoint objects in N -space, $\mathbb{R}^N$ (i.e., compact subsets of $\mathbb{R}^N$ without holes). Let $d : \mathbb{R}^N \times \mathbb{R}^N \rightarrow \mathbb{R}^+$ be a convex distance function, where $\mathbb{R}^+$ is the set of positive real numbers. For two objects $s, s' \in S$, define $d(s, s')$ as the minimum distance $d(x, x')$ between two points $x \in s, x' \in s'$. Finally, let $c(P, r) = \{x \in \mathbb{R}^N \mid d(P, x) \leq r\}$ denote the ball with center $P \in \mathbb{R}^N$ and radius r .

Consider two objects $s_i, s_j \in S$. We say that s_i and s_j are *r-connected*, denoted by $s_i \odot_r s_j$, if and only if there exists a ball $c(P, r')$, with $r' \leq r$, such that $c(P, r') \cap s_i \neq \emptyset$ and $c(P, r') \cap s_j \neq \emptyset$.

Figure 3.2: Illustration of the relation r -connected.

Since the transitive closure of the r -connected relation, denoted $cl(\odot_r)$, is an equivalence relation, we define the *optical clusters with respect to separation parameter r* as the equivalence classes of $cl(\odot_r)$.

Figure 3.2a illustrates the optical clusters of an 11 object set S' in $\mathbb{R}^2$. Notice that the optical clustering of S' for the given value of r (illustrated by the balls of radius r) results in three clusters, namely, $\{s_1, \dots, s_5\}$, $\{s_6, \dots, s_{10}\}$ and $\{s_{11}\}$.

Let $m(S, r)$ denote the number of optical clusters of S with respect to r . Clearly, as r increases to infinity, the number of optical clusters $m(S, r)$ decreases to 1. In fact, $m(S, r)$ is a monotonically decreasing function in r . That is, $r \leq r' \Rightarrow m(S, r) \geq m(S, r')$.

Given the task of constructing the optical clusters of a set S of geometric objects with respect to a separation parameter r , a naive solution would be to compute the graph $(S, \odot_r)$ (see Figure 3.2b), and then find the connected components of $(S, \odot_r)$. The drawback to this approach is that it involves computing the connected components of the graph $(S, \odot_r)$, which has size $O(n^2)$, since in the worst case it may be a complete graph on the n vertices. As an illustration of this, consider the optical clusters of the set $S'' = \{s_6, \dots, s_{10}\}$ from Figure 3.2a. Every object in S'' is r -connected to every other object in S'' (witness the ball in the center of S''),

Figure 3.3: Illustration of the relation *Delaunay connected*.

implying that $(S'', \odot_r)$ has $|S''|^2$ edges. The $O(n^2)$ size of $(S, \odot_r)$ implies that the processor-time product of the naive algorithm is $\Omega(n^2)$ in the worst case.

Consider $s_i, s_j \in S$. We say that s_i and s_j are *Delaunay connected with respect to r* , denoted by $s_i \Delta_r s_j$, if and only if there exists an $r' \leq r$ and $P \in \mathbb{R}^n$ such that $d(P, s_i) = r' = d(P, s_j)$ and for all $s_k \in S - \{s_i, s_j\}$, $d(P, s_k) > r'$ [28]. Figure 3.3a illustrates the *Delaunay connected* relation with respect to the same set of objects S' given in Figure 3.2a.

It has been shown in [28] that $cl(\Delta_r) = cl(\odot_r)$ and, for object sets in $\mathbb{R}^2$, $|\Delta_r| = O(n)$. Hence, relation Δ_r induces the same clustering of S as relation $\odot_r$, but it has only a linear number of elements.

Since we are only interested in the equivalence classes of $cl(\odot_r)$, we can avoid the $\Omega(n^2)$ worst-case time-processor bound of the naive algorithm by using instead the relation Δ_r . The optical clusters of S with respect to separation parameter r are exactly the connected components of the graph (S, Δ_r) , where (S, Δ_r) denotes the graph with vertex set S and edge set containing all edges between all pairs of vertices $(s, s') \in \Delta_r$. (See Figure 3.3b.)

Let $V(S)$ and $DT(S)$ denote the *Voronoi Diagram* and its dual, the *Delaunay Triangulation*, of S , respectively [23, 99]; see Figure 3.1. It is easy to see that

Figure 3.4: A Voronoi diagram (dashed lines) and Delaunay triangulation (solid lines) from [85].

$DT(S) = \bigcup_{r \geq 0} \Delta_r$. Define for every edge $(s, s') \in DT(S)$ with corresponding dual edge e in $V(S)$ a label $min(s, s') = \min\{d(s, x) = d(s', x) \mid x \in e\}$ and call the labeled graph $(S, DT(S))$, with labeling $(s, s') \rightarrow min(s, s')$, the *cluster graph* of S , denoted $CG(S)$. As shown in [28], it follows that $\Delta_r = \bigcup_{((s, s') \in DT(S), min(s, s') \leq r)} (s, s')$.

3.2 Parallel Optical Clustering of Points Sets in

E^2

In this section we present solutions to the following optical clustering problems:

1. Determine the optical clusters of S with respect to a given separation parameter.
2. Given an interval $[a, b]$ representing the number of optical clusters desired in the clustering of S , determine the range of the separation parameter that will result in such an optical clustering.

3.2.1 Computing Optical Clusters with Respect to a Given Separation Parameter r

Let $S = \{s_1, \dots, s_n\}$ be a set of distinct planar points, arbitrarily distributed one per processor on a mesh of size n . In this section, we consider the problem of determining the optical clusters of S with respect to a given separation parameter r . The algorithm consists of first constructing $CG(S)$, and then computing the connected components of an edge-restricted subgraph of $CG(S)$.

Lemma 3.1 *The cluster graph $CG(S)$ of a set of n points in the Euclidean plane can be computed in optimal $O(\sqrt{n})$ time on a mesh of size n .*

Proof: In $O(\sqrt{n})$ time, we can construct the Voronoi diagram $V(S)$ and its dual, the Delaunay triangulation $DT(S)$, using the algorithm given in [65]. At this stage, every processor stores one (arbitrary) edge of $DT(S)$ and the corresponding edge of $V(S)$. To complete the construction of $CG(S)$, we compute in $\Theta(1)$ time, simultaneously for every edge $(s, s') \in DT(S)$, its label $\min(s, s')$. $\square$

Theorem 3.1 *Given a set S of n points in the Euclidean plane, and a real number $r > 0$, the optical clusters of S with respect to separation parameter r can be computed in optimal $O(\sqrt{n})$ time on a mesh of size n . In addition, the number $m(S, r)$ can be computed in optimal $O(\sqrt{n})$ time.*

Proof: Given the cluster graph $CG(S)$ and a real value $r > 0$, we can compute the optical clusters with respect to separation parameter r as follows. Delete all edges $(s, s') \in CG(S)$ with label $\min(s, s') > r$, and compute the connected components of $CG(S)$ with respect to the remaining edges. Note that this operation gives us the optical clusters of S with respect to r and allows us to compute the number of optical clusters, $m(S, r)$, by a parallel prefix operation. Since the connected components of a graph with $O(n)$ edges can be computed in $O(\sqrt{n})$ time on a mesh of size n [54, 102], Theorem 3.1 follows. $\square$

3.2.2 Computing Optical Clusters with Respect to a Given Range in the Number of Clusters Desired

In this section, we consider the problem of determining the maximal interval for the separation parameter r such that the number of clusters of the respective optical clusterings of a planar point set S is within a desired range.

Consider Figure 3.1. If we perform optical clustering (as described in Section 3.2.1) with respect to an r value that is too small, each cluster will consist of exactly one point, providing us with little additional information about the structure of the image. On the other hand, if we perform optical clustering with respect to a value r that is too large, then all the points in the image will form a single cluster, again providing little additional information about the structure of the image. Clearly, knowledge of a suitable r implies considerable knowledge concerning the structure of the image. In some applications, whereas we may not know a "suitable" value for r , we may have knowledge with respect to the number of optical clusters we expect in the image. For example, suppose one is to process the digitization of some preprinted form. The fields might include information such as a social insurance number, credit card number, date of birth, age, height, weight, income, and so forth. For such fields, we can exploit our knowledge with respect to an acceptable range of the number of clusters that is anticipated, as was previously discussed.

We will now show how knowledge about the expected number of clusters can be used to compute a suitable range of values for the separation parameter r . Let $[a, b]$, $a \leq b$, $a, b \in \{1, \dots, n\}$, be an interval denoting the desired range of $m(S, r)$, the number of clusters of S with respect to r . Let $R(a, b) \subset \mathbb{R}^+$ denote the corresponding set of separation parameters. That is, $r \in R(a, b)$ implies that $m(S, r) \in [a, b]$. Let $R(a)$ be the largest value $r' \in \mathbb{R}^+$ such that $m(S, r') \geq a$. Similarly, let $R(b)$ be the smallest value $r' \in \mathbb{R}^+$ such that $m(S, r') \leq b$. Since $m(S, r)$ is monotone and decreasing with respect to r , $R(a, b)$ is the closed interval $[R(b), R(a)]$. Let $CG(S)$ have k edges, and let $\min_1, \dots, \min_k$ be the values of the labels of these edges in increasing sorted order. Thus, $m(S, \min_1) \geq m(S, \min_2) \geq \dots \geq m(S, \min_k)$, and

for all i , $1 \leq i < k$, if $r \in [\min_i, \min_{i+1})$, then $m(S, r) = m(S, \min_i)$. Given the definition of $CG(S)$, notice that $R(b)$ must be an element of $\{\min_1, \dots, \min_k\}$, and $R(a)$ must be a maximum end-point of some interval $[\min_i, \min_{i+1})$. Therefore, we can determine $R(a, b)$ by computing $R(a)$ and $R(b)$, where $R(a)$ and $R(b)$ can be computed independently.

We will use a binary search type algorithm to determine $R(a)$ and $R(b)$. Our algorithm will consist of $0 \leq t \leq \log_2 n$ phases, where in each phase, the original graph CG will be further *compressed*. Let CG_t denote the compressed version of CG immediately following the t^{th} stage of the algorithm. Let $CG_t^{r'}$ denote a copy of CG_t , where all edges with labels $m(S, s') \geq r'$ have been removed. Phase 0 of the algorithm is defined by (i) setting $CG_0 = CG(S)$, where $CG(S)$ is constructed as in Section 3.2.1, and (ii) setting q_0 to either a or b , depending on which one is being searched for. The non-increasing sequence $\langle q_i \rangle = q_0, \dots, q_{\log_2 n}$ is used to keep track of the number of optical clusters remaining to be identified following the termination of phase i . The t^{th} phase of the algorithm is defined as follows.

Figure 3.5: The space in which the binary search algorithm for determining $R(x)$ is conducted.

Algorithm Phase t :

1. Let $CG_{t-1} = (V, E)$ and $CG_t = (V', E')$, where CG_{t-1} is given and CG_t is the graph to be constructed in the t^{th} phase. Determine r' , the median over the set of edge labels of CG_{t-1} .
2. (a) Construct the graph $CG_{t-1}^{r'}$ by eliminating all edges with labels greater than or equal to r' .
 (b) Label the connected components of $CG_{t-1}^{r'}$.
 (c) Compute the number of connected components of $CG_{t-1}^{r'}$.
3. TERMINATING CASES: (T1) If the number of connected components of $CG_{t-1}^{r'} = q_{t-1}$, then return r' . EXIT. (T2) If $CG_{t-1}^{r'}$ consists of a single edge with label l , then return either the label just greater than l or just less than l in the sequence $\min_1, \dots, \min_k$, depending on whether we are searching for $R(a)$ or $R(b)$, respectively. EXIT.
4. CASE A: The number of connected components of $CG_{t-1}^{r'}$ is less than q_{t-1} , i.e., r' is too large. Set E' equal to the set of edges in $CG_{t-1}^{r'}$. Notice that any vertex $v \in V$ not incident on an edge in E' will remain as a separate cluster for the remaining phases of the algorithm, since in this binary search procedure the set of edges is never increased. Therefore, identify the set U of connected components of $CG_{t-1}^{r'}$ which contain exactly one vertex, and compute $|U|$, the number of single vertex components in $CG_{t-1}^{r'}$. Set V' equal to $V - U$ and $q_t = q_{t-1} - |U|$.
5. CASE B: The number of connected components of $CG_{t-1}^{r'}$ is greater than q_{t-1} , i.e., r' is too small. This means that none of the edges in $CG_{t-1}^{r'}$ has a label that results in q_{t-1} clusters, and that every edge in $CG_{t-1}^{r'}$ links vertices that are members of the same optical cluster. Therefore, collapse the connected components of $CG_{t-1}^{r'}$ to form the set V' of "Super-Vertices", and let E' be the set of edges (s, s') between elements of V' such that there exists an edge e

in CG_{t-1} connecting the components represented by s and s' . (Note: Remove all loops and collapse multiple edges in E .) Notice that, as in CASE A, any singleton vertex in (V', E') will remain as a separate cluster for the remaining phases of the algorithm. Therefore, label the connected components of (V', E') , and identify the set W of connected components of (V', E') with exactly one vertex. Set $V' = V' - W$ and $q_t = q_{t-1} - |W|$.

End of Algorithm Phase t .

For the above algorithm, the correctness and time complexity depend on the relationship between the size of graphs CG_t and CG_{t-1} . We show that in each phase, the graph is compressed by at least a factor of $\frac{6}{5}$.

Lemma 3.2 $|CG_t| \leq \frac{5}{6} |CG_{t-1}|$.

Proof: Let CG_{t-1} have e edges and v vertices, and let CG_t have e' edges and v' vertices. Since r' is the median over all edge labels in CG_{t-1} , the result of either CASE A or CASE B is that half of the edges of CG_{t-1} are removed. Therefore, $e' \leq e/2$. Further, since singleton vertices are removed from either the connected components of CG_{t-1}' (CASE A) or the connected components of (V', E') (CASE B), we know that $v' \leq v$. Let $M = e + v$ be the size of CG_{t-1} . Since there are no isolated vertices in CG_{t-1} , we have $v \leq 2e$. Therefore, $M = e + v \leq 3e$. Finally, we obtain $M' = e' + v' \leq e' + v \leq e/2 + v = e/2 + M - e = M - e/2 \leq M - M/6 = \frac{5}{6}M$. $\square$

Theorem 3.2 *Given a set S of n points in the Euclidean plane and an interval $[a, b]$ for the number $m(S, r)$ of optical clusters desired in the completed clustering of S , the range of values for the separation parameter r that will result in such an optical clustering can be determined in asymptotically optimal $O(\sqrt{n})$ time on a mesh of size n .*

Proof: Given Lemma 3.2, we compress the new graph constructed at the end of each phase into the upper-left $m \times m$ region of the mesh, where $m^2 \leq n/(\frac{6}{5})^{t-1}$. This can be accomplished by ranking the vertices and edges remaining in the graph,

having each vertex and each edge compute its new position/processor (using the its rank and knowledge of the new graphs size) and then having each edge use a random access read operation to update itself before a final sort is used to move all vertices and edges to their new positions. Notice that after the above compression step, standard mesh operations (see Chapter 1) such as sorting, broadcasting, component labeling, random access read, random access write, and parallel prefix can be used to complete the t^{th} phase of the algorithm on a mesh of size n in $O(\sqrt{\frac{n}{t-1}})$ time. Therefore, the running time of the algorithm is given by the recurrence $T(n) = T(\frac{5}{6}n) + O(\sqrt{n})$, which implies $T(n) = O(\sqrt{n})$. $\square$

3.3 Optical Clustering for Chain Code Representations

In practice, the input is often not given as an image but in its *chain code* representation. Chain codes are a classical approach in image processing for representing regions by border codes [41, 49]. A chain code representation of a standard 8-connected component without holes is defined by the starting location (x and y coordinates) of a pixel on the border, followed by a sequence of directions indicating where the remaining border pixels are with respect to, say, the counterclockwise direction. Chain codes are used in many situations, such as handwriting analysis, where the image is relatively sparse. This not only saves space, but because the representation is often more compact, the algorithms that work on the chain codes are much more efficient than algorithms that would work on the raw image. Optical clustering methods can also easily be applied to chain codes without expanding the chain code to the (in general much larger) image. Suppose that an image is initially represented as a set of chain codes of total length c , where each 8-connected component, as well as all holes of the component, is represented by a labeled chain code. Further, suppose that each such labeled chain code is stored in a contiguous set of processors on a mesh of size c (see [41]). In time $O(\sqrt{c})$, the chain codes can be transformed into a labeled set S representing the coordinates of the border pixels.

Then, instead of solving the clustering problems for the chain codes, we simply apply our clustering methods to S , which yields the required result. Note that the size of s is still $O(c)$.

3.4 Conclusion

In this chapter, we presented asymptotically optimal mesh algorithms to solve two problems in optical clustering, both of which are important in certain areas of image analysis. To the best of our knowledge, these are the first parallel algorithms presented for these problems. These techniques allow one to determine on a mesh multicomputer (i) the optical clusters of a digitized picture for a given separation parameter r , and (ii) the maximal interval for the separation parameter r such that the number of clusters produced is within a certain range. This is important in applications such as on-line processing of handwriting (e.g., in the processing of handwritten forms) or determining the location of a handwritten address on an envelope [83].

Part II

Algorithms for Fine-Grained Hypercube Multicomputers

Chapter 4

The Hypercube Multicomputer: Definitions and Basic Operations

In this chapter we first define the hypercube multicomputer and briefly describe some standard hypercube operations. We then outline the Multi-Way Search problem [38, 87] and its hypercube solution. Note that the definition of the Multi-Way Search problem is almost identical to the Multisearch problem described in Chapter 2. The principle difference between these problems is that in the Multi-Way Search problem queries move “synchronously” through the shared data structure, while in the Multisearch problem one query may move several steps down its search path, while another is stationary.

4.1 The Hypercube Multicomputer

A d -dimensional hypercube multicomputer is a set $P_1, \dots, P_p$ of p processors ($p = 2^d$) connected in a hypercube topology; i.e., P_i and P_j are connected by a communication link if and only if the binary representations of i and j differ in exactly one bit. In a hypercube, there is no shared memory. The entire storage capability consists of local memories, one attached to each processor.

Figure 4.1: Hypercubes of dimensions $d = 1, 2, 3$ and 4 .

In the following chapters we assume, unless otherwise specified, a fine-grained model of computation, in which local memories are of size $O(1)$, and arithmetic operations on each processor and communication between processors which are adjacent along a fixed dimension are elementary instructions and take time $O(1)$.

A second model which we will occasionally consider is the *pipelined hypercube*. In this model, like in the fine-grained model, local memories are of size $O(1)$ and arithmetic operations again take time $O(1)$, but processors can communicate with *all* adjacent processors in time $O(1)$, not just a single fixed neighbor. The CM2 from Thinking Machines Corporation is an example of such a machine [100].¹

4.1.1 Standard Hypercube Operations

In the following chapters we will use many *standard hypercube operations* such as parallel prefix [56], monotonic routing [79] and bitonic merge [12].

The *parallel prefix sum* of a vector V is the vector W , with $W[k] := \sum_{i=0}^k V[i]$, $0 \leq k < p$. Instead of summing we can perform any binary associative operation, e.g. copying. A further generalization is the *segmented parallel prefix* in which the vector is divided into segments and a parallel prefix operation is computed in each segment independently. In the following chapters we also use an operation called

¹The operation that supports communication with all adjacent processors in $O(1)$ time is called a "cube swap" in Thinking Machine's documentation. Unfortunately, this operation can only be accessed in CMIS, a high level microcode.

segmented broadcast, whereby in each segment all elements are replaced by the first element in the segment. All of these parallel prefix based operations require time $O(\log p)$ on a p processor fine-grained hypercube [56].

Monotonic routing refers to a class of data movement operations originally developed by D. Nassimi and S. Sahni [79]. This class of operations includes:

Concentrate Each selected processor P_i , $0 \leq i < n$, stores a record G_i and a rank R_i . At the conclusion of a concentrate operation each record G_i is stored on processor P_{R_i} .

Distribute The distribute operation is the inverse of a concentrate operation. Initially, each processors $P_0, \dots, P_r$, $r < n$ stores data record G_i with a destination field D_i such that $D_0 < D_1 < \dots < D_r$. At the conclusion of a distribute operation each record G_i is stored on processor P_{D_i} .

Generalize The generalize operation is a modified version of the distribute operation. Initially, each processors $P_0, \dots, P_r$, $r < n$ stores data record G_i with a destination field D_i such that $D_0 < D_1 < \dots < D_r$. At the conclusion of a generalize operation each record G_i is stored on processors $P_{D_{i-1}+1}, \dots, P_{D_i}$.

Bit-Permute-Complement (BPC) In a BPC operation, initially each processor P_i , $0 \leq i < n$ of a d -dimensional hypercube stores a data record G_i and a copy of the permutation vector $B = [B_{d-1}, B_{d-2}, \dots, B_0]$. The permutation vector B specifies how the binary representation of i is to be permuted and possibly have some of its bits complemented in order to derive a destination address i' . At the conclusion of a BPC operation each record G_i is stored on processors $P_{i'}$.

For a more detailed discussion of these operations and their implementations see [86, pages 23–93]. All of these operations require $O(\log p)$ time on a p processor fine-grained hypercube.

A bitonic sequence is any sequence that can be derived by performing a cyclic shift of a unimodal sequence (See [12]). The *bitonic merge* operation transforms a

bitonic sequence into a sorted sequence in $O(\log p)$ time on a p processor fine-grained hypercube [12, 78]. A special case of a bitonic sequence that we will frequently consider is a sequence which consists of a set of non-decreasing values followed by a set of non-increasing values.

All of the basic hypercube operations considered above can be implemented in terms of a loop that scans sequentially through the dimensions of a hypercube. This being the case, on the pipelined hypercube these operations can be implemented using pipelining to achieve $O(\frac{n}{p} + \log p)$ time bounds, when $n \geq p$. For a further discussion of the pipelined hypercube model see [71, 108].

4.1.2 Sorting on the Hypercube

There are many different approaches to sorting on fine-grained hypercube multicomputers [1, 12, 17, 26, 78]. From an asymptotic perspective Cypher and Plaxton's Share Sort [26] is the algorithm of choice with a complexity of $O(\log n \log^2 \log n)$ for sorting n items. A slightly modified version of this algorithm, which makes an additional assumption about the computational model, appears in Leighton's book [69] and requires only $O(\log n \log \log n)$ time. In practice, for hypercubes having less than about 2^{24} processors, much simpler algorithms, like those based on Batcher's bitonic sort [12] are likely to be more efficient. For a practical examination of sorting for the CM-2, see [17].

4.1.3 Multi-Way Search on Trees and General Graphs

Before presenting our new parallel algorithms, we introduce some notations and previous results which will be used in the remainder. In particular, we use a procedure called *multi-way search*: Given a tree stored on a hypercube multicomputer, m search queries on that tree are to be executed independently and in parallel. At each time step, each query currently visiting a node of the tree decides which adjacent node to visit next, and is then moved to that node. Note that, each node can be concurrently visited by an arbitrary number of queries.

More formally, let $T = (V, E)$ be a tree of size k , height h , and out-degree $O(1)$, and let U be a universe of possible search queries on T . A *search path* for a query $q \in U$ is a sequence $path(q) = (v_1, \dots, v_h)$ of h vertices of T defined by a *successor function* $f : (V \cup start) \times U \Rightarrow O(V)$ (i.e., a function with the property that $f(start, q) \in V$ and for every vertex $v \in V$, $(v, f(v, q)) \in E$). A *search process* for a query q with search path $(v_1, \dots, v_h)$ is a process divided into h time steps $t_1 < t_2 < \dots < t_h$ such that at time t_j , $1 \leq j \leq h$, query q is *matched* with node v_j . A *match* of a query q and a node v_i at time t_j is defined as a situation where there exists a processor which contains a description of both, the query q and the node v_i . Note, however, that we do not assume that the search path is given in advance; we assume that it is constructed during the search by successive applications of the functions f . Given a set $Q = q_1, \dots, q_m \subseteq U$ of m queries, $m = O(k)$, then the *multi-way search problem* consists of executing (in parallel) all m search processes induced by the m queries. In the following chapters we will also refer to queries as messages.

The best way to visualize this process is to depict each search process as a pebble, representing the respective query and moving through the tree T . A pebble may only move along edges of T , but it can traverse them in both directions. The multi-way search problem consists of m such pebbles moving simultaneously through the tree. At each time step, every pebble decides which node to "visit" next, and then all pebbles are simultaneously moved. Note that, each node of the tree may be "visited", at any time, by an arbitrary number of pebbles.

On a PRAM with $\max(k, m)$ processors multi-way search can be easily implemented in time $O(h)$. Each query is simply represented by one processor, navigating it through the tree, stored in the shared memory. The PRAM's concurrent read capability ensures that queries "visiting" the same node do not interfere.

For hypercube multicomputers, it was shown in [34, 38] that the multi-way search problem can be solved in time $O(\log p \log^2 \log p + h \log p)$ on a hypercube of size $p = \max(k, m)$. The algorithm applies to a class of graphs called *ordered h -level graphs* (see [38] for a precise definition) which includes the class of all trees with

constant degree. The following outlines the global structure of the algorithm (applied to the special case of search trees): Consider the *level numbering* of the nodes of a tree as indicated in Figure 4.2. Each node with level number i , together with its links and data, is assumed to be stored on processor P_i . The m search queries

Figure 4.2: Level numbering of the nodes of a tree.

are assumed to be stored in arbitrary order (with each processor storing at most one query). Initially, the m queries are sorted by the level number of the first node on their search paths². The m search processes for the m queries $q_1, \dots, q_m$ are then executed simultaneously in h phases, each requiring time $O(\log(\max(k, m)))$. Each phase moves all queries one step ahead in their search paths. In each phase, the queries are permuted such that they are sorted with respect to the level order number of the respective node they want to visit next. Furthermore, a copy of the search tree is created and its nodes are permuted such that, at the end of each phase, each processor containing a query q_i also stores a copy of the node the query wants to visit next. See [38] for a full description of the algorithm.

Consider the problem of changing the tree T or the set Q of queries during the execution of a multi-way search. That is, during the search (more precisely, at the end of each phase of the algorithm outlined above) leaves may be added to T ,

²This step can clearly be avoided if all queries start at the root of the tree

subtrees may be deleted from T , and queries may duplicate or delete themselves. This problem is referred to as the *dynamic multi-way search* problem.

On the PRAM, the dynamic multi-way search problem can be solved in time $O(h \log(\max(k, m)))$. The reason for the increased time complexity is that the assignment of processors to new queries and the assignment of storage space of deleted nodes to newly created ones may require a partial sum operation for each phase of the algorithm, which slows down the static solution by a factor of $O(\log(\max(k, m)))$.

For the hypercube it has been shown in [87] that this problem can be solved such that the time complexity of each phase is still $O(\log(\max(k, m)))$. That is, the time complexity of the entire multi-way search procedure for the dynamic case is still $O(\log p \log^2 \log p + h \log(p))$ on a hypercube of size $p = \max(k, m)$. For trees with unbounded out-degree, as well as arbitrary graphs, the time complexity increases to $O(h \log p \log^2 \log p)$ on a hypercube of size $p = \max(k, m)$.

Chapter 5

Parallel Construction and Manipulation of Quadtrees on a Hypercube

A *region quadtree* is a well known hierarchical data structure for representing a binary image of size $\sqrt{M} \times \sqrt{M}$ ($\sqrt{M} = 2^r$ for some positive integer r). The root of the quadtree represents the entire image and has a value “black”, “white”, or “gray” depending on whether the entire image is black, white, or composed of both types of pixels, respectively. If the root is gray, it has four children which are the roots of quadtrees recursively representing the four quadrants of the image; otherwise it has no children. For the remainder, we do not differentiate between a node of a quadtree and the portion of the image represented by that node. There are two widely used representations of quadtrees. A *pointer based quadtree* uses the standard tree representation. A *linear quadtree* can be either a preorder traversal of the nodes of a quadtree, or the sorted sequence (with respect to the preorder of the tree) of the quadtree’s leaves. Some linear quadtree representations of the second type store with each leaf also a code sequence representing the path from the root to that leaf (*linear quadtree with path encoding*) while others store for each leaf only its size and location (*linear quadtree without path encoding*). For an overview and bibliography on quadtrees and applications we refer to the work of Samet [92, 93].

Quadrees are an efficient and widely used data structure for image processing applications, and many quadtree algorithms image processing tasks have been

t	time complexity
s	total memory space
p	number of processors
M	number of pixels in original image
b	length of the boundary code
N	size of pointer based quadtree
N'	size of linear quadtree with path encoding
n	size of linear quadtree without path encoding
h	height of the quadtree

Table 5.1: Overview of parameters used in quadtree algorithms.

Problem	pointer based quadtree	
	hypercube ^a	PRAM ^a
binary image to quadtree	$s = p = M$ $t = O(\log^2 M)$	$s = M$ $t = O(\log M)$ $p = \frac{M}{\log M}$
boundary code to quadtree	$s = p = b$ $t = O(h \log b)$	$s = p = b$ $t = O(h \log b)$

Table 5.2: Parallel pointer based quadtree construction algorithms.

^aAlgorithms presented in this thesis.

developed [92]. Recently, researchers have also started to consider quadtree algorithms for parallel models of computation [15, 43, 57, 70, 72]. While some papers [70, 72] consider parallel architectures designed (or reconfigured) particularly for quadtree manipulation, others [15, 57] consider the general purpose architectures mesh-connected computer and PRAM, respectively. Hung and Rosenfeld [57] study mesh-connected computer algorithms for constructing and manipulating linear quadrees without path encoding; the obtained results are listed in Tables 5.3 and 5.5 (rightmost column). Table 5.1 lists the parameters that will be used for the remainder of this chapter. PRAM algorithms for manipulating linear quadrees with path encoding were studied by Bhaskar, Rosenfeld and Wu [15]; the obtained results are listed in Table 5.5 (second column from the right).

Problem	linear quadtree		
	hypercube ^a	PRAM ^a	mesh ^b
binary image to quadtree	$s = p = M$ $t = O(\log^2 M)$	$s = M$ $t = O(\log M)$ $p = \frac{M}{\log M}$	$s = p = M$ $t = O(\sqrt{M})$
boundary code to quadtree	$s = p = b$ $t = O(h \log b)$	$s = p = b$ $t = O(h \log b)$	

Table 5.3: Parallel linear quadtree construction algorithms.

^aAlgorithms presented in this thesis.^bAlgorithms presented in [57].

Problem	pointer based quadtree	
	hypercube ^a	PRAM ^a
determine neighbors of all leaf nodes or compute perimeter	$s = p = N$ $t = O(h \log N)$	$s = p = N$ $t = O(h)$
compute area or compute centroid	$s = p = N$ $t = O(\log N)$	$s = N$ $t = O(\log N)$ $p = \frac{N}{\log N}$
compute complement	$s = p = N$ $t = O(1)$	$s = p = N$ $t = O(1)$
compute union compute intersection	$s = p = N$ $t = O((h + \log^2 \log N) \log N)$	$s = p = N$ $t = O(h + \log N)$
rotate by 90°	$s = p = N$ $t = O(h + \log N \log^2 \log N)$	$s = p = N$ $t = O(h + \log N)$

Table 5.4: Parallel pointer based quadtree manipulation algorithms.

^aAlgorithms presented in this thesis.

Problem	linear quadtree		
	hypercube ^a	PRAM ^b	mesh ^c
determine neighbors of all leaf nodes or compute perimeter	$s = p = N'$ $t = O(h \log^2 N' \log^2 \log N')$	$s = p = N'$ $t = O(h \log N')$	$s = p = n$ $t = (\sqrt{n})$
compute area/ compute centroid	$s = p = N'$ $t = O(\log N')$	$s = p = N'$ $t = O(\log N')$	$s = p = n$ $t = (\sqrt{n})$
compute complement	$s = p = N'$ $t = O(h \log N' \log^2 \log N')$	$s = p = N'$ $t = O(h)$	$s = p = n$ $t = (\sqrt{n})$
compute union compute intersection	$s = p = N'$ $t = O(h \log^2 N' \log^2 \log N')$	$s = p = N'$ $t = O(h \log N)$	$s = p = n$ $t = (\sqrt{n})$
determine connected component			$s = p = n$ $t = (\sqrt{n})$

Table 5.5: Parallel linear quadtree manipulation algorithms.

^aObtained from [15] by using standard PRAM simulation on a hypercube as described by Nassimi and Sahni [79], together with Cypher and Plaxton's deterministic hypercube sorting algorithm [26, 69].

^bAlgorithms presented in [57].

^cAlgorithms presented in [15].

In this chapter, we extend the previous results in two directions. Firstly, we study parallel quadtree construction algorithms for the hypercube and PRAM. We describe algorithms for converting images, represented either by a binary array or a boundary code, into pointer based as well as linear quadrees. Tables 5.2 and 5.3 summarize the obtained results. Secondly, we study parallel manipulation methods for pointer based quadrees. We show that, for parallel processing on a hypercube or PRAM, pointer based quadrees are an attractive alternative to the linear quadrees studied in the previous literature. The parallel manipulation algorithms for pointer based quadrees presented in this chapter improve, in the expected case, on previously presented methods. Table 5.4 summarizes the obtained results. Note that, the algorithms in [15] apply to linear quadrees *with* path encoding. In the expected case, the height, h , of the quadtree is $O(\log N)$ [14, 47, 63]. Hence, $N = O(N')$; i.e., the linear and pointer based quadrees have, asymptotically, the same space requirement. In this case, we obtain improvements in the time complexity for several problems, such as the following:

1. Computing the neighbors of all leaf nodes and the perimeter of an image, and
 - hypercube: $O(h \log N)$ vs. $O(h \log^2 N \log^2 \log N)$
 - PRAM: $O(h)$ vs. $O(h \log N)$
2. Computing the union/intersection of two quadrees.
 - hypercube: $O(\log N(h + \log^2 \log N))$ vs. $O(h \log^2 N \log^2 \log N)$
 - PRAM: $O(h + \log N)$ vs. $O(h \log N)$.

In the *worst case*, $h = O(N)$, the linear quadtree with path encoding needs to store one path requiring $O(h)$ bits, while the pointer based quadtree needs $O(h)$ pointers of $O(\log h)$ bits each; that is, $N = O(N' \log h)$. In this case, we obtain a time space trade-off between the above time complexity improvements and increased storage for pointer based quadtree algorithms. Note that, for the hypercube, the space increases by a factor smaller than the time complexity improvement, and for the PRAM both factors are equivalent. In [59] the conversion of binary images to quadrees was considered for a different hypercube model with secondary memory devices.

The remainder of this chapter is organized as follows. In Section 5.1, we discuss some preliminaries concerning the storage of quadrees on a hypercube multicomputer. In Section 5.2, we present efficient hypercube and PRAM algorithms for *constructing* a (pointer based or linear) quadtree from a binary image or from an image represented by its boundary code. In Section 5.3, we introduce efficient parallel hypercube and PRAM algorithms for manipulating pointer based quadrees. The results described in this chapter have appeared in [36].

5.1 Storing Pointer Based Quadrees on a Hypercube Multicomputer

On a PRAM, a pointer based quadtree can be stored in the same way as on a standard sequential machine. For the hypercube, however, we require a scheme for distributing a quadtree over its local memories. Consider the *level order numbering* of the nodes of a quadtree as described in Section 4.1.3. For the remainder we will

assume that each node with level order number i , together with the attached data and pointers to its children, is stored at processor P_i .

5.2 Constructing Quadtrees from Binary Images and Boundary Codes

5.2.1 Quadtree from Binary Image

Consider a $\sqrt{M} \times \sqrt{M}$ binary image stored on a PRAM (with M processors and $O(M)$ storage space) or a hypercube (with M processors) in row-major numbering (see Figure 5.1a). That is, for the hypercube, processor P_i stores the pixel with row-major number i .

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

(a)

1	2	5	6
3	4	7	8
9	10	13	14
11	12	15	16

(b)

Figure 5.1: (a) Row-Major numbering (b) Shuffled Row-Major numbering.

The following is an outline of a general parallel algorithm, for the hypercube and PRAM, for computing a pointer based quadtree from such a binary image representation. The implementation details follow.

Algorithm Binary Image to Quadtree:

1. For each pixel (in parallel) its shuffled row major number (as indicated in Figure 5.1b) is computed.
2. All pixels are permuted according to shuffled row-major number.
3. A complete 4-ary tree, with the sorted sequence of pixels as leaves, is built.

4. From each leaf a message is sent along the path to the root of the tree. The messages move synchronously upwards from level to level. At each level, the following is executed: If all four messages reaching a node x come from black white children, then x is set to black white and its children are marked "to be deleted". If the messages reaching x are from children with different color, x is set to gray.
5. All nodes marked "to be deleted" are deleted, the remaining nodes are compressed to form a consecutive sequence, and all pointers are updated.

End of Algorithm Binary Image to Quadtree.

Theorem 5.1 *The pointer based quadtree representation of a $\sqrt{M} \times \sqrt{M}$ binary image can be computed in time $O(\log^2 M)$ and $O(\log M)$ on a hypercube and PRAM, respectively, with $s = p = M$.*

Proof: From the definition of quadrees it follows that the tree generated by the above algorithm is the correct quadtree. What remains to be shown is that the above algorithm can be implemented, on the hypercube and PRAM, within the claimed time complexity bounds. Step 1 requires only the local computation of the shuffled row-major number of the respective pixel at each processor. For a $\sqrt{M} \times \sqrt{M}$ image, this takes $O(\log M)$ local computation steps (for the PRAM and hypercube). Step 2 can be performed using a BPC operation in time $O(\log M)$ on either the hypercube [80] or PRAM [25]. Step 3 can be implemented by building the tree level by level, starting with the leaves (which are given). Since it is a complete tree, at each stage the addresses of the nodes of the subsequent level can be immediately computed. This results in an $O(h) = O(\log M)$ time algorithm for the PRAM. For the hypercube, because the tree is stored in level order, routing the nodes of the subsequent level to their respective positions requires only a concentrate followed by a distribute operation. Thus, Step 3 requires time $O(\log^2 M)$ on the hypercube. Step 4 is a multi-way search operation as outlined in Section 4.1.3, with traveling messages represented by query processes. Hence, it requires time

$O(h \log M) = O(\log^2 M)$ and $O(\log M)$ on the hypercube and PRAM, respectively. Note that Step 4 does not change the topology of the tree but marks only the nodes to be deleted. In Step 5 the marked nodes are deleted by compressing the sequence of the remaining (non marked) nodes. This can be accomplished, again due to the level ordering of the tree, by a $O(\log M)$ time concentrate operation on either the hypercube or PRAM. The problem of updating the pointers (address references between tree nodes) can be solved using a constant number of concentrate and distribute operations as illustrated by the updateTree procedure given in [33, 87]. Hence, Step 5 requires time $O(\log M)$ on both the hypercube and PRAM. $\square$

Linear quadtrees without path encoding can be constructed in essentially the same way by marking in Step 4 also gray nodes as "to be deleted". For linear quadtrees with path encoding, we also need to compute (between Steps 4 and 5) the path encoding for each leaf by applying one additional multi-way search procedure.

Corollary 5.1 *The linear quadtree representation (with or without path encoding) of a $\sqrt{M} \times \sqrt{M}$ binary image can be computed in time $O(\log^2 M)$ and $O(\log M)$ on a hypercube and PRAM, respectively, with $s = p = M$.*

5.2.2 Quadtree from Boundary Code

Consider an image I described by a *boundary code* of length b ; i.e., a sequence $a_1, \dots, a_b$ of b *boundary elements* $a_i \in \{r, l, u, d\}$ as shown in Figure 5.2 (see [90]). The image I consists of the entire area inside the *boundary line* defined by the boundary code. The unit size pixels of I that are adjacent to the boundary line are called *boundary pixels* (see Figure 5.2). For the remainder, let S_I denote a smallest (isothetic) square containing I . Note that S_I has a width of at most b .

Samet in [90] gives a sequential algorithm for building a pointer based quadtree from an image described by a boundary code. His algorithm consists of two phases. Phase 1 computes a quadtree representing only the boundary pixels of I , which Samet refers to it as the *quadtree template*. The missing children of an internal node x , at the end of Phase 1, are referred to as *absent children* of x . Note that, all absent

Figure 5.2: Boundary code and boundary pixels of an image.

children are leaves. In Phase 2 of Samet's algorithm, leaf nodes corresponding to the black and white area inside and outside the boundary line, respectively, are created. In this section we describe a parallel algorithm with the same high level structure as Samet's. It consists of two phases - each phase takes the same input as Samet's corresponding phase and produces the same output. Our contribution is in showing how each phase can be efficiently implemented in the parallel setting.

Algorithm Boundary Code to Quadtree — Phase 1:

1. For each boundary element, its absolute address (in S_I) is computed, and the adjacent boundary pixels are created (see [90]).
2. The shuffled row-major number of each boundary pixel with respect to S_I is computed.
3. All boundary pixels are permuted with respect to their shuffled row-major number.
4. A quadtree with the above sequence of boundary pixels as leaves is built. For nodes with less than four children, for each missing child a pointer with value "absent" is created.

End of Algorithm Boundary Code to Quadtree — Phase 1.

We now have a quadtree template as defined by Samet in [90]. We can therefore make use of the following lemmas.

Lemma 5.1 ([90]) *After Algorithm Boundary Code to Quadtree: Phase 1, if an absent child of a node is black {white}, then all other absent children of a node are black {white}.*

Lemma 5.2 ([90]) *After Algorithm Boundary Code to Quadtree: Phase 1, consider a node, x , with at least one absent child. Choose an absent child R adjacent to a non-absent (black or gray) sibling Q , and a non-absent leaf q in the subtree rooted at Q which is adjacent to R . If q is white then R is white. If q is black and adjacent to the boundary line, then R is white if the boundary line is between q and R , and black if the boundary line does not separate them. If q is black and not adjacent to the boundary line, then R is black.*

The following outlines the remainder of the algorithm. Implementation details follow.

Algorithm Algorithm Boundary Code to Quadtree — Phase 2:

1. From each leaf, a message is sent to the root of the tree. The messages move synchronously upwards from level to level. For each node, a value *Nodetype* is determined which indicates for each side of its respective quadrant whether it is completely inside the image I , completely outside of I , or intersected by the border line (see also [90]). Note that the *Nodetype* value for every boundary pixel (leaf of template quadtree) is given; for every internal node, given the *Nodetype* values of all its children, its *Nodetype* value can be easily determined in constant time. For each internal node x with at least one absent child, the absent children are created and their values are determined as follows:
 - (a) An absent child R adjacent to a non-absent child Q is selected. The color of R is determined according to Lemma 5.2. However, the color of R

is determined directly from the *Nodctype* value of Q rather than from the leaf q referred to in Lemma 5.2. All other absent children of x are assigned the same color as R (Lemma 5.1).

(b) The *Nodctype* values of the previously absent children are determined. Finally, the *Nodctype* value of x is computed.

2. From each leaf, a message is sent to the root of the tree. The messages move synchronously upwards from level to level. (This is ensured by wait loops for messages starting at leaves of smaller depth.) At each level, the following is executed: If all four messages reaching a node x come from black {white} children, then x is set to black {white} and its children are marked "to be deleted". If the messages reaching x are from children with different color, then x is set to gray.

3. All nodes marked "to be deleted" are removed, the remaining nodes are compressed to form a consecutive sequence, and all pointers are updated.

End of Algorithm Algorithm Boundary Code to Quadtree — Phase 2.

Theorem 5.2 *The pointer based quadtree representation of a binary image described by a boundary code of length b can be computed in time $O(h \log b)$ on either a hypercube or PRAM, with $s = p = b$.*

Proof: The correctness of the algorithm follows from Samet [90]. What remains to be shown is that the individual steps listed in the above two phases can be implemented with the claimed time complexity.

We start by describing a hypercube and PRAM implementation of Phase 1. For Step 1, the x -coordinates of the absolute addresses are computed by assigning a value 1, -1, 0, and 0 to the boundary elements r, l, u, d , respectively, and computing the partial sums of this sequence. All y -coordinates are computed analogously. For each boundary element, the creation of the boundary pixels requires only information about the directly adjacent boundary elements; otherwise, it is a local $O(1)$ time operation. Hence, for the hypercube and PRAM, Step 1 can be executed in $O(\log b)$

time. Step 2 requires $O(\log b)$ local computation steps at each processor. Step 3 requires time $O(\log b \log^2 \log b)$ on the hypercube [26] or time $O(\log b)$ on the PRAM [25]. Step 4 can be implemented by building the tree level by level, starting with the leaves (which are given). At each level, every node (initially leaves) examines its three neighbors to the right and left and determines (using the shuffled row-major numbering and current level information) with whom a common ancestor is to be created. This can be implemented on the hypercube with $O(\log b)$ time per level (and obviously in the same time on the PRAM), by using a constant number of partial sum as well as concentrate and distribute [79] operations.

At the beginning of Phase 2, we have a quadtree template representing only the boundary pixels of the image I . The nodes corresponding to the black and white area inside and outside the boundary line, respectively, are now created by successive dynamic multi-way search procedures. In Step 1, a dynamic multi-way search procedure is used to add and update the absent children. [hypercube, PRAM: $O(h \log b)$]. Steps 2 and 3 are identical to Steps 4 and 5 of the algorithm Binary Image to Quadtree. Therefore, Step 2 can be implemented on a hypercube and PRAM in time $O(h \log b)$ and $O(h)$ respectively; Step 3 requires time $O(\log b)$ on either architecture. □

Linear quadrees without path encoding can be constructed in essentially the same way by also marking in Step 2 of Phase 1 gray nodes as "to be deleted". For linear quadrees with path encoding, we also need to compute (between Steps 2 and 3 of Phase 2) the path encoding for each leaf by applying one additional multi way search procedure.

Corollary 5.2 *The linear quadtree representation (with or without path encoding) of a binary image represented by a boundary code of length b can be computed in time $O(\log b(h + \log^2 \log b))$ and $O(h \log b)$ on a hypercube and PRAM, respectively, with $s = p = b$.*

5.3 Operations on Quadrees

Numerous region properties of images such as the area or centroid, which are simply associative functions of the leaves (and do not need neighboring information), can be immediately calculated by partial sum operations (see [15]). This requires time $O(\log N)$ on a hypercube with $s = p = N$ or a PRAM with $s = N$ and $p = N \log N$. In the remainder of this section we study the following more complex Quadtree operations.

- Finding Neighbors in Quadrees.
- Rotating Quadrees By 90° .
- Forming the Union or Intersection of Quadrees.

5.3.1 Finding Neighbors in Quadrees and Computing Region Properties

One of the main advantages of using the pointer based quadtree is that, once the quadtree has been constructed, parallel searching algorithms on quadrees can be easily adapted from the existing sequential methods by using the dynamic multi-way search technique outlined in Section 4.1.3. One of the most important building blocks of quadtree applications are neighbor finding techniques. For a leaf x representing a quadrant χ , a neighbor of x is a leaf y representing a quadrant that is adjacent to χ (with respect to the image) and has at least the same size as χ [91]. The *multiple neighbor finding problem* consists of finding the neighbors of all leaves of the quadtree.

Theorem 5.3 *Given a pointer based quadtree of size N stored on a hypercube or PRAM with $s = p = N$, then the multiple neighbor finding problem can be solved in time $O(h \log N)$ and $O(h)$, respectively.*

Proof: The sequential method described in [91] for finding the neighbor y of one single leaf x traverses the tree from x upwards, along path $\eta(x)$, to the lowest common ancestor of x and y ; then it descends downwards to y by using the “mirror image” of the upwards path $\eta(x)$. The main problem with parallelizing this method to parallel traversals for all leaves of the tree, using multi-way search, is that a message used in multi-way search may only be of constant size and, thus, cannot store the path $\eta(x)$. Assume W.L.O.G. that the right neighbor of x is to be determined. Let α denote the right border of the quadrant associated with x , and let β denote the line defined by extending α . We observe that a query can also be routed from a leaf x to its right neighbor y (along the same path as described above) as follows: The query moves upwards from x until it reaches a node whose associated quadrant intersects β . Then, it descends downwards by selecting always the child whose associated quadrant is adjacent to α . Hence, a query process to be routed from x to its neighbor y needs to store only α and β . With this, multiple neighbor finding reduces to multi-way search and, thus, the theorem follows. $\square$

Note that a black leaf node represents a region on the perimeter of the associated image if and only if it has at least one white neighbor. Once the neighbors of each leaf in all four directions have been determined, the calculation of the perimeter of the image therefore follows immediately (see [15]).

Corollary 5.3 *Given a pointer based quadtree of size N stored on a hypercube or PRAM with $s = p = N$, then the perimeter of the associated image can be computed in time $O(h \log N)$ and $O(h)$, respectively.*

5.3.2 Rotating Quadrees By 90°

Given a pointer based quadtree T , the following algorithm computes the quadtree T' for the image of T rotated by 90° on a hypercube or PRAM, with $s = p = N$.

Algorithm Rotate Quadtree By 90° :

1. For each node, the position of the rotated associated quadrant is computed.

2. For each rotated quadrant, the shuffled row-major number (with respect to the partitioning into quadrants of the same size) is computed.
3. The nodes are permuted by major key level and minor key shuffled row-major number.
4. All nodes are resorted to their original position in the old tree. Each node sends its new address to its parent.
5. All nodes are again sorted by major key level and minor key shuffled row-major number.

End of Algorithm Rotate Quadtree By 90° .

Theorem 5.4 *Given a pointer based quadtree T of size N stored on a hypercube or PRAM with $s = p = N$, then the quadtree T' representing the image, associated with T , rotated by 90° can be computed in time $O(h + \log N \log^2 \log N)$ and $O(h + \log N)$, respectively.*

Proof: The correctness of the algorithm follows from the observation that if a node v is the parent of a node w in T then the node in T' representing the rotated quadrant of v is also the parent of the node in T' representing the rotated quadrant of w . The computation of the shuffled row-major number in Step 2 requires $O(h)$ local computation steps at each processor. The remainder of the algorithm reduces to a constant number of sorting operations. Therefore, the time complexities follow. $\square$

5.3.3 Forming the Union, Intersection, and Complement of Quadrees

The union (intersection) of two quadrees T_A and T_B is defined as the quadtree $T_{A \cup B}$ ($T_{A \cap B}$) representing the image composed of the bitwise OR (AND) of the two original images. The complement of a quadtree T_A is defined as the quadtree $T_{\neg A}$ representing the image composed of the bitwise NOT of the original image.

In this section, we study the parallel computation of the union, intersection, and complement of two pointer based quadtrees. Below, we introduce some definitions that will be used in the remainder of this section. A tree T_{A+B} is called an *overlay* of T_A and T_B if it is the smallest 4-ary tree such that for each node v of T_A or T_B there exists a node $\delta(v)$ in T_{A+B} representing the same image area (assuming that T_{A+B} represents an image subdivision defined in standard quadtree fashion). The *combined level order numbering* of T_A and T_B is defined as follows: For each node v of T_A and T_B , the combined level order number $\eta_{A+B}(v)$ is the level order number of $\delta(v)$ in T_{A+B} . The *shuffled row major number* of a node v of T_A (or T_B) is the shuffled row major number of the associated quadrant with respect to the subdivision of the image plane into quadrants of the same size.

Figure 5.3: Combined level order numbering scheme.

We assume that both quadtrees are stored by level order number as indicated in Section 5.1. As a preprocessing step, we convert this storage scheme into a *combined level order numbering scheme* where every node v of T_A or T_B is stored at processor number $\eta_{A+B}(v)$. Note that every processor stores at least one node, but at most two nodes, one of each tree. The new relative order of the nodes of one tree, say T_A , is the same as their order in the initial level order numbering of T_A . The combined level order numbering scheme can be obtained as follows: All nodes are sorted by major

key level (height within their tree) and minor key shuffled row major number. For any two nodes with the same major and minor keys stored in two adjacent processors P_i and P_{i+1} , the node in P_{i+1} is moved to P_i . Finally the contents of the processors are shifted leftwards so that processors without data are avoided. Given this storage scheme for the two quadrees T_A and T_B , the following is an outline of a parallel algorithm for computing the quadtree $T_{A \cup B}$. Our algorithm uses dynamic multi-way search (see Section 4.1.3) with three different types of messages: "compare", "copy" and "update" messages.

Algorithm Compute Union of Two Quadrees:

1. From each of the roots of T_A and T_B , a wave of "compare" messages is sent towards the leaves. That is a "compare" message is sent to each root and each node, receiving a message, duplicates it and sends one to each child (within its own tree). Messages move synchronously downwards from level to level. During this process, a new tree T is created, which will subsequently be converted into $T_{A \cup B}$. At each level, the following is executed:

- (a) Each node x receiving a "compare" message, compares itself with the respective node y (representing the same image area) of the other tree. The node y is stored at the same processor P as node x and receives a "compare" message at the same time as node x does. Unless x and y are the roots of T_A and T_B respectively, let $\text{parent}(x)$ and $\text{parent}(y)$ denote their respective parents. Note that, $\text{parent}(x)$ and $\text{parent}(y)$ are both "gray" nodes stored at the same processor P' and, previously, received a "compare" message at the same time.

Case 1: x and y are both "gray":

A new "gray" node z for T representing the same quadrant as x and y is created and stored at processor P . Note that, $\text{parent}(x)$ and $\text{parent}(y)$ previously created a "gray" node z' for T . This node z' is made the parent of z in T .

Case 2: x or y is "black":

A new "black" node z for T representing the same quadrant as x and y is created and stored at processor P . The "gray" node z' created by $\text{parent}(x)$ and $\text{parent}(y)$ is made the parent of z in T . The two "compare" messages which reached x and y are not forwarded but deleted.

Case 3: One node, x or y , is gray and the other node is white:

A new "gray" node z for T representing the same quadrant as x and y is created and stored at processor P . The "gray" node z' created by $\text{parent}(x)$ and $\text{parent}(y)$ is made the parent of z in T . The "compare" message which reached the "white" node is deleted. The "compare" message which reached the "gray" node, is changed to a "copy" message, duplicated, and forwarded to all children.

Case 4: x and y are both "white":

A new "white" node z for T representing the same quadrant as x and y is created and stored at processor P . The "gray" node z' created by $\text{parent}(x)$ and $\text{parent}(y)$ is made the parent of z in T . The two "compare" messages which reached x and y are not forwarded but deleted.

- (b) Each node x receiving a "copy" message (in the other tree there exists no node y representing the same quadrant) creates a new node z for T with the same color as x and representing the same quadrant. The node z' created by $\text{parent}(x)$ and $\text{parent}(y)$ is made the parent of z in T . A "copy" message is sent to each child, or the message is deleted if x is a leaf.

2. From each leaf an "update" message is sent to the root of the tree. The "update" messages move synchronously upwards from level to level. (This is ensured by wait loops for messages starting at leaves of smaller depth.) At each level, the following is executed: If all four "update" messages reaching a

node x come from black {white} children, then x is set to black {white} and its children are marked "to be deleted". If the "update" messages reaching x are from children with different color, x is set to gray.

3. All nodes marked "to be deleted" are deleted, the remaining nodes are compressed to form a consecutive sequence, and all pointers are updated.

End of Algorithm Compute Union of Two Quadrees.

Computing the intersection of two pointer based quadrees is analogous. All steps of the above algorithm remain unchanged except for Cases 2, 3, and 4 where "black" and "white" should be exchanged.

Theorem 5.5 *Given two pointer based quadrees with a total number of N nodes stored on a hypercube or PRAM with $s = p = N$, then the union {intersection} of these quadrees can be computed in time $O((h + \log^2 \log N) \log N)$ and $O(h + \log N)$, respectively, where h denotes the maximum height of the two trees.*

Proof: In order to observe the correctness of Algorithm Compute Union of Two Quadrees we first study the intermediate tree T created at the end of Step 1. Consider two nodes x and y in T_A and T_B representing the same quadrant. Then, a node z in T is created in Step 1a (a "compare" message reaches x and y), and it is easy to see that through Cases 1 to 4 the right color, representing the union {intersection} of x and y , is assigned to z . Consider, on the other hand, a node x for quadrant χ in, say, T_A with no node in T_B representing the same quadrant. Then, T_B has a leaf y for a quadrant γ containing χ . Let x' be the ancestor of x representing quadrant γ . If γ is "black" {"white"} then no node needs to be created in T , which is guaranteed by the deletion of the "compare" messages reaching x' and y (Step 1a, Case 2). If γ is "white" {"black"} then the entire subtree rooted at x' has to be copied into T . This is achieved by the "copy" messages started at x' (Case 3 of Step 1a and Step 1b).

In order to prove the claimed time complexity, we first observe that the preprocessing reduces to a sorting operation followed by a concentrate [79] and partial

sum for the hypercube and PRAM, respectively. Hence, its time complexity is $O(\log N \log^2 \log N)$ and $O(\log N)$ on the hypercube and PRAM, respectively. The combined level order numbering scheme used to store the trees T_A , T_B , and T allows simultaneous multi-way search on all three trees, because T_A , T_B and T are subtrees of T_{A+B} , and all nodes are stored with respect to their level order number in T_{A+B} , (see Section 4.1.3 and Section 5.1). Hence, Step 1 can be implemented on a hypercube using the dynamic multi-way search procedure outlined in Section 4.1.3. That is, Step 1 requires time $O(h \log N)$ on the hypercube. We observe that, during Step 1, at any time no tree node is visited by more than one message. Therefore advancing all messages from one level of the tree to the next level can be implemented, on the PRAM, in time $O(1)$. This is due to the fact that for assigning processors to messages we do not require a partial sum operation as in the general case, but we can use a fixed scheme where every processor is assigned to one node and responsible for the message visiting that node. Hence, Step 1 requires time $O(h)$ on the PRAM. Steps 2 and 3 are equivalent to Steps 4 and 5 of the algorithm in Section 5.2.1. Hence, from Theorem 5.1, their time complexity is $O((h + \log^2 \log N) \log N)$ and $O(h + \log N)$ on the hypercube and PRAM, respectively. $\square$

5.4 Conclusion

In this chapter we have shown that, for parallel processing on a PRAM or hypercube, pointer based quadrees are an attractive alternative to linear quadrees. We presented efficient PRAM and hypercube algorithms for constructing pointer based (and also linear) quadrees, either from a binary image or from a boundary representation. We also presented, for pointer based quadrees, efficient parallel manipulation algorithms such as finding the neighbors of all leaves in a quadtree and computing the union/intersection of two quadrees.

Chapter 6

Constructing d -dimensional Hyperoctrees from Slices on a Hypercube

In many applications a description of an object is given as a set of cross sections, e.g. in medical diagnosis planar cross sections are obtained via tomography systems. In this chapter we consider the problem of computing the 3-dimensional representation of an object given as a set of cross sections. There are two primary versions of this problem which can be considered depending on what is known about the input. If the cross sections are known to be widely spaced or we have no spacing information at all, i.e. the information about the 3-dimensional object consists only of planar contours extracted from the cross sectional images, then we have an interpolation problem, see for example [19]. Alternatively, if the cross sections are known to lie close together, i.e. they can be considered as interpolations of the volume lying between themselves and the following cross section, then we have a merging problem. It is this latter version of the problem that we will consider.

In this chapter we will use an octree variant originally described by Yau and Srihari [112] called hyperoctrees, as we are interested in d -dimensional representations, where d may be any constant. We describe an algorithm for constructing d -dimensional hyperoctrees from sets of $(d - 1)$ -dimensional hyperoctrees (called *slices*). Given n slices of width one, the basic algorithm consists of $\log n$ phases. In

each phase pairs of adjacent slices are joined to form slices of twice the width. In all phases the slices are represented by “hybrid trees”, a data structure in between $d-1$ and d -dimensional hyperoctrees (to be defined later). Note that both the input and output hyperoctrees may either be based on pointer or linear representations. The sequential algorithm has a running time of $O(m \log n)$ [112], where m is the maximum of input and output size. The time complexity of our parallel algorithm for a SIMD hypercube with p processors is $O(\frac{m \log n}{p} \log p)$. Our parallel algorithm follows the basic merging strategy of the sequential algorithm given in [112]. The main contribution of this chapter is to solve the non-trivial problem of merging “hybrid trees” in parallel.

The remainder of this chapter is organized as follows. In Section 6.1 we define hyperoctrees and a new data structure called hybrid trees. Section 6.2 briefly recalls the sequential algorithm and is followed by a section describing a merging algorithm for hybrid trees. Section 6.4 describes in detail our parallel hypercube algorithm.

6.1 Hyperoctrees and Hybrid Trees

Hypercubes [112] are a generalization of the quadtree data structure (see Chapter 5 for definition and description). The d -dimensional hyperoctree, or for short 2^d tree, is a hierarchical data structure for the representation of a d dimensional discrete grid of side length n , where n is assumed to be a power of 2. The total grid is split in 2^d subgrids of half the side length. The subgrids are split recursively and recursion stops when a subgrid of uniform color is reached. Nodes are either grey, black or white, representing nonuniform, black or white subgrids. The level of a node v is defined recursively. It is $\log_2 n$, if v is the root, and $\text{level}(v) := \text{level}(\text{father}(v)) - 1$ otherwise.

For our algorithm we need a further generalization of 2^d trees, since we want to represent discrete grids where the side length of the side in the d th dimension varies.

To understand why this generalization is necessary consider the following naive algorithm for combining n 2^{d-1} -trees to form a 2^d -tree.

1. Convert each of the 2^{d-1} -trees that represents a slice into an equivalent 2^d -tree.
2. Perform a union operations on all the 2^d -trees formed in the previous step.

The problem with the naive algorithm is that it can result in a considerable and unnecessary “blow up” in the amount of data that must be processed. Consider for example the situation in which all of the $(d - 1)$ -dimensional slices are completely black. In the first step of the naive algorithm each of the fully black $(d - 1)$ -dimensional slices would be broken up in n^{d-1} volume elements of side length 1, as the width along dimension d is only 1. Thus we would return to a representation in image space. When in the second step of the naive algorithm the union of all n slices is computed the resulting 2^d -tree is of size $O(1)$. It is exactly this unnecessary increase or “blow up” in data size that any efficient algorithm must seek to avoid.

We need to represent not only d -dimensional cubes but also cuboids with the same side length k in dimensions 1 through $d - 1$ and side length $j \leq k$ in dimension d , where j is the number of already joined slices, and k and j are powers of 2.

Definition 6.1 *A phase i hybrid tree T^i is a colored tree with two kind of nodes. The nodes at levels l , $i \leq l \leq \log n$, form a 2^{d-1} -tree and represent cuboids of size $(2^l)^{d-1} \times 2^i$, and nodes at levels 0 through i form a forest of 2^d -trees and represent cubes of size $(2^l)^d$.*

Notice that phase 0 or $\log n$ hybrid trees can be considered as 2^{d-1} -trees or 2^d -trees, respectively. Further note that nodes at level i are at the same time leaves of the 2^{d-1} -subtree and roots of the 2^d -subtrees. Figure 6.1 shows a phase $i = 2$ hybrid tree for $d = 2$. The nodes at level $l > 2$ represent rectangles of height $2^i = 4$ and length 2^l , nodes at levels $l \leq 2$ represent squares of side length 2^l .

Figure 6.1: A hybrid tree.

6.2 Sequential Algorithm

The above definition of hybrid trees formalizes an intermediate state of the sequential algorithm presented in [112]. The basic structure of their algorithm is to consider each 2^{d-1} -tree as a phase 0 hybrid tree and perform $\log n$ pairwise merge phases. More precisely, in phase $i + 1$ of the merge algorithm, $0 \leq i < \log n$, pairs T_1^i and T_2^i of phase i hybrid trees are merged into a phase $i + 1$ hybrid tree $T_{1,2}^{i+1}$.

In the following section, we present a modified version of this merge phase, which we will use in our parallel algorithm.

6.3 Merging Hybrid Trees

We describe an algorithm to join two hybrid trees T_1^i and T_2^i in phase $i + 1$ of the merge. For this algorithm to be efficient it must avoid performing work that will later have to be undone. In particular, the decision to break up a cuboid should be postponed as long as possible, i.e., until it is certain that the cuboid will never become part of a uniformly colored cube. When a cuboid is broken up, it must be broken up into cuboids which are maximal and which also fit this criterion.

Before we can join the trees we overlay the 2^{d-1} subtrees of T_1^i and T_2^i and expand them to the same shape. While expanding the trees, we consider three different cases for all corresponding nodes v_1 and v_2 at levels $l \geq i$ in the two trees.

**PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT**

Case 1: If v_1 is black or white and v_2 is grey, then the cuboid of v_1 must be broken up to the same level of resolution as the cuboid of v_2 . Therefore we replace v_1 with a copy of the 2^{d-1} -subtree rooted at v_2 , where leaf nodes get the color of v_1 . We do not have to copy nodes below level i , as nodes in level i already represent cubes.

Figure 6.2: Case 1 for $d = 2$.

Case 2: If v_1 is black and v_2 white, they must both be broken up to cubes of side length 2^i . We thus replace each of them with the complete 2^{d-1} -tree of height $l - i$ with leaf nodes colored as v_1 and v_2 , respectively.

Figure 6.3: Case 2 for $d = 2$.

Case 3: If they are both either black or white, they represent cuboids of the same length and form together a cuboid of height 2^{i+1} , and thus no expansion is necessary.

The 2^{d-1} -subtrees of the two trees are now of the same shape and corresponding nodes on levels $l \geq i+1$ have the same color. We use the overlayed trees to construct the 2^d -tree nodes on level $i+1$, by joining the children of corresponding grey 2^{d-1} -tree nodes at level $i+1$. Keeping only one copy of the expanded 2^{d-1} -trees we obtain the hybrid tree $T_{1,2}^{i+1}$.

On a sequential machine the above merge operation can be easily implemented in time $O(\max(|T_1^i| + |T_2^i|, |T_{1,2}^{i+1}|))$ by performing simultaneous tree traversals [112]. As the overall algorithm consists of $\log n$ phases, the input set of n $(d-1)$ -dimensional hyperoctrees can be transformed into the corresponding d -dimensional hyperoctrees T in time $O(m \log n)$, with $m = \max(|T|, \sum_{j=1}^n |T_j|)$.

6.4 Parallel Algorithm for Linear Hyperoctrees

In the parallel setting, an efficient implementation of hybrid tree merge is not as straightforward. Since the sequential time complexity is $\Theta(m)$, an optimal parallel algorithm, even for the PRAM model, can achieve a time not better than $O(\frac{m}{p})$. In the following, we present an $O(\frac{m}{p} \log p)$ time hypercube algorithm.

In the following we will use a linear representation for hyperoctrees. We will later show how to convert between this linear representation and the level-order pointer-based representation. Both of these representation schemes were defined for quadrees in Chapter 5. In this chapter we use the natural generalization of these schemes to hyperoctrees.

For each node v in the 2^{d-1} -subtree of a hybrid tree, we define the position of v , $pos(v)$, to be the inorder traversal number of its corresponding node w in the complete $(d-1)$ -ary tree, $\bar{T}$, of height $\log n$. We define $cover(v)$ to be the interval $[lm(v), rm(v)]$, where $lm(v)$ and $rm(v)$ are the inorder traversal numbers (with respect to the complete tree $\bar{T}$) of the leftmost and rightmost descendant of w . For each node in a 2^d -subtree we define $anc_i(v)$ to be the position of the ancestor of v in level i . Note that the size of these numbers is at most $(d-1) \log n$ bits.

We now show how to join two hybrid trees T_1^i and T_2^i to perform phase $i + 1$ of the algorithm. We assume W.L.O.G. that the slice represented by T_1^i lies above the one represented by T_2^i . The following three procedures describe the parallel implementation of cases 1, 2 and 3 from Section 6.1, respectively. Taken together they define procedure $\text{Merge}(T_1^i, T_2^i)$. Note that, in our parallel algorithm the expansion step and the join step are not performed separately.

Figure 6.4: Merging linear trees T_1 and T_2 (thick black lines). In the first merge v_1 ends up between covered nodes. In the second merge its two copies delineate the start and end of the sequence of covered nodes. Further the trees u_1 , T_1' , u_2 and T_2' become neighbors and form thus a linear quadtree rooted in level $i + 1$.

Procedure $\text{Merge}(T_1^i, T_2^i)$, Case 1:

In this case each black or white node, v_1 , in T_1^i corresponding to a grey node, v_2 , in T_2^i is replaced by a tree of the shape of T_{v_2} , the 2^{d-1} -subtree of the tree T_2^i rooted at v_2 . As internal nodes are not represented by the linear representation, this is equivalent to replacing v_1 by the leaves of T_{v_2} . We perform this operation in two steps.

1. Detect all black and white nodes in levels $l > i$ of T_1^i , which correspond to grey nodes of T_2^i . These black and white nodes are called *covering nodes*. We merge the trees by assigning $\text{pos}(v)$ as the key to each node in a level $l > i$ and

$anc_i(v)$ as key to each node in a level $l \leq i$, and observe the following. A node v_1 in a level $l > i$ is a covering node, if and only if one of its neighbors in the resulting sequence has a different pos value and falls in the interval $cover(v_1)$. Thus, every node performs a comparison with its two neighbors. We then unmerge the sequence, i.e. extract the two original sequences.

2. Determine for each covering node v_1 the set of leaves of the subtree T_{v_2} rooted at the corresponding node v_2 . We first make a copy of each covering node and store it in an adjacent position. Every node v in T_1^i or T_2^i is then assigned a key, $key(v) = (x, y)$, where x and y are the primary and secondary keys, respectively. We define $key(v)$ as follows:

$$key(v) = \begin{cases} (lm(v), 0) & \text{if } v \text{ is the 1}^{st} \text{ copy of a covering node in } T_1^i \text{ at level } l > i, \\ (rm(v), 3) & \text{if } v \text{ is the 2}^{nd} \text{ copy of a covering node in } T_1^i \text{ at level } l > i, \\ (pos(v), j) & \text{if } v \text{ is a noncovering node in } T_j^i, j=1,2, \text{ at level } l > i, \\ (anc_{i+1}(v), j) & \text{if } v \text{ is a node in } T_j^i, j=1,2, \text{ at level } l \leq i. \end{cases}$$

Lemma 6.1 *The nodes in the linear hybrid trees form a monotonic sequence with respect to $key(v)$.*

Proof: The nodes are ordered by their pos value. As only leaf nodes are represented, all $lm(v)$ and $rm(v)$ values are unique and no pos value of another node falls in the interval $cover(v)$. Noncovering nodes at level $l > i$ do not change their key. Let $(u_1 \dots u_k)$ be a maximal subsequence of nodes of the linear tree T_j , which are at level $l \leq i$ and receive the same key. Let w be their ancestor in level $i + 1$, then $pos(u_1) < pos(w) < pos(u_k)$. $\square$

3. Thus, we can join the two trees using a stable bitonic merge.

The following lemma states some observations on the resulting sequence S . We omit the proof of the statements as they are a direct consequence of the choice of $key(v)$.

Lemma 6.2 *The sequence S has the following properties:*

- (a) *The first copy of a covering node appears immediately to the left of the first node in the sequence of covered nodes, the second copy immediately to the right of the last node in the sequence of covered nodes.*
 - (b) *The 2^d -trees which become children of a node in level $i + 1$ of the resulting hybrid tree $T_{1,2}^{i+1}$ are in the order required for $T_{1,2}^{i+1}$.*
 - (c) *Corresponding black or white 2^{d-1} -tree nodes of T_1^i and T_2^i , respectively, are neighbors in S .*
4. Use Lemma 6.2(a) to compute for all covered nodes their covering node. We do so by performing a segmented broadcast of covering nodes, where the segments are defined by the two copies of the covering nodes. As cover intervals do not overlap, any node is covered by at most one node. Then both copies of covering nodes are removed from S .
 5. Create the copies of the leaves of T_{v_2} , the 2^{d-1} -subtree rooted at v_2 , for the covering node v_1 . Each covered subsequence $\hat{S}$ of 2^d -tree nodes with the same key forms the set of leaves of a subtree rooted in level $i + 1$. We thus insert 2^{d-1} copies of the covering node before or behind $\hat{S}$, depending on whether the covering node stems from T_1^i or T_2^i .

End of Procedure Merge(T_1^i, T_2^i), Case 1.

Procedure Merge(T_1^i, T_2^i), Case 2:

In this case we expand corresponding nodes v_1 and v_2 where one of the nodes is black the other node white. According to Lemma 6.2(c) corresponding nodes can be determined by a comparison with the neighbor in the sequence S . Let the two corresponding nodes be at level l . Instead of replacing them first by two 2^{d-1} -trees which are then joined in a second step, we directly construct the 2^d -tree. To achieve this we replace them by 2^{i-l} groups of 2^{d-1} nodes at level i . The groups alternate their colors as v_1 and v_2 do.

End of Procedure Merge(T_1^i, T_2^i), Case 2.

Procedure Merge(T_1^i, T_2^i), Case 3:

The removal of one copy of corresponding black or white nodes completes the joining of two 2^{d-1} -trees and yields the tree $T_{1,2}^{i+1}$.

End of Procedure Merge(T_1^i, T_2^i), Case 3.

This concludes our discussion of procedure Merge(T_1^i, T_2^i), i.e. the phase $i + 1$ merge of two trees T_1^i and T_2^i .

Theorem 6.1 *The construction of a linear d -dimensional hyperoctree from a set of n linear $(d-1)$ -dimensional hyperoctrees on a hypercube with p processors takes time $O(\frac{m \log n}{p} \log p)$, where m is the maximum of input and output size.*

Proof: As we start with n hyperoctrees $T_1, \dots, T_n$, we need $\log n$ “merge” phases to obtain the final 2^d -tree, T . The time we need to perform one phase depends on the input size as well as on the output size. This can be seen by considering the following examples. If all slices are black, we start with n single node trees and end up with one node representing a black block. On the other hand, if slices are alternating black and white, we start with n single node trees and end up with one tree of size n^d . As the algorithm only constructs nodes which either appear in the final output tree T or which are further refined, we can estimate the data size with $m = \max(|T|, \sum_{i=1}^n |T_i|)$.

The processor load may change in each phase and the insertion and deletion of nodes is in fact a reallocation of processors. This can be performed with parallel prefix and monotonic routing operations. Since the nodes always remain in the same order it requires time $O(\frac{m}{p} \log p)$.

In a preprocessing step we compute the keys pos with local operations in time $O(\frac{m}{p} \log m)$. For each node v we start with the position of the root and follow the path from the root to v . It takes time $O(1)$ to compute the position of a child of node v , given the position of v . The height of the tree is $O(\log m) = O(d \log n) = O(\log n)$.

Given pos , the value of lm and rm can be computed in time $O(1)$ using closed formulas as the tree is a complete tree.

In each phase we have to compute keys for nodes which were either inserted or which changed from the 2^{d-1} -subtree to a 2^d -subtree. Nodes which are inserted obtain their keys from the nodes which caused their insertion. Given anc , the value of anc_{i+1} can be computed in time $O(1)$. $\square$

6.5 Parallel Algorithm for Pointer-Based Hyperoctrees

We now describe how to construct a d -dimensional *pointer-based hyperoctree* from a set of n $(d-1)$ -dimensional pointer-based hyperoctrees. The pointer-based representation is at times an attractive alternative to the linear representation, even in the parallel setting as was illustrated in Chapter 5. Our approach will be to convert the given pointer-based representation into a linear representation, apply the algorithm given in the previous section to perform the merging, and then convert back to a pointer-based representation. We will assume that the input, i.e., the n pointer-based trees, are stored in level-order as described in Chapter 5.

Algorithm Construct Pointer-Based Hyperoctree from Slices:

1. To convert a pointer-based tree T_p into its equivalent linear representation T_l , all we need to compute is the inorder number of each node v in T_p and then sort the leaves of T_p by this value. The resulting sequence of nodes is T_l . This method can be used to convert our input of n $(d-1)$ -dimensional pointer-based hyperoctrees into n $(d-1)$ -dimensional linear-based hyperoctrees.
2. Now we can apply the linear-hyperoctree construction algorithm of Section 6.4 to the n $(d-1)$ -dimensional linear-based hyperoctrees created in the previous step to produce a single d -dimensional linear-based hyperoctree, T_l .
3. To convert the linear tree T_l into its equivalent pointer-based representation T_p , we start by computing for each node v of T_l the values $pos(v)$ and $anc_{i+1}(v)$ as previously described. The tree T_p is constructed level-by-level starting with

the leaves. To form each level i of T_p , we concentrate all level i nodes of T_l , and identify segments (groups) of nodes within this set having the same parents (i.e. $anc_{i+1}(v)$). The first node in each such segment then creates its parent node, complete with pos and anc values. Finally, the newly created parent nodes are merged with the nodes of T_l at level $i + 1$ (and should be considered in the construction of the next level).

End of Algorithm Construct Pointer-Based Hyperoctree from Slices.

Theorem 6.2 *The construction of a pointer-based d -dimensional hyperoctree from a set of n pointer-based $(d - 1)$ -dimensional hyperoctrees on a hypercube with p processors takes time $O(\frac{m \log n}{p} \log p)$, where m is the maximum of input and output size.*

Proof: The correctness of the algorithm Construct Pointer-Based Hyperoctree from Slices follows directly from the definition of linear and pointer based hyperoctrees and Theorem 6.4. Consider the complexity of this algorithm. Both Steps 1 and 3 require time $O(\frac{m \log n}{p} \log p)$ as they consist of $\log n$ phases, each requiring at most $O(\frac{m}{p} \log p)$ time. Step 2 also requires time $O(\frac{m \log n}{p} \log p)$ (as stated in Theorem 6.1), therefore the theorem follows. $\square$

Given that all of the algorithms described in this chapter use only basic hypercube operations that can be pipelined, we immediately get the following corollary applying to both linear and pointer based representations.

Corollary 6.1 *The construction of a d -dimensional hyperoctree from a set of n $(d - 1)$ -dimensional hyperoctrees on a pipelined hypercube with p processors takes time $O(\frac{m \log n}{p} + \log p \log n)$, where m is the maximum of input and output size.*

6.6 Conclusion

In this chapter we have presented a parallel algorithm for the construction of a d -dimensional hyperoctree from a set of n $(d - 1)$ -dimensional hyperoctrees, representing adjacent "slices". On a p -processor SIMD hypercube the time complexity of

our algorithm is $O(\frac{m \log n}{p} \log p)$, where m is the maximum of input and output size. This parallel algorithm represents, to our knowledge, the first parallel algorithm given for the construction of octree based data structures from “slices”.

Chapter 7

A Massively Parallel Knowledge-Base Server on a Hypercube

As outlined in [39, 107, 109], massively parallel architectures are essential for computationally intensive artificial intelligence (AI) applications. Since knowledge representation is an essential part of AI [82, 110], several researchers have studied parallel architectures for implementing knowledge bases [16, 39, 44, 45, 58, 62, 77, 88, 94, 107, 109]. The parallel knowledge representation systems presented in the literature have, however, either been based on special purpose parallel architectures or support only the parallelization of one query at a time. The latter implies the (economically infeasible) dedication of a massively parallel computer to one single user (e.g. [44, 62]).

This chapter is concerned with the design and implementation of a traditional frame based knowledge representation system [18, 20, 46, 61, 76] on a fine-grained hypercube multicomputer like the 64K processor Connection Machine (CM-2) [55, 107]. We show that, using a general purpose multicomputer, it is possible to provide efficient shared access of multiple users to a large scale knowledge base server; see Figure 7.1.

We will consider a parallel implementation of a standard frame based knowledge representation system which answers elementary queries such as top-down and bottom-up inference queries and assert/retract queries [44]. Such a system could be utilized as the foundational layers of a truly parallel reasoning system, see Figure 7.2. That is, it could be used as a parallel implementation of Layers 1 and 2

Figure 7.1: Many users sharing a single large knowledge-base server.

in Figure 7.2¹. This could lead to an architecture where Layers 3 and 4 reside on multiple workstations being connected to a central SIMD hypercube multicomputer which supports layers 1 and 2.

Figure 7.2: A layered view of a knowledge-base application.

More specifically, we show in this chapter how to execute in parallel a set of inference and assert/retract queries on a knowledge base (with n frames) stored on a *fine-grained* hypercube multicomputer (with $N = n$ processors). In Section 7.2 we study single inheritance hierarchies with implicit storage [104]. We consider, in Section 7.2.2, multiple bottom-up inference queries on single inheritance hierarchies. We show that $m \leq N$ bottom-up inference queries [44] can be answered, in parallel,

¹Parallelization of the higher level layers has been extensively studied [40, 53, 88].

in time $O(T_s(n) + h \log n)$, where h is the height of the inheritance hierarchy and $T_s(n)$ denote the time required to sort n items on a n -processor hypercube multicomputer. If frames can have an unbounded number of children the time increases to $O(hT_s(n))$. Note that, in contrast to [44], our time complexity results also account for the inter-processor communication time. In Section 7.2.3, we present a heuristic algorithm for answering multiple top-down inference queries [44]. Our experimental results, obtained from extensive testing of a prototype implementation, show that a nearly optimal (100%) processor utilization is obtained for a 70% load factor (number of processors divided by number of queries). In our experiments, the utilization never dropped below 70%, regardless of the load factor and other parameters. Our system adapts flexibly and automatically to varying work loads in a close to optimal way (providing a nearly constant product of response time and number of queries). In Section 7.2.5 we study assert and retract queries, and show that they can be executed in essentially the same time complexity as top-down inference queries. Note that, our system can process all four kinds of queries simultaneously, and that only direct communication between adjacent processors is used in our algorithms.

7.1 Frame-Based Knowledge Representation

Semantic nets alternatively known as frame-based systems have been widely studied [16, 18, 20, 39, 44, 46, 61, 76, 94, 101, 104], and several general purpose knowledge representation tools have been designed based on them [18, 20, 44, 46, 61, 101]. There are many advantages to a frame based approach for knowledge representation as detailed in [16, 76].

A frame language provides the designer of knowledge based systems with an easy way to describe the domain objects to be modeled and their relationships. In a frame-based system each frame is used to describe an individual object or a class of objects. For example in Figure 7.3, the class *Ocean-Liner* is used as a prototype to describe all of the properties that are common to all Ocean liners, such as the fact that they carry paying passengers. The instance *QE2*, on the other hand, is a

Figure 7.3: An example of an inheritance hierarchy.

frame that represents an individual instance of the class *Ocean-Liner*. It specifies knowledge about a particular ocean liner, the *QE2*, such knowledge might include the number of passengers that the liner carries, or the *QE2*'s transatlantic crossing time. Both classes and instances are represented by frames. Each frame consists of a series of slots, where each slot is used to represent a single fact about a particular class or instance. Some slots may be explicit while others may inherit their values from their predecessors in the hierarchy (implicit). We will restrict our focus to parallel single inheritance frame based systems. In such systems the inheritance hierarchy can be represented by a k -ary tree.

7.2 A Parallel Frame Based Knowledge Server Supporting Single Inheritance Hierarchies

In this section, we will study the efficient hypercube implementation of a knowledge base server supporting a single inheritance hierarchy. We first describe, in Section 7.2.1, how to store a frame based system on a hypercube multicomputer, and then, how this representation can be effectively used for inference. We will be interested in answering two basic types of elementary queries: bottom-up and

top-down inference queries [44]. These queries are elementary and intended to serve as a base upon which more complex query types can be defined by higher level inference mechanisms (as depicted in Figure 7.2). In the following Section 7.2.2 and Section 7.2.3, we show how multiple bottom-up and top-down inference queries can be processed efficiently in parallel. To simplify exposition, we will describe our inference methods for both query types separately; it is however easy to see that both type of queries can be processed simultaneously.

7.2.1 Storing an Inheritance Hierarchy on a Hypercube Multicomputer

Our scheme for distributing an inheritance hierarchy over the local memories of a hypercube will be based on the *level numbering* (see Section 4.1.3) of the frames of an inheritance hierarchy. For the remainder we will assume that each frame with level number i , together with its links and data, is stored at processor P_i .

In some systems (e.g. [61,101]), inheritance is “precompiled” such that little or no searching is required to find the value of a slot, even if the value is derived from some superclass via inheritance. While this approach makes answering queries relatively easy, it has two main drawbacks. Firstly, it implies the duplication of many values and is consequently very space inefficient. Secondly, in a dynamic system it poses the problem of how to efficiently maintain this “precompilation”. In order to avoid these problems, we will use an implicit representation for our single inheritance system. That is, when storing a frame on a processor, we store only the explicitly valued slots. For implicitly valued slots there is no reference necessary since these values will be determined by the inference mechanism.

7.2.2 Answering Multiple bottom-up Inference Queries

We first consider the parallel implementation of multiple bottom-up queries. Each bottom-up query, $q(X)$, is of the form “Does frame X meet conditions a through z ” or “What are the values of slots a through z of frame X ?”. “Is the Exxon-Valdez

Frame Name:	Oil-Tanker	Frame Index:	9	Frame Type:	Class
Slots					
Name:	Cargo	Explicit Value:	Oil	Inherited Value:	Unknown
Name:	Max-Speed	Explicit Value:	Unknown	Inherited Value:	Unknown
Links					
SuperClass-Index:	5	SubClass-Indices:	None	Instances-Indices:	15

Figure 7.4: A frame record representing the class Oil-Tankers.

a vehicle with color black and current-location Alaska?" or "What is the weight of the QE2?" are examples of bottom-up queries based on Figure 7.3. Bottom-up queries are always about a particular instance or class frame but, since we are using an implicit representation, may require the examination of all superclasses of that frame.

Queries will be represented by records of the form depicted in Figure 7.5. Each "Current Value" field of a query's slot is used to store the value of the respective slot at the frame the query is currently examining.

Query Type: Bottom-up		Root Name: Exxon-Valdez	
Slot1:	Name: color	Value:	black
Slot2:	Name: current-location	Value:	Alaska
Completed: False			

Figure 7.5: A query record representing the bottom-up query "Is the Exxon-Valdez a vehicle with color black and current-location Alaska?"

Consider an inference hierarchy with n frames, stored on a hypercube multicomputer with N processors as described in Section 7.1 (W.L.O.G., $n = N$), and a set of $m \leq N$ bottom-up inference queries where each query is stored at one arbitrary processor. For the remainder, $\text{frames}(i)$ and $\text{query}(i)$ refer to the query and frame

(currently) stored at processor $PE(i)$.

Algorithm Bottom-Up:

1. Match each query $q(X)$ with the frame X it refers to.
2. As long as there is still a processor $PE(i)$ storing an unanswered query(i), repeat the following:
 - (a) Every $PE(i)$: If any slot in frame(i) is explicitly valued, and query(i) refers to the same slot but is currently unvalued, set the value of that slot of query(i) to the value given in the frame.
 - (b) Every $PE(i)$: If all necessary slots referred to in query(i) have been instantiated, check the condition, report the result, and delete the query.
 - (c) Use multi-way search to advance all query(i) with Completed = False one step along their path (in the inheritance hierarchy) towards root; i.e., match them with the parent of the frame currently examined.

End of Algorithm Bottom-Up.

Theorem 7.1 *All $m \leq N$ bottom-up inference queries can be answered, in parallel, in time $O(T_s(n) + h \log n)$ [or $O(hT_s(n))$ if frames can have an unbounded number of children], where h is the height of the inheritance hierarchy.*

Proof: Algorithm Bottom-Up describes a hypercube method for answering, in parallel, m bottom-up inference queries. It works essentially by first matching the queries to the frames they refer to and then shifting them through the tree towards the root until all slots referred to in the query have been instantiated. As indicated in Section 4.1.3, the problem of advancing all m queries one step along their path (in the inheritance hierarchy) towards root (Step 2c) can be solved on a hypercube multicomputer of size n in time $O(\log n)$ if the number of children of each frame is bounded by a small constant. For unbounded number of children, each advancement of all m queries takes time $O(T_s(n))$ [26, 34, 38]. The remainder of Step 2 consists of simple local, $O(1)$ time, operations. From [34, 38] and [26] it also follows that the initial match in Step 1 can be executed in time $O(T_s(n))$. $\square$

7.2.3 Answering Multiple Top-Down Inference Queries

Top-down inference queries, $q(X)$, are of the general form "Identify all frames in the subtree (of the inheritance hierarchy) rooted at X such that conditions a through z are true". For example, "Identify all instances of Sea-Vehicles with weight > 1000 tons and paying passengers $= 0$ ", or "Identify all classes who are subclasses of Vehicle and have less-than 10 paying passengers" are top-down inference queries based on the hierarchy in Figure 7.3.

Algorithm Top-Down:

1. Match each query $q(X)$ with the frame X it refers to.
2. As long as there is still a processor $PE(i)$ storing a query with Completed = False, repeat the following:
 - (a) Every $PE(i)$: If any slot in $frame(i)$ is explicitly valued, and $query(i)$ refers to the same slot but is currently unvalued, set the value of that slot of $query(i)$ to the value given in the frame.
 - (b) Every $PE(i)$: If all necessary slots referred to in $query(i)$ have been instantiated, check the condition, and set Completed to True.
 - (c) Use multi-way search to advance all $query(i)$ with Completed = False one step along their path (in the inheritance hierarchy) towards root; i.e., match them with the ancestor of the frame currently examined.
3. Match each query $q(X)$ with the frame X it refers to.
4. Split each query $q(X)$ into two tokens, a search token *search-token* and a control token *control-token*. Each token contains a copy of the original query. Each search token at frame X is responsible for searching the subtree rooted at the first child of frame X ; each control token at frame X is responsible for searching (or having searched) the subtrees rooted at the other children of frame X .
5. As long as there is still a processor $PE(i)$ storing an unanswered query $q(X)$, repeat the following:
 - (a) Count the number F of free processors. A free processor is a processor that is currently not supporting any search token.

- (b) Each token t calculates the number $a(t)$ of assistants it could currently use. A control token can always use as many assistants as it has remaining subtrees to search. A Search token can always use as many assistants as there are unsearched subtrees at the frame it is currently visiting. F new search tokens (assistants) are created and matched with the existing (search and control) tokens in order of the level numbering of their frames, each receiving $a(t)$ assistants until all new tokens are distributed.
- (c) For each token that has been allocated assistants in Step 4, assign a child whose subtree has not been searched yet to each assistant, match the assistants with those children, create for each a corresponding control token, and have them search the respective subtrees.
- (d) Execute "Algorithm Process-All-Search-Tokens" as shown below.
- (e) Use multi-way search to advance all search tokens.

End of Algorithm Top-Down.

Algorithm Process-All-Search-Tokens:

1. Every PE(i) storing a search token:

For every slot j in frame(i) which is explicitly valued, if query(i) refers to the same slot then

if the last node visited by the search token was the parent of frame(i)
 then the inherited value for slot j of frame(i) is the current value of slot j of the search token; the explicit value of slot j (if any) of frame(i) becomes the current value of slot j of the search token; check the query condition and report the result (if condition = true).

else the explicit value of slot j (if any) of frame(i) becomes the current value of slot j of the search token.

If the search token has not yet arrived at its control token

then select, as frame to be visited next, the next node in the preorder traversal

else If the search token's corresponding control token has additional subtrees to be searched

then start traversing one of those subtrees

else delete both, the search token and the corresponding control token, and release the processor.

End of Algorithm Process-All-Search-Tokens.

Again, queries are represented by records of the form depicted in Figure 7.5. Each "Current Value" field of a query's slot is used to store the (implicit) value of the respective slot at the frame the query is currently examining.

Taken together Algorithms Top-Down and Process-All-Search-Tokens describe a hypercube method for answering multiple top-down inference queries. Again, we assume an inheritance hierarchy of n frames, stored on a hypercube multicomputer with N processors as described in Section 7.1, and a set of $m \leq N$ top-down inference queries where each query is initially stored at one arbitrary processor. Steps 1-3 of Algorithm Top-Down are similar to our bottom-up inference algorithm. The result of these steps is that each query, $q(X)$, has for all the slots which are specified in it, explicitly stored the implicit values at frame X . What is left to do in the remaining steps is to search, for every $q(X)$, the subtree rooted at X . To this end, a search token and control token are created for every query. Each search token, for a query $q(X)$, traverses (independently and in parallel) in preorder the subtree rooted at frame X and determines the answers to be reported; the details are described in Steps 5d and 5e, together with Algorithm Process-All-Search-Tokens. Each control token remains at the root of the respective subtree to be traversed, indicates to the respective search token the end of its traversal, and creates new assistant processes in the same way as search tokens do. (Note that, the number of control tokens never exceeds the number of search tokens.) The main idea leading to a near optimal speedup (as will be shown in Section 7.2.4) is to re-use processors released by queries, which need to traverse smaller subtrees, to improve the performance of the search processes for the larger subtrees. This rescheduling mechanism is described in Steps 5a-5c. After each "round", i.e. parallel advancement of all search tokens by one edge in the preorder of their subtree, processors from finished traversal processes are given to unfinished traversal processes. Every token determines, from the outdegree of the frame it is currently visiting, how many "assistants" it could currently utilize, i.e. ask them to search those subtrees independently and in parallel. For the distribution of available processors, the following heuristic is used: higher priority is given to those search

tokens with smaller level number, i.e. tokens that have (in the expected case) the largest subtrees still to be searched.

The correctness of the algorithm follows from the following two observations: 1) the above algorithm searches for every query $q(X)$ the entire subtree rooted at frame X and 2) at every time a frame is examined, all inherited values are present. Due to the rescheduling procedure, the performance analysis for this algorithm is more complicated than in Section 7.2.2 and will be discussed separately in the next section.

7.2.4 Analysis and Experimental Results

In the previous sections we introduced two inference algorithms. In the case of the bottom-up inference algorithm it was possible to get a worst case bound on the algorithm's time complexity. In the case of the top-down inference algorithm, worst case analysis is more difficult. As in Section 7.2.2, advancing all m tokens one step along their path can be executed on a hypercube multicomputer of size n in time $O(\log n)$ (if the number of children of each frame is bounded by a small constant) or in time $O(T_s(n))$ (for unbounded number of children); see Section 4.1.3. The problem lies in determining the number of such parallel steps required by our algorithm since at the heart of the method is a heuristic rescheduling scheme that reallocates processors to queries. The challenge is to quantify how effective this reallocation technique is. In order to test our mechanism, we have implemented a sequential simulator and have performed extensive tests using randomly generated hierarchies and sets of top-down inference queries. Our simulators primary purpose was to help us predict the number of parallel steps needed to satisfy a given query set on a given inheritance hierarchy. Steps in this context refer to the operation of advancing all search tokens to the next node in their search paths (i.e. one step of multiseach). The simulator was written in Pascal on a Sparc Station 1+ and consists of about 25 pages of code. It should be noted that the term simulator is in some ways misleading in this context. The simulator is an exact implementation

of our technique for answering top-down queries under our rescheduling scheme in that

- **search tokens take exactly the same path through the hierarchy as they would in a parallel implementation**
- **assistants are allocated in exactly the same manner as they would in a parallel implementation**

What is being simulated, in some sense, is the parallel machine rather than the parallel algorithm.

We experimented with several heuristics for determining the priority with which search tokens should be provided with assistants (i.e., extra processors). Ideally, one would like to give priority to those search tokens with the "most remaining work". Unfortunately, this is hard to compute and potentially computationally expensive. We experimented with the following schemes

1. assign highest priority to search tokens that are currently visiting nodes with low level numbers.
2. assign highest priority to search tokens that are currently visiting nodes with high level numbers.
3. precompute for each node the weight of its subtrees and use this number to assign priority.
4. assign highest priority at random.

After some experimentation, the first heuristic was selected as it appeared to provide the best results for the lowest computational overhead. This heuristic appears, on average, to favor search tokens "high in the tree" which are likely to have the most remaining work.

As stated earlier, we had to use randomly generated input because no working knowledge base system of this nature was available to us. We considered the following three input parameters: 1) n = number of frames = number of processors, 2) m = number of queries, and 3) k = maximum number of children of a frame.

Figure 7.6: Experimental results for Top-Down inference.

Figure 7.6 shows the result of our experiments for hierarchies consisting of $n=16,000$ nodes. The graph on the left depicts results for hierarchies with unbounded k , while the graph on the right shows results for hierarchies with a small value of k ($k=8$). The x -axis in each diagram represents m , the number of queries, ranging from 1 to 16,000 in 1% increments. For each value of m , 1000 experiments were performed, each with a new randomly generated hierarchy and set of queries. The two curves show the average number of parallel steps as well as the average speed up. The speed up was measured by comparing the number of parallel steps with the total number of steps necessary for sequentially processing the same query set on the same hierarchy. It measures the utilization of the massive parallel architecture and, as our results show, a nearly optimal utilization is obtained for a 75% load factor (number of processors, n , divided by number of queries, m). The utilization never dropped below 70%, regardless of the load factor and other parameters.

The shape of the curves in Figure 7.6 can be explained by two opposing effects. If there are only a few queries to be processed (small load factor), these cannot immediately request enough assistants (due to the constraints of the hierarchy) to

utilize all processors. On the other hand, it is important for large subtrees to receive assistants early in the traversal process. Hence, if the number of queries is close to the number of processors, there are no (or only very few) assistants available until the smaller trees have been traversed. Therefore, it becomes likely that for the larger trees, that late arriving assistants cannot be efficiently applied to the traversal.

It is difficult to predict how well these results would reflect the performance of an operational system in which real users are performing inferences on real data. The issue is not so much that fact that a simulator is being used in our experiment, but rather that the experiments are being performed using randomly generated input data. The results reported in these experiments should be considered as an indication that the underlying algorithms are sound and exhibit good performance in the face of randomly generated hierarchies and query sets. More study of realistic hierarchies and query sets is needed.

7.2.5 Dynamic Knowledge Representation: Supporting Assert and Retract Queries

We have described how m inference queries can be answered efficiently on a static frame based inheritance hierarchy of n nodes. We will now sketch how assert and retract queries can easily be added to the system, thereby producing a truly dynamic knowledge representation scheme. Assert queries are queries that add knowledge to our representation. There are three basic types of assertions: assertion of a new slot value, assertion of a new slot (with initial value), or lastly, assertion of a new class or instance frame (complete with slots and values). Retract queries fall into three analogous types: retraction of a slot value, retraction of a slot, and retraction of a class or instance frame. Assert and retract queries are executed simultaneously to our search queries. From the semantic viewpoint, however, all assert and retract queries submitted in one round (set of inference/assertion/retraction queries to be processed in parallel) will be performed only after all inference queries in that round have been answered. In addition, assertion and retraction queries processed in parallel must be prioritized. For example, it is possible that several assert queries may

attempt to change the value stored in a particular slot and frame in the same round. Which if any of these changes should have a lasting effect? We assume query priorities based on their position in the set of submitted queries, i.e. queries submitted towards the end of the list of queries (to be processed in parallel) are considered to be executed (logically) after those submitted earlier in the list. Assert and retract queries can then be easily processed as follows: Match the queries with the respective frames using the sorting algorithm in [26]; use the concentrate/distribute operations in [79] to remove redundant assert and retract queries; apply the dynamic multi-way search algorithm referred to in Section 4.1.3 to insert/delete the required frames. All of these operations can be computed, for all assert and retract queries in parallel, in time $O(T_s(n))$.

7.3 Conclusion

In this chapter we showed how to execute in parallel a set of inference and assert/retract queries on a shared knowledge base (with n frames) stored on a *fine-grained* SIMD hypercube multicomputer (with $N = n$ processors). We studied single inheritance hierarchies and showed that $m \leq N$ bottom-up inference queries can be answered, in parallel, in time $O(T_s(n) + h \log n)$ [or $O(hT_s(n))$ if frames can have an unbounded number of children], where h is the height of the inheritance hierarchy. We presented a heuristic algorithm for answering multiple top-down inference queries; our experimental results showed that a nearly optimal (100%) processor utilization is obtained for a 75% load factor (number of processors divided by number of queries). We also outlined how assert and retract queries can be executed with essentially the same time complexity as top-down inference queries.

Part III

**Algorithms for Coarse-Grained
Multicomputers**

Chapter 8

Coarse-Grained Multicomputers: Definition and Basic Operations

In the remainder of this thesis we are interested in a class of computer architectures which we will call Coarse-Grained Multicomputers, or CGM for short. A Coarse-Grained Multicomputer is a distributed memory MIMD parallel computer in which each processor has a significant, non-constant, amount of local memory (in contrast to the more commonly studied fine-grained multicomputer).

There are three main parameters in defining a CGM:

1. The number of processors, p .
2. The total memory size of the machine¹, n .
3. The architecture of the CGM's interconnection network.

More formally, the *Coarse-Grained Multicomputer (CGM)* is a MIMD multicomputer having $O(n)$ memory evenly distributed over its p processors. The memory on each processor consists of $O(\frac{n}{p})$ $\Theta(\log n)$ bit words or registers (see Figure 8.1). Each processor can perform standard arithmetic and Boolean operations on the content of a register in unit time. Processors communicate via an interconnection network in which a processor may communicate with any *one* of its immediate neighbors in $O(1)$ time. We define a CGM interconnection network as any interconnection

¹In general, we will assume $n > p$ and $n \bmod p = 0$.

Figure 8.1: A Coarse-Grained Multicomputer (CGM).

network in which a linear array may be embedded. We make this assumption in our computational model in order to place a “very coarse” upper bound of $O(p)$ on the time complexity of our elementary communication operations (see Section 8.1). For all commonly studied (practical) interconnection networks, including mesh, hypercube, CCC, perfect shuffle, pyramid, fat tree etc., the complexity of these elementary operations is a sublinear function of p .

Commonly used interconnection networks for CGM include mesh (e.g. Intel Paragon) and hypercube (e.g. iPSC/860) networks as defined in Chapters 1 and 4, respectively. Another topology used for the recent CM-5 machine from Thinking Machines is the fat-tree introduced by Leiserson [52]. A fat tree is essentially a binary tree with the processors connected to its leaves, and increasing edge bandwidth for edges higher in the tree.

There has been a long history of CGM machines ranging from the Cosmic Cube [96] in the early 1980s to the much more recent Connection Machine CM-5 and Intel iPSC 860 based machines. There are sound technical and economic factors that suggest we may see many more parallel machines based on CGM architectures. Firstly, typically a CGM can be designed to use “off-the-shelf sequential” microprocessors for the processing elements, allowing the limited resources of the parallel machine

producer to be focused on the design of fast interconnection networks and system software. Secondly, this design approach has the added advantage that machines utilizing it may then be able to "ride the technology curve", plugging-in new faster sequential microprocessors as they become available. Recent machines, like the CM-5 from Thinking Machines Corporation and the Intel Paragon, have adopted this design philosophy.

8.1 Primitive (CGM) Operations

In the remaining chapters, we will use four *elementary operations*: sort, partial sum, broadcast and segmented broadcast [75,86]. All of the algorithms we describe in the remaining chapters could be implemented, without loss of asymptotic efficiency, strictly in terms of local computation, plus these four elementary (communication) operations.

Sort: Let $T_s(p, l)$ denote the time required to sort p sorted list of length l , one stored per processor on a p -processor CGM into a complete sorted list of lp elements (l per processor, ordered according to some processor numbering). Note that, such a coarse-grained sort can always be implemented by "simulating" a fine-grained comparison/exchange sort, like Bitonic sort [12,78]. The simulation consists of modifying the comparison/exchange procedure to compare and merge lists of sorted items instead of single items. If for any given network $T_s(p, 1)$ is the time required for the fine-grained sort then this simulation yields $T_s(p, l) \leq lT_s(p, 1)$.

Partial Sum: Let $T_{ps}(p, l)$ denote the time required to perform a partial sum operation on a list of length lp stored evenly distributed on a p -processor CGM. Items in the list are considered to be ordered first by processor number then by position in some local list. A partial sum operation computes for each element of the list the sum of all the previous elements.

Operation	Mesh	Hypercube ^a	Fat-trees (Randomized)
$T_s(p, l)$	$O(l\sqrt{p})$	$O(l \log^2 p)$	$O(l \log p)$
$T_{ps}(p, l)$	$O(l + \sqrt{p})$	$O(l + \log p)$	$O(l + \log p)$
$T_b(p, l)$	$O(l + \sqrt{p})$	$O(l + \log p)$	$O(l + \log p)$
$T_{sb}(p, l)$	$O(l + \sqrt{p})$	$O(l + \log p)$	$O(l + \log p)$
$T_c(p, l)$	$O(l + \sqrt{p})$	$O(l \log^2 p)$	$O(l \log p)$

Table 8.1: Time required for each elementary operation on CGM having different interconnection networks.

^aNote that, the time complexity given for $T_s(p, l)$ on a hypercube is based on a coarse-grained adaptation of Batcher's Bitonic Sort. Cypher and Plaxton's Share Sorting [26] is not appropriate in this context due to the large constants involved and the fact that it is not a comparison/exchange sort.

Broadcast: Let $T_b(p, l)$ denote the time required for one processor to broadcast l values to all other processors on a p -processor CGM.

Segmented Broadcasting: Let $T_{sb}(p, l)$ denote the time required to perform a segmented broadcast of $O(l)$ data on a p -processor CGM. In a segmented broadcast $p' \leq p$ processors, p_j , broadcast $O(l)$ values to destination processors $p_{j+1} \dots p_{j+k}$ ($0 < j + k < n$), where all such destination ranges are disjoint.

Let $T_c(p, l)$ denote the maximum time for any elementary operation given a particular interconnection network.

Lemma 8.1 For all CGM: $T_c(p, l) \leq O(lp)$.

Proof: By definition all CGM interconnection networks can embed a linear array and all elementary operations require at most $O(lp)$ time a linear array [75]. $\square$

Clearly, most CGM interconnection networks used in practice can implement these elementary operations in much less than $O(lp)$ time. Table 8.1 shows the complexity for each elementary operation on coarse-grained machines having different common interconnection networks.

For ease of description we will also use a small set of basic operations. These basic operations can be implemented strictly in terms of sorting, although in practice direct implementation would likely result in faster code. These basic operations include all-to-all broadcast and static load rebalancing. Table 8.2 shows the time required for each basic operation on coarse-grained machines having different interconnection networks.

All-to-All Broadcasting: Assume that each processor on a p -processor CGM has a single value that it wishes to send to all $p - 1$ other processors. Let $T_{aa}(p)$ denote that time required for this operation. Note that such an operation may be implemented in terms of a single sort of p^2 values in time $T_s(p, p)$, or as a rotation on a linear array in time $O(p)$, (although given a particular network much better implementations may be possible). Note that use of this operation requires $n \geq p^2$.

Load Balancing: Assume that each processor i on a p -processor CGM stores a list of c_i values, where these values are considered ordered first by processor number and then by their list rank. Let $C = \sum_{i=0}^{p-1} c_i$. By *load balancing* we refer to the operations of redistributing such values so that each processor stores $w = \lceil \frac{C}{p} \rceil$ or $w - 1$ values and the relative ordering of values remains unchanged. Let $T_{lb}(p, l)$ denote the time required for this operation, $l = \max_{i=0}^{p-1} c_i$. The load balancing operation may be implemented in terms of a constant number of sorting operations in time $O(T_s(p, l))$ given $l \geq p$, (although given a particular network faster implementations may be possible). The approach is as follows². First distribute a copy of all c_i values to all processors by means of an all-to-all broadcast operation and have each processor compute l , C , and w . Let each processor i now form a list of l records of which c_i are data records and contain (in order) the locally stored values and $l - c_i$ are dummy records used to ensure correct spacing in the result. The idea is to assign as keys to these records their rank in the desired load balanced result. Let the k^{th} data record on processor i be assigned $\text{key}(k, i) = ((\sum_{j < i} c_j) + k - 1) + \lfloor \frac{((\sum_{j \leq i} c_j) + k - 1)}{w} \rfloor (l - w) + 1$.

²Note that we assume for ease of exposition that C is a multiple of p if this is not the case the appropriate number of dummy values should be added.

Operation	Mesh	Hypercube	Fat-trees (Randomized)
$T_{sa}(p)$	$O(p)$	$O(p)$	$O(p)$
$T_b(p, l)$	$O(l\sqrt{p})$	$O(l \log p)$	$O(l \log p)$

Table 8.2: Time required for each basic operation on CGM having different interconnection networks.

Let the k_{ih} dummy record on processor i be assigned $\text{key}(k, i) = ((\sum_{j < i} l - c_j) + k - 1) + \lfloor \frac{((\sum_{j < i} l - c_j) + k - 1)}{l - w} \rfloor w + w + 1$. Having sorted these records each processor need only remove the $l - w$ dummy records it received in order achieve the desired result.

In the remainder of this thesis we will often sort data which is not evenly distributed over the processors. In such cases we assume that a load balancing step is first performed so that when the sort is complete the same number (± 1) of items reside on each processor.

Given the above discussion it is now possible to state the following lemma.

Lemma 8.2 *On any CGM $O(T_s(p, l) + T_{sa}(p) + T_b(p, l)) = O(T_s(p, l))$, given $l \geq p$.*

8.2 Algorithm Design for CGM: Asymptotic Optimality

Many of the geometric problems addressed in the following chapters have optimal $\Theta(n \log n)$ time sequential solutions. For these problems we describe parallel solutions requiring $T_p = O(\frac{n \log n}{p} + T_s(p, \frac{n}{p}))$ time, for $n \geq p^2$. Clearly if the first term dominates, these algorithms are “truly” optimal, as they achieve linear speedup. If the second term dominates they may still be optimal if the problem being considered reduces to sorting and $T_s(p, \frac{n}{p})$ is optimal given the CGM interconnection topology³. Note that, even when our algorithms are not “truly” optimal they require at most sort time.

³This would indicate, in general, that the routing time on the given network is the dominating factor.

For very "coarse-grained machines", i.e. $n \gg p$, we obtain algorithms which are always optimal on a CGM, as is suggested by the following lemma.

Lemma 8.3 For $n \geq 2^{T_s(p,1)}$ and comparison/exchange sort: $T_p = O(\frac{n \log n}{p} + T_s(p, \frac{n}{p})) = O(\frac{n \log n}{p})$.

Proof: If for coarse-grained sorting a simulation of fine-grained comparison/exchange sorting is used, as described in Section 8.1, then $T_s(p, \frac{n}{p}) \leq \frac{n}{p} T_s(p, 1)$. Hence $\frac{n \log n}{p} \geq \frac{n \log 2^{T_s(p,1)}}{p} \geq \Theta(\frac{n T_s(p,1)}{p}) \geq \Theta(T_s(p, \frac{n}{p}))$. $\square$

Note that, Lemma 8.3 represents a sufficient but not a necessary condition for our algorithms to be optimal. The use of a faster coarse-grained sort than the trivial simulation of fine-grained comparison exchange sorting would increase the range of values of n and p for which our algorithms are optimal.

p	$\frac{n}{p}$
8	1
16	1
32	2
64	4
128	20
256	256
512	12,678

Table 8.3: Minimum values of $\frac{n}{p}$, given p , for which our algorithms are optimal on a mesh-connected CGM.

It is interesting to examine what Lemma 8.3 implies in practice. Let us consider mesh-connected CGM for example. Table 8.3 illustrates for given values of p what the minimum load factor $\frac{n}{p}$ must be for our algorithms to be optimal.⁴

Given that the typical coarse-grained machine has several megabytes of memory per processor (See [60] as an example), notice that even in the case of $p = 512$ the number of items per processor is very reasonable.

⁴In practice slightly larger minimum load factors would be required to see the local computation dominate the sort time. This is due to the small constants that are hidden in the $O(\sqrt{p})$ time sorting.

Note that, optimal time complexities achieved by our CGM algorithms would not be achieved by straight forward simulation of existing fine-grained algorithms. Consider, for example, the 2D-Visibility problem to be described in Section 9.2. In [29] an optimal $\Omega(\sqrt{n})$ algorithm for a $p = n$ fine-grained mesh-connected multicomputer was described. Standard simulation of this algorithm on a p -processor CGM using Brent's theorem [69] requires time $O(\frac{n}{p}\sqrt{n})$, which is clearly not optimal for $p < n$. In fact such a simulation is not competitive with our algorithms for any values of p and n ($p < n$) because even if it is the sort term that dominates in the CGM algorithm's complexity, our time complexity, $O(\frac{n}{p}\sqrt{p})$, is clearly less than $O(\frac{n}{p}\sqrt{n})$.

8.3 Algorithm Design for CGM: Practical Considerations

In algorithm design for fine-grained machines it is usually assumed that the time to communicate $O(1)$ data between two processors that are neighbors in the communication topology is the same as the time required to multiply two $O(1)$ sized numbers. In practice this is rarely the case, even on supposedly fine-grained machines like the CM-2 [17]. It is much more common for communications to be one or two orders of magnitude slower than processor arithmetic. Although the difference between the time required for communication and arithmetic may be "only a constant" in asymptotic terms, this constant in practice greatly influences which algorithms are of practical value. On a CGM the difference between these times will typically be even more dramatic.

Given that communication is a very computationally expensive operation on a CGM, practical algorithms for this architecture should attempt to minimize the amount of data movement required. All of the algorithms described in the following chapters apply this design philosophy. Each requires only a small constant number of communication steps. The algorithms described in the remaining chapters consist of sequential code plus a small number of calls to the communication operations

described in Section 8.1. This approach has the added advantage that the communication operations can be carefully implemented, and optimized once and then used to provide an efficient base on which to implement a variety of algorithms.

The design philosophy described above has advantages other than those related to speed. Firstly, algorithms designed in this way tend to be relatively easy to implement since they are essentially sequential algorithms with calls to parallel communication primitives. Secondly, this approach to parallel algorithm design, which abstracts the parallel data communication operations, can simplify the design of algorithms for CGM employing non-standard communication models. Essentially the details of the interconnection network become secondary as long as it is able to efficiently implement the communication operations described in Section 8.1.

Chapter 9

Scalable Parallel Geometric Algorithms for Coarse-Grained Multicomputers

Much of the theoretical literature focuses on algorithms for fine-grained models of parallel computation. In practice, parallel machines are often coarse-grained. If, for some model, the fine-grained algorithm is optimal, i.e. $T_p = \frac{T_{seq}}{p}$, where T_{seq} is the time required by the optimal sequential algorithm, T_p is the parallel time and p is the number of processors, then simulation via Brent's Theorem [69] gives an optimal coarse-grained algorithm. For distributed memory machines (network machines) most algorithms are *not* optimal (in contrast to PRAM algorithms). Hence, simulation of fine-grained algorithms on coarse-grained machines might be improved upon. In this chapter, we show that improvement is possible, for the following geometric problems.

- 2D-Visibility from a point.
- 2D-Weighted Dominance counting.
- 3D-Maxima.
- All Nearest Neighbors in the plane.
- Convex Hull of planar points.

Furthermore, we show that in many practical cases “truly optimal” algorithms can be designed for coarse-grained machines. Even when our algorithms are not optimal they run in at most sort time. In addition to being efficient in the asymptotic sense these algorithms tend to be efficient and relatively easy to implement in practice, as has been demonstrated by a sample implementation (see Section 9.7).

All of the algorithms described in this chapter compute properties of geometric objects of size n in Euclidean space. The fundamental technique used is the partitioning of Euclidean space into regions such that each region contains an equal sized subset of the geometric data being processed.

All of the algorithms in this chapter solve geometric problems that have been extensively studied in both the sequential [21, 42, 51, 67, 68, 84, 85, 97, 98, 111] and parallel [1–3, 5, 11, 24, 27, 50, 66, 74, 81] settings. In the parallel setting Atallah and Tsay [11] studied many of these problems for a model in which a fine-grained mesh connected multicomputer is attached to a single sequential machine. Atallah and Tsay’s approach was to decompose geometric problems such that subproblems could be solved one after another, using the attached mesh of processors. In contrast, the decomposition schemes we propose permits each subproblem to be solved independently and in parallel on individual CGM processors.

For a given n and p all of these algorithms solve their respective problems in time $O(\frac{n \log n}{p} + T_s(p, \frac{n}{p}))$. The added restriction $n \geq p^2$ is required by many of our algorithms in order to ensure that each processor has $O(p)$ local registers.

9.1 Preliminaries

For all problems in this and the following Chapter involving points (or line segments), we assume that initially there are n points (line segments) represented by Cartesian coordinates, where $n \geq p$ and $n \bmod p = 0$. Initially, such points (line segments) are evenly distributed over the memory of the CGM such that each processor stores $\frac{n}{p}$ points (line segments) selected at random. Given a point v , in the plane, let $x(v)$ and $y(v)$ denote the x -coordinate and y -coordinate of v , respectively. The coordinates

of a point v in 3-space will be denoted similarly with the addition of $z(v)$ to denote the z -coordinate.

(a) Partitioning given unique x -coordinates. (b) Partitioning given non-unique x -coordinates.

Figure 9.1: Partitioning of the plane given unique and non-unique x -coordinates.

For ease of description, in the remainder, we assume that for data represented by points, such as points or line segment sets, all points have unique x and y (z) coordinates. Our algorithms can easily be modified to handle the general case. The assumption of unique coordinates is only used by our algorithms to permit the plane to be partitioned into slabs containing equal sized subsets of the input data by either vertical or horizontal lines. Let us consider what happens if, for example, the x -coordinates are not unique. Instead of using a set of vertical lines to partition the data, as in Figure 9.1a), we can use “vertical lines” containing a single jog, as is Figure 9.1b). In fact our implementation of this partitioning, coarse-grained sorting, already achieves this. In cases where points with the same x -coordinate contribute to each others results, Weighted Dominance counting for example, partial results can be communicated between processors storing points that share x -coordinates, without adversely effecting the time complexity, using an All-to-All broadcast operation.

9.2 2D-Visibility from a Point

Given a set S of n opaque line segments in the Euclidean plane that do not intersect, (except possibly at their end points) and a point P in the plane, the 2D-Visibility Problem is to determine $\text{vis}(S)$, the region of the plane visible from P . This problem can be solved in the sequential setting using the plane sweep paradigm [85] in time $O(n \log n)$. In the parallel setting optimal fine-grained mesh solutions were given in [66]. In solving the 2D-Visibility problem we will assume W.L.O.G. that $P = (0, -\infty)$ and solve the parallel projection version of this problem. See [48] for a discussion of the mapping between perspective and parallel projections. See Figure 9.2 for an example of line segment set in which the region visible from $P = (0, -\infty)$ has been marked in bold.

Figure 9.2: 2D-Visibility from a point at $P = (0, -\infty)$. (Result in bold).

In this section we show how to compute on a CGM $\text{vis}(S)$, the portion of S visible from P . Initially, each processor stores $\frac{n}{p}$ line segments drawn from S at random. Let S_i denote the subset of S originally stored in processor i . Let $m' = |\text{vis}(S)|$. The computation of $\text{vis}(S)$ will be considered complete when each processor i stores $\frac{m'}{p}$ consecutive segments from $\text{vis}(S)$.

Algorithm 2D-Visibility:

1. Arbitrarily partition the set S into p sets, S_i ($1 \leq i \leq p$) and compute $\text{vis}(S_i)$, $1 \leq i \leq p$. Let $S' = \sum_{i=1}^p \text{vis}(S_i)$ and $m = |S'|$. Note that, $m \leq 2n$,

since each end point of a segment in S can cause the creation of at most one new segment in $\text{vis}(S)$.

2. Determine p vertical lines $l_0 \dots l_p$ that partitions the plane into p vertical slabs such that each slab contains the right endpoint of $\frac{m}{p}$ line segments from S' . Let G_i denote the set of segments with right endpoint in the range $(l_{i-1} \dots l_i]$.
3. For each vertical slab determine the set C_i of line segments from S' that properly intersect the vertical line l_i . Note that for each i , the size of C_i is at most p because each line segment set $\text{vis}(S_i)$ consists of a single x -monotone chain.
4. Compute the visibility problem within each vertical slab by calculating $\text{vis}(G_i \cup C_i)$, $1 \leq i \leq p$.
5. Concatenate the results of the previous step to form $\text{vis}(S)$ and redistribute the $m' = |\text{vis}(S)|$ segments of $\text{vis}(S)$ such that each processor i stores $O(\frac{m}{p})$ consecutive segments.

End of Algorithm 2D-Visibility.

Theorem 9.1 *Given a set S of n opaque line segments in the Euclidean plane that do not intersect, (except possibly at their end points) and a point P in the plane, $\text{vis}(S)$, the region of the plane visible from P can be computed on a p -processor CGM, $n \geq p^2$, in time $O(\frac{n \log n}{p} + T_s(p, \frac{n}{p}))$.*

Proof: The correctness of Algorithm 2D-Visibility is straight-forward. In the remainder we prove the claimed time complexity by giving implementation details of the above steps.

Step 1: Compute $\text{vis}(S_i)$ on every processor i , sequentially and in parallel. This requires $O(\frac{n}{p} \log \frac{n}{p})$ time [51, 111].

Step 2: On each processor i , let G_i be a copy of $\text{vis}(S_i)$. Sort the set of segments $\cup_{j=1}^p G_j$ by x -coordinate of their right end points such that $|G_i| = O(\frac{m}{p})$, where $m = \sum_{j=1}^p |\text{vis}(S_j)|$. Store the result back into G_i .

Step 3: To compute the sets C_i , first identify l_i on every processor i . Each l_i is defined by the largest x-coordinate of a right end point of a segment in G_i . Then perform an All-to-All broadcast operation so that each processor has a copy of all l_i ($0 \leq i < p$). (Note that this step requires $n \geq p^2$ in order to ensure that there is sufficient local storage on each processor.) Every processor that stores an element of $\text{vis}(S_i)$ that is intersected by the vertical line defined by a received l_i should now create a record containing a copy of the intersected segment and an address field set to i . Finally, all records created in the preceding step can be sorted by their address fields and the result stored in C_i .

Step 4: Each processor i now stores the sets G_i and C_i and can solve the 2D-visibility problem for the vertical slab $(l_{i-1} \dots l_i]$ by solving $\text{vis}(G_i \cup C_i)$ independently and in parallel.

Step 5: All that remains is to concatenate the visibility regions in each slab to form $\text{vis}(S)$ and to load balance the number of segments stored on each processor. The concatenation of visibility regions can be done by having processors responsible for adjoining regions communicate once. Finally, the m' vertices of $\text{vis}(S)$ can be redistributed over the p -processor machine so that each processor stores $\frac{m}{p}$ vertices by performing a load balancing operation.

The time complexity of finding $\text{vis}(S)$ can now be expressed as $T_{\text{vis}}(p, \frac{n}{p}) = O(\frac{n \log n}{p} + T_s(p, \frac{n}{p}) + (T_{\text{a2a}}(p) + T_s(p, p)) + T_b(p, \frac{n}{p}))$, where the first term comes from computing $\text{vis}(S_i)$ and $\text{vis}(G_i \cup C_i)$ for all i independently, the second term from the global sort used to create the sets G_i , the third term from broadcasting all l_i and the sort used to create the sets C_i and the fourth term from the final rebalancing. Note that, by Lemma 8.2 and the above analysis, $T_{\text{vis}}(p, \frac{n}{p}) = O(\frac{n \log n}{p} + T_s(p, \frac{n}{p}))$. $\square$

The above proof shows that our algorithm for the 2D-Visibility problem is either optimal (under the conditions described in Lemma 8.3) or requires sort time for the given GGM interconnection network.

9.3 2D-Weighted Dominance Counting

Given a set $V = \{v_1, v_2, \dots, v_n\}$ of n weighted points in the Euclidean plane, the weighted dominance problem is to determine for all points $v \in V$, $wdom(v, V)$, the sum¹ of all points $u \in V$ dominated by v . A point u is *dominated* by a point v if and only if $x(u) < x(v)$ and $y(u) < y(v)$. This problem has been previously studied in both the sequential [68, 98] and fine-grained parallel [27, 50] settings.

In this section, we show how to compute on a p processor CGM for all points $v \in V$, $wdom(v, V)$. Initially, each processor stores $\frac{n}{p}$ points drawn from V at random.

Our approach will be to divide the plane first into vertical slabs $V_1 \dots V_p$ and then into horizontal slabs $H_1 \dots H_p$ such that the weighted dominance for each point $v \in V$ can be computed from the weighted dominance within these regions. Consider Figure 9.3. Let $wsum(X)$ for some set X of points denote the sum of the

Figure 9.3: Weighted dominance for point u .

weights of all points $x \in X$. Observe that given a point $u \in V_i \cap H_j$, $wdom(u, V) = wdom(u, V_i) + (wdom(u, H_j) - wdom(u, V_i \cap H_j)) + \sum_{j'=1}^{j-1} \sum_{i'=1}^{i-1} wsum(V_{i'} \cap H_{j'})$.

¹Sum may be replaced by any associative operation.

Algorithm Weighted Dominance:

1. Using p vertical lines partition the plane into p vertical slabs such that each slab contains a set V_i of $\frac{n}{p}$ points from V and for all points $v \in V_i$ and $u \in V_{i+1}$ the condition $x(v) < x(u)$ holds.
2. For each point $u \in V_i$ compute $wdom(u, V_i)$ and let $C(u)$ denote this result. In addition associate with each point $v \in V_i$ the index, i , of the vertical slab it is a member of.
3. Using p horizontal lines partition the plane into p horizontal slabs such that each slab contains a set H_j of $\frac{n}{p}$ points from V and for all points $v \in H_j$ and $u \in H_{j+1}$ the condition $y(v) < y(u)$ holds. Partition each set H_j into p disjoint sets $H_j^i = H_j \cap V_i$.
4. For each point $u \in H_j^i$, compute $(wdom(u, H_j) - wdom(u, H_j^i))$ and let $A(u)$ denote this result.
5. Compute, H_j , the p partial sums $B_i^j = \sum_{i'=1}^{j-1} wsum(H_j^{i'})$, $1 \leq j \leq p$, within each horizontal slab.
6. Finally, compute for all points $v \in H_j^i$ $wdom(v, V) = C(v) + A(v) + \sum_{j'=1}^{j-1} B_{i-1}^{j'}$.

End of Algorithm Weighted Dominance.

Theorem 9.2 *Given a set $V = \{v_1, v_2, \dots, v_n\}$ of n weighted points in the Euclidean plane, the weighted dominance for all points $v \in V$ can be computed on a p -processor CGM, $n \geq p^2$, in time $O(\frac{n \log n}{p} + T_s(p, \frac{n}{p}))$.*

Proof: The correctness of Algorithm Weighted Dominance follows directly from the observation that $wdom(u, V) = wdom(u, V_i) + (wdom(u, H_j) - wdom(u, V_i \cap H_j)) + \sum_{j'=1}^{j-1} \sum_{i'=1}^{i-1} wsum(V_i \cap H_{j'})$, given a point $u \in V_i \cap H_j$. In the remainder we prove the claimed time complexity by giving implementation details of the above steps.

- Step 1:** To form the vertical partitions sort V by x -coordinate and let V_i denote the subset of V stored on processor i .
- Step 2:** On each processor i , independently and in parallel, compute $wdom(u, V_i)$. Also store with each point its current vertical partition number, i .
- Step 3:** To form the horizontal partitions, sort V by y -coordinate and let H_j denote the subset of V stored on processor j .
- Step 4 and 5:** Each processor j now stores the set of points H_j . Sort each H_j by the vertical partition number i to obtain a partitioning into H_j^i , $1 \leq i \leq p$. Compute for each point $u \in H_j^i$, $A(u) = (wdom(u, H_j) - wdom(u, H_j^i))$, in time $O(\frac{n}{p} \log \frac{n}{p})$ using the standard sequential algorithm². Each processor j can also compute the p values B_i^j , $1 \leq i \leq p$, trivially in linear time.
- Step 6:** Sort the points and their associated C values back to their vertical partitions. Also sort all B_i^j values by key $i + 1$. Finally, each processor has all the data it needs to determine $wdom(v, V) = C(v) + A(v) + \sum_{i=1}^{p-1} B_{i+1}^i$.

The time complexity of computing for all points $v \in V$ the value $wdom(v, V)$ can now be expressed as $T_{wdom}(p, \frac{n}{p}) = O(\frac{n \log n}{p} + T_s(p, \frac{n}{p}))$, given $n \geq p^2$, where the first term results from computing the weighted sum in the vertical and horizontal partitions, and the second term from the global sorts used to create the partitions. $\square$

The above proof shows that our algorithm for the Weighted Dominance problem is either optimal (under the conditions described in Lemma 8.3), or requires sort time for the given CGM interconnection network.

²Weighted Dominance can be solved sequentially by first sorting the points by x coordinate and then running a modified mergesort algorithm to sort the points by y -coordinate. The modified merge procedure first takes a left list and a right list and merges them. It then adds to the weighted sum of each element that originated in the right list the sum of all weights of points from the left list with smaller y -coordinates[98].

9.4 3D-Maxima

Given a set P of n points in the Euclidean space, the 3D-Maxima Problem, $3Dmax(P)$, is to determine the set of points $v \in P$, such that there is no point $w \in P \setminus \{v\}$ with $x(w) > x(v)$ and $y(w) > y(v)$ and $z(w) > z(v)$. We say point v is not *dominated* by any other point of P . This problem has been previously studied in both the sequential and fine-grained parallel settings; see for example [27, 85, 97, 98]. In the sequential setting both the 2D and 3D Maxima problems can be solved in $O(n \log n)$ time [85].

In this section we show how to compute $3Dmax(P)$ on a CGM. Initially, each processor stores $\frac{n}{p}$ points drawn from P at random.

Figure 9.4: The region of the plane dominated by the maximal elements (shown as $\bullet$) can be described by the x -monotonic chain of line segments, shown as —, parallel to the x -axis.

Algorithm 3D-Maxima:

1. Using p planes parallel to the xy -plane partition 3D-space into p vertical slabs such that each slab contains a set P_i of $\frac{n}{p}$ points from P and for all points $v \in P_i$ and $u \in P_{i+1}$ the condition $z(v) < z(u)$ holds.

2. Compute $3Dmax(P_i)$, for $1 \leq i \leq p$, and call each sets of resulting maxima H_i , respectively.
3. Compute $2D-Maxima(P'_i)$, $1 \leq i \leq p$, where P'_i denotes the projection of P_i to the xy -plane. Note that the region of the plane dominated by $2D-Maxima(P'_i)$ can be represented by the x -monotonic chain, $C(P_i)$, made up of line segments parallel to the x -axes as is illustrated in Figure 9.4. Note also that such chains are non-increasing in y by definition.
4. Using p vertical planes, $(s_1, \dots, s_p)$, parallel to the xy -plane partition the set $\bigcup_{i=1}^p H_i$ into p slabs such that each slab contains a set V_j of $O(\frac{\sum_{i=1}^p |H_i|}{p})$ points and for all points $v \in V_j$ and $u \in V_{j+1}$ the condition $x(v) < x(u)$ holds.
5. For every vertical plane s_j , $1 \leq j \leq p$, compute the set I_j which consists of the intersection point of s_j with each chain $C(P_i)$, $1 \leq i \leq p$. Since each chain is x -monotonic each set I_j , $1 \leq j \leq p$, consists of at most p points.
6. Compute $3Dmax(V_j \cup I_j)$, $1 \leq j \leq p$.

End of Algorithm 3D-Maxima.

Theorem 9.3 *Given P a set of n points in the Euclidean space, the 3D-Maxima Problem, $3Dmax(P)$, can be computed on a p -processor CGM, $n \geq p^2$, in time $O(\frac{n \log n}{p} + T_s(p, \frac{n}{p}))$.*

Proof: We first establish the correctness of algorithm 3D-Maxima. Assume a point $v \in P_i$ is not a maximal point in P . We must show that v is not a maximal point in any $V_j \cup I_j$ after Step 6. Note that if v is dominated by a point in P_i , then v is not in any H_i after Step 2. If v is in V_j and dominated by a point in V_j , then this is detected in Step 6.

The interesting case is the following: point v is in V_j and dominated by a point w . Assume w is in H_i . Note that v must be dominated by a point on $C(P_i)$, because w is dominated by a point on $C(P_i)$ with respect to its x and y -coordinate (or is itself on $C(P_i)$), and v is dominated by all points in H_i with respect to the z -coordinate.

Thus v must be dominated by the intersection of s_j with $C(P_i)$, because otherwise $C(P_i)$ is not y -monotonic, which is a contradiction. This proves the correctness of the algorithm.

In the remainder we prove the claimed time and space complexity by giving implementation details of the algorithm's steps.

Step 1: To form the initial partitions sort P by z -coordinate and let P_i denote the subset of P stored on processor i .

Step 2 and 3: On each processor i , independently and in parallel, compute the value $3Dmax(P_i)$ and call the set of resulting maxima H_i , $1 \leq i \leq p$. Also determine $C(P_i)$, $1 \leq i \leq p$, by computing $2D-Maxima(P'_i)$, $1 \leq i \leq p$, where P'_i denotes the projection of P_i to the xy -plane.

Step 4: Globally sort the points in $\bigcup_{j=1}^p H_j$ by z -coordinate and let V_j denote the subset of $\bigcup_{j=1}^p H_j$ stored finally on processor j . Note that the splitting plane s_j , $1 \leq j \leq p$, is defined by the largest x -coordinate of any point in V_j .

Step 5: Perform an All-to-All broadcast of s_j , $1 \leq j \leq p$. Each processor i can now determine for each s_j its intersection with the chain $C(P_i)$ trivially in $O(\frac{n}{p})$ time. The set of all such intersection points can then be sorted, using the index of the intersecting plane as key to form the set I_j on each processor j .

Step 6: Finally, $3Dmax(V, \cup I_j)$, can be computed on processor $1 \leq j \leq p$.

The time complexity of computing of $3Dmax(P)$ can now be expressed as $T_{3Dmax}(p, \frac{n}{p}) = O(\frac{n \log n}{p} + T_s(p, \frac{n}{p}))$, given $n \geq p^2$, where the first term results from performing local $3Dmax$ and $2Dmax$ computations, and the second term from the global sorts used to create the partitions and determine the set I_j , $1 \leq j \leq p$. $\square$

The above proof shows that our algorithm for the 3D Maxima problem is either optimal (under the conditions described in Lemma 8.3), or requires sort time for the given CGM interconnection network.

9.5 All Nearest Neighbors

Given P be a set of n points in the Euclidean plane, the Nearest Neighbor problem, $NN(P)$, is to determine for each point $v \in P$ a point $w \in P \setminus \{v\}$, such that $\text{dist}(v, w) \leq \text{dist}(v, u)$ for all $u \in P$, where $\text{dist}(v, w)$ denotes the Euclidean distance between points v and w . This problem has been well studied in both the sequential [85, 97] and fine-grained parallel settings [74]. In the sequential domain the all nearest neighbors problem can be solved in terms of the Voronoi diagram of the same point set giving rise to an $O(n \log n)$ time algorithm [85].

In this section we show how to compute $NN(P)$ on a CGM. Initially, each processor stores $\frac{n}{p}$ points drawn from P at random.

We first recall the following lemma, which is a reformulation of lemmata 3.1 and 3.2 from [11].

Lemma 9.1 (See Figure 9.5). *Let H and V be a horizontal and a vertical slab, let I be the set of four intersection points defined by the limiting lines of the two slabs and let C_{VH} be the set $\{w \in V \setminus H; \min_{p \in I} (\text{dist}(w, p)) < \text{dist}(w, NN_H(w))\}$. Then the following holds:*

1. *For all $w \in V \setminus H$ which are the nearest neighbor of $v \in H \setminus V$: $w \in C_{VH}$.*
2. $|C_{VH}| \leq 8$.

Our approach is to subdivide the plane in p vertical and horizontal slabs, each containing point sets V_i and H_j , $1 \leq i, j \leq p$, of size $O(\frac{n}{p})$, respectively. According to Lemma 9.1 the nearest neighbor of a point $v \in V_i \cap H_j$ is either a point in V_i , H_j or $C_j := \bigcup_{k=1}^p C_{V_k H_j}$, where $C_{V_k H_j}$ denotes $C_{V_k H_j}$. Note that $|C_j| \leq 8p$, by Lemma 9.1(2).

Algorithm All Nearest Neighbors:

1. Using p vertical lines partition the plane into p vertical slabs such that each slab contains a set V_i of $\frac{n}{p}$ points from P and for all points $v \in V_i$ and $u \in V_{i+1}$ the condition $x(v) < x(u)$ holds.

Figure 9.5: Illustration of Lemma 9.1. The points $\circ$ belong to the set C_{VH} , the points $\times$ denote the set I .

2. Within each vertical slab V_i compute $NN(V_i)$ and let $N_1(v)$ denote this result for each point $v \in V_i$.
3. Using p horizontal lines partition the plane into p horizontal slabs such that each slab contains a set H_j of $\frac{n}{p}$ points from P and for all points $v \in H_j$ and $u \in H_{j+1}$ the condition $y(v) < y(u)$ holds.
4. Within each vertical slab V_i compute all sets C_{ij} , $1 \leq j \leq p$ from the set V_i and the y -coordinates of the horizontal lines defining the horizontal partitioning.
5. In each horizontal slab H_j form the set $C_j = \bigcup_{k=1}^p C_{kj}$. Now compute $NN(H_j \cup C_j)$ and let $N_2(v)$ denote this result, for each point $v \in H_j$. Note that by Lemma 9.1, $|H_j \cup C_j| = O(\frac{n}{p} + p)$ and for each point $v \in P$ its value in $NN(P)$ is the closer of $N_1(v)$ and $N_2(v)$.

End of Algorithm All Nearest Neighbors.

Theorem 9.4 *Given P a set of n points in the Euclidean plane, the Nearest Neighbor Problem, $NN(P)$, can be computed on a p -processor CGM, $n \geq p^2$, in time $O(\frac{n \log n}{p} + T_s(p, \frac{n}{p}))$.*

Proof: The correctness follows immediately from lemma 9.1(1), and the observation that the nearest neighbor of a point $v \in V_i \cap H_j$ must lie either in V_i , H_j or C_j .

In the remainder we prove the claimed time complexity by giving implementation details of the above steps.

Step 1: To form the vertical partitions sort P by x -coordinate and let V_i denote the subset of P stored on processor i .

Step 2: On each processor i , independently and in parallel, compute $NN(V_i)$ and denote the result $N_1(v)$, for all $v \in P$.

Step 3: To form the horizontal partitions, make a copy of P , call it H and associate with each point $v \in H$ the value $N_1(v)$. Sort H by y -coordinate and let H_j denote the subset of H now stored on processor j .

Step 4: Each processor j now stores the set of points H_j and can determine h_j , the largest y -coordinate of any point in H_j . Perform an all-to-all broadcast of all h_j values. Using the values h_j , $1 \leq j \leq p$, each processor i storing a vertical slab V_i can now compute the values $C_{i,j}$ as follows: Sort V_i by y -coordinate, scan through the sorted set, for each point v determine which (if any) sets $C_{i,j}$ it belongs to by comparing $NN_1(v)$ with the distance between v and the points I defined by the values h_j , $1 \leq j \leq p$. The time complexity of this step is $O(\frac{n}{p} \log n)$ as it is dominated by local sorting.

Step 5: Globally sort the sets $C_{i,j}$ by key j to form the sets C_j on processor j , $1 \leq j \leq p$. The computation of $NN(H_j \cup C_j)$ can now be performed on processor j , $1 \leq j \leq p$, in time $O(\frac{n}{p} \log n)$. The final result may now be computed in $O(\frac{n}{p})$ time by comparing for all $v \in P$ the values $N_1(v)$ and $N_2(v)$.

The time complexity of computing of $NN(P)$ can now be expressed as $T_{NN}(p, \frac{n}{p}) = O(\frac{n \log n}{p} + T_s(p, \frac{n}{p}))$, given $n \geq p^2$, where the first term results from performing local nearest neighbor and local sorting steps, and the second term from the global sorts used to create the partitions. $\square$

The above proof shows that our algorithm for the All Nearest Neighbors problem is either optimal (under the conditions described in Lemma 8.3), or requires sort time for the given CGM interconnection network.

9.6 Convex Hull in the plane

The problem of determining the convex hull of a set of points in the plane has been studied extensively in both the sequential [21, 42, 51, 67, 84, 97] and fine-grained parallel [2, 3, 5, 24, 74, 81] settings. In the parallel setting the focus has been on algorithms for fine-grained architectures, although some implementation oriented studies have been completed for CGMs [73].

The *convex hull* of a set S of points, denoted $\text{convex-hull}(S)$, is defined to be the smallest convex set containing S . A point $q \in S$ is an *extreme point* of S if and only if q is on the boundary of $\text{convex-hull}(S)$. The extreme points of S define a convex polygon containing $\text{convex-hull}(S)$. Let $\text{hull}(S)$ be the set of extreme points of S in clockwise ordering with respect to this convex polygon (starting with the extreme point with minimum x -coordinate). The computation of $\text{hull}(S)$ will be considered complete when the p processors store $\text{hull}(S)$ in sorted order, with $\frac{|\text{hull}(S)|}{p}$ points stored on each processor.

Algorithm Convex Hull:

1. Partition the plane into p vertical slabs such that each slab contains $\frac{n}{p}$ points. Let S_i denote the points contained in slab i .
2. Independently in each slab i , $0 \leq i < p$, compute $\text{hull}(S_i)$. (See Figure 9.6).
3. Calculate for every $\text{hull}(S_i)$ the upper and lower common tangent lines between $\text{hull}(S_i)$ and $\text{hull}(S_j)$, $0 \leq i, j < p$ and $i \neq j$.

Figure 9.6: Convex hulls within each vertical slab.

4. Combine the tangent lines computed in the previous step, and select from the convex hulls $\text{hull}(S_i)$, the extreme points of the entire point set S .
5. Redistribute the $m = |\text{hull}(S)|$ points of $\text{hull}(S)$ such that each processor i stores $\frac{m}{p} \pm 1$ extreme points with consecutive labels.

End of Algorithm Convex Hull.

Theorem 9.5 *Given a set S of n points in the plane, $\text{hull}(S)$ can be computed on a p -processor CGM, $\frac{n}{\log \frac{n}{p}} \geq p^2$, in time $O(\frac{n \log n}{p} + T_s(p, \frac{n}{p}))$.*

Proof: The correctness of Algorithm Convex Hull is straight-forward. In the remainder we prove the claimed time complexity by giving implementation details of the above steps.

Step 1: The partitioning of S into p vertical slabs can be achieved by sorting the points in S by x -coordinates as described in Section 8.1. Having completed such a sort, each processor stores S_i , a set of $\frac{n}{p}$ sorted points.

Step 2: The convex hull of each set S_i , $\text{hull}(S_i)$, can then be computed independently and in parallel on each processor using a Graham Scan in time $O(\frac{n}{p})$ [51, 85]. As a result of Step 2, a Boolean value *local-extreme* associated with each point q is set to true for all extreme points of S_i and to false for all other points. In addition, all extreme points have been labeled in a clockwise fashion as previously described.

Figure 9.7: Finding the upper and lower tangent lines between two convex hulls.

Step 3: The most critical step of the algorithm is the identification of the tangent lines between all $\text{hull}(S_i)$. Our method is an adaptation and extension of the technique described in [1] and [74].

To simplify exposition, consider for the moment the problem of computing the upper tangent line between just two convex hulls, call them A and B as shown in Figure 9.7. To identify the upper tangent line $\overline{ab}$, the points $a \in \text{hull}(A)$ and $b \in \text{hull}(B)$ must be found. Note that, for a and b to be on the upper tangent line they must be between t (t') and u (u') the eastmost and westmost extreme points, respectively. Let x and y be the points immediately before and after a in the ordering of the extreme points of A . The point a is on the upper tangent line if and only if the following two conditions are met (See Figure 9.8). *Condition 1:* all points in $\text{hull}(B)$ are below the line $\overline{xa}$. *Condition 2:* at least one point in $\text{hull}(B)$ is above the line $\overline{ay}$ [74, 84]. This characterization of a and the analogous characterizations of b, a', b' can be used in a binary search procedure to find tangent lines as follows. Initially, set the search range to $t \dots u$ and tentatively set a to be the point on $\text{hull}(A)$ “halfway” between t and u . Check if a meets the previously described characterization, if so we are done, otherwise if condition 1 was not satisfied continue the search recursively

Figure 9.8: Characterization of left endpoint of upper tangent line.

on the range to $t \dots x$. If condition 2 was not satisfied then continue recursively on the range $y \dots u$. Since this binary search procedure halves the length of the search range in each round, $O(\log n)$ rounds are required.

We now describe how the general method given above for computing an upper tangent line between two convex hulls can be adapted to find all of the upper tangent lines between p convex hulls stored on a p -processor CGM. As a pre-processing step, each processor i creates a list of the line segments of $\text{hull}(S_i)$ ordered by angle. During the algorithm each processor will maintain a table with one entry for each of the $p - 1$ other convex hulls it is attempting to find tangent lines with. Entries in the table consist of a range value, initial set to $t \dots u$, and a tentative value for the point a , initially set to the midpoint of the range. The algorithm proceeds in $\log \frac{n}{p}$ phases. In each phase, every processor creates $2(p - 1)$ messages, two messages addressed to each of the other processors. Each of these messages contains the equation for either $\overline{xa}$ or $\overline{ay}$, where the value of x , a and y are determined from the appropriate table entries. These messages are then transmitted by performing a global sort of

the $2p(p-1)$ messages, where the first key is the destination processors number and the second key is the angle of the included line. Each processor, on receiving the $2(p-1)$ messages addressed to it, forms $2(p-1)$ return messages. Each return message consists of the result of evaluating either condition 1 or condition 2 with respect to the provided line equations. The evaluation of conditions 1 and 2 can be performed by merging the sorted set of received messages with the ordered list of line segments of $\text{hull}(S_i)$ that were created in the preprocessing step. Since each processor must merge a list of length $2(p-1)$ with a list of length $\frac{n}{p}$ the total time required for this step is $O(\frac{n}{p})$. The return messages are then transmitted by another sorting step and, based on the return messages, every processor updates its table entries and resets search ranges where necessary. Clearly, the problem of determining all lower tangent lines can be solved in a completely analogous manner. Therefore all tangent lines between p convex hulls stored of a p -processor CGM can be computed in time $O(\log \frac{n}{p}(T_s(p, p) + \frac{n}{p}))$.

Step 4: What remains to be shown is how to combine the tangent lines with parts of the p convex hulls $\text{hull}(S_i)$ to form $\text{hull}(S)$. Recall that the computation of $\text{hull}(S)$ requires that all extreme points be labeled in a clockwise fashion. Having computed all upper tangent lines, each processor can now compute r , the “rightmost” upper tangent line and l , the “leftmost” upper tangent line in $O(p)$ time. It is exactly these tangent lines and the points on the convex hulls $\text{hull}(S_i)$ that fall between them that contribute to $\text{hull}(S)$. Note that $\text{hull}(S_i)$ contains no extreme points of $\text{hull}(S)$ if the right endpoint of r lies to the right of the left endpoint of l . The computation and labeling of $\text{hull}(S)$ can be achieved as follows. First, compute on each processor the number of points that fall between r and l and then perform an All-to-All broadcast to distribute these values. Each processor i can then use this information to compute the number of extreme point that fall before any extreme points stored on processor i and can then assign the correct labels to such locally

stored extreme points. This entire step can be completed in $O(T_{a2a}(p) + \frac{n}{p})$ time.

Step 5: Finally, the m vertices of $\text{hull}(S)$ can now be redistributed over the p -processor machine so that each processor stores $\frac{m}{p}$ vertices by performing a load balancing operation (See Section 8.1).

The time complexity of finding $\text{hull}(S)$ can now be expressed as $T_{\text{hull}}(p, \frac{n}{p}) = O((\frac{n \log n}{p} + T_s(p, \frac{n}{p})) + (\log \frac{n}{p} T_s(p, p) + \frac{n}{p} \log \frac{n}{p}) + (T_{a2a}(p) + \frac{n}{p}) + T_{\text{lb}}(p, \frac{n}{p}))$ where the first term comes from the original sort step, the second term from the computation of all tangent lines, the third term from the local Graham scans and the relabeling process and the last term from the redistribution of the final result. By Lemma 8.2 and the above clearly $T_{\text{hull}}(p, \frac{n}{p}) = O(\frac{n \log n}{p} + T_s(p, \frac{n}{p}) + \log \frac{n}{p} T_s(p, p))$, when $n \geq p^2$. For $\frac{n}{\log \frac{n}{p}} \geq p^2$, we have $\frac{n}{p} T_s(p, 1) \geq p \log \frac{n}{p} T_s(p, 1)$ and hence $T_s(p, \frac{n}{p}) \geq \log \frac{n}{p} T_s(p, p)$. Thus, $T_{\text{hull}}(p, \frac{n}{p}) = O(\frac{n \log n}{p} + T_s(p, \frac{n}{p}))$ when $\frac{n}{\log \frac{n}{p}} \geq p^2$. $\square$

The above proof shows that given $\frac{n}{\log \frac{n}{p}} \geq p^2$, our algorithm for solving the convex hull problem is either optimal or requires sort time for the given CGM interconnection network. Since the sorting problem can be reduced to the convex hull problem [111], our algorithm is optimal when the conditions described in Lemma 8.3 hold, or for CGMs based on interconnection networks having optimal comparison/exchange sorting algorithms (e.g. mesh).

We also obtain the following simple lemma.

Lemma 9.2 *Given a set S of n points in the plane the area, perimeter and centroid of $\text{hull}(S)$ can be determined on p -processor CGM, $\frac{n}{\log \frac{n}{p}} \geq p^2$, in time $O(\frac{n \log n}{p} + T_s(p, \frac{n}{p}))$.*

Proof: Consider the follow algorithm sketches. Each begins with the computation of $\text{hull}(S)$ as previously described.

Area: The point q_w , the westmost point on $\text{hull}(S)$, is broadcast to all processors.

Each processor, independently and in parallel, computes for each pair of consecutively labeled extreme points $q_i q_{i+1}$, the area of the triangle $q_i q_{i+1} q_w$. In

addition, the last point stored by each processor is passed onto the next processor so that the area of the triangle “between” them can be accounted for. The area of $\text{hull}(S)$ equals the summation of the area of all such triangles and can clearly be calculated using a partial sum.

Perimeter: Each processor, independently and in parallel, computes the sum of the length of all line segments on the part of $\text{hull}(S)$ it stores. The last point stored by each processor is passed onto the next processor so that the length of the line segments that join parts of $\text{hull}(S)$ stored on neighboring processors can be accounted for. The perimeter of $\text{hull}(S)$ equals the summation of all such sums, and can be easily calculated using a partial sum.

Centroid: The centroid of $\text{hull}(S)$ can be computed (using simple addition) from the area and centroid of each triangle in a triangulation of $\text{hull}(S)$ [74]. Such a triangulation can be computed as described in the area algorithm above.

□

9.7 Experimental Results

To demonstrate the practicality of our CGM algorithms, we implemented the 2D-Visibility algorithm of Section 9.2 in FORTRAN for an Intel iPSC 860 multicomputer. Our code is under 400 lines long and is largely unoptimized. We used Intel's standard FORTRAN compiler, as we did not have available to us any of the high performance i860 compilers. Our input data was randomly generated line-segment sets ranging in size between 2^8 and 2^{19} . As Figure 9.9 and Table 9.1 illustrate our asymptotic analysis was an accurate predictor of actual performance. After passing the point where communication dominates, our algorithm achieves near linear speedup.

Figure 9.9: Graph of experimental results for 2D-Visibility on an 8 node Intel iPSC 860 multicomputer.

total number of line segments	communication time [msec]	total time [msec]
256	347	350
512	350	356
1024	365	376
2048	373	394
4096	365	410
8192	356	455
16384	424	635
32768	333	797
65536	366	1402
131072	316	2600
262144	421	5505
524288	323	11891

Table 9.1: Experimental results for 2D-Visibility on an 8 node Intel iPSC 860 multicomputer.

9.8 Conclusion

In this chapter, we have given efficient CGM algorithms for the following geometric problems.

- 2D-Visibility from a point.
- 3D-Maxima.
- All Nearest Neighbors in the plane.
- 2D-Weighted Dominance counting.
- Convex Hull of planar points.

These algorithms have been based on spatial partitioning and are either optimal (i.e. require $O(\frac{n \log n}{p})$ time) or require at most sort time on any coarse-grained multi-computer, $n \geq p^2$ ($\frac{n}{\log \frac{n}{p}} \geq p^2$ for Convex Hull). Experimental results were given to support the asymptotic analysis.

Chapter 10

Data Structures on Coarse-Grained Multicomputer with Applications to Geometric Separability Problems

The basic technique used in the algorithms described in the previous chapter was to apply a spatial partitioning to geometric data. In contrast, the geometric algorithms described in this chapter have as their basic technique the partitioning of a data structure that represents the geometric data.

We first describe, in Section 10.1.1, a technique called CGM multisearch which allows the efficient CGM implementation of a large class of sequential data structures. The multisearch problem was previously solved for both fine-grained meshes (Section 2) and fine-grained hypercubes (Section 4.1.3), and in both models lead to a large number of efficient geometric algorithms. In Sections 10.2 we show how CGM multisearch can be used to implement parallel segment trees (with catalog counts) and describe a sample geometric application based on this parallel data structure. In Sections 10.3 and 10.4, we use parallel data structures to describe efficient CGM solutions to both the uni-directional and multi-directional separability problems.

10.1 CGM Multisearch: Parallel Data Structures on a CGM

10.1.1 The Multisearch Problem for balanced k -ary trees

Let $T = (V, E)$ be a balanced k -ary tree of size n , where k is a constant. The edges of T are assumed to be either all oriented towards the root or towards the leaves of the tree. Let U be a universe of possible *search queries* on T . Let us recall the definition of multisearch with small adaptations for this special case.

Define the *search path* of a query $q \in U$, denoted $path(q)$, to be a sequence of h vertices $(v_1, \dots, v_h)$ of T defined by a successor function $f : (V \cup start) \times U \rightarrow V$ as

- $f(start, q) = v_1$, and
- $f(v_i, q) = v_{i+1}$ for $i = 1, \dots, h - 1$.

The function f has the following properties.

- For every vertex $v \in V$ and query $q \in U$, $(v, f(v, q)) \in E$.
- $f(v, q)$ can be computed in $\Theta(1)$ time by a single processor that contains the information pertinent to q and v .

We say that a query $q \in U$ *visits* a node $v \in V$ at time t if and only if, at time t , the CGM is in a state where there exists a processor which contains a description of both the query q and the node v . (Note that this definition implies that many queries can simultaneously visit node v in one of two ways. Either all queries visiting a node v are stored on the same processor as v , or copies of v 's information exists on more than one processor.) The *search process* for a search query q with search path $path(q) = (v_1, \dots, v_h)$ is a process divided into h time steps, $t_1 < t_2 < \dots < t_h$, such that at time t_i , $1 \leq i \leq h$, query q visits node v_i . We will refer to the change of state between t_i and t_{i+1} , $1 \leq i < h$, as *advancing query q one step in its search path*. It is important to note that we do not assume the search path to be given in advance. In fact, we assume that the search path for each query is constructed

online during the search by successive applications of the function f . Note that a query can be advanced along an edge in T only in the direction of that edge.

Given a set $Q = \{q_1, \dots, q_m\} \subseteq U$ of m search queries where $m = O(n)$, then the *multisearch problem* for Q on T consists of executing (in parallel) all m search processes induced by the m search queries. It is important to note that the m search processes can overlap arbitrarily. That is, at any time t , any node of T may be visited by an arbitrary number of queries, which may, in fact, be at very different time steps in their respective search paths. Notice that we do not require all queries to be advanced synchronously.

10.1.2 Solving the Multisearch Problem on a CGM

In this section, we present a CGM solution to the multisearch problem for balanced k -ary trees. We define some notation, describe how data structures are to be represented on CGM, and then give an algorithm for CGM multisearch.

Let $T = (V, E)$ to be a balanced k -ary tree with vertex set V , edge set E , and size $n = |V| + |E|$. The tree T consists of $h = O(\log_k n)$ levels where the level of a node v is defined recursively as follows: $\text{level}(v) = 0$, if v is the root, and $\text{level}(v) = \text{level}(\text{parent}(v)) + 1$, otherwise. Let $\text{rank}(v)$ denote the rank of vertex v in the left to right ordering of v 's level (starting from zero). Finally, we define $Q = \{q_1, \dots, q_m\}$ to be a set of $m = O(n)$ search queries. Every vertex $v \in V$ should store the following values.

level(v): The level of vertex $v \in V$.

children(v): An array of pointers to the k children of v . (Possible set to nil for some v).

data(v): The application specific data that will guide the search queries through T .

We now discuss the manner in which T and Q will be initially represented on the CGM. Note that since each processor has only $\frac{n}{p}$ local storage, T will be partitioned

Figure 10.1: The partitioning of T into $p + 1$ subtrees of size $O(\frac{n}{p})$.

into subtrees such that each subtree has size $O(\frac{n}{p})$ and no processor stores more than $O(1)$ such subtrees (see Figure 10.1). We define for a node $v \in V$ the value $subtree(v) \in \{1 \dots p\}$ as follows:

$$subtree(v) = \begin{cases} p & \text{if } 1 \leq \text{level}(v) \leq \log_k p - 1 \\ \text{rank}(v') & \text{otherwise, where } v' \text{ is the ancestor of } v \\ & \text{such that } \text{level}(v') = \log_k p \end{cases}$$

Let $V_i = \{v \in V \mid subtree(v) = i\}$. Let T_i denote the subtree of T with vertex set V_i and edge set $E_i = \{(v, v') \in E \mid subtree(v) = subtree(v') = i\}$.

Initially, T should be stored on the CGM such that $T_i, (1 \leq i < p)$ resides on processor i and T_p is stored on processor 0. In addition all processors should store $\frac{m}{p}$ queries drawn from Q at random. Note that under this storage scheme no processor stores more than $O(\frac{n}{p})$ vertices and $O(\frac{n}{p})$ queries. The assignment of vertices and queries to processors may change during the course of the algorithms, but the $O(\frac{n}{p})$ memory bound will always be respected.

Our algorithm for solving the Multisearch problem for k -ary search trees is given below. Note that the method used is similar to the fine-grained method described in Chapter 2. We assume W.L.O.G. that T is a tree with the edges directed towards

the leaves. If the edges are directed towards the root, Steps 1 and 2 should be performed after Steps 3 - 5.

Algorithm CGM Multisearch:

1. Make p copies of T_p and distribute them one per processor. Let T_p^i and Q_i denote the copy of T_p stored on processor i and the set of queries stored on processor i , respectively.
2. Each processor i now advances the queries Q_i , $\log_k p$ steps down their respective search paths, in T_p^i .
3. For each tree T_i ($0 \leq i \leq p-1$) compute $c(T_i) = \left\lceil \frac{| \{q \in Q : v(q) \in V_i\} |}{p} \right\rceil$, where $v(q)$ denotes the next vertex in the search path of q . Note that $c(T_i)$ is the total number of copies of T_i that will be required in the next step and that $p \leq \sum_{i=0}^{p-1} c(T_i) \leq 2p$.
4. Make $c(T_i)$ copies of each subtree T_i and redistribute Q such that every query $q \in Q$ is stored on a processor i that also stores a copy of the subtree containing $v(q)$, the next node in q 's search path. Note that each processor should store the copies of at most two subtrees and $O(\frac{n}{p})$ queries.
5. Let each processor now advance the $O(\frac{n}{p})$ queries it stores, the remaining steps down their respective search paths using the copied subtrees.

End of Algorithm CGM Multisearch.

Theorem 10.1 *Given a balanced tree T of size $O(n)$ and bounded degree k , and a set Q of $m = O(n)$ search queries, $n \geq p^2$, the multisearch problem for Q on T can be solved on a p -processor CGM in time $O(\frac{n \log_k n}{p} + T_c(p, \frac{n}{p}))$.*

Proof: The correctness of Algorithm CGM Multisearch is straight-forward. In the remainder we prove the claimed time complexity by giving implementation details of each of its steps.

Step 1: Use a broadcast operation to distribute copies of T_p to each processor in time $O(T_b(p, \frac{n}{p}))$.

Step 2: Each processor i now advances each query $q_j \in Q_i$ down its search path until the next element in q_j 's search path is outside T_p . Note that each query will have to be advanced at most $\log_k p$ steps as this is the height of T_p . This step therefore requires $O(\frac{n}{p} \log_k p)$ time.

Step 3: The computation of $c(T_i)$ can be done in two steps. First, each processor computes $c_i(T_j) = |\{q \in Q_i : v(q) \in V_j\}|$ for $0 \leq j \leq p-1$ and then these values are sorted globally such that each value $c_i(T_j)$ is stored on processor j . The value $c(T_i)$ can now be computed locally for T_i on processor i . This step requires $\frac{n}{p} + T_s(p, p)$ time. Note that this step also requires $O(p)$ local memory per processor, requiring $n \geq p^2$.

Step 4: We need to make $c(T_i)$ copies of each subtree T_i and distribute them evenly over the processors. Our approach is to first compute the address (i.e. processor numbers) of where the first copy of each subtree should go and then to use this address to actually perform the distribution. We first compute the value $\text{address}(T_i)$ by performing a prefix sum over the value $c(T_i)$ and halving the result on every processor using integer division. The integer division by two ensures that for all T_i , $0 \leq i \leq p$, $\text{address}(T_i) \in \{0 \dots p\}$. We also define $\text{address}(T_p) := p$ and perform an all-to-all broadcast of the values, $\text{address}(T_i)$. We then globally sort one copy of the nodes of all subtrees, T_i , using $\text{address}(T_i)$ as the key. The final distribution of subtree copies can then be completed using a segmented broadcast in which the copy of subtree T_i on processor $\text{address}(T_i)$ is broadcasted to all processors j , where $\text{address}(T_i) < j < \text{address}(T_{i+1})$. Finally the query set Q can be sorted with the key for a query $q \in Q$ being the value $\text{address}(T_i)$, where T_i is the subtree containing the next node on q 's search path. Let Q_i again denote the query set now stored on processor i . This step consists of a constant number of basic elementary operations and therefore requires time $T_c(p, \frac{n}{p})$.

Step 5: Each processor i now advances each query $q_j \in Q_i$ to the end of its search path. Note that, in this step each query will be advanced down its search path at most $O(\log_k \frac{n}{p})$ steps as this is the height of the current tree implying a time complexity of $O(\frac{n}{p} \log_k \frac{n}{p})$

The time complexity of CGM multisearch as a whole is $O(\frac{n \log n}{p} + T_c(p, \frac{n}{p}))$, where the first term comes from the local operations required to advance queries and the second mainly from Step 4. Note that, a search query may elect to terminate its search and be deleted at any stage in its search path without increasing the time complexity of CGM multisearch. $\square$

The above proof shows that our algorithm for the multisearch problem is either optimal (under the conditions described in Lemma 8.3) or requires sort time for the given CGM interconnection network.

10.1.3 CGM Multisearch with Dynamic Node Values

In many applications it is useful for queries to be able to write data values to nodes in addition to being able to read them. We now describe how CGM multisearch can be extended to support the maintenance of dynamic node values. During an application of multisearch, many queries may write to the same data field of some node. To specify how such values are to be combined, a *combining-operation* must be specified. Since the order in which different queries visit a node is undefined, we will consider only associative combining-operations like, e.g., sum, min, max, or, and, not.

Lemma 10.1 *For associative combining-operations, dynamic node values can be maintained during an application of CGM multisearch without an increase in the time required.*

Proof: The maintenance of dynamic node values requires the following changes to the CGM multisearch algorithm:

1. In Steps 2 and 5 when individual processors are advancing queries using local copies of subtrees the combining-operation should be applied.
2. Immediately after Steps 2 and 5 all copies of subtrees stored on different processors must be combined to establish the correct node values. This can be achieved as follows:
 - Let C denote the set of vertices consisting of all copied subtrees. Sort C by first key level and second key rank. This will group all copies of the same node in a block that covers at most three processors, because there are at most $2p$ such copies.
 - Let each processor use the specified combining operation to combine the data values from all copies of a single vertex it stores.
 - Note that, the first and the last node stored may require additional information from other processors. Perform an all-to-all broadcast of the data values of the first and last node stored on each processor. Given this additional data the final data values for each node can be determined and all other values from C discarded.
 - Lastly, rebuild the subtrees T_i such that $T_i, 0 \leq i \leq p - 1$, is stored on processor i and subtree T_p is stored on processor 0. This can be achieved by a single sort.

These changes to the CGM multisearch algorithm can clearly be implemented using only a constant number of elementary operations and therefore do not increase the algorithm's time complexity. $\square$

10.2 Segment Trees using CGM Multisearch

In this section we describe a CGM algorithm for a line-segment stabbing problem. Our solution is based on using CGM multisearch to implement a parallel version of Bentley's segment tree data structure. This technique is similar to the fine-grained

hypercube method described in [87]. Parallel segment trees will then be used in the following sections to solve other geometric problems. In order to use multisearch to parallelize a sequential data structure (in this case a segment tree) we will have to describe:

1. how to construct (in parallel) the data structure on a CGM, and
2. how the data structure traversals are to be realized by online search processes.

10.2.1 Parallel Segment Trees

The *segment tree*, originally introduced by Bentley [13], is a data structure designed for storing intervals. See Preparata and Shamos's book [85] for a full description and catalog of applications.

A segment tree for a set, $\mathcal{S}$, of n intervals (n a power of 2) is a complete binary tree with n leaves in which each node v stores the following:

- an interval x -range equal to the union over all intervals stored in the leaves of the subtree rooted at v , and
- $\text{count}(v) = |\{I \in \mathcal{S} \mid x\text{-range}(v) \text{ is in } I, \text{ but } x\text{-range}(\text{parent}(v)) \text{ is not in } I\}|$

In the following algorithm for constructing a segment tree T on a p -processor CGM, we assume that initially the set $\mathcal{S}$ of n intervals is distributed evenly over the processors.

Algorithm Construct Segment Tree:

1. Create a list, E , of all unique endpoints from $\mathcal{S}$ and sort and load balance it. We assume W.L.O.G. that $|E| = 2^x$ for some positive integer x . If this is not the case E should be padded to the next larger power of 2.
2. Construct a complete binary tree T having $|E|$ leaves and store it to meet the requirements of CGM multisearch as described in Section 10.1.2.

3. In order to convert T into a segment tree we must first initialize the x -range of each node. Form a set of $|E|$ queries each containing the x -value of an endpoint. Using multisearch, each query should traverse from the leaf corresponding to its x -value towards the root of the tree, initializing the x -range of nodes along its path. A query that starts on a leaf that is a left [right] child should continue its upwards traversal until it visits the root or a node that is a right [left] child.
4. To initialize the value $\text{count}(v)$ for each node of T , a set of $2n$ queries is formed, one for each endpoint of an original interval. These queries (each consisting of an interval) traverse from the root of T to the leaf of T having the x -range that contains the endpoint. While traversing this path, each query increments $\text{count}(v)$ where appropriate. Note that, a query q at node v may also have to increment $\text{count}(v')$ where v' is the child of v that is not on q 's search path. This can be achieved locally (i.e. on a single processor without interprocessor communications) on all levels of T , except level $\log_k p$. At this level each query may split into two copies, where the second copy is used just to update the second child and is then terminated.

End of Algorithm Construct Segment Tree.

Theorem 10.2 *A segment tree for a set S of n intervals can be constructed on a p -processor CGM, $n \geq p^2$, in time $O(\frac{n \log n}{p} + T_c(p, \frac{n}{p}))$.*

Proof: The correctness of Algorithm Construct Segment Tree follows immediately from the definition of segment trees. The segment tree construction algorithm consists of a sort step, plus two calls to CGM multisearch and therefore requires time $O(\frac{n \log n}{p} + T_c(p, \frac{n}{p}))$, $n \geq p^2$, on a p -processor CGM. $\square$

10.2.2 The Multiple Stabbing Problem

Having described how to construct a segment tree, we now give an example application. This algorithm is included for illustrative purposes only as the problem it solves can be also solved by simpler sort-based methods in the same time complexity.

Another more significant application follows in Section 10.3. Consider the following problem, which we will, for the sake of brevity, refer to as the *multiple stabbing problem*: Given a line segment set S of size $O(n)$, and a set M of vertical query lines ($m = |M| = O(n)$), determine for each vertical query line, $l \in M$, the number of elements of S it intersects.

Algorithm Multiple Line Segment Stabbing:

1. Project the line segments of S onto the x -axis and build a segment tree of the resulting intervals, as described in Section 10.2.1.
2. Using CGM multisearch, let each vertical line query traverse the segment tree from the root to the leaf containing the x -coordinate of the query. As each query traverses the tree it should sum the values $\text{count}(v)$ for every node v it visits. For each query this sum is the result.

End of Algorithm Multiple Line Segment Stabbing.

Theorem 10.3 *The multiple stabbing problem can be solved for a line segment set S of size $O(n)$ and a query set M of size $m = O(n)$ in time $O(\frac{n \log n}{p} + T_c(p, \frac{n}{p}))$, $n \geq p^2$, on a p -processor CGM.*

Proof: The correctness of Multiple Line Segment Stabbing follows directly from the definition of segment trees and the time complexity follows from the previously described algorithms for constructing and traversing a segment tree. □

10.3 The Uni-Directional Separability Problem

In this section we describe how CGM multisearch can be used to determine the set of all directions in which a set, $\mathcal{P}$, of planar polygons is uni-directionally separable. We call this the *uni-directional separability problem* or *UDS-Problem* for short. Our approach is to show how techniques first described by Dehne and Sack [30] for solving this problem in the sequential domain can be combined with CGM multisearch to provide an efficient parallel solution.

10.3.1 Definitions and Notation

We follow the notation given in [30]. Let $\wp = \{P_1, \dots, P_\eta\}$ be a set of η Γ -vertex polygons (in the Euclidean plane), with pairwise non-intersecting interiors. A translation of a polygon $P_i \in \wp$ is specified by a translation direction and distance. A *separating motion* of P_i is a translation of P_i in some direction d by an arbitrarily large distance. P_i is said to collide with polygon P_j , $i \neq j$, if, at any distance during the separating motion, the interiors of P_i and P_j intersect; otherwise, P_i and P_j are said to be separable in the given direction. A polygon P_i is *separable from the set* $\wp$, if there exists some direction d such that in this direction P_i is separable from each of the remaining polygons P_j , $i \neq j$, ($1 \leq j \leq \eta$). A set of polygons $\wp = \{P_1, \dots, P_\eta\}$ is *uni-directionally separable* if there exists a *uni-directional translation ordering* $(P_{\pi(1)}, P_{\pi(2)}, \dots, P_{\pi(\eta)})$ of $\wp$ such that for $i = 1, \dots, \eta - 1$ polygon $P_{\pi(i)}$ is separable from the set $\{P_{\pi(i+1)}, \dots, P_{\pi(\eta)}\}$ of remaining polygons by a translation in direction d . See Figure 10.2.

Figure 10.2: Polygon 1 is uni-directionally separable in direction d . The set of polygons is not uni-directionally separable in direction d as can be seen by examining polygons 4 and 5. (from [30])

10.3.2 Review of Sequential Literature

We will use the following theorems originally derived by researchers studying sequential algorithms for separability problems.

Theorem 10.4 ([105]) *A set of polygons $\mathcal{P} = \{P_1, \dots, P_n\}$ admits a translation ordering in a direction d , if and only if, every pair of polygons, viewed in isolation, is separable with a single translation in direction d .*

The problem of determining all directions of separability for two polygons P and Q has been addressed by Sack and Toussaint [89]. They defined for two polygons P and Q , the *relative moveability wedge*, $W_P(Q)$, which describes all directions in which P is separable relative to Q . They also define the *moveability wedge*, $W(P, Q)$, which is the union of $W_P(Q)$ and $W_Q(P)$, and describes all directions in which P and Q are separable relative to each other, (see Figure 10.3 for an example). The

Figure 10.3: The moveability wedge, $W(P, Q)$ for polygons P and Q . Note that, all directions of separability fall in sectors A, C, and E.

moveability wedge for two simple Γ -vertex polygons can be computed in $O(\Gamma)$ time.

This result is based on Sack and Toussaint's observation (see [30]) that given a triangulation of P and Q , $W(P, Q)$ can be computed in time $O(\Gamma)$, combined with Chazelle's $O(\Gamma)$ time triangulation algorithm [22]. Alternatively, the moveability wedge, $W(P, Q)$, can be computed in $O(\Gamma \log \Gamma)$ time by a much simpler method as described in [89]. Note that, the moveability wedge for a pair of polygons can be described by three or less sectors in the range $[0, \dots, 360)$, see Figure 10.3.

Dehne and Sack [30] introduce the concept of a moveability wheel, $W(\rho)$, for a set, ρ , of polygons. The moveability wheel $W(\rho)$ for ρ , is the set of all directions d for which a uni-directional translation ordering exists.

Theorem 10.5 ([30]) *The moveability wheel $W(\rho)$ has the following properties:*

- a) *Let ω denote the set of all pairwise moveability wedges, $\{W(P_i, P_j) | 1 \leq i < j \leq \eta\}$, then $W(\rho)$ is the intersection of all pairwise moveability wedges $W(P_i, P_j) \in \omega$.*
- b) *$W(\rho)$ consists of at most $\eta(\eta - 1)$ disjoint sectors.*

In order to represent the moveability wheel $W(\rho)$ for ρ , Dehne and Sack propose the use of a modified segment tree [85]. Let $\mathcal{S} = \{w_1, \dots, w_{\eta^2}\}$ be a set of η^2 moveability wedges and let $w_i^c := [0 \dots 360) - w_i$. Each w_i^c consists of at most three intervals, therefore the set $\mathcal{S}^c$ of all such intervals consists of at most $3\eta^2$ intervals. A *modified segment tree* for a set of moveability wedges $\mathcal{S}$ is a segment tree on $\mathcal{S}^c$ in which each node v stores, in addition to $x\text{-range}(v)$ and $\text{count}(v)$, a single bit called $\text{mark}(v)$ defined as follows:

$$\text{mark}(v) = \begin{cases} \text{true} & \text{if } v \text{ is a leaf and } \text{count}(v) = 0 \\ & \text{or } (v \text{ is an internal node, } \text{count}(v) = 0, \\ & \text{and at least one child } v' \text{ is marked)} \\ \text{false} & \text{otherwise} \end{cases}$$

Theorem 10.6 ([30]) *Let T be the modified segment tree for a set, S , of moveability wedges, and let d be a direction in $[0, 360)$. Furthermore let $\text{INT}(S)$ denote the intersection of all wedges in S .*

- a) $INT(S) \neq \emptyset$ if and only if the root of T is marked.
- b) d is in $INT(S)$ if and only if all nodes along the path from the root of T to the leaf containing d are marked.

Based on Theorem 10.6, Dehne and Sack give an algorithm for solving the UDS-Problem. Given a set ρ of η polygons of size Γ , they first build the modified segment tree for the set S of all pairwise moveability wedges. They then perform $O(\eta^2)$ searches in the modified segment tree to determine all directions of uni-directional separability. The construction of the modified segment tree and the $O(\eta^2)$ searches are performed in time $O(\eta^2(\Gamma + \log \eta))$ on a sequential machine.

10.3.3 Solving the Uni-Directional Separability Problem on a CGM

In this section we describe how CGM multisearch can be used to solve the UDS-problem in parallel on a CGM. Our solution is based on a parallel version of the modified segment tree data structure. It assumes that initially the set ρ of η polygons is distributed evenly over the processors.

Algorithm Construct Modified Segment Tree:

1. Create the set S of moveability wedges for all pairs of polygons in ρ .
2. Build a segment tree for the complement S^c of S .
3. Convert the segment tree into a modified segment tree by initializing the bit, $\text{mark}(v)$, for each node of the segment tree as defined in Section 10.3.2.

End of Algorithm Construct Modified Segment Tree.

Theorem 10.7 *Given a set, ρ , of η Γ -vertex polygons, a modified segment tree representing the moveability wheel $W(\eta)$ can be constructed on a p -processor CGM in time $O(\frac{\eta^2(\Gamma + \log \eta)}{p} + T_e(p, \frac{\eta^2}{p}))$, where the total memory size $n = O(\eta^2 + \eta\Gamma + p^2)$ and $\frac{n}{p} \geq \Gamma$, $\eta \geq p^2$.*

Proof: The correctness of Algorithm Construct Modified Segment Tree follows directly from [30]. In the remainder we prove the claimed time complexity by giving implementation details of each of its steps.

Step 1: In order to create the set S , every polygon $P_i \in \wp$ must be matched with every polygon $P_j \in \wp$ ($i \neq j$), where “matched” means that the two polygons reside on the same processor at the same time. Each processor stores $O(\frac{n}{p})$ polygons and requires $O((\frac{n}{p})^2 \Gamma)$ time to sequentially compute all of their pairwise moveability wedges. The $O(\frac{n}{p})$ polygons each processor stores can be “rotated” using a sort in time $T_s(p, \frac{n\Gamma}{p})$. Clearly, after p such rotations, all polygons will have been matched with all other polygons. This algorithm for generating S requires $O(p(\frac{n}{p})^2 \Gamma + pT_s(p, \frac{n\Gamma}{p}))$ time on a p -processor CGM.

Step 2: First, (trivially) compute the set S^c from S . Then construct a segment tree on S^c , as previously described in Section 10.2.1.

Step 3: The segment tree created in the previous step can be converted into a modified segment tree by initializing the $\text{mark}(v)$ value for each node v . Initially, set the value of $\text{mark}(v)$ to true for all leaves v with $\text{count}(v) = 0$, and false for all other nodes of T . Start a query at each marked leaf of T . Using multisearch each query traverses from its starting position towards the root of the tree initializing the value $\text{mark}(v)$ of nodes v along its path. A query may stop its upwards traversal as soon as it encounters a node v with $\text{count}(v) \neq 0$.

Step 1 requires $O(p(\frac{n}{p})^2 \Gamma + pT_s(p, \frac{n\Gamma}{p})) = O(\frac{n^2 \Gamma}{p} + pT_s(p, \frac{n\Gamma}{p}))$ time. Therefore, given positive constants c_1, c_2 and $n_0, n \geq n_0$, when $\frac{n^2 \Gamma}{p} c_1 \geq pT_s(p, \frac{n\Gamma}{p}) c_2$ we have an optimal parallel algorithm for this step. Note that this is the case when $\eta \geq p^2$ because $\frac{n^2 \Gamma}{p} c_1 \geq pT_s(p, \frac{n\Gamma}{p}) c_2 \Leftrightarrow \frac{n^2 \Gamma}{p} c_1 \geq \frac{n\Gamma}{p} pT_s(p, 1) c_2 \Leftrightarrow \eta c_1 \geq pT_s(p, 1) c_2$ which is clearly the case because $T_s(p, 1) \leq O(p)$, for all CGM interconnection networks.

Steps 2 and 3 of the above algorithm involve only a constant number of calls to CGM multisearch in which $O(\eta^2)$ query processes are executed on an $O(\eta^2)$ sized tree. Therefore, these steps require $O(\frac{\eta^2 \log \eta}{p} + T_c(p, \frac{\eta^2}{p}))$ time on a p -processor CGM.

□

Note that all queries that reach the root in Step 3 of the above algorithm originate from leaves having x -coordinates in $W(\varphi)$. In fact if all such queries are sorted by x -coordinate of their originating leaf, every block of three queries will define an interval in $[0 \dots 360]$ in which φ is uni-directionally separable.

Theorem 10.8 *For any set φ of η Γ -vertex polygons, the set $W(\varphi)$ of all directions d , for which a uni-directional translation ordering of φ exists, can be computed in time $O(\frac{\eta^2(\Gamma + \log \eta)}{p} + T_e(p, \frac{\eta^2}{p}))$, where the total memory size $n = O(\eta^2 + \eta\Gamma + p^2)$ and $\frac{n}{p} \geq \Gamma$, $\eta \geq p^2$.*

10.4 The Multi-Directional Separability Problem

In this section, we describe a CGM solution to the multi-directional separability problem (MDS-Problem). Our approach is to show how techniques first described in the sequential domain [30] can be adapted to provide an efficient parallel solution. Unlike our solution to the uni-directional separability problem we will not utilize a single parallel version of the sequential data structure. Instead, we will use many copies of a simpler data structure.

10.4.1 Definitions and Notation

Throughout this section, we will continue to use the definitions established in Section 10.3. In addition we require the following new definitions.

A set $\varphi = \{P_1, \dots, P_\eta\}$ of polygons is *multi-directionally separable* if there exists a *multi-directional translation ordering* $(P_{\pi(1)}, P_{\pi(2)}, \dots, P_{\pi(\eta)})$ of φ such that for $i = 1, \dots, \eta - 1$ polygon $P_{\pi(i)}$ is separable from the set of remaining polygons $P_{\pi(i+1)} = \{P_{\pi(i+1)}, \dots, P_{\pi(\eta)}\}$ by a translation in direction d_i . The *multi-directional separability problem* is to determine for a set of polygons, φ , if a multi-directional translation ordering exists and, if so, to provide an example ordering. See Figure 10.4.

Given a set $\varphi = \{P_1, \dots, P_\eta\}$ of polygons, a *wedge-tree*, T_P , [30] is a modified segment tree (See Section 10.3.2) with respect to $\{W_{P_i}(P_j) | P_j \in \varphi \setminus \{P_i\}\}$.

Figure 10.4: Polygons 1–5 are separable in the indicated directions. The set of polygons is multi-directionally separable as no two polygons are interlocked. (from [30])

A *MDS-graph*, G_φ , for set of polygons $\varphi = \{P_1, \dots, P_n\}$ is the forest of wedge-trees $T_{P_1}, \dots, T_{P_n}$.

10.4.2 Solving the Multi-Directional Separability Problem on a CGM

In this section we describe a CGM solution to the MDS-problem. Our solution is based on the following observations [30]:

Given a MDS-graph for set of polygons $\varphi = \{P_1, \dots, P_n\}$

1. Polygon P_i is separable from φ if and only if the root of its wedge tree, T_{P_i} , is marked.
2. A necessary condition for φ to be multi-directional separable is that the MDS-graph for φ contains at least one wedge tree, T_{P_j} , whose root is marked.

3. If more than one wedge tree, T_{P_i} , has a marked root an arbitrarily chosen corresponding polygon can be selected as the next polygon in the translation ordering. The removal of a polygon never reduces the movability of the remaining polygons.

Algorithm Multi-Directional Separability which generates a multi-directional translation ordering of $\wp$, if one exists, is given below. It assumes that initially the set $\wp$ of η polygons is distributed evenly over the processors.

Algorithm Multi-Directional Separability:

1. For each polygon $P_i \in \wp$, create the set of all relative moveability wedges $\mathcal{S}_i = \{W_{P_i}(P_j) | P_j \in \wp \setminus \{P_i\}\}$.
2. Using the sets $\mathcal{S}_i$, construct the corresponding MDS-graph, $G_\wp$.
3. To compute the next polygon in a multi-directional translation ordering of $\wp$ perform the following steps. If at any repetition there exists no wedge-tree whose root is marked, then no multi-directional translation ordering exists for $\wp$. Otherwise, perform the following steps:
 - (a) Pick a wedge tree T_{P_i} whose root is marked.
 - (b) Record P_i as the next polygon in the multi-directional translation ordering.
 - (c) Delete from $G_\wp$ the wedge tree T_{P_i} .
 - (d) Delete from all remaining wedge trees in $G_\wp$ all relative moveability wedges $W_{P_i}(P_j) (i \neq j)$.

End of Algorithm Multi-Directional Separability.

Theorem 10.9 *Given a set, $\wp$, of η Γ -vertex polygons, the MDS-Problem for $\wp$ can be solved on a p -processor CGM in time $O(\frac{\eta^2(\Gamma + \log \eta)}{p})$, where the total memory size $n = O(\eta^2 + \eta\Gamma + p^2)$ and $\frac{n}{p} \geq \Gamma$, $\eta \geq p^2$.*

Proof: The correctness of Algorithm Multi-Directional Separability follows directly from [30]. In the remainder we prove the claimed time complexity by giving implementation details of each of its steps.

Step 1: In order to create for each polygon $P_i \in \wp$ the set of all relative moveability wedges $\mathcal{S}_i = \{W_{P_i}(P_j) | P_j \in \wp \setminus \{P_i\}\}$ every polygon $P_i \in \wp$ must be “matched” with every polygon $P_j \in \wp$ ($i \neq j$), where matched means that the two polygons reside on the same processor at the same time. Each processor stores $O(\frac{n}{p})$ polygons and requires $O((\frac{n}{p})^2 \Gamma)$ time to sequentially compute all of their pairwise relative moveability wedges. The $O(\frac{n}{p})$ polygons each processor stores can be “rotated” using a sort in time $T_s(p, \frac{n\Gamma}{p})$. Clearly, after p such rotations all polygons will have been matched with all other polygons. This algorithm for generating the sets $\mathcal{S}_i$ requires $O(p(\frac{n}{p})^2 \Gamma + pT_s(p, \frac{n\Gamma}{p}))$ time on a p -processor CGM.

Step 2: Each processor locally will construct the wedge trees T_{P_i} from $\mathcal{S}_i$, for each polygon P_i it was originally allocated. This can be achieved in time $O(\frac{n^2 \log n}{p})$ on a p -processor CGM.

Step 3: Each repetition of Steps 3a-3d can be implemented as follows. We will assume that each processor maintains a list of locally stored wedge trees whose roots are marked. A polygon P_j corresponding to a wedge tree with marked root is globally selected (using a partial sum operation), and the value P_j is broadcast. All processors update all wedge trees $\{T_{P_i} | i \neq j\}$ deleting all segments from relative moveability wedges $W_{P_i}(P_j) (i \neq j)$. The processor storing the wedge tree T_{P_j} marks it as deleted. Given this implementation of Steps 3a-3d it is easy to see that Step 3 requires time $O(\frac{n^2 \log n}{p} + \eta T_b(p, 1))$.

Step 1 requires $O(p(\frac{n}{p})^2 \Gamma + pT_s(p, \frac{n\Gamma}{p})) = O(\frac{n^2 \Gamma}{p})$ time by the same reasoning as in Theorem 10.7. Steps 2 and 3 require time $O(\frac{n^2 \log n}{p} + \eta T_b(p, 1) + \eta T_{ps}(p, 1))$. Therefore, given positive constants c_1, c_2, c_3 , and $n_0, n \geq n_0$, when $\frac{n^2 \log n}{p} c_1 \geq \eta T_b(p, 1) c_2 + \eta T_{ps}(p, 1) c_3$ we have an optimal algorithm for these steps. Note that,

by Lemma 8.1, $\frac{\eta^2 \log \eta}{p} c_1 \geq \eta T_b(p, 1) c_2 + \eta T_{p^2}(p, 1) c_3 \Leftrightarrow \frac{\eta^2 \log \eta}{p} c_1 \geq \eta p (c_2 + c_3) \Leftrightarrow \frac{\eta \log \eta}{p} c_1 \geq p c_2$, which is clearly the case given $\eta \geq p^2$. Therefore this algorithm as a whole requires time $O(\frac{\eta^2 \Gamma}{p} + \frac{\eta \log \eta}{p}) = O(\frac{\eta^2 (\Gamma + \log \eta)}{p})$, which represents an optimal parallelization of the sequential algorithm presented in [30], under the assumption $\frac{\eta}{p} \geq \Gamma$ and $\eta \geq p^2$.

□

10.5 Conclusion

In this chapter we have described a technique called CGM multisearch which allows the efficient CGM implementation of a large class of sequential data structures. To illustrate the applicability of this technique we have shown how it can be used to implement a parallel version of the well known segment tree data structure. We have also studied both the uni-directional and multi-directional separability problems and shown how parallel CGM based data structures can be combined with sequential techniques to produce efficient parallel geometric algorithms for solving these problems.

Part IV

Conclusion

Chapter 11

Conclusion

This thesis has been concerned with two different, yet often intertwined themes. We have studied multisearch techniques for efficiently constructing and manipulating parallel versions of standard sequential data structures on multicomputers. We have also shown how such parallel data structures can be used to efficiently solve problems from such diverse application domains as artificial intelligence and solid modeling. Our second theme has been concerned with how multicomputers can be used to solve geometric problems ranging from clustering to separability problems. In some cases we have used parallel data structuring techniques to solve these geometric problems, while in other cases different parallel techniques proved more effective.

11.1 Multisearch

In this thesis, we have presented two new multisearch algorithms, one for fine-grained mesh connected multicomputers and the other for coarse-grained multicomputers. Table 11.1 summarizes the current state of the art for multisearch algorithms.

To date, the multisearch technique has been applied to a wide range of parallel data structuring problems. Below we give a list of many of these applications in order to demonstrate the wide applicability of the multisearch technique.

Model	Complexity	Graph class	Dynamic
Fine-grained Hypercube	$O(r \log n + T_{\text{sort}})$ for $n = p$	Monotonic	Yes
Fine-grained Mesh	$O(\sqrt{n} + r \frac{\sqrt{n}}{\log_2 n})$ for $n = p$	Partitionable hierarchical-DAGs	No
Coarse-grained Multicomputer	$O(\frac{nr}{p} + T_{\text{sort}})$ for $n \geq p^2$	Partitionable	No

Table 11.1: Summary of multisearch algorithms where r is maximum path length of any query, p is number of processors, and n is size of data structure and number of queries.

Applications of multisearch

- Search trees (See Chapter 2)
 - next element search in a slab
 - best common ancestor
- Quadtrees (See Chapter 5)
 - construction from binary matrices
 - construction from boundary codes
 - neighbor finding
 - computation of the union/intersection of Quadtrees
- Frame-based inheritance hierarchies (See Chapter 7)
 - top-down inference
 - bottom-up inference
 - assert/retract queries
- Separability (modified segment trees with catalog counts) (See Chapter 10)
 - uni-directional separability
 - multi-directional separability
- Segment trees (See [38, 87])
 - next element search

- trapezoidal decomposition
 - triangulation of a simple polygon
- **Branch-and-Bound** (See [33, 87])
- **subdivision hierarchies** (See [10])
 - lines-polyhedron intersection queries
 - intersecting convex polyhedra
 - 3-d convex hulls
- **Bipartite Graphs** (See [7])
 - determination of maximum matching
 - determination of maximum independent vertex set

Multisearch can be a powerful and efficient tool for solving parallel data structuring problems. The “trick”, as with many tools, is knowing when to apply it. In Figure 11.1 we give, in the form of a decision tree, a set of questions worth considering when trying to decide if multisearch is applicable to a specific data structuring problem.

The first question is to ascertain if the problem can be solved in terms of a query data structure, and if, so how many queries will be processed by the data structure. Multisearch is designed to allow a multicomputer to efficiently process many simultaneous queries on a shared data structure rather than using it to speed the processing of a single query. The second question is aimed at determining the relationship between queries. If queries must be processed in a fixed order (i.e. they are dependent on each other) it will be hard to extract parallelism at the query level and either the queries must be made to be independent or one should seek parallelism at some other level of the problem. The third question aims at determining if it is really necessary for each query to actually traverse its search path. Consider, for example, the problem of parallelizing n search queries on a two three tree [95] of size n . Clearly, multisearch can be used to perform this parallelization as two three trees and searches on them meet the basic requirements for all of the multisearch algorithms given in Figure 11.1. Yet there is another approach that should be

Figure 11.1: Deciding if multisearch is applicable.

considered: Sort the n queries with the n nodes of the tree and for each query, the closest node to it in the resulting sorted sequence is the final node of its search path. This sort based approach has the advantage that, although it is not asymptotically more efficient, it is simpler to explain and implement and has significantly lower constants than multisearch. For this application multisearch performs too much work. It would actually advance all queries down their search path when this was not necessary. Multisearch should in general only be applied to the many situations when the traversal, not just the final destination of queries is important. As with all "general" rules there is at least one exception. Question four considers a situation

when it may still be useful to apply multisearch despite the fact that only the final destination of each query is important. There may not be a complete ordering of the set of queries and nodes and in this case the sort based approach is not applicable. It should be noted that the decision tree given in Figure 11.1 is only a guide to when multisearch might be applicable; one must examine each possible application individually.

11.2 Directions for Future Research

There are several directions in which one could attempt to further generalize the parallel data structuring techniques presented in this thesis. For example,

1. Can the class of graphs used to represent the data structure on each parallel model be extended?
2. Can a dynamic version of multisearch, like the one defined for the hypercube, be defined for mesh and coarse-grained multicomputers?
3. Can catalog structures, like the ones defined for the hypercube version of multisearch, be added to mesh or coarse-grained versions of multisearch?

Another direction for possible further research is in the area of scalable algorithms for coarse-grained multicomputers. In Chapters 9 and 10 of this thesis we give scalable algorithms for solving several important geometric problems on multicomputers including 2D-visibility from a point, weighted dominance counting of planar points, 3D-maxima, all nearest neighbors in the plane and convex hull of planar points. There are many other important geometric problems that should be considered. In particular, algorithms concerned with decomposition of the plane such as, trapezoidal decomposition, triangulation and vornoi diagrams, are worth consideration.

Bibliography

- [1] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap. Parallel computational geometry. *Algorithmica*, 3(3):293–327, 1988.
- [2] S. G. Akl. Getting convex hulls in constant time. Technical report, Queen's University, Department of Computing and Information Science, Queen's University, Kingston Ontario, Canada, 1981.
- [3] S. G. Akl. A constant-time parallel algorithm for computing convex hulls. *BIT*, 22:130–134, 1982.
- [4] S. G. Akl. *Parallel Sorting Algorithms*. Notes and Reports in Computer Science and Applied Mathematics. Academic Press, 1985.
- [5] S. G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.
- [6] S. G. Akl and K.A. Lyons. *Parallel Computational Geometry*. Prentice Hall, 1993.
- [7] M. G. Andrews and D.T. Lee. Parallel algorithms for convex bipartite graphs and related problems. In *Allerton Conference Communications, Control, and Computing*, pages 195–204, 1992.
- [8] M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: a technique for designing parallel algorithms. Technical Report CSD-TR-665, Purdue University, Department of Computer Science, Purdue University, 1987.

- [9] M. J. Atallah, F. Dehne, R. Miller, A. Rau-Chaplin, and J.-J. Tsay. Multisearch techniques for implementing data structures on a mesh-connected computer. In *ACM Symposium on parallel algorithms and architectures*, pages 204-214, Hilton Head, South Carolina, 1991.
- [10] M. J. 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*, 1992. To appear.
- [11] M. J. Atallah and J.-J. Tsay. On the parallel decomposability of geometric problems. Technical report, Purdue University, Department of Computer Science, Purdue University., West Lafayette., IN, USA 47907, 1992.
- [12] K.E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference*, pages 307-314, 1968.
- [13] J. L. Bentley. Algorithms for Klee's rectangle problems. Carnegie Mellon, Department of Computer Science. Unpublished notes, 1977.
- [14] J. L. Bentley and D. F. Stanat. Analysis of range searches in quad trees. *Information Processing Letters*, 3(6):170-173, 1975.
- [15] S. K. Bhaskar, A. Rosenfeld, and A. Y. Wu. Parallel processing of regions represented by linear quadtrees. *Computer Vision, Graphics and Image Processing*, 42:371-380, 1988.
- [16] L. Bic. Processing of semantic nets on dataflow systems. *Artificial Intelligence*, 27:219-227, 1985.
- [17] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *ACM Symposium on parallel algorithms and architectures*, pages 3-16, 1991.

- [18] D. G. Bobrow and T. Winograd. An overview of KRL, a knowledge representation language. *Cognitive Science*, 1:3-46, 1977.
- [19] J.D. Boissonnat. Shape reconstruction from planar cross sections. *Computer Vision, Graphics and Image Processing*, 44:1-29, 1988.
- [20] R. J. Brachman and J. G. Schmolze. An overview of the KL-One knowledge representation system. *Cognitive Science*, 9(2), 1985.
- [21] A. Bykat. Convex hull of a finite set of points in two dimensions. *Information Processing Letters*, 7:296-298, 1978.
- [22] B. Chazelle. Triangulating a simple polygon in linear time. In *Symposium on Foundations of Computer Science*, pages 220-230, 1990.
- [23] P. Chew and R. L. Drysdale. Voronoi diagrams based on convex distance functions. In *1st Symposium on Computational Geometry*, 1985.
- [24] A. Chow. A parallel algorithm for determining convex hulls set of points in two dimensions. In *Allerton Conference Communications, Control, and Computing*, pages 214-233, March 1981.
- [25] R. Cole. Parallel merge sorting. *SIAM Journal of Computing*, 17(4):770-785, 1988.
- [26] R. Cypher and C. G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. In *ACM Symposium on Theory of Computing*, pages 193-203. ACM, 1990.
- [27] F. Dehne. $O(n^{\frac{1}{2}})$ algorithms for the maximal elements and ECDF searching problem on a mesh-connected computer. *Information Processing Letters*, 22:303-306, 1986.
- [28] F. Dehne. Optical clustering. *The Visual Computer*, 2:39-43, 1986.

- [29] F. Dehne. Solving visibility and separability problems on a mesh-of-processors. *The Visual Computer*, 4(6):356-370, 1988.
- [30] F. Dehne and J.-R. Sack. Translation separability of sets of polygons. *The Visual Computer*, 3:227-235, 1987.
- [31] F. Dehne, A. Fabri, M. Nassar, A. Rau-Chaplin, and R. Valiveti. Construction of d-dimensional hyperoctrees on a hypercube multiprocessor. In *Allerton Conference Communications, Control, and Computing*, pages 373-382, 1992.
- [32] F. Dehne, A. Ferreira, and A. Rau-Chaplin. A massively parallel knowledge-base server using a hypercube multiprocessor. In *IEEE Int. Workshop on Tools for Artificial Intelligence*, pages 660-666, 1990.
- [33] F. Dehne, A. Ferreira, and A. Rau-Chaplin. Parallel branch and bound on fine grained hypercube multiprocessors. *Parallel Computing*, 15:201-209, 1990.
- [34] F. Dehne, A. Ferreira, and A. Rau-Chaplin. Parallel fractional cascading on hypercube multiprocessors. *Computational Geometry: Theory and Applications*, 1991. To appear.
- [35] F. Dehne, A. Ferreira, and A. Rau-Chaplin. Parallel processing of pointer based quadrees. In *International Conference on Parallel Processing*, pages 255-262, St. Charles, Ill, 1991.
- [36] F. Dehne, A. Ferreira, and A. Rau-Chaplin. Parallel processing of pointer based quadrees. In *International Conference on Parallel Processing*, pages 255-262, 1991.
- [37] F. Dehne, R. Miller, and A. Rau-Chaplin. Optical clustering on a mesh connected computer. *International Journal of Parallel Programming*, 1992. To appear.
- [38] 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(4):367-375, 1990.

- [39] J. G. Delgado-Frias and W. R. Moore. Parallel architectures for AI semantic network processing. *Knowledge-Based Systems*, 1(5):259-265, 1988.
- [40] M. Dixon and J. d. Kleer. Massively parallel assumption-based truth maintenance. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 199-204. American Association for Artificial Intelligence, 1988.
- [41] T. Dubitzki, A. Wu, and A. Rosenfeld. Parallel computation of contour properties. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3:331-337, 1981.
- [42] W. Eddy. A new convex hull algorithm for planar sets. *ACM Tran. Math. Software*, 3(4):398-403, 1977.
- [43] S. Edelman and E. Shapiro. Quadrees in concurrent prolog. In *International Conference on Parallel Processing*, pages 544-551, 1985.
- [44] M. Evett and J. Hendler. Parallel knowledge representation on the Connection Machine. In *Proceedings of Parallel Computing 1989*, pages 381-386, Leiden, The Netherlands, 1989.
- [45] S. E. Fahlman, G. E. Hinton, and T. J. Sejnowski. Massively parallel architectures for AI: NETL, Thistle and Boltzman machines. In *AAAI Annual Conference on Artificial Intelligence*, pages 109-113, 1983.
- [46] R. Fikes and T. Kehler. The role of frame-based representation in reasoning. *Communications of the ACM*, 28(9):904-920, 1985.
- [47] R. A. Finkel and J. L. Bentley. Quad trees - a data for retrieval on composite keys. *Acta Informatica*, 4(1):1-9, 1974.
- [48] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice*. The System Programming Series. Addison-Wesley, 1990.

- [49] H. Freeman. Computer processing of line-drawing images. *ACM Computing Surveys*, 6:57-97, 1974.
- [50] M. Goodrich. *Efficient Parallel Techniques for Computational Geometry*. PhD thesis, Purdue University, Department of Computer Science, 1987.
- [51] R. L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Information Processing Letters*, 1:132-133, 1972.
- [52] R. I. Greenberg and C.E. Leiserson. Randomized routing on fat-trees. *Advances in Computing Research*, 5:345-374, 1989.
- [53] A. Gupta. *Parallelism in production systems*. PhD thesis, Carnegie-Mellon, 1986.
- [54] S.E. Hambrusch and F. Dehne. Parallel geometric algorithms on mesh-connected computers. In *International Parallel Processing Symposium*, pages 234-241, 1992.
- [55] W. D. Hillis. *The Connection Machine(Ed.)*. MIT Press, 1985.
- [56] W.D. Hills and G.L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170-1183, 1986.
- [57] Y. Hung and A. Rosenfeld. Parallel processing of linear quadtrees on a mesh connected computer. *Journal of Parallel and Distributed Computing*, 7:1-27, 1989.
- [58] K. Hwang, J. Gosh, and R. Chowkwanyun. Computer architectures for artificial intelligence. *Computer*, 20(1):19-27, 1987.
- [59] O. H. Ibarra and M. H. Kim. Quadtree building algorithms on a SIMD hypercube. Technical Report 90-27, University of California, Department of Computer Science, University of California, Santa Barbara, California 93106, 1990.

- [60] Intel Scientific Computers. *iPSC System Overview*.
- [61] IntelliCorp. *KEE reference manual*, 1986.
- [62] B. Israel and J. Hendler. A highly parallel implementation of a marker passing algorithm. Technical Report CS-TR-2089, University of Maryland, Department of Computer Science, University of Maryland, College Park, 1988.
- [63] K. J. Jacquemain. The complexity of constructing quad-tree^s in arbitrary dimensions. In J. Muehlbacher, editor, *Workshop on Graphtheoretic Concepts in Computer Science*, pages 293–301, 1982.
- [64] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Advanced Reference series. Prentice Hall, USA, 1988.
- [65] C. Jeong and D. T. Lee. Determining maximum k-width connectivity on meshes. In *Proc. of Fall Joint Computer Conference*, 1987.
- [66] C. Jeong and D. T. Lee. Parallel geometric algorithms on a mesh connected computer. Technical Report 87-02-FC-01, Northwestern University, Department of Electrical Engineering and Computer Science, Northwestern University, 1987. To appear in *Algorithmica*.
- [67] D. G. Kirkpatrick and R. Seidel. The ultimate planar convex hull? Technical Report 83 577, Cornell University, Department of Computer Science, Cornell University, October 1983.
- [68] H. T. Kung, F. Luccio, and F.P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM*, 22(4):469–476, 1975.
- [69] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.
- [70] M. Martin, D. M. Chiarulli, and S. S. Iyengar. Parallel processing of quadtrees on a horizontally reconfigurable architecture computing system. In *International Conference on Parallel Processing*, pages 452–454, 1986.

- [71] Ernst W. Mayr and C. Greg Plaxton. Pipelined parallel computation, and sorting on a pipelined hypercube. Technical Report STAN-CS-89-1261, Stanford University, Department of Computer Science, Stanford University, Stanford California 94305, 1989.
- [72] G.-G. Mei and W. Liu. Parallel processing for quadtree problems. In *International Conference on Parallel Processing*, pages 895-902, 1986.
- [73] R. Miller and Q. F. Stout. Efficient parallel convex hull algorithms. *IEEE Transactions on Computers*, 37(12):1605-1618, December 1988.
- [74] R. Miller and Q. F. Stout. Mesh computer algorithms for computational geometry. *IEEE Transactions on Computers*, 38(3):321-340, January 1989.
- [75] R. Miller and Q. F. Stout. *Parallel Algorithms for Regular Architectures*. MIT Press, 1991.
- [76] M. Minsky. *A framework for representing knowledge*, pages 211-277. McGraw Hill, New York, 1975.
- [77] D. I. Moldovan and Y.-W. Tung. SNAP: a VLSI architecture for artificial intelligence. *Journal of Parallel and Distributed Computing*, 2(2):109-131, 1985.
- [78] D. Nassimi and S. Sahni. Bitonic sort on a mesh-connected parallel computer. *IEEE Transactions on Computers*, C-27(2):2-7, January 1979.
- [79] D. Nassimi and S. Sahni. Data broadcasting in SIMD computers. *IEEE Transactions on Computers*, C-30(2):101-107, February 1981.
- [80] D. Nassimi and S. Sahni. Optimal BPC permutations on a cube connected computer. *IEEE Transactions on Computers*, C-31(4):338-341, April 1982.
- [81] D. Nath, S. N. Maheshwari, and P. C. P. Bhatt. Parallel algorithms for convex hull in two dimensions. In *Conf. Anal. Problem Classes Programming Parallel Comput.*, pages 358-372, 1981.

- [82] A. Newell. The knowledge level. *Artificial Intelligence Magazine*, 2(2):1-20, 1981.
- [83] P.W. Palumbo, S.N. Srihari, and J. Soh. Real-time address block location using pipelining and multiprocessing. *IEEE Computer*, 25(7):34-42, July 1992.
- [84] F. P. Preparata and S. J. Hong. Convex hulls of finite sets in two and three dimensions. *Communications of the ACM*, 2(20):87-93, February 1977.
- [85] F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, 1985.
- [86] S. Ranka and S. Sahni. *Hypercube Algorithms: with Applications to Image Processing and Pattern Recognition*. Bilkent University Lecture Series. Springer-Verlag, 1990.
- [87] A. Rau-Chaplin. Efficient parallel data structures for fine-grained hypercube multiprocessors. Master's thesis, Carleton University, School of Computer Science, Carleton University, Ottawa Canada, 1989.
- [88] J. Rice. The advanced architectures project. *Artificial Intelligence Magazine*, pages 27-39, 1989.
- [89] J-R. Sack and G.T. Toussaint. Moveability of objects. In *IEEE International Symposium Information Theory*, pages 71-72, St. Jovite, Canada, 1983.
- [90] H. Samet. Region representation: quadrees from boundary codes. *Communications of the ACM*, 23(3):163-170, 1980.
- [91] H. Samet. Neighbor finding techniques for images represented by quadrees. *Computer Graphics and Image Processing*, 18(1):37-57, 1982.
- [92] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187-260, June 1984.
- [93] H. Samet. *Applications of Spatial Data Structures*. Addison-Wesley, 1989.

- [94] P. S. Sapaty. A wave language for parallel processing of semantic networks. *Comput. Artificial Intelligence*, 5(4):289-314, 1986.
- [95] R. Sedgewick. *Algorithms*. Addison-Wesley Publishing Company, 1988.
- [96] C. L. Seitz. The cosmic cube. *Communications of the ACM*, 28(1):22-33, January 1985.
- [97] M. I. Shamos. *Computational Geometry*. PhD thesis, Yale University, Department of Computer Science, 1978.
- [98] M.I. Shamos. *Algorithms and Complexity*, pages 251-280. Academic Press, New York, 1976.
- [99] M.I. Shamos and D. Hoey. Closest point problems. *SIAM Journal of Computing*, pages 744-757, 1980.
- [100] C. Stanfill. Communications architecture in the Connection Machine system. Technical Report TR HA87-3, Thinking Machines Corporation, Cambridge, Mass., 1987.
- [101] M. J. Stefik, M. Bobrow, D. G. Mittal, and L. Conway. Knowledge programming in LOOPS: Report on an experimental course. *Artificial Intelligence*, 4(3):3-14, 1983.
- [102] Q.F. Stout. Optimal component labeling algorithms for mesh computers and VLSI. *Abstracts of the American Math Society*, 5:148, 1981.
- [103] C. D. Thompson and H. T. Kung. Sorting on a mesh-connected parallel computer. *Communications of the ACM*, 20(4):263-271, April 1977.
- [104] D. S. Touretzky. *The Mathematics of Inheritance Systems*(Ed.) Morgan Kaufmann Publishers, Los Altos, CA, 1986.
- [105] G.T. Toussaint. *Computational Geometry*, pages 335-376. North Holland, 1985.

- [106] J. J. Tsay. Searching tree structures on a mesh of processors. In *Third Annual ACM-SIAM Symposium on Discrete Algorithms*, 1992.
- [107] L. Uhr. *Multi-Computer Architectures for Artificial Intelligence(Ed.)*. John Wiley and Sons, 1987.
- [108] P. Varman and K. Doshi. Sorting with linear speedup on a pipelined hypercube. Technical Report TR-8802, Rice University, Department of Electrical and Computer Engineering, Rice University, 1988.
- [109] B. W. Wah and G.-J. Li. A survey on special purpose computer architectures for AI. *SIGART News*, 4(96):28 -46, 1986.
- [110] W. A. Woods. What's important about knowledge representation? *Computer*, 15(10):222 -29, 1983.
- [111] A. C. Yao. A lower bound to finding the convex hulls. *Journal of the ACM*, 28:780 -787, 1981.
- [112] M. Yau and S. N. Srihari. A hierarchical data structure for multidimensional digital images. *Communications of the ACM*, 26(1), July 1983.

END

1 2 - 1 2 - 9 3

FIN