Solving Large Graph Problems on Coarse-Grained Multicomputers

By
Albert Chan

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

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

March 2003

© Copyright
2003, Albert Chan

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-83514-6

Our file Notre référence

ISBN: 0-612-83514-6

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

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

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

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

Canada

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

Solving Large Graph Problems on Coarse-Grained Multicomputers

submitted by

Albert Chan

Dr. Frank Dehne
(Director, School of Computer Science)

Dr. Frank Dehne
(Thesis Supervisor)

Dr. Selim Akl
(External Examiner)

Carleton University
April 2003

Abstract

Parallel graph algorithm design is a very well studied topic. Many results have been presented, mostly PRAM algorithms, which are inherently fine grained. However, experiments show that PRAM algorithms may not achieve the expected speedup on real machines [5,177].

Recently, researchers have shown interest in developing algorithms for more realistic, coarse grained models such as BSP and CGM, which more accurately reflect the architecture of real machines. We are interested in developing efficient parallel CGM/BSP *graph* algorithms. This is a new area that not many researchers have touched on before. We are particularly interested in graph recognition and maximum matching problems.

We present here CGM/BSP algorithms to recognize bipartite graphs, convex bipartite graphs, and planar graphs. For a graph of size n and for a CGM with p processors, the bipartite graph recognition algorithm runs in $O(\frac{n}{p} \log p)$ time and $O(\log p)$ rounds. The other two algorithms run in $O(\frac{n}{p} \log^2 p)$ time and $O(\log^2 p)$ rounds.

For the maximum matching problems, we have a CGM/BSP algorithm for finding maximum matching in convex bipartite graphs. This algorithm runs in $O(T_s(\frac{n_A}{p}, \frac{n_B}{p}) + \frac{n_A}{p} \log p)$ time and $O(\log p)$ rounds where n_A and n_B are the numbers of vertices in the two groups of the bipartite graph. We also have a CGM/BSP algorithm for finding maximum weight matching in trees. It runs in $O(\frac{n}{p} \log p)$ time and $O(\log^2 p)$ rounds.

We also present in this thesis optimal CGM/BSP algorithms for solving the next element search problem and for parallel sorting. The next element search algorithm runs in $O(\frac{n}{p} \log n)$ local computation. The parallel sorting algorithm runs in either $O(\frac{n}{p} \log n)$ or $O(\frac{n}{p})$, depending on the type of data to be sorted. Both algorithms use $O(1)$ rounds.

We have implemented and measured the performance of some CGM/BSP graph algorithms on PC clusters. The implementation is packaged into two libraries: *CGMlib* provides basic operations for CGM algorithms. *CGMgraph* includes several graph algorithms such as list ranking, Euler tour computation, finding connected components and spanning forest, and bipartite graph recognition. The libraries can be downloaded from <http://www.scs.carleton.ca/~cgm>.

Acknowledgement

First of all, I would like to thank my supervisor, Dr. Frank Dehne, for providing help during the course of the research for this thesis.

Thanks should also be extended to those members on my examining board: Dr. Selim Akl, Dr. Frank Dehne, Dr. Lucia Moura, Dr. Jörg-Rüdiger Sack, and Dr. Murray Woodside. I really do appreciate their willingness to be on the committee.

Many of the algorithms presented in this thesis are the results of joint research works with various co-authors, including: P. Bose, E. Cáceres, F. Dehne, M. Latzel, G. Prencipe, A. Rau-Chaplin, and S. Song. I would like to thank them for their contribution.

Many people have provided technical helps and lab supports during my research works and the preparation of this thesis. I would like to acknowledge their support:

- O. Baltzer (CGMLab, Dalhousie University)
- D. Bickle (HPCVL, Carleton University)
- T. Eavis (CGMLab, Dalhousie University)
- J. Fourier (Nortel Networks)
- D. Marteinson (David Marteinson Consulting)
- J.-R. Sack (HPCVL, Carleton University)
- P. Taillon (HPCVL, Carleton University)

This research is partially supported by Natural Sciences and Engineering Research Council of Canada and Nortel Networks Canada.

Finally, I must thank all my family members, especially my wife Rita, for her tolerance during my long working hours through the whole course of my Ph.D studies, and her support for this thesis.

Contents

Abstract	iii
Acknowledgement	iv
1 Introduction	1
1.1 Motivation	1
1.2 Models	3
1.2.1 Von Neumann Sequential RAM Model	3
1.2.2 PRAM Model	5
1.2.3 BSP Model	6
1.2.4 CGM Model	7
1.2.5 Other Related Models	10
1.3 Hardware Architectures	11
1.4 Overview of Results	18
1.5 Thesis Structure	21
2 Graph Terminology	22
2.1 General Graph Terminology	22
2.2 Bipartite Graphs and Planar Graphs	23
2.3 Matchings	24
2.4 Miscellaneous	25
3 Survey of previous Parallel Graph Algorithms	27
3.1 PRAM Graph Algorithms	27

3.2	CGM Graph Algorithms	30
3.3	Implementation of Parallel Graph Algorithms	33
4	Graph Recognition	34
4.1	Problem Overview	34
4.2	Recognizing <i>Bipartite</i> Graphs	35
4.3	Recognizing <i>Convex</i> Bipartite Graphs	37
4.3.1	The PQ-Trees	37
4.3.2	Multiple <i>Disjoint</i> Reduction on PQ-Trees	39
4.3.3	MReduce	45
4.3.4	Convex Bipartite Graph Recognition	59
4.4	Planarity Testing	60
4.4.1	Problem Overview	60
4.4.2	Coarse-Grained Parallel Graph <i>st</i> -Numbering	61
4.4.3	Coarse-Grained Parallel Graph Planarity Testing	63
4.5	Summary of Results	70
5	Maximum Matching on Graphs	71
5.1	Problem Overview	71
5.2	Max. Matching in Convex Bipartite Graphs	72
5.2.1	Algorithm Description	73
5.2.2	Special Case: All Intervals Start at the Same Point	73
5.2.3	The General Case	76
5.3	Maximum Weighted Tree Matching	79
5.3.1	Partitioning a Tree via its Centroid	80
5.3.2	Re-rooting a Tree	81
5.3.3	Maximum Gain Alternating Path	82
5.3.4	Maximum Weight Matching for Trees	84
5.4	Summary of Results	87
6	Next Element Search	88
6.1	Problem Overview	88

6.2	Segment Tree Definition	89
6.3	Parallel Segment Tree Construction	90
6.4	Parallel Query Processing	94
6.5	A Simplified Algorithm for Axis Parallel Line Segments	96
6.6	Applications	97
6.7	Summary of Results	98
7	Coarse Grained Sorting	99
7.1	Problem Overview	99
7.2	The Algorithm	100
7.3	Generalization	103
7.4	Summary of Results	104
8	CGMlib and CGMgraph	105
8.1	Introduction	105
8.2	Library Overview and Experimental Setup	106
8.3	CGMlib: Basic Infrastructure and Utilities	109
8.3.1	CGM Communication Operations	109
8.3.2	CGM Utilities	110
8.3.3	Performance Evaluation	112
8.4	CGMgraph: Parallel List Ranking and Euler Tour	113
8.4.1	Randomized List Ranking	115
8.4.2	Deterministic List Ranking	116
8.4.3	Euler Tour	116
8.4.4	Performance Evaluation	117
8.5	CGMgraph: Parallel Connected Components and Spanning Forest . .	119
8.5.1	Outline of Method	120
8.5.2	Performance Evaluation	121
8.6	CGMgraph: Parallel Bipartite Graph Recognition	123
8.6.1	Outline of Method	123
8.6.2	Performance Evaluation	123
8.7	Summary	125

9 Conclusion	126
9.1 Summary of CGM/BSP Algorithms	127
9.2 Future Work	129
Bibliography	133

List of Figures

1.1	The von Neumann model	3
1.2	System bus model	4
1.3	Coarse Grained Multicomputer model	8
1.4	h -Relation in CGM Model	9
1.5	The LogP model	10
1.6	Shared memory processors (SMP) machine	12
1.7	Distributed shared memory (DSM) machine	13
1.8	A completely connected network with 8 processors	14
1.9	A star connected network with 9 processors	14
1.10	Linear array and ring	15
1.11	A 2-D mesh	16
1.12	A complete binary tree network	17
1.13	A 4-dimension hypercube	17
1.14	A beowulf cluster (PC cluster)	18
1.15	Performance of the bipartite graph recognition algorithm	20
2.1	A bipartite graph	23
2.2	A convex bipartite graph	24
2.3	A planar graph	24
2.4	A matching of a graph	25
4.1	A typical PQ-tree	38
4.2	A universal PQ-tree.	39
4.3	The algorithm at a glance	56

4.4	An st -numbered graph	61
4.5	PQ-trees representing several permutations	64
4.6	B_3 and the corresponding PQ-tree	65
4.7	A set of joinable nodes	67
4.8	Illustration of the rotate operation	68
5.1	Re-rooting a tree	82
5.2	Gain computation through Euler tour list ranking	85
6.1	Illustration of next element search	89
6.2	Decomposition of the segment tree	91
6.3	The tree T_0 with catalogs partitioned into lists l_i	92
8.1	General use of <i>CGMlib</i> and <i>CGMgraph</i>	107
8.2	Hierarchy of main classes in <i>CGMlib</i> and <i>CGMgraph</i>	108
8.3	Performance of the h -Relation implementation	113
8.4	Performance of the prefix sum implementation	114
8.5	Performance of the parallel sort implementation	115
8.6	Performance of the randomized list ranking algorithm	118
8.7	Performance of the deterministic list ranking algorithm	119
8.8	Performance of the Euler tour algorithm	120
8.9	Performance of the connectd components algorithm	121
8.10	Performance of the spanning forest algorithm	122
8.11	Performance of the bipartite graph recognition algorithm	124

Chapter 1

Introduction

1.1 Motivation

In order to solve bigger and more complex problems, multiple processors are linked together to gain more computing power. However, ordinary algorithms cannot take advantage of multiple processors since they are designed to run on a single computer.

There is a need to develop algorithms that take advantage of multiple processors but it is impossible to design algorithms that can work on all processor connection patterns (except for very simple parallel algorithms called embarrassingly parallel algorithms, see [4]). It is necessary to make some assumptions on how the processors are connected together, in order to simplify algorithm design. These assumptions are collectively called the parallel computing models.

A very common parallel computing model that has been well studied since the 1960s is the “Parallel Random Access Machine” or “PRAM” model. See Section 1.2.2 for definition of the PRAM model. In this model, each processor has limited local memory, but has access to a large amount of global memory. Processors are not connected to each other, but have direct access to global memory. Due to the fine grained property of this model, we also assume the number of processors available to be unlimited.

Theoretical results from PRAM algorithms do not necessarily match the speedups observed on *real* parallel machines [5, 177]. This is because each processor only has

limited (usually $O(1)$) local memory. That is, the size of memory cannot be increased when the input size increases. As a result, the processors must communicate with each other frequently to exchange information in order to continue execution. The problem here is that communication between processors is far slower than the computation speed.¹ Therefore the more communication, the slower the algorithms.

Newer models have been proposed to minimize this communication overhead. One of these models we study here is the “Coarse Grained Multicomputers” (CGM) model. See Section 1.2.4 for definition of the CGM model. We found that this model is more practical than the PRAM model in that the assumption and cost model used reflect better the reality of commercially available multiprocessors. More precisely, the CGM model is a version of the BSP model [185] (see Section 1.2.3 for definition). In comparison to the BSP model, the CGM [55] model allows only bulk messages in order to minimize message overhead costs. A CGM is comprised of a set of p processors $P_1, \dots, P_p$ with $O(\frac{N}{p})$ local memory per processor (where N is the total memory capacity of all processors) and an arbitrary communication network (or shared memory). All algorithms consist of alternating local computation and global communication rounds. Each communication round consists of routing a single h -Relation with $h = O(\frac{N}{p})$, that is, each processor sends and receives $O(\frac{N}{p})$ data (See Section 1.2.3 for a description of g). We require that all data sent from a given processor to another processor in one communication round is packed into one long message, thereby minimizing the communication overhead. A CGM computation / communication round corresponds to a BSP superstep with communication cost $g\frac{N}{p}$ (plus the above “packing requirement”). Finding an optimal parallel algorithm in the CGM model is thus equivalent to minimizing the number of communication rounds as well as the total local computation time. The CGM model has the advantage of producing results which correspond much better to the actual performance on commercially available parallel machines. In addition to minimizing communication and

¹Even with the advance of high speed switch, like the recent 10-gigabit Ethernet card, the communication overhead problem still persists as processors have to exchange handshaking messages before a connection can be established. For short messages, the handshaking overhead can be much larger than the message itself, and therefore, making communication appears to be much slower than it should be.

computation volume, it also minimizes other important costs like message overheads and processor synchronization.

Our goal here is to design efficient CGM graph algorithms with results that match those of real parallel machines.

1.2 Models

1.2.1 Von Neumann Sequential RAM Model

The computing model for sequential algorithms is commonly referred to as a von Neumann machine [135]. The von Neumann model consists of five major components as illustrated in Figure 1.1. The Input Unit provides instructions and data to the system, which are subsequently stored in the Memory Unit. The instructions and data are processed by the Arithmetic and Logic Unit (ALU) under the direction of the Control Unit (CU). The results of the calculation are sent to the Output Unit. The ALU and CU are frequently collectively referred to as the central processing unit (CPU). Most commercial computers can be decomposed into these five basic units.

Figure 1.1: The von Neumann model

The stored program is the most important aspect of the von Neumann model. A program is stored in the computer's memory along with the data to be processed. That means the applications are not hard-wired. All programs and data stored in the memory unit can be accessed at any time. As a result, the von Neumann model is also referred to as a "Random Access Machine" or "RAM".

The random access of any memory unit is important in algorithm design as it allows constant time access to any memory location. Consequently, all the basic operations in an algorithm such as additions, subtractions, or assignments take constant time.

Modern computers usually streamline the von Neumann Model into a System Bus Model. Figure 1.2 shows the system bus model of a computer system. This model partitions a computer system into three subunits: CPU, Memory, and Input/Output (I/O). This version of the von Neumann model combines the ALU and the CU into one functional unit, the CPU. The input and output units are also combined into a single I/O unit.

Figure 1.2: System bus model

In the system bus model, communication between the components is by means of a shared pathway called the system bus. The system bus is made up of the data bus (which carries the information being transmitted), the address bus (which identifies where the information is being sent), and the control bus (which describes aspects of how the information is being sent, and in what manner). Some architectures may also have a separate I/O bus.

Notice that although the communication among the three units (CPU, memory, and I/O) is through the system bus, the access of *any* memory location still takes constant time. Therefore, a machine that uses the system bus model is still a RAM.

1.2.2 PRAM Model

For many years, the “Parallel Random Access Machine” or “PRAM” model [4] has been the de facto model for studying parallel algorithms. A PRAM consists of p synchronous processors with shared memory. This model ignores synchronization problems and communication issues. It concentrates on the task of parallelization of the problem. Based on how two or more processors are permitted to access the same memory location at the same time, several variations of this model are available:

- EREW = Exclusive Read Exclusive Write, only one processor can read from and write to a location in any single step.
- CREW = Concurrent Read Exclusive Write, any number of processors can read from a location in a step, but only one processor can write to a location in any single step.
- ERCW = Exclusive Read Concurrent Write, only one processor can read from a location in any single step, but any number of processors can write to a location in any single step.
- CRCW = Concurrent Read Concurrent Write, any number of processors can read from and write to a location in a step.

In the case of concurrent write, what happens if two or more processors try to write to the same location? Several strategies can be used to resolve the possible conflicts:

- Common: All processors must be trying to write the same value. Otherwise, an error will occur.
- Arbitrary: An arbitrary processor succeeds in the case of a write conflict.

- Priority: The lowest numbered processor succeeds.

We define $T(n, p)$ to be parallel execution time for the algorithm under consideration with p processors on input of size n . Let $T_s(n)$ be the time complexity of the *best* known sequential algorithm that solves the particular problem. The speedup of a parallel algorithm is defined by $S(n, p) = \frac{T_s(n)}{T(n, p)}$ [108]. Speedup can range from 0 to p . Under normal situations, the speedup will not be greater than p . Generally, we prefer algorithms with speedup proportional to p , that is, $S(n, p) = k * p$ for some constant k . Asymptotically, we will write this as $\Omega(p)$, with the understanding that a constant factor is hidden inside the notation. The folding principle states that we can always use fewer processors and get similar efficiency.

Although very simple, the PRAM model has a serious limitation. As each processor has only relatively little (usually $O(1)$) local memory, the PRAM algorithm is inherently fine grained. That is, the problem size is usually not much larger than the number of processors. In fact, it is not unusual that the number of processors is in the order of or much larger than the problem size. It is very difficult to build a real PRAM machine, as every processor must be connected to each memory location.

Although we can simulate a PRAM machine with fewer processors, but experience shows that such simulations usually result in large amounts of small messages being sent and received between processors. As communication is still the bottleneck of parallel algorithms, it will actually slow down our algorithms. In some cases this slow down will come to a point that the parallel algorithms will run even slower than the sequential algorithms.

1.2.3 BSP Model

The Bulk Synchronous Parallel (BSP) model is designed to overcome the connectivity problem (as mentioned in the previous section) of the PRAM model. It was proposed by Valiant in 1990 [185].

A BSP machine consists of p processors. Each processor has a large amount of local memory. There is no global memory. The processors are connected through a network. The processors communicate with each other by sending and receiving

messages.

BSP algorithms consist of a sequence of supersteps. In each superstep, the processors operate independently performing local computation and global communication by sending and receiving messages, but the messages sent in one superstep can only be received in the next superstep. At the end of each superstep, a barrier synchronization is done. This barrier synchronization keeps all processors synchronized.

The analysis of BSP algorithms is based on two parameters: g and L . L is an upper bound on the latency, or delay, incurred in communicating a message from its source processor/memory module to its target processor/memory module. g is the gap, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal of g , $\frac{1}{g}$, corresponds to the available per-processor communication bandwidth. Since all the messages in a superstep can be grouped into a large message, if a superstep consists of w local computation and h messages being sent and received, we have $T_{superstep} = w + gh + L$. For practical purpose, we assume the running time is dominated by g , and therefore, L can be ignored.

1.2.4 CGM Model

The Coarse Grained Multicomputer (CGM) Model is a variant of Valiant's BSP model [185]. It was proposed by Dehne, Fabri, and Rau-Chaplin in 1993 [58].

A Coarse Grained Multicomputer $CGM(N, p)$ consists of p processors $P_1, \dots, P_p$, where each processor has $O(\frac{N}{p})$ local memory. Here N refers to the total memory capacity of the CGM. The processors can communicate with each other either by sending and receiving messages over the network, or by being connected to shared memory. If the processors are connected through shared memory, the size of the shared memory should be large enough for an h -Relation (see below). That is, the size of global memory should be $\Omega(n)$.

CGM algorithms usually make the assumption that the problem size is *much* larger than the number of processors ($N \gg p$). A common criterion in many algorithms is that $\frac{N}{p} \geq p$. In some algorithms, this assumption is generalized to $\frac{n}{p} \geq p^\epsilon$, where ϵ is a

parameter describing the scalability (which measures how well the algorithms behave when the CGM gradually degenerates into a fine-grained machine) of the algorithms.

A CGM algorithm consists of alternating local computation and global communication rounds, as illustrated in Figure 1.3. In the local computation rounds, only local computation on local data is allowed. In each communication round, a single h -Relation (with $h = O(\frac{n}{p})$) is routed, i.e., each processor sends and receives $O(\frac{n}{p})$ data, all packed into a single message.

Figure 1.3: Coarse Grained Multicomputer model

The h -Relation is the basic communication operation used in CGM algorithms. It performs an “all-to-all” operation for each processor. During an h -Relation operation, every processor can send out messages of size $h = O(\frac{n}{p})$, and receive messages of size $h = O(\frac{n}{p})$. The individual sizes of messages being sent to and received from each processor need not be the same, but the total size of all messages being sent to or received from any processor cannot exceed the prescribed bounds. Figure 1.4 illustrates an h -Relation.

A CGM algorithm is essentially a BSP algorithm with additional “packing” and “coarse grained” requirements. It is very easy to convert CGM algorithms into BSP algorithms. Because of the added requirements, the CGM model is slightly more powerful than the BSP model.

Figure 1.4: h -Relation in CGM Model

Lemma 1 *A CGM algorithm with c communication rounds corresponds to a BSP algorithm with communication cost $c \times g_p^n$ where n is the input size and p is the number of processors.*

The BSP model assumes h messages (of constant size) to be sent and received in a superstep. Since the BSP model assume all messages are available only until the next superstep, we can in practice group all messages into a single message that is sent and received at the end of each superstep. Therefore, the h messages in the BSP superstep has the same meaning (up to a constant factor) as the CGM h -Relation and each CGM round corresponds to a BSP superstep with communication cost g_p^n .

The goal of the CGM algorithm design is to both minimize the local computation and the number of communication rounds. It will be ideal if we can achieve $O(1)$ communication rounds and $T(n, p) = \frac{T_s(n)}{p}$ where n is the problem size and p is the number of processors.

From a practical point of view, i.e. for implementing this model on commercial multiprocessors, it is very important that the problem size is not a parameter for the

number of communication rounds. This is usually achievable, but for many problems, it is very difficult or even impossible to reduce the number of rounds down to $O(1)$, and we have to settle for $O(\log p)$ or $O(\log^2 p)$ rounds. Since p is usually fixed or grows very slowly in practice, and $\log p$ is a slowly growing function, the number of communication rounds is essentially a fixed constant for most practical arrangements. This is important because empirical studies show that the number of communication rounds is the most important parameter influencing the observed running time.

1.2.5 Other Related Models

There are other models introduced in order to overcome the weakness of the PRAM model. Some of them are discussed in this section.

LogP Model

Culler et al. proposed a new model to overcome the difficulty of the PRAM model. This model is called “LogP” [2, 35, 36, 115, 115]. It is very similar to the BSP model. This model measures the performance of an algorithm using four parameters (see Figure 1.5):

Figure 1.5: The LogP model

L : An upper bound on the Latency, or delay, incurred in communicating a message containing a word (or small number of words) from its source processor/memory module to its target processor/memory module.

- o: The *overhead*, defined as the length of time that a processor is engaged in the transmission or reception of each message. During this time, the processor cannot perform other operations.
- g: The *gap*, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal of g corresponds to the available per-processor communication bandwidth.
- P: The ratio of the number of *Processors* to memory modules.

Bilardi et. al. showed that the BSP model is slightly stronger than the logP model [12].

QSM Model

Ramachandran et al. introduced the Queuing Shared Memory (QSM) Model [78–81, 170]. This model is similar to the BSP model and the CGM model. The communication vehicle used in this model is shared memory.

Algorithms using this model are divided into phases. Each phase consists of local computation, as well as global communication by reading and writing to the shared memory. Values returned by a shared memory read cannot be used in the same phase. Also, the same location cannot be read from and written to within the same phase. Furthermore, if two or more processors want to write to the same location, the requests are queued.

It has been shown that it is relatively easy to convert algorithms from the QSM model into BSP and logP models [87].

1.3 Hardware Architectures

As mentioned before, there can be many ways to connect processors together. When designing algorithms, we usually ignore the topology of the connection. However, the way the processors are connected together will have an impact on the performance of the parallel algorithms.

The distance between two processors is defined as the smallest number of processors a message has to go through from one processor to the other. The diameter d of

a network is defined as the largest distance of the network. Obviously, the worst case performance cannot be better than $O(d)$ because messages have to be relayed from processor to processor. Therefore, one of the goals in building a parallel machine is to minimize its diameter.

In the following, we briefly discuss some common connection patterns [4, 86, 132].

Shared Memory Processors (SMP)

The shared memory processors (SMP) architecture is the hardware architecture that is well suited to the PRAM model. Since all processors are communicated by global memory, the theoretical diameter of SMPs is $O(1)$. Figure 1.6 shows a block diagram of a typical SMP arrangement.

Figure 1.6: Shared memory processors (SMP) machine

A real shared memory machine with more than a few processors is relatively difficult to build. This is because each processor has to be wired into each memory location. To overcome this problem, manufacturers partition the shared memory into groups. Each group is local to one processor. This local memory is treated as global

memory by a memory manager. If a processor needs to access a piece of data that is located inside another processor, the processor will issue a regular memory read/write instruction. The memory manager will intercept this instruction, and send a message to access the required data. When the message is returned, the memory manager will signal the originating processor the result of the memory access operation. Figure 1.7 shows a block diagram of this Distributed Shared Memory structure.

Figure 1.7: Distributed shared memory (DSM) machine

Completely Connected Networks

A completely connected network is a network in which each processor has a direct connection to every other processor. Obviously, the diameter of a completely connected network is always 1. Figure 1.8 shows an example of a completed connected network with 8 processors.

Figure 1.8: A completely connected network with 8 processors

Star Connected Networks

In a star connected network, a single processor is placed in the center of the network. Every other processor has a direct connection to this processor. All communication must go through this central processor. The diameter of a star connected network is always 2. See Figure 1.9 for an example of a star connected network with 9 processors.

Figure 1.9: A star connected network with 9 processors

Linear Arrays and Rings

A linear array is a set of processors connected in a row in which each processor, except for the first and the last processors, is connected to two neighboring processors. The diameter of a p processor linear array is $p - 1$.

If we connect the first and the last processors together, we get a ring. The diameter of a p processor ring is $\lfloor \frac{p}{2} \rfloor$.

Figure 1.10 shows a block diagram of a linear array and a ring.

Figure 1.10: Linear array and ring

Meshes

If the processors are laid out in a grid and are connected in such a way that each processor is connected to the four neighboring processors, except for the processors on the four edges (which are connected to three neighboring processors) and the processors on the four corners (which are connected to only two neighboring processors), the processors form a 2-D mesh. A $\sqrt{p} \times \sqrt{p}$ mesh has a diameter of $2\sqrt{p} - 2$. Figure 1.11 shows a block diagram of a 2-D mesh.

We can extend the 2-D mesh into a torus by “wrapping” the connections on the edge- and the corner- processors to the other side. The diameter of a $\sqrt{p} \times \sqrt{p}$ torus is $2\lfloor \frac{\sqrt{p}}{2} \rfloor$.

We can also extend the 2-D mesh into a 3-D mesh (cube) by putting all the processors in a 3-D grid, and connecting each processor to its six neighboring processors, except for the processors on the six surfaces (which are connected to five neighboring

Figure 1.11: A 2-D mesh

processors), the processors on the twelve edges (which are connected to four neighboring processors) and the processors on the eight corners (which are connected to only three neighboring processors). The diameter of a $\sqrt[3]{p} \times \sqrt[3]{p} \times \sqrt[3]{p}$ cube is $3\sqrt[3]{p} - 3$.

Tree Networks

Another popular network topology is the tree network. In a tree network, one processor is identified as the root. All other processors are fanned out from the root, or from other processors that are directly or indirectly fanned out from the root.

The diameter of a tree network depends on the actual topology. For example, a p processor complete binary tree network (where $p + 1$ is an exact power of 2) has a diameter of $2 \log_2(p + 1) - 2$.

Figure 1.12 shows a block diagram of a complete binary tree network.

Hypercube Networks

Hypercubes were popular in the 1980s. An r -dimension hypercube has 2^r processors. The processors are numbered from 0 to $2^r - 1$. Each processor has r connections. Two

Figure 1.12: A complete binary tree network

processors P_i and P_j are directly connected if and only if their processor numbers differ only by one digit in the binary format. For example, in a 4-dimension hypercube, P_5 (0101) and P_{13} (1101) are directly connected.

The diameter of a $p = 2^r$ processor hypercube is $r = \log_2 p$.

Figure 1.13 shows a block diagram of a 4-dimension hypercube.

Figure 1.13: A 4-dimension hypercube

Beowulf Clusters

Beowulf clusters have become very popular recently because of their low cost. The Beowulf clusters consist of groups of low cost computers (usually personal computers)

connected together through a fast switch or router. Therefore, they are also called PC clusters. Figure 1.14 shows a block diagram of the Beowulf cluster.

Figure 1.14: A beowulf cluster (PC cluster)

We are particularly interested in developing algorithms that can be run efficiently on PC clusters. The CGM model is very suitable to PC clusters because PC clusters are inherently coarse grained. Experiments show that the observed results match those predicted by the model (see next section).

1.4 Overview of Results

Our goal is to design efficient CGM algorithms for graph (and related) problems; with an emphasis on graph matching and graph recognition.

- Matching:

We present a CGM algorithm for maximum matching in convex bipartite graphs. The local computation of this algorithm is $O(T_s(\frac{n}{p}, \frac{m}{p}) + \frac{n}{p} \log p)$ and the number of communication rounds is $O(\log p)$. This result is published in [14].

We present a CGM algorithm for maximum weight matching in trees. The local computation of this algorithm is $O(\frac{n}{p} \log p)$ and the number of communication rounds is $O(\log^2 p)$. This result is published in [25].

- Graph recognition:

We present a CGM algorithm for recognizing whether a given graph is bipartite. The algorithm runs in $O(\frac{n}{p} \log p)$ time and uses $O(\log p)$ communication rounds. This result is published in [17].

We present a CGM algorithm for recognizing whether a given bipartite graph is convex. The algorithm runs in $O(\frac{n}{p} \log^2 p)$ time and uses $O(\log^2 p)$ communication rounds. This result is published in [17].

We present a CGM algorithm for recognizing whether a given graph is planar. The algorithm runs in $O(\frac{n}{p} \log^2 p)$ time and uses $O(\log^2 p)$ communication rounds. This result is published in [18].

- Other algorithms:

We present two CGM algorithms for next element search. Both algorithms have local computation of $O(\frac{n}{p} \log n)$ and require $O(1)$ communication rounds. The first algorithm can be used to solve the general case of the next element search problem. This algorithm requires $O(\frac{n}{p} \log n)$ storage. If the line segments are all axis parallel, the second algorithm can be used and this algorithm requires only $O(\frac{n}{p})$ storage. These results are published in [27].

We present a CGM algorithm for sorting integers in parallel. The algorithm runs in $O(\frac{n}{p})$ time and uses $O(1)$ communication rounds. This result is published in [24]. The algorithm was originally designed for sorting integers. However, with a minor modification, the same algorithm can also be used to sort general data. In this case, the local computation of the algorithm is $O(\frac{n}{p} \log n)$ and the number of communication rounds is still $O(1)$.

- Implementation:

In order to verify that the algorithms we have developed run efficiently on PC clusters, we have implemented some of the algorithms and collected timing data on PC clusters. The implementation is packaged into two libraries: *CGMlib* and *CGMgraph*. *CGMlib* provides basic operations for implementing CGM algorithms. *CGMgraph* implements several graph algorithms, including

list ranking, Euler tour computation, finding connected components/spanning forest, and recognizing bipartite graphs. The libraries are available for download at <http://www.scs.carleton.ca/~cgm>.

The measured results are consistent with our analysis. For example, we have run the bipartite graph recognition algorithm on the HPCVL PC cluster called *THOG*. Figure 1.15 summarizes the data collected.

Figure 1.15: Performance of the bipartite graph recognition algorithm

The theoretical running time of the bipartite graph recognition is $O(\frac{n}{p} \log p)$ time and $O(\log p)$ communication rounds. From Figure 1.15, we can see that the computation time is very close to $\frac{1}{p}$. Also, except for some noise when p is small, the communication time remains almost flat. This shows that $\log p$ is a very slowly growing function, and for practical purposes can be regarded as a constant. More details on our experiments will be presented Chapter 8.

1.5 Thesis Structure

The remainder of this thesis is divided into the following chapters: Chapter 2 explains some of the graph terminology used in this thesis; Chapter 3 contains a survey of previous work done on parallel graph algorithms; Chapter 4 discusses our algorithms for recognizing bipartite graphs, convex bipartite graphs and planar graphs; Chapter 5 discusses our algorithms for finding maximum matching in convex bipartite graphs and trees; Chapter 6 presents an algorithm for the next element search problem; Chapter 7 presents an algorithm for performing sorting on a CGM; Chapter 8 discusses the implementation of *CGMlib* and *CGMgraph*, as well as our experimental results; Chapter 9 outlines the plans for future research and summarizes our results.

Chapter 2

Graph Terminology

We now explain some of the terminology that will be used in this thesis. For a complete list of graph terminology, please refer to graph theory books, such as [64, 99, 148, 181].

2.1 General Graph Terminology

A graph $G = (V, E)$ is an ordered pair of sets V and E , where $E \subseteq V \times V$. We usually refer to V as vertices and E as edges. When drawing a graph, the vertices are usually drawn as circles. The edges will be lines connecting vertices. Two vertices $u \in V$ and $v \in V$ are connected if and only if $(u, v) \in E$.

Unless otherwise specified, we assume $G = (V, E)$ and u, v are vertices in V .

An edge $(u, v) \in E$ is said to be undirected if $(v, u) \in E$ and both (u, v) and (v, u) refer to the same edge.

An edge $(u, v) \in E$ is said to be directed if either $(v, u) \notin E$ or $(v, u) \in E$ but (u, v) and (v, u) refer to two different edges.

A graph G is said to be undirected if and only if all the edges in E are undirected. Similarly, G is directed if and only if all the edges in E are directed. Otherwise, we say G is a mixed graph. In this thesis, we will only consider undirected graphs.

A graph G is weighted if every edge in E is associated with a number. This number is referred to as the weight of the edge. If no weights are associated with the

edges, then G is called an unweighted graph.

In a graph G , two vertices u and v are said to be adjacent if $(u, v) \in E$. An edge $e \in E$ is said to be incident on a vertex $u \in V$ if u is one of the endpoints of e ; that is, $e = (u, v)$ for some $v \in V$.

2.2 Bipartite Graphs and Planar Graphs

A bipartite graph $G = (A, B, E)$ is defined as an undirected graph $G = (V, E)$ such that $V = A \cup B$ and $A \cap B = \emptyset$. Moreover, $\forall e = (a, b) \in E$, either $a \in A$ and $b \in B$ or vice versa. Figure 2.1 shows a bipartite graph.

Figure 2.1: A bipartite graph

A convex bipartite graph is a bipartite graph $G = (A, B, E)$ with an ordering $B = (b_1, b_2, \dots, b_{n_B})$ such that $\forall a \in A$, if $(a, b_i) \in E$ and $(a, b_j) \in E$ ($i \leq j$) then $(a, b_k) \in E$ for all $i \leq k \leq j$. Figure 2.2 shows a convex bipartite graph.

A planar graph is a graph that can be drawn on a plane such that no two edges intersect each other. Figure 2.3 shows a planar graph. A planar graph with n vertices will have at most $3n - 6$ edges [99, 181].

Figure 2.2: A convex bipartite graph

Figure 2.3: A planar graph

2.3 Matchings

A path in a graph G is an ordered list of edges $P = (e_1, e_2, \dots, e_n)$ such that for each $e_k = (u_k, v_k)$, we have $v_k = u_{k+1}$. If $u_1 = v_n$, the path is called a cycle.

A simple path is a path with no repeated internal vertex. Similarly, a simple cycle is a cycle with no repeated vertex. Note that a simple cycle is also a simple path.

A tree is a connected (sub)graph with no cycle. We sometimes call this a free tree to distinguish it from a rooted tree. A (free) forest is a graph which consists only of (free) trees.

A matching M in a graph $G = (V, E)$ is a subset of E such that no two edges in M are incident to the same vertex. Figure 2.4 shows a matching of a graph.

Figure 2.4: A matching of a graph

A matching M' in a graph $G = (V, E)$ is maximum if for any matching M in G , $|M'| \geq |M|$. The matching shown in Figure 2.4 is also maximum.

A matching M in a graph G is perfect if $|M| = \frac{|V|}{2}$. Note that a perfect matching must be maximum.

2.4 Miscellaneous

A graph G is connected if for every vertex u , there is at least one path leading from u to v for any vertex v in G . A connected component of G is a connected subgraph of G such that the size of the subgraph cannot be increased.

Let $G = (V, E)$ be a graph with k connected components, a spanning forest of G is a subgraph that consists of all the vertices in V and $|E| - k$ edges from E .

The Euler tour of a tree T is a traversal of T from its root, travelling all edges exactly twice (once going down and once going up), and finally returning to the root of T .

Given a tree T and two leaves v_1, v_2 of T , the lowest common ancestor of v_1, v_2 is a node u of T that u is an ancestor of both v_1 and v_2 but any child w of u is not.

Given a graph $G = (V, E)$, an *ear decomposition* of G is an ordered partition of E into a set of r simple paths $P_1, P_2, \dots, P_r$ such that P_1 is a cycle, and for each $2 \leq i \leq r$, P_i is a simple path with endpoints belonging to $P_1 \cup P_2 \cup \dots \cup P_{i-1}$ but with none of its internal vertices belonging to P_j ($j < i$). The paths P_i are called ears. If none of P_i ($i > 1$) is a cycle, then the decomposition is called an *open ear decomposition*. For each edge e in P_i , i is referred to as the *ear number* of e [20].

Chapter 3

Survey of previous Parallel Graph Algorithms

Parallel graph algorithms have been extensively studied in the literature (see [173] for a survey). However, most of the published parallel graph algorithms have traditionally been designed for the theoretical PRAM model. In this chapter, we will give a brief review of some of the work done in the past.

3.1 PRAM Graph Algorithms

Graph Recognition

Many graph problems are difficult to solve for general graphs but easier for certain classes of graphs. There are many algorithms designed specifically for these graphs. Therefore, it is an important task to recognize these graphs. There are many classes of graphs that we need to recognize. Diks et. al. presented an optimal parallel algorithms for recognizing outerplanar graphs [66]. Edenbrandt published a paper on chordal graph recognition [68, 123, 179]. Schoenmakers provided an algorithm to recognize series parallel graphs [174]. Klein et. al. have algorithms for planarity testing [122, 127].

Matching

Matching is an important graph problem. It has many applications and has been extensively studied. Andrews et. al. presented an algorithm for maximum matching in interval graphs [6]. Dahlhaus et. al. presented algorithms on matchings for various graphs [40–42, 46–48]. Goldberg et. al. have a sublinear-time parallel algorithm for matching [83]. Grover published an article on bipartite matching [89]. Kao et. al. presented a nearly optimal parallel algorithm for maximum matching in planar graphs [111]. Karpinski et. al. also have published on parallel algorithms about graph matching [116]. Kelsen presented an algorithm for finding matching in expander graphs [119] and an optimal parallel algorithm for maximum matching [120]. Kim et. al. provided an optimal parallel matching algorithm for convex bipartite graph on mesh [121]. Klein has matching algorithms for chordal graphs [123–126]. Moitra et. al. presented an algorithm for maximum matching on interval graphs [153]. Nakanishi et. al. have a parallel algorithm for matching in general [155]. Wu et. al. provided a distributed algorithm for finding maximum matching in general graphs [188]. Bauernöppel et. al. described an $O(n \log^2 n)$ algorithm for maximum matching in permutation graphs [7, 8].

Graph Coloring

Graph coloring is also an interesting problem. We can color either the vertices or the edges. In vertex-coloring, we want to color the vertices such that every edge will be incident to two vertices that are of different colors. In edge-coloring, we want to color the edges such that no two edges of the same color will be incident to the same vertex. Bhandari et. al. presented a parallel algorithm for coloring graphs [11]. Kadem et. al. have parallel algorithms for coloring random graphs [118]. Kucera et. al. presented a constant time parallel algorithm for coloring k -colorable graphs [128, 186]. Caspi et. al. provided a near-optimal parallel algorithm for edge-coloring outerplanar graphs [21]. Gibbons et. al. have published parallel algorithms for optimal edge-coloring trees [77]. Ogunyode presented parallel algorithms for approximate edge-coloring simple graphs [161]. Diks presented a parallel algorithm for vertex-coloring

planar graphs using only six colors [65]. Later, Hagerup et. al. published a parallel algorithm solving the same problem, but using only five colors [94–96, 98, 156]. He et. al. presented parallel algorithms for vertex-coloring perfect planar graphs [100, 180].

Spanning Forest

Finding a spanning forest of a graph provides a convenient way to traverse the graph. Hagerup presented an optimal parallel algorithm for finding minimum spanning forests (MSF) in planar graphs [97]. Nassimi has published a parallel algorithm for finding MSF for planar graph using a mesh [157]. Dessmark et. al. provided parallel algorithms for finding spanning forests in dense graphs [61].

Graph Partitioning

In order to apply the famous divide-and-conquer technique, we sometimes need to partition a graph into subgraphs. A good partitioning algorithm will partition the graphs into subgraphs of approximately the same size. However, this is not an easy task. Gazit et. al. presented a parallel algorithm for finding a separator in planar graphs [75]. Gilbert et. al. provided a parallel graph partitioning algorithm for a message passing parallel machine [82]. Bui et. al. provided parallel algorithms for simple graph partitioning [15]. Lee et. al. studied the parallel k -way graph partitioning problem [131]. Natarajan studied the graph partitioning problem in shared memory processors [158]. Lor et. al. used divide-and-conquer to perform minimal cut bisectioning of task graphs [142]. Ou et. al. used linear programming to perform parallel incremental graph partitioning [162]. Dasdan et. al. presented efficient multi-way graph partitioning algorithms [53]. Karypis et. al. provided a multi-level graph partitioning scheme [117]. Tetelbaum provided an estimation of the graph partitioning for a hierarchical system [183]. Recently, Kumar et. al. presented algorithms on performing parallel graph partitioning in heterogeneous grid engines [129].

Other Graph Problems

There are many other interesting graph problems. The following are just a few of them:

- Perfect eliminating order [44, 45]
- Maximal independent sets [84, 102, 114]
- Connectivity [34, 72, 74, 88, 107, 112]
- Cliques [3, 37]
- Graph contraction [50, 90, 144, 149–151, 168, 187]
- Hamiltonian cycle [38, 39, 43, 49, 70, 145, 159, 178]
- Graph search [51, 93]
- Shortest paths [28, 105, 152, 164, 182]
- Graph isomorphism [31, 73, 76, 134, 137–139, 143]

3.2 CGM Graph Algorithms

The area of CGM graph algorithms is less explored. Dehne et al. presented the first CGM graph algorithms [19]. Recently, scholars have started to show interest in this topic, and we are beginning to see some of their results [14, 16, 17, 19, 23–25, 27, 55]. These results, however, are still very limited and there is a lot of room for improvement.

Our goal here is to design CGM graph algorithms that are efficient on real parallel machines.

[19] presented several CGM algorithms that are commonly used. We will use several of them as subroutines in our algorithms. The following subsections provide a brief description on each of these algorithms that we use as subroutines. For details on these algorithms, please see [19]. In the following subsections, we assume n to be the input size and p to be the number of processors available in a CGM.

List Ranking

Definition 1 *The rank of an element a_i in a list of n elements $A = \{a_1, a_2, \dots, a_n\}$ is defined as its distance to the end of the list. For example, the last element in the list has a rank of 0 and the first element in the list has a rank of $n - 1$.*

Definition 2 *Given a list of n elements $A = \{a_1, a_2, \dots, a_n\}$, each element in A is associated with a pointer which points to the next element in the list, the list ranking operation is to find for each element a_i its rank in the list.*

The list ranking algorithms presented in [19] consists of three phases. In the first phase, we select a set of $\frac{n}{p}$ samples. Depending on how the samples are selected, we can have a randomized list ranking algorithm or a deterministic list ranking algorithm. In the second phase, using simulated pointer jumping [173], every element obtains a pointer to the next selected sample in the list, together with the distance to that element. All selected sample are then compressed into one processor and sequential weighted list ranking is applied where the weights are the distances obtained in the previous step. In the last phase, all other (non-selected) nodes determine their rank by obtaining the rank of the next selected sample (as determined in the previous step) and adding to it the distance obtained through the simulated pointer jumping. This algorithm runs in $O(\frac{n}{p} \log p)$ local computation and $O(\log p)$ communication rounds.

Euler Tour Computation

The Euler tour traversal of a tree is simple. We first convert each edge in the tree into two directed edges. These edges will form a linked list that each edge knows which one is the next edge. Then, we can apply the list ranking algorithm to determine the traversal order. This algorithm has the same performance as the list ranking algorithm. That is, $O(\frac{n}{p} \log p)$ local computation and $O(\log p)$ communication rounds.

Connected Components and Spanning Forest

The algorithm for finding the connected components and spanning forest consists of two phases. In the first phase, we simulate $\log p$ iterations of the standard PRAM

algorithm for connected component computation [173] on the CGM. Each round of the PRAM algorithm creates “super vertices” by combining adjacent vertices (or super vertices). Each round reduces the number of vertices or super vertices by at least a constant factor. Therefore, after $\log p$ iterations, the number of (super) vertices is reduced to $O(\frac{n}{p})$. In the second phase, each processor obtains a copy of the $O(\frac{n}{p})$ (super) vertices and stores a subset of the edges (which could be still more than $\frac{n}{p}$ in total). Each processor first builds a spanning forest for its subset of edges and then $\log p$ “binary merge” steps are performed in which pairs of processors merge their two spanning forests into one spanning forest. After $\log p$ merging steps, we obtain one single spanning forest with at most $O(\frac{n}{p})$ edges. Each tree in the spanning forest represents a connected components. This algorithm runs in $O(\frac{n}{p} \log p)$ local computation and $O(\log p)$ communication rounds.

Lowest Common Ancestor

The algorithm for finding the lowest common ancestors for the vertices in a tree T consists of four steps:

- Compute an Euler tour for T .
- Find the levels, in T , for all vertices of the Euler tour.
- For each vertex v in T , find $l(v)$ and $r(v)$ which denote the leftmost and rightmost appearances, respectively, of v in the Euler tour.
- Solve the range-minima problem defined as follow: given a list of members $\{b_1, b_2, \dots, b_n\}$ and an interval $[i, j]$, with $1 \leq i \leq j \leq n$, find the minimum of $\{b_i, \dots, b_j\}$.

This algorithm runs in $O(\frac{n}{p} \log p)$ local computation and $O(\log p)$ communication rounds.

Tree Contraction

The algorithm for contracting a tree can be done by simulating the PRAM algorithm for $O(\log p)$ rounds to reduce the tree to the size of $O(\frac{n}{p})$. The resulting tree then can be sent to a single processor to finish the contraction. Therefore, the algorithm runs in $O(\frac{n}{p} \log p)$ local computation and $O(\log p)$ communication rounds.

Open Ear Decomposition

The CGM algorithm for open ear decomposition is a simulation of the PRAM algorithm in [146], as described in [19].

This algorithm runs in $O(\frac{n}{p} \log p)$ local computation and $O(\log p)$ communication rounds.

3.3 Implementation of Parallel Graph Algorithms

Various experimental results have been presented for the list ranking problem. Reid-Miller [172] reported on experiments for a simple randomized list ranking method on the Cray C-90. Dascal and Vishkin [52] studied list ranking methods for multi-threaded architectures. Sibeyn et. al. reported on parallel list ranking implementations for shared memory machines [92] and processor arrays [176]. For the connected component problem, experimental results for the MASPARE MP-1 were presented in [54].

Recently, parallel graph algorithms for more “PC cluster like” parallel models like the BSP [185] and CGM [55, 58] have been presented in [14, 17, 19, 20, 59]. Gustadt et. al. [130] implemented a randomized list ranking algorithm for the CGM and concluded that “the coarse grained multicomputer model (CGM) is well suited to design competitive algorithms, and that it is thereby now possible to aim to develop portable, predictable and efficient parallel algorithms code for graph problems.”

Chapter 4

Graph Recognition

4.1 Problem Overview

Many graph problems are difficult to solve for general graphs, but are easier for certain classes of graphs. If we know in advance that the graphs we are interested in are within the given category, we can apply specially designed algorithms to take advantage of properties that are unique to that category. Therefore, it is important to develop algorithms to recognize and categorize graphs.

In this chapter, we study the problem of graph recognition. We will study the following problems:

- Given an arbitrary graph G , determine whether G is *bipartite*.
- Given a bipartite graph G , determine whether G is *convex*.
- Given an arbitrary graph G , determine whether G is *planar*.

These and other closely related problems have been extensively studied for the sequential [13, 184] and the shared memory (PRAM) parallel [29, 30, 101, 124–126] domain. In this chapter, we present parallel CGM algorithms for recognizing bipartite, convex bipartite, and planar graphs.

On a CGM with p processors, to solve the bipartite graph recognition problem of size n , our bipartite graph recognition algorithm requires $O(\log p)$ communication

rounds and linear sequential work per round. This implies a BSP algorithm with $O(\log p)$ supersteps, $O(g \frac{n}{p} \log p)$ communication time, and $O(\frac{n}{p} \log p)$ local computation time.

The other two algorithms (convex bipartite graph recognition and planarity testing) require $O(\log^2 p)$ communication rounds and linear sequential work per round. Our results imply BSP algorithms with $O(\log^2 p)$ supersteps, $O(g \frac{n}{p} \log^2 p)$ communication time, and $O(\frac{n}{p} \log^2 p)$ local computation time.

The algorithm for recognizing bipartite graphs is fairly simple and is essentially a combination of tools developed in [19]. The larger part of this chapter deals with the problem of recognizing *convex* bipartite graphs and *planar* graphs. These are clearly more difficult problems. They have been extensively studied in the literature and are closely linked to the *consecutive ones* problem for $(0, 1)$ -matrices as well as chordal graph recognition [13, 29, 30, 101, 124–126, 184].

Our algorithm for determining whether a bipartite graph is convex includes a coarse grained parallel version of the *PQ tree* data structure introduced by Booth and Lueker [13]. Hence, our algorithm also solves, with the same time complexity as indicated above, the problem of testing the consecutive-ones property for $(0, 1)$ -matrices as well as the chordal graph recognition problem. These, in turn, have numerous applications in graph theory, DNA sequence assembly, database theory, and other areas [13, 29, 30, 101, 124–126, 184].

4.2 Recognizing *Bipartite* Graphs

In this section, we present a fairly simple CGM algorithm for recognizing bipartite graphs. It is essentially a combination of tools developed in [19].

Algorithm 1 Recognition of bipartite graphs

Input: A Graph $G = (V, E)$ with vertex set V and edge set E , $|G| = |V| + |E| = N$, stored on a CGM with p processors and $O(\frac{N}{p})$ memory per processor; $\frac{N}{p} \geq p^\epsilon$ for some fixed $\epsilon > 0$. V and E are arbitrarily and evenly distributed over the memory of the CGM.

Output: A Boolean value indicating whether G is a bipartite graph and, if it is, a partition of V into two disjoint sets A and B such that $E \subseteq (A \times B)$ (Note that we assume $A \times B$ is the same as $B \times A$ since the graph is undirected).

- (1) Compute a spanning forest of G [19].
- (2) For each tree in the forest, select one arbitrary node as the root. Apply the CGM Euler Tour algorithm in [19] to determine the distance between each node and the root of its tree. Classify the nodes into two groups: the nodes with an odd numbered distance to the root, and the nodes with an even numbered distance to the root.
- (3) Each processor examines the edges stored in its local memory. If any edge has its two vertices belong to the same group, the result for that processor is “failure”; otherwise, the result is “success”.
- (4) By applying CGM sort (see Chapter 7 for details) to all “failure”/“success” values, it is determined whether there was any processor with a “failure” result. If there was any “failure”, the graph G is *not* bipartite. Otherwise, G is a bipartite graph, and the two groups of vertices identified in Step 2 are the sets A and B .

— End of Algorithm —

Theorem 1 *Algorithm 1 recognizes whether $G = (V, E)$, $|G| = |V| + |E| = N$, is a bipartite graph and, if so, partitions V into sets A and B ($A \cup B = V$ and $A \cap B = \emptyset$) such that $E \subseteq (A \times B)$ in $O(\log p)$ communication rounds and $O(\frac{N}{p})$ local computation per round on a CGM with p processors and $O(\frac{N}{p})$ memory per processor, $\frac{N}{p} \geq p^\epsilon$ for some fixed $\epsilon > 0$.*

Proof. The correctness of the algorithm is based on the fact that G is bipartite if and only if it has no odd length cycle. If G is not bipartite, then there exists a non-tree edge $e = (v_1, v_2)$ (with respect to the spanning trees constructed in Step 1) that closes an odd length cycle. The vertices v_1 and v_2 must both have either odd or even distance to the root (as calculated in Step 2) because otherwise the cycle would be of even length. Hence, e generates a “failure” in Step 3. On the other hand, if an

edge e generates a “failure” in Step 3, then it is an edge in an odd length cycle and, therefore, G is not bipartite.

Steps 1 and 2 require $O(\frac{N}{p})$ storage, $O(\log p)$ communication rounds, and $O(\frac{N}{p})$ local computation per round for $\frac{N}{p} \geq p^\epsilon$ for some fixed $\epsilon > 0$ [19]. Step 3 requires $O(\frac{N}{p})$ local computation and $O(1)$ communication rounds. Step 4 requires $O(1)$ communication rounds and $O(\frac{N}{p})$ local computation per round for $\frac{N}{p} \geq p^\epsilon$ for some fixed $\epsilon > 0$ [85]. Therefore, the algorithm can determine if a graph is bipartite within the given bounds. $\square$

Theorem 1 implies a BSP algorithm with local computation of $O(\frac{n}{p} \log p)$ and communication cost of $O(g \frac{n}{p} \log p)$.

We can calculate the speedup for this algorithm as follow. Let $T_s(n) = O(n)$ be the running time of the best sequential algorithm. Let $T_p(n, p) = O(\frac{n}{p} \log p) + O(g \frac{n}{p} \log p)$ be the running time of our algorithm. Let c , d and k be the constants associated to the different big-O notations. The speedup is therefore $s = \frac{T_s(n)}{T_p(n, p)} = \frac{cn}{\frac{dn}{p} \log p + \frac{kgn}{p} \log p} = \frac{p}{\log p} (\frac{c}{d+kg}) = \frac{p}{g \log p} (\frac{c}{k+\frac{d}{g}}) = \Omega(\frac{p}{g \log p})$.

4.3 Recognizing *Convex* Bipartite Graphs

We now turn our attention to the problem of testing whether a given bipartite graph is a convex bipartite graph. The sequential solution, presented by Booth and Lueker [13], introduced a data structure called *PQ-tree*. Our solution will include a coarse grained parallel version of the PQ-tree. We will first review Booth and Lueker’s PQ-tree definition.

4.3.1 The PQ-Trees

A PQ-tree [13] is a tree data structure that represents a class of permissible permutations over a universal set.

Definition 3 *A tree T is a PQ-tree if every internal node of T can be classified as either a P-node or a Q-node. A P-node is an internal node that has at least 2 children, and the children can be permuted in any order. A Q-node is an internal node that has*

at least 3 children, and the children can only be permuted in two ways: the original order or the reverse order. The leaves of the PQ-tree are elements of a universal set $S = \{a_1, \dots, a_n\}$, usually called the ground set.

Figure 4.1 shows a typical PQ-tree.

Figure 4.1: A typical PQ-tree

The order of the ground set in the PQ-tree, from left to right, is called its *frontier*. The frontier of a PQ-tree is clearly a permutation of the ground set. Given a PQ-tree T and using only permissible permutations of its internal nodes, we can generate a number of permutations of S . We will denote with $L(T)$ the set of all these permissible permutations. A PQ-tree T' is *equivalent* to T if T' can be transformed into T using only permissible permutations of the internal nodes (and, of course, if $L(T')$ and $L(T)$ have the same elements).

Given a set $A \subset S$, we define that $\lambda \in L(T)$ *satisfies* A if all elements of A appear consecutively in λ . The main operation on a PQ-tree T is *reduction*. Given a *reduction set* $\mathcal{A} = \{A_1, \dots, A_m\}$ of subsets of S and a PQ-tree T , the reduction operation transform T into another PQ-tree T' , if it exists, such that each permutation in $L(T')$ satisfies every A_i , $1 \leq i \leq m$.

Two particular PQ-trees are the *universal* and the *empty* tree: the first one has only one internal node (the root of T) and that internal node is a P-node (Figure

4.2); the second one (also called a *null* PQ-tree) is used to represent an impossible reduction, that is when it is impossible to reduce a PQ-tree with respect to a given reduction set.

Figure 4.2: A universal PQ-tree.

4.3.2 Multiple *Disjoint* Reduction on PQ-Trees

In this section, we outline a coarse grained parallel algorithm for performing multiple disjoint reductions on a PQ-tree. More precisely, given a PQ-tree T , we wish to perform the *reduction* operation for a set $\mathcal{A} = \{B_1, \dots, B_r\}$, which are subsets of the universal set S where $B_1, \dots, B_r$ are disjoint. That is $B_i \cap B_j = \emptyset$ if $i \neq j$. We shall refer to our algorithm as Algorithm MDREDUCE. For ease of discussion, each set B_i is assigned a unique color, and we color the leaves of the PQ-tree accordingly. As shown in [124], MDREDUCE can be partitioned into the following phases: (1) pre-process the PQ-Tree (Algorithm 2); (2) process all the P-nodes (Algorithm 3); (3) process all the Q-nodes (Algorithm 4); (4) if T is not a null tree, each processor examines its nodes and relabels each orientable node (to be defined later) to become an R-node; (5) post-process the PQ-Tree (Algorithm 5). In the following subsections, we present coarse grained parallel algorithms for each phase. The PQ-tree definitions used are from [13, 124–126].

Pre-Processing the PQ-Tree

The pre-processing phase extends the coloring δ of the leaves to a coloring Δ of all nodes of the PQ-tree T .

For an internal node v of T , we say that a color is *complete* at v if all the leaves with that color are descendants of v . We say a color is *incomplete* at v if some, but not all, of the leaves of that color are descendants of v . We say that a color *covers* v if all the leaves below v are of that color, and that v is *uncovered* if no color covers v .

Let $LCA(c)$ be the lowest common ancestor of all leaves with color c . Let $COLORS(v)$ denote the set of colors assigned to leaves that are descendants of v . Let $INC(v)$ be the set of colors which are incomplete at v . Then $INC(v) = COLORS(v) - \{c: LCA(c) \text{ is a descendent of } v\}$.

Algorithm 2 Pre-processing the PQ-tree

Input: The input PQ-Tree of Algorithm MDREDUCE.

Output: The original PQ-tree T in which each node is assigned a “coloring” Δ , or, if failure occurs, a null tree.

- (1) Apply the coarse grained parallel Lowest Common Ancestor (LCA) algorithm [19].
- (2) Expand T into a binary tree B , as described in Klein [124].
- (3) Perform tree contraction on B [19]. For each node v_b in B , let v_p be the node in T from which v_b is created. Let w_1 and w_2 be the children of v_b . The operation for the tree contraction is $INC(v_b) = INC(w_1) \cup INC(w_2) - \{c: LCA(c) \text{ is a descendent of } v_p\}$. If at any point the size of INC is more than two, stop and return a null tree.
- (4) Let c_v be a new color unique to node v . Each processor, for all its nodes, v , calculates $\Delta(v) = \langle c_1, c_2 \rangle$ as follows: if two colors are incomplete at v , then c_1 and c_2 are these colors; if only one color c is incomplete at v but c does not cover v , then $c_1 = c$ and $c_2 = c_v$; if one color c is incomplete at v and covers v , then $c_1 = c_2 = c$; if no color is incomplete at v , then $c_1 = c_2 = c_v$.

— End of Algorithm —

Lemma 2 *Let n be the input size of Algorithm 2. On a coarse grained multiprocessor with p processors and $O(\frac{n}{p})$ storage per processor, Algorithm 2 can be completed in $O(\log p)$ communication rounds with $O(\frac{n}{p})$ local computation per round.*

Proof. It is obvious that Step 1 and Step 3 are within all the bounds [19]. Steps 2 and 4 will require $O(\frac{n}{p})$ storage, $O(\frac{n}{p})$ local computation, and $O(1)$ communication rounds, respectively. Therefore, the whole algorithm is within all the bounds. $\square$

Processing P-Nodes

A node v in a PQ-tree is *orientable* if it is a Q-node and the two colors in $\Delta(v) = \langle c_1, c_2 \rangle$ are different, i.e. $c_1 \neq c_2$.

For a color c , define $h_v(c) = \begin{cases} c & \text{if } c \in INC(v) \\ c_v & \text{if } c \notin INC(v) \end{cases}$

For a PQ-tree T where w_1 and w_k are the leftmost and rightmost elements, respectively, of the frontier $fr_T(v)$, let $l_T = h_v(\delta(w_1))$ and $r_T = h_v(\delta(w_k))$ (recall that $\delta(v)$ is the original color assigned to the leave v). We define $\langle a, b \rangle \sim \langle a', b' \rangle$ if $\{a, b\} = \{a', b'\}$.

Algorithm 3 Processing P-nodes

Input: The PQ-Tree output from Algorithm 2.

Output: The original PQ-tree T in which all the P-nodes have been processed, or, if failure occurs, a null tree.

- (1) If the input PQ-tree T is a null tree, return T .
- (2) Each processor sets variable *FAILURE* to *FALSE*
- (3) Each processor, for each P-node v , reorders the children of v such that for each color c all children covered by c are consecutive.
- (4) Each processor, for each P-node v and each color c , does the following: if there are at least two children covered by c (and at least one child not covered by c) then insert a new P-node w_c between these c -covered children and v .
- (5) Each processor, for each P-node v , constructs an auxiliary graph G_v whose nodes are the children of v and where for each color c there is an edge between children v_i and v_j at which c is incomplete if either v_i or v_j is covered by c , or there is no child covered by c . If any node has more than 2 neighbors, set *FAILURE* to *TRUE* to indicate a failure condition.
- (6) Perform a multi-broadcast of the variable *FAILURE*. If at least one of the broadcast values is *TRUE*, return a null tree.

- (7) Each processor uses list-ranking to identify the connected components of each G_v and verifies that each of these connected components is a simple path. If any of these components is a cycle, set *FAILURE* to *TRUE* to indicate a failure condition. We call these paths *color chains*.
- (8) Perform a multi-broadcast of the variable *FAILURE*. If at least one of the broadcast values is *TRUE*, return a null tree.
- (9) Each processor, for each color chain χ containing at least 2 nodes, chooses one of the 2 orientations of χ arbitrarily. Reorder the children of v so that the nodes of χ are consecutive, and insert a new Q-node between these nodes of v .
- (10) Each processor, for each P-node v , sets $S = \{v_i : v_i \text{ is a child of } v, \text{ and } INC(v_i) = \emptyset\}$. If every child of v is in S , then return. Otherwise, reorder the children of v to make S consecutive, insert a new P-node v' between v and the subset S (if $|S| > 1$), and change v to be a Q-node.

— End of Algorithm —

Lemma 3 *Let n be the input size of Algorithm 3. On a coarse grained multiprocessor with p processors and $O(\frac{n}{p})$ storage per processor, Algorithm 3 can be completed in $O(\log p)$ communication rounds with $O(\frac{n}{p})$ local computation per round.*

Proof. It is obvious that all steps use at most $O(\frac{n}{p})$ storage and $O(\frac{n}{p})$ local computation. Step 1 does not need communication, Step 6 needs $O(\log p)$ communication rounds and all other steps require $O(1)$ communication rounds. Therefore, total communication requirement is $O(\log p)$ rounds. $\square$

Processing Q-Nodes

For each Q-node v , we define an orientation $\overline{LR}[v]$ which is either $\Delta(v_i)$ or $\Delta(v_i)^R$. Note that if $\Delta(v_i) = \langle c_1, c_2 \rangle$ then $\Delta(v_i)^R = \langle c_2, c_1 \rangle$.

For $\langle a, b \rangle \sim \langle a', b' \rangle$ and $a \neq b$, we define $\langle a, b \rangle \text{swap} \langle a', b' \rangle$ as *TRUE* if $\langle a, b \rangle = \langle b', a' \rangle$, and *FALSE* if $\langle a, b \rangle = \langle a', b' \rangle$.

For a Q-node v , *flip* is defined as the operation which re-orders all its children in reverse order.

Algorithm 4 Processing Q-nodes**Input:** The PQ-tree output from Algorithm 3.**Output:** The original PQ-tree T in which all the Q-nodes have been processed, or, if failure occurs, a null tree.

- (1) If the input PQ-tree T is a null tree, return T .
- (2) Each processor sets variable *FAILURE* to *FALSE*.
- (3) Each processor, for each Q-node v of T and its children $v_1, \dots, v_s$, assigns to each $\overline{LR}[v_i]$ either $\Delta(v_i)$ or $\Delta(v_i)^R$, such that every color in the sequence $\overline{LR}[v_1], \dots, \overline{LR}[v_s]$ occurs consecutively, and $h_v(\langle \overline{L}[v_1], \overline{R}[v_s] \rangle) \sim \Delta(v)$. If this is impossible, set *FAILURE* to *TRUE* to indicate a failure condition, otherwise, set $\overline{LR}[v]$ to $h_v(\langle \overline{L}[v_1], \overline{R}[v_s] \rangle)$.
- (4) Perform a multi-broadcast of the variable *FAILURE*. If at least one of the broadcast values is *TRUE*, return a null tree.
- (5) Each processor, for each node v , does the following: if v is orientable, then set $OPP[v]$ to $\overline{LR}[v]$ swap $\overline{LR}[v]$; otherwise, set $OPP[v]$ to *FALSE*.
- (6) Each processor, for each node v , sets $REV[v]$ to $\bigoplus_{u \text{ is an ancestor of } v} OPP[u]$ ($\oplus$ denotes “exclusive-or”).
- (7) For each orientable node v , if $REV[v]$ is *TRUE*, then flip v .

— End of Algorithm —

Lemma 4 *Let n be the input size of Algorithm 4. On a coarse grained multiprocessor with p processors and $O(\frac{n}{p})$ storage per processor, Algorithm 4 can be completed in $O(\log p)$ communication rounds with $O(\frac{n}{p})$ local computation per round.*

Proof. It is obvious that all steps use at most $O(\frac{n}{p})$ storage and $O(\frac{n}{p})$ local computation. Step 1 does not need communication, Step 5 needs $O(\log p)$ communication rounds and all other steps require $O(1)$ communication rounds. Therefore, total communication requirement is $O(\log p)$ rounds. $\square$

Post-Processing the PQ-Tree**Algorithm 5** Post-processing the PQ-tree

Input: The PQ-tree output from Algorithm 4, with all R-nodes renamed.

Output: Result of Algorithm MDREDUCE.

- (1) If T is a null tree, return.
- (2) Each processor temporarily cuts the links of its Q-nodes to their parents.
- (3) Each processor performs pointer jumping for all its nodes that are children of R-nodes to determine their lowest Q-node ancestor.
- (4) Each processor restores the links cut in Step 2.
- (5) Each processor eliminates its R-nodes by setting the parents of their children to their lowest Q-node ancestors.

— End of Algorithm —

Lemma 5 *Let n be the input size of Algorithm 5. On a coarse grained multiprocessor with p processors and $O(\frac{n}{p})$ storage per processor, Algorithm 5 can be completed using $O(\log p)$ communication rounds with $O(\frac{n}{p})$ local computation per round.*

Proof. It is obvious that all the steps require $O(\frac{n}{p})$ storage, and thus the total storage requirement is also $O(\frac{n}{p})$. Step 1 requires $O(1)$ local computation and all the other steps requires $O(\frac{n}{p})$ local computation, so the total local computation is $O(\frac{n}{p})$. For communication requirements, Steps 2 and 4 do not require communication, Steps 1 and 5 require $O(1)$ communication rounds and Step 3 requires $O(\log p)$ communication rounds, so the total requirements are $O(\log p)$ communication rounds. $\square$

Analysis

Theorem 2 *On a coarse grained multiprocessor with p processors and $O(\frac{n}{p})$ storage per processor, Algorithm MDREDUCE performs a multiple disjoint reduction for a PQ-tree T in $O(\log p)$ communication rounds with $O(\frac{n}{p})$ local computation per round.*

Proof. The correctness of the algorithm follows Klein [124]. The complexity of Phases 1, 2, 3 and 5 follows Lemmas 2, 3, 4 and 5. Phase 4 requires $O(\frac{n}{p})$ storage, $O(\frac{n}{p})$ local computation, and no communication. Therefore, the total complexity is $O(\frac{n}{p})$ storage, $O(\frac{n}{p})$ local computation, and $O(\log p)$ communication rounds. $\square$

4.3.3 MReduce

Using the coarse grained parallel MDREDUCE algorithm presented in the previous section, we will now develop a coarse grained parallel algorithm for the general *MReduce* operation: given a PQ-tree T over the ground set S with n elements, perform the reduction operation for an arbitrary reduction set $\mathcal{A} = \{A_1, \dots, A_k\}$.

Our CGM algorithm for the general *MReduce* operation consists of two phases. In the first phase, we execute $3 \log_{\frac{4}{3}} p$ times (see Page 54 for explanation) an algorithm which is a CGM implementation of a PRAM algorithm proposed by Klein [124]. We call this operation $\text{MREDUCE}_1(T, \{A_1, \dots, A_k\}, 0)$. Our contribution here is the implementation of the various shared memory PRAM steps on a distributed memory CGM, which is nontrivial. After this first phase, we have reduced the problem to one in which we are left with a set of smaller PQ-trees, of which the sizes are at most n/p . Hence, each tree can be stored in the local memory of one processor. However, we cannot guarantee that all the *reduction sets* of these PQ-trees also fit in the local memory of one processor. In the second phase of our algorithm, we use a merging strategy to complete the algorithm. We will refer to this phase as the *Merging Phase*.

Algorithm 6 MReduce

Input: A PQ-tree T over the ground set S and the reduction sets $A_1, \dots, A_k$.

Output: T reduced with respect to $A_1, \dots, A_k$.

- (1) Execute Algorithm 7 with $(T, \{A_1, \dots, A_k\}$, and $i = 0)$.
- (2) Merging phase.

— End of Algorithm —

First Phase: the MREDUCE_1

Before describing the algorithm, we recall the following definition from Klein [124].

Definition 4 For a node v of a PQ-tree, $\text{leaves}_T(v)$ denotes the set of pendant leaves of v , i.e. leaves of T having v as ancestor. Let $\text{lca}_T(A)$ denote the least common ancestor in T of the leaves belonging to A . Suppose that $v = \text{lca}_T(A)$ has children $v_1, \dots, v_s$ in order. We say A is contiguous in T if:

- v is a Q -node, and for some consecutive subsequence $v_p, \dots, v_q$ of the children of v , $A = \bigcup_{p \leq i \leq q} \text{leaves}_T(v_i)$; or
- v is a P -node or a leaf, and $A = \text{leaves}_T(v)$.

Suppose that E is contiguous in T ; $T|E$ denotes the subtree consisting of $\text{lca}_T(E)$ and those children of $\text{lca}_T(E)$ whose descendants are in E (it is still a PQ-tree whose ground set is E).

For a set A , define

$$A_i|E = \begin{cases} A_i \cap E & \text{if } A_i \cap E \neq E \\ \emptyset & \text{if } A_i \cap E = E \end{cases}$$

Let $\star_E$ denote $\text{lca}_T(E)$. T/E denotes the subtree of T obtained by omitting all the proper descendants of $\text{lca}_T(E)$ that are ancestors of elements of E (it is still a PQ-tree whose ground set is $S - E \cup \{\star_E\}$).

For a set A , define

$$A_i/E = \begin{cases} A_i - E \cup \{\star_E\} & \text{if } A_i \supseteq E \\ A_i - E & \text{otherwise} \end{cases}$$

From now on we indicate with n the size of the ground set S of the input PQ-tree T with $m = \sum_{i=1}^k |A_i|$, where each A_i s is one of the input reduction sets (that is, we want to reduce T , having S as ground set, with respect to A_i). The main procedure of the reduction algorithm is the following:

Algorithm 7 MREDUCE_1

Input: A PQ-Tree T , a reduction set $A = \{A_1, \dots, A_k\}$ and an integer i .

- (1) If $i = 3 \log_{\frac{4}{3}} p$, return.
- (2) Purge the collection of input sets A_i of empty sets. If no sets remain, return.
- (3) Let n be the size of the ground set of T . If $n \leq 4$, carry out the reduction one by one. If the size of the input is smaller than the size of the local memory of the processors, then solve the problem sequentially using Booth and Lueker algorithm [13].

- (4) Otherwise, let $\mathcal{A}$ be the family of (nonempty) sets A_i . Let $\mathcal{S}$ consist of the sets A_i such that $|A_i| \leq \frac{n}{2}$. We call such sets “small.” Let $\mathcal{L}$ be the remaining “large” sets in $\mathcal{A}$. Find the connected components of the intersection graph of $\mathcal{A}$, find a spanning forest of the intersection graph of $\mathcal{S}$, and find the intersection $\cap \mathcal{L}$ of the large sets.
- (5) Proceed according to the following cases:
- (5.1) The intersection graph of $\mathcal{A}$ is disconnected. In this case, let $\mathcal{C}_1, \dots, \mathcal{C}_r$ be the connected components of $\mathcal{A}$. For $i = 1 \dots r$, let E_i be the union of sets in the connected component $\mathcal{C}_i$. Call MDREDUCE to reduce T with respect to the disjoint sets $E_1, \dots, E_r$. Next, for each $i = 1 \dots r$ in parallel, recursively call this algorithm with $T = T|E_i$, $A = \mathcal{C}_i$ and $i = i + 1$.
- (5.2) The union of sets in some connected components of $\mathcal{S}$ has cardinality at least $\frac{n}{4}$. In this case, from the small sets making up this large connected component, select a subset whose union has cardinality between $\frac{n}{4}$ and $\frac{3n}{4}$. Let E be this union, and call Algorithm 9 (see below) with $(T, E, \{A_1, \dots, A_k\}, \text{ and } i)$.
- (5.3) The cardinality of the intersection of the large sets is at most $\frac{3n}{4}$. In this case, from the large sets choose a subset whose intersection has cardinality between $\frac{n}{4}$ and $\frac{3n}{4}$. Let E be this intersection, and call Algorithm 9 with $(T, E, \{A_1, \dots, A_k\}, \text{ and } i)$.
- (5.4) If Cases 5.1 to 5.3 do not hold, let E be the intersection of the large sets, and call Algorithm 9 with $(T, E, \{A_1, \dots, A_k\}, \text{ and } i)$.

— End of Algorithm —

CGM Implementation of Step 3 in Algorithm 7

If $n \leq 4$, in a constant number of rounds we can eliminate from the input all A_j such that $\exists A_i | A_j \equiv A_i$. In this way, we have a constant number of A_i sets and the input can fit in only one processor, where the reduction can be processed sequentially using Booth and Lueker algorithm [13].

CGM Implementation of Step 4 in Algorithm 7

If we sort A_i , $1 \leq i \leq k$, by the index, then we can compute $|A_i|$ and know which sets belong to $\mathcal{S}$ and which to $\mathcal{L}$. This requires $O(1)$ rounds.

Finding the connected components of the intersection graph of $\mathcal{A}$ is equivalent to finding the connected components of a new bipartite graph $G_b = (V_1, V_2, E)$, with $|V_1| = |\mathcal{A}|$ (that is we have one node for each reduction set in $\mathcal{A}$), $V_2 = S$, with S being the ground set of T , and $|E| = \sum_{A_i \in \mathcal{A}} |A_i|$ (we have one edge for each element in the reduction sets in $\mathcal{A}$).¹ Since $|V_1| + |V_2| + |E| = |\mathcal{A}| + \sum_{A_i \in \mathcal{A}} |A_i| + |S|$, we have that the size of G_b is as small as the input size of the current call to Algorithm 7. Moreover, G_b can be built in $O(1)$ communication rounds and in $O(\frac{n+m}{p})$ time (Section 3.2).

Analogously, we can compute the spanning forest (Section 3.2) of the intersection graph of $\mathcal{S}$, building a graph G_s similarly to G_b . The connected components algorithm runs in $O(\log p)$ rounds.

Finally, we can obtain $\cap \mathcal{L}$ by first computing and broadcasting $|\mathcal{L}|$ (sorting $A_i \in \mathcal{L}$ by index) and then sorting the elements of the large sets by value.² We mark an element as belonging to the intersection if the number of sets that contain the element is $|\mathcal{L}|$ (that is if all the large set have that elements). $\cap \mathcal{L}$ can be found in a constant number of rounds.

CGM Implementation of Step 5.1 in Algorithm 7

We can compute E_i , $1 \leq i \leq r$, by sorting the components of $\mathcal{A}$ according to the label assigned to each component by the Connected Component algorithm (Section 3.2); with MDREDUCE we can reduce T with respect to the disjoint sets $E_1, \dots, E_r$.

Since all the E_i are disjoint, we can use an algorithm similar to Algorithm 10 (as described in next section) to compute $T|E_i$. Both MDREDUCE and Algorithm 10 require $O(\log p)$ rounds, so this step can be carried out in $O(\log p)$ rounds.

¹When we use PQ-tree reduction for solving the “bipartiteness” problem, we have G as a sub graph of the input graph for that particular problem.

²Each A_i contains elements from the ground set S . We can suppose without loss of generality that each element of S has a distinct integer value. So, *sorting A_i by value* means sorting the elements in the sets according to these values.

Computation of the Set E in Step 5.2 in Algorithm 7

The set E of this case can be computed with the following CGM algorithm.

Algorithm 8 Computing the set E in Step 5.2 of Algorithm 7

Input: The set $\mathcal{S}$ and the spanning forest of its intersection graph.

Output: The set E of the Step 5.2 of Algorithm 7.

- (1) In order to find a connected component $\mathcal{C}$ in the spanning forest of $\mathcal{S}$, such that the union of its sets has cardinality at least $\frac{n}{4}$, order all the components according to the labels given by the Spanning Forest algorithm (Section 3.2).
- (2) Sort each component with respect to the values of its elements and mark only one element as “valid” for each distinct value.
- (3) Sort again with respect to the components’ labels. Compute the cardinality of the union of the elements of each component (that is the *size* of each component), with a prefix-sum computation counting only the “valid” elements (in this way we do not count the elements with the same values twice, computing in a correct way the cardinality of the union).
- (4) If a processor finds a component whose size is greater than or equal to $\frac{n}{4}$, then it broadcasts the label of this component. Otherwise it broadcasts a “not-found” message.
- (5) If every processor sends “not-found,” go to Step 5.3 of Algorithm 7. Otherwise, among all the labels received in the previous step, choose as $\mathcal{C}$ the component with the smallest label.
- (6) For each of the sets comprising $\mathcal{C}$, compute the distance in the spanning tree from the root using the Euler tour algorithm (Section 3.2).
- (7) Sort the sets according to distance, and let $B_1, \dots, B_s$ be the sorted sequence. Sort each set with respect to the values of its elements and mark only one element as “valid” for each distinct value. Sort the sets according to distance and let $\hat{i}$ be the minimum i such that $|\bigcup_{j=1}^i B_j| \geq \frac{n}{4}$ ($\hat{i}$ can be found with a prefix-sum computation on the “valid” elements). Broadcast $\hat{i}$.
- (8) All the “valid” elements in $B_1, \dots, B_{\hat{i}}$ are marked as belonging to E .

— End of Algorithm —

Each step is done in a constant number of communication rounds except for Step 6, which requires $O(\log p)$ communication rounds; consequently, this step also determines the bounds of Algorithm 8.

Computation of the Set E in Step 5.3 in Algorithm 7

We computed $\bigcap \mathcal{L}$ in Step 4, so we can easily check if the size of the intersection is $\geq \frac{n}{4}$. If this is the case, after having ordered the large sets arbitrarily, we can compute the maximum i , say $\hat{i}$, such that the intersection of the first $\hat{i}$ sets has cardinality at least $\frac{n}{4}$, with a technique similar to the one used in Step 7 of Algorithm 8 (here we perform the “valid” marking operation only on the elements belonging to the intersection). Hence Step 5.3 also requires $O(\log p)$ rounds.

The SUBREDUCE Routine

The SUBREDUCE routine, after reducing T with respect to E , D and F (these last two sets are from GLUE, described in next section), calls recursively Algorithm 7.

Algorithm 9 Subreduce

Input: A PQ-Tree T , a set E contiguous in T , a reduction set $A = \{A_1, \dots, A_k\}$ and an integer i .

- (1) Reduce T with respect to E using MDREDUCE (Section 4.3.2).
- (2) Let $(D, F) \leftarrow \text{GLUE}(E, \{A_1, \dots, A_k\})$.
- (3) If D is well-defined, reduce T with respect to D using MDREDUCE.
- (4) If F is well-defined, reduce T with respect to F using MDREDUCE.
- (5) In parallel, make the following recursive calls to Algorithm 7:
 - (5.1) Modify T by calling Algorithm 7 with $(T|E, \{A_1|E, \dots, A_k|E\}, i + 1)$.
 - (5.2) Modify T by calling Algorithm 7 with $(T/E, \{A_1/E, \dots, A_k/E\}, i + 1)$.
- (6) Return T if each reduction set A_i is consecutive in the frontier of the resulting tree T . Return the null tree otherwise.

— End of Algorithm —

CGM Implementation of $T|E$ and T/E

$A_i|E$ and A_i/E , $1 \leq i \leq k$, can be computed with sorting. Moreover, given a PQ-tree T and a contiguous set E in T , we can compute $T|E$ and T/E in the following way.

Algorithm 10 Computing $T|E$ and T/E

Input: A PQ-tree T and a set E contiguous in T .

Output: $T|E$ and T/E .

- (1) Find the leftmost and the rightmost element of E in T (say u and v).
- (2) Run the Euler tour algorithm on T and let $l(u)$ and $l(v)$ be the labels assigned to u and v respectively (u and v have unique labels because they are leaves). Mark as belonging to $T|E$ all the nodes that have at least one of the labels assigned by the Euler tour between $l(u)$ and $l(v)$.
- (3) All the other nodes belong to T/E .

— End of Algorithm —

Since we have to compute the Euler tour, Algorithm 10 runs in $O(\log p)$ rounds.

CGM Implementation of Step 6 in Algorithm 9

We assume that in Step 4 of Algorithm 7, after having computed $|A_i|$, we can substitute each element $x \in A_i$ with a pair $(x, |A_i|)$.

For checking if each set A_i is consecutive in the frontier of T , we first compute an Euler tour, to give each leaf of T (the element of its frontier) a unique label. Without loss of generality, we can assume that the values of the labels are increasing from the left to the right of the frontier. Two cases occur:

- If a processor has a sub-sequence of elements of A_i stored completely in its local memory, it can check whether this sub-sequence is exactly the entire set A_i . If not, broadcast a “failure” message.
- If a processor has a sub-sequence of elements of A_i that spans over two or more processors, it broadcasts two pairs: $(i, label_{leftmost})$ and $(i, label_{rightmost})$, where $label_{leftmost}$ and $label_{rightmost}$ are the labels of the leftmost and rightmost

elements of the sub-sequence of A_i , respectively (note that since each processor can have at most two spanning sequences, it broadcasts at most 4 pairs). After the broadcast, using the ordering of the labels, each processor can decide if there is another sub-sequence of A_i not consecutive to the one that it has. If so, it broadcasts a “failure” message.

All these can be done in $O(\log p)$ rounds.

The GLUE Routine

The GLUE routine computes two special sets, D and F , from the original reduction sets. For the properties of these two sets, refer to Klein [124].

Algorithm 11 GLUE

Input: A set E contiguous in T and a reduction set $A = \{A_1, \dots, A_k\}$.

- (1) Let $\mathcal{A}$ be the collection of sets in $\{A_1, \dots, A_k\}$ that have a nontrivial intersection with E .
- (2) If $\mathcal{A}$ is empty, return the pair $(\emptyset, \emptyset)$.
- (3) Let A_p be a set in $\mathcal{A}$ that minimizes $|A_p \cap E|$.
- (4) Let A_q be a set in $\mathcal{A}$ that minimizes $|A_q - E|$.
- (5) Let $D = A_p \cap A_q$.
- (6) Let $\mathcal{A}'$ be the collection of sets A_i in $\mathcal{A}$ such that $A_i \cap E \not\supseteq A_p \cap E$.
- (7) If $\mathcal{A}'$ is empty then return $(D, \emptyset)$.
- (8) Let A_r be a set in $\mathcal{A}'$ that minimizes $|A_r \cap E|$.
- (9) Let A_s be a set in $\mathcal{A}'$ that minimizes $|A_s - E|$.
- (10) Let $F = A_r \cap A_s$.
- (11) Return (D, F) .

— End of Algorithm —

CGM Implementation of GLUE

The GLUE procedure can be implemented on the CGM with the following algorithm. Note that in the following algorithm, we replace A_p with A_ρ since p has been used for the number of processors.

Algorithm 12 CGM implementation of GLUE

Input: E and $A = \{A_1, \dots, A_k\}$.

Output: The sets D and F .

(1) In order to compute $\mathcal{A}$, do the following:

(1.1) Compute $|E|$.

(1.2) Sort all the elements in $E, A_1, \dots, A_k$ by value. Mark all the elements that are adjacent in the sorted sequence to an element in E with the same value (we are computing $A_i \cap E$, for every i).

(1.3) Sort A_i by index and, $\forall A_i$, count the number h_i of the marked elements; if $0 < h_i < |A_i|$ and $0 < h_i < |E|$, then A_i has a nontrivial intersection with E .

(2) All the processors that recognize that an A_i has a nontrivial intersection with E broadcast an “exist” message; the others broadcast an “absent” message. If all processors send “absent”, then return the pair $(\emptyset, \emptyset)$.

(3) In Step 1.2 we computed $A_i \cap E$. Therefore, after sorting A_i in some order, we can compute in parallel $|A_i \cap E|$, $1 \leq i \leq k$. Finally, compute $A_\rho = \min_{1 \leq i \leq k} (|A_i \cap E|)$ and broadcast the index ρ of the corresponding set A_ρ .

(4) In a similar way to the previous step, compute the index q of the set that minimizes $|A_i - E|$.

(5) Sort the elements of A_ρ and A_q by value and compute $D = A_\rho \cap A_q$.

(6) For this step, we observe that

$$\begin{aligned} \mathcal{A}' &= \{A_i \in \mathcal{A}, i \neq \rho | ((A_i \cap E) \cap (A_\rho \cap E)) \text{ is nontrivial}\} \cup \\ &\quad \{A_i \in \mathcal{A}, i \neq \rho | (A_i \cap E) \cap (A_\rho \cap E) = \emptyset\}. \end{aligned}$$

Since we already have $(A_i \cap E)$ and $(A_\rho \cap E)$ (from Step 1), we can compute $\mathcal{A}'$ in a similar way to Step 1.

(7) Similarly to Step 2.

(8) Similarly to Step 3.

(9) Similarly to Step 4.

(10) Similarly to Step 5.

(11) Return.
 — End of Algorithm —

Each step of Algorithm 12 can be done in a constant number of communication rounds.

Second Phase: The Merging Phase

Klein analyzed Algorithm 7 by considering the tree R of recursive calls to it. The root of R is the initial invocation of the procedure and the internal nodes represent all the subsequent recursive calls. Let u be a node of R and $T(u)$ denote the PQ-tree to which the invocation u is applied. Let $n(u)$ denote the size of the ground set of $T(u)$. Klein showed that the descendants of u in R that are 3 levels below u are *smaller* than u by a factor of $\frac{1}{4}$, that is $n(u) \leq \frac{3n(w)}{4} + 1$, if w is such a descendant.

From these observations, we can conclude that after $l = 3 \log_{\frac{4}{3}} p$ levels of R (that is when the first part of our algorithm stops), the sizes of the ground sets associated with the nodes in R at level l is at most $\frac{n}{p}$, so each PQ-tree obtained so far can fit completely in the local memory of one processor. So, from now on we can assume that we have a certain number of PQ-trees, each of them stored entirely on one processor.

Unfortunately, we cannot use the same argument for the reduction set. In fact, it is not true that the size of reduction sets $m = \sum_{i=1}^k |A_i|$ reduces in a regular way as n does. Let u be an internal node of R , let $A_{u_1}, \dots, A_{u_j}$ be its reduction sets, and let $m_u = \sum_{i=1}^j |A_{u_i}|$. Since the sizes of the reduction sets of the children of u depend strictly on how large A_{u_i} s are and on how they intersect with E (computed by u), it is possible that A_{u_i} s are split in an unbalanced way. We can have $\sum_{i=1}^j |A_{u_i} \cap E| = O(m_u)$ and $\sum_{i=1}^j |A_{u_i} \setminus E| = O(1)$ (or vice versa). If the imbalance continues until the level $3 \log_{\frac{4}{3}} p$ of R , we can have a recursive call associated to a node v at level l such that $\sum_{i=1}^f |A_{v_i}| > \frac{m}{p}$. Therefore, while the ground set of $T(v)$, and hence $T(v)$, can fit in one processor, the reduction sets cannot, and consequently we cannot simply use Booth and Lueker [13]'s sequential algorithm to complete the reduction.

Let us consider a node v at level l in R , that has $m_u > \frac{m}{p}$. Since Klein shows that, at any level of recursion, the sum of the sizes of all reduction sets is $\leq 2m$, we

can make α_v copies of $T(v)$, with $\alpha_v = \lfloor \frac{m_v}{p} \rfloor$. In fact

$$\sum_{v \in l} \alpha_v = \sum_{v \in l} \lfloor \frac{m_v}{p} \rfloor \leq \sum_{v \in l} \frac{m_v}{p} \leq \frac{p}{m} \sum_{v \in l} m_v \leq \frac{p}{m} \cdot 2m = 2p.$$

Therefore we have in total at most $2p$ copies, or at most two copies per processor.

From now on, the reduction problem of each v node at level l of R will be solved by α_v processors that have the copies of $T(v)$. The next step is the *distribution* of the reduction sets associated to v among α_v processors. So now each of these processors can solve locally the problem of reducing $T(v)$, with respect to its reduction sets, using Booth and Lueker's algorithm (let us call $T'(v)$ these independently reduced trees).

Now we need to *merge* all the $T'(v)$. Recall that $L(T)$ is the set of permissible permutations of the leaves of T . We want to obtain a PQ-tree $\hat{T}(v)$ such that $L(\hat{T}(v)) = L(\bar{T}(v))$, where $\bar{T}(v)$ is the PQ-tree that we would have obtained by reducing $T(v)$ directly with respect to its reduction sets. $\hat{T}(v)$ can be obtained by merging $T'(v)$ in pairs in binary tree fashion, as depicted in Figure 4.3.

Algorithm 13 Merging phase

Input: h PQ-trees $T(i)$, with $|T(i)| \leq \frac{n}{p}$ and $\sum_i |T(i)| \leq n$, and their reduction sets.

Output: $T(i)$ reduced with respect to their reduction sets.

- (1) Let m_i be the sum of the sizes of the reduction sets of T_i . Make $\alpha_i = \lfloor \frac{m_i}{p} \rfloor$ copies of each $T(i)$. Distribute the reduction sets of each T_i between the processors that have the copies of $T(i)$.
- (2) Each processor runs Booth and Lueker's algorithm on the PQ-trees that it has with respect to the corresponding reduction sets. Let us call $T''(i)$ the trees obtained after these reductions.
- (3) All the α_i processors associated with $T(i)$ start the merging phase in a binary tree fashion, as depicted in Figure 4.3.

— End of Algorithm —

PQ-tree Intersection

Definition 5 E is segregated in T if $E = \text{leaves}(\text{lca}_T(E))$.

Figure 4.3: The algorithm at a glance

If E is contiguous in T , we can modify T , obtaining T' , so that E is segregated in T' and $L(T) = L(T')$. If E is not already segregated, then $v = lca_T(E)$ is a Q-node (from Definition 4). Insert an R-node z between the children $v_p, \dots, v_q$ and v . In the resulting tree T' , $z = lcat_{T'}(E)$ and $leaves(z) = \bigcup_{i=p}^q leaves(v_i) = E$. Moreover, it follows from the definition of the R-node that $L(T') = L(T)$.

A new operation on PQ-trees is defined by Klein [124]. A PQ tree $\hat{T}$ is the *intersection* of two PQ-trees T and T' over the same ground set if $L(\hat{T}) = L(T) \cap L(T')$.

We will need to intersect two PQ-trees that have the same ground set S . If $|S| \leq \frac{n}{p}$,

then the PQ-trees we want to intersect can fit on one processor.

Definition 6 For a tree T with n nodes, a node $v \in T$ with s children determines a separation of T into $s + 1$ subtrees: $T_0, T_1, \dots, T_s$, where $T_1, \dots, T_s$ are the subtrees of T rooted at the children of v , and T_0 is the subtree obtained from T by deleting $T_1, \dots, T_s$. We say v is a good separator of T if each subtree $T_0, T_1, \dots, T_s$ has no more than $\frac{n}{2}$ nodes.

Every tree with at least 2 nodes has a good separator, which can be found by computing $\text{size}(v)$ (the number of descendants of v) for each node $v \in T$. This can be easily computed with the Euler tour technique.

Let T and T' be two PQ-trees on the same ground set S and let n be the size of T . The following algorithm is based on the one proposed by Klein [124]:

Algorithm 14 Intersect

Input: Two PQ-trees T and T' .

- (1) If T has only one node, return.
- (2) Find a good separator v of T , with children $v_1, \dots, v_s$. Set $A = \text{leaves}_T(v)$, and $A_i = \text{leaves}_T(v_i)$, for $i = 1 \dots s$. Let $T_0 = T/A$ and, for $i = 1 \dots s$, let $T_i = T|A_i$.
- (3) Reduce T' first with respect to A and then with respect to the disjoint sets $A_1, \dots, A_s$. If v is a Q-node, also reduce T' with respect to the disjoint sets $A_1 \cup A_2, A_3 \cup A_4, \dots$ and with respect to the disjoint sets $A_2 \cup A_3, A_4 \cup A_5, \dots$.
- (4) Segregate T' with respect to A , and let $T'_0 = T'/A$.
- (5) For $i = 1 \dots s$, segregate T' with respect to A_i , and let $T'_i = T'|A_i$.
- (6) If we identify v with $\text{lca}_{T'}(A)$, then the corresponding subtrees T_i and T'_i have an identical ground set. Recursively intersect corresponding subtrees.

— End of Algorithm —

The correctness of Algorithm 14 follows immediately from [124]. Before computing the complexity, we introduce the following:

Lemma 6 Let T be a PQ-tree over a ground set S . For the size of T (the total number of its nodes), we have $|S| \leq |T| \leq 2|S|$ or equivalently $\frac{|T|}{2} \leq |S| \leq |T|$.

Proof. It follows trivially by observing that each node in a PQ-tree has at least 2 children. $\square$

Let us now compute the complexity of each step.

- Step 2: Finding a good separator can be done by using the Euler tour technique. Since we are assuming that the two PQ-trees to intersect can fit in the local memory of one processor, this computation can be done sequentially in $O(n)$ time and $O(1)$ communication rounds. The same bounds hold for the computation of T_0 and T_i .
- Step 3: The reduction of T' required in this step can be computed by using Booth and Lueker's algorithm [13]. Since T' has the same ground set as T , from Lemma 6 we have $|T'| \leq 2n$; therefore we can compute this step in $O(1)$ communication rounds and $O(n)$ time.
- Step 4: Similarly, segregating T' with respect to A and all the A_i can be done sequentially in $O(n)$ time. The same bounds hold for computing T'_0 and all the T'_i .

Hence, each step but the recursive one takes $O(n)$ time. Because a good separator is used in Step 2, the recursion depth is at most $\log n$. Since we can pass from one level of the recursion to the next one in $O(n)$ time, we can conclude that $O(n)$ is also the time for executing Algorithm 14.

Theorem 3 *Let T be a PQ-tree over ground set S and let T' be a copy of T . Let T^* and T'^* be the result of the reduction of T with respect to $\{A_1, \dots, A_r\}$ and of T' with respect to $\{B_1, \dots, B_t\}$ respectively. Let $\bar{T}$ be the PQ-tree obtained by reducing T with respect to $\{A_1, \dots, A_r, B_1, \dots, B_t\}$. Let λ be a permissible permutation of the leaves in $\bar{T}$. Then $\lambda \in L(\bar{T}) \Leftrightarrow \lambda \in L(T^*) \cap L(T'^*)$.*

Proof. $L(\bar{T})$ is the intersection between the sets of all the orderings that satisfy $A_1, \dots, A_r$, $B_1, \dots, B_t$, and $L(T)$. Since $L(\bar{T})$ is always the same, independent from the order in which we reduce T , if $\lambda \in L(\bar{T})$, then λ must belong to the intersection

between the sets of all the orderings that satisfy $A_1, \dots, A_r$ and $L(T)$ and must belong also to the intersection between the sets of all the orderings that satisfy $B_1, \dots, B_t$ and $L(T)$; that is $\lambda \in L(T^*)$, $\lambda \in L(T'^*)$ and $\lambda \in L(T)$. Hence λ belongs to $L(T^*) \cap L(T'^*)$.

The other side of the theorem can be proved in a similar way. $\square$

In conclusion, when the α_i processors associated to $T(i)$ finish their merging phase on the $T'(i)$ s, we know that the original T has been reduced correctly with respect to the original reduction sets $A_1, \dots, A_k$. Algorithm 6 runs in $O(\log p)$ communication rounds with $O(\frac{n}{p} \log \frac{n}{p})$ local computation time.

4.3.4 Convex Bipartite Graph Recognition

Given a bipartite graph $G = (A, B, E)$ with vertex sets $A = \{a_1, a_2, \dots, a_k\}$ and $B = \{b_1, b_2, \dots, b_n\}$, let $\mathcal{A} = \{A_1, \dots, A_k\}$ where $A_i = \{b \in B \mid (a_i, b) \in E\}$, and let T be a PQ-tree over the ground set B consisting of a root with children $b_1, b_2, \dots, b_n$. The problem of determining whether G is convex and, if it is, computing the correct ordering of the elements in B , is equivalent to the *MReduce* operation on T with respect to $\mathcal{A}$.

Theorem 4 *Let n be the size of the input bipartite graph. On a CGM with p processors and $O(\frac{n}{p})$ storage per processor, the problem of determining whether G is convex (and computing the correct ordering of the elements in B) can be solved in $O(\log^2 p)$ communication rounds with $O(\frac{n}{p})$ local computation time per round.*

Theorem 4 implies a BSP algorithm with local computation of $O(\frac{n}{p} \log^2 p)$ and communication cost of $O(g \frac{n}{p} \log^2 p)$.

We can calculate the speedup for this algorithm as follow. Let $T_s(n) = O(n)$ be the running time of the best sequential algorithm. Let $T_p(n, p) = O(\frac{n}{p} \log^2 p) + O(g \frac{n}{p} \log^2 p)$ be the running time of our algorithm. Let c , d and k be the constants associated to the different big-O notations. The speedup is therefore $s = \frac{T_s(n)}{T_p(n, p)} = \frac{\frac{dn}{p} \log^2 p + \frac{kg n}{p} \log^2 p}{\frac{p}{\log^2 p} (\frac{c}{d+kg})} = \frac{p}{g \log^2 p} (\frac{c}{k+\frac{d}{g}}) = \Omega(\frac{p}{g \log^2 p})$.

4.4 Planarity Testing

4.4.1 Problem Overview

Now we consider the problem of recognizing whether a graph G is planar and, if so, reporting a planar embedding of G .

The first sequential linear-time algorithm for planarity testing and planar embedding was presented by Hopcroft and Tarjan [103]. Booth and Lueker [13] described another linear time algorithm which is based on a previous version by Lempel, Even and Cederbaum [22] but uses PQ-trees and st -numbering. JáJá and Simon [110] showed that the problem is in NC.³ Using [13, 22], Klein and Reif [127] presented an efficient PRAM algorithm which requires $O(\log^2 n)$ time using a linear number of processors. Using open ear decomposition and st -numbering, Ramachandran and Reif [171] presented a PRAM algorithm with linear work.

In this section, we present a parallel CGM algorithm for planarity testing and planar embedding. The algorithm requires $O(\log^2 p)$ communication rounds and linear sequential work per round. It assumes that the local memory per processor, $\frac{N}{p}$, is larger than p^ϵ for some fixed $\epsilon > 0$. This assumption is true for all commercially available multiprocessors. Our result implies a BSP algorithm with $O(\log^2 p)$ supersteps, $O(g \log^2(p) \frac{N}{p})$ communication, and $O(\log^2(p) \frac{N}{p})$ local computation. Our algorithm is based on the general structure of the PRAM method in [122], using a parallel implementation of PQ-trees. However, many of the PRAM steps in [122] are very difficult on a CGM. The main contribution of this section lies in the study of these individual steps and how they can be implemented on a CGM. We present nontrivial efficient CGM implementations of the various parts of the overall strategy in [122]. Our main result is a parallel planarity testing algorithm which is much more practical and efficient on commercially available multiprocessors.

³NC is defined as the class of problems that can be solved by PRAM algorithms within $O(\log^k n)$ time using no more than $O(n^l)$ processors. Here n is the problem size, and k and l are fixed constants independent to n [108, 165].

4.4.2 Coarse-Grained Parallel Graph *st*-Numbering

Before we can proceed with our planarity testing algorithm, we require a parallel *st*-numbering method for the CGM. An *st*-numbering of a graph G with vertices $1, 2, \dots, n$ is a numbering of the vertices where the following is verified: vertices 1 and n are adjacent and each other vertices j is adjacent to two vertices i and k such that $i < j < k$. Figure 4.4 shows an example (from [160]) of a graph and an *st*-numbering.

Figure 4.4: An *st*-numbered graph

We observe that a parallel *st*-numbering method can be obtained by combining the CGM graph algorithms of [19] and the PRAM algorithm by Maon, Schieber and Vishkin [146]. We now give a brief outline of the method.

Let $G = (V, E)$ be a graph with an open ear decomposition $G = [P_0, P_1, \dots, P_l]$ such that $P_0 = (s, t)$ and the endpoints $L(P_i)$ and $R(P_i)$ of each P_i are in P_j and P_k where $i > j \geq k \geq 0$. The vertex in an ear P_i , $i \geq 1$, which is adjacent to $L(P_i)$ (respectively $R(P_i)$), is denoted by $LS(P_i)$ (respectively $RS(P_i)$). The vertex $L(P_i)$ is called the *anchor* of P_i ; if $v \in P_i$, $v \neq L(P_i)$ and $v \neq R(P_i)$, then v is called an *internal vertex*.

Algorithm 15 Computing the *st*-numbering of a graph [146]

Input: An open ear decomposition of graph $G = (V, E)$, $G = [P_0, P_1, \dots, P_l]$, where

$$P_0 = (s, t).$$

Output: A valid st -numbering of graph G .

- (1) Compute the ear tree $ET(V_T, E_T)$ where $V_T = \{P_i | P_i \text{ is an ear in } G\}$ and $E_T = \{(P_i, P_j) | L(P_i) \text{ is an internal vertex of } P_j, i \geq 1\}$.
- (2) Compute an orientation of the ears of G .
- (3) Compute the st -numbering of the ears of G .

— End of Algorithm —

To implement Step 1, we apply the CGM open ear decomposition method in [19]. The adjacency lists of graph ET is constructed using CGM integer sorting [23, 24]. Step 2 uses a *hinge tree* (see definition below). Finding the hinges reduces to l lowest common ancestor computations in ET , using [19]. Step 3 uses a *numbering tree* (see definition below) N and computation of a pre-order numbering of N . (Many of the details can be seen in [146]). The construction of N can be done in $O(p)$ rounds. The pre-order numbering can be done using the Euler Tour algorithm for computing tree functions [19].

Thus the CGM st -numbering algorithm can be done in $O(\log p)$ communication rounds and linear work per round since it is based on [19] ($O(\log p)$ rounds) and [23, 24] (constant number of rounds) and both require linear work per round.

In the following, we explain some of the definitions we have used in the algorithm.

Definition 7 Given a graph $G = (V, E)$ and its open ear decomposition $P_1, \dots, P_r$, the end points of an ear P_i are called the hinges of the ear.

Definition 8 Given a graph $G = (V, E)$ and its open ear decomposition $P_1, \dots, P_r$, Let the hinges of P_i be on P_j and P_k , with $j \leq k$. We define the hinge that is on P_j to be $L(P_i)$, and the hinge that is on P_k to be $R(P_i)$.

Definition 9 Given a graph $G = (V, E)$ and its open ear decomposition $P_1, \dots, P_r$, Let the hinges of P_i be on P_j and P_k , with $j \leq k$. We define $LS(P_i)$ to be the vertex on P_i that is adjacent to $L(P_i)$, and $RS(P_i)$ to be the vertex on P_i that is adjacent to $R(P_i)$.

Definition 10 Given a graph $G = (V, E)$ and its open ear decomposition $P_1, \dots, P_r$, the hinge tree of G is a graph $H = (V_H, E_H)$ such that $V_H = \{P_i | P_i \text{ is an ear of } G\}$ and $E_H = \{(P_i, P_m) | P_m \text{ is a hinge of } P_i\}$ (that is, one (or both) of the end points of P_i is on P_m).

[146] showed that H is a directed tree rooted at P_0 .

Definition 11 Given a graph $G = (V, E)$, the numbering tree N is defined as follows. Let v be a vertex of G . Each vertex $v \in V$ induces two vertices v_{in} and v_{out} in N . Also, v_{in} and v_{out} are both connected by an edge. The rest of the edges of N are defined by the following rules:

- Assume that v is in P_i and let u be the neighbor of v in P_i which precedes v (if such a vertex exists). Then u_{out} is connected to v_{in} .
- For each incoming ear P_i whose anchor is v the vertex v_{in} is connected to w_{in} , where $w = RS(P_i)$. For each outgoing ear P_j whose anchor is v the vertex v_{out} is connected to w_{in} , where $w = LS(P_i)$.

4.4.3 Coarse-Grained Parallel Graph Planarity Testing

In this subsection, we present the main result of this section, a coarse-grained parallel algorithm for graph planarity testing. We first give an outline of the main structure of the algorithm and then detail how to implement the main steps on a CGM within the claimed time bounds.

Planarity testing using PQ-trees is very well described in [160]. The PRAM algorithms for planarity testing of [127] are also based on the PQ-tree data structure. See the previous section for a precise definition of the PQ-tree.

Figure 4.5(a) shows a PQ-tree whose frontier is the permutation $abcde$. Figure 4.5(b) shows a reordering of the P -node giving the permutation $bacde$. If we also flip the children of the Q -node we get the permutation of Figure 4.5(c). Notice that in some permutations (for example (c)), the leaves a and e are consecutive.

We can now illustrate how st -numbering and PQ-trees can be used in planarity testing.

Figure 4.5: PQ-trees representing several permutations

Consider G to be an st -numbered graph of vertices in $V = \{1, 2, \dots, n\}$ (see example in Figure 4.4), and G_k the subgraph induced by the vertex set $V_k = \{1, \dots, k\}$. The *bush* graph B_k is formed by adding to G_k all edges, called virtual edges, with one end in V_k and the other in $V - V_k$. Such vertices in $V - V_k$ are called *virtual vertices* and are kept separate (so there may be several virtual vertices with the same label) and placed on the outer face, on a horizontal line.

B_k can be represented by a PQ-tree. A P -node represents a cut-vertex of B_k . A Q -node represents a bi-connected component of G_k . A leaf represents a virtual vertex of B_k . It can be shown [160] that if B_k is any bush graph of a planar graph G , then there exists a sequence of permutations and reversions so that all the vertices labeled $k + 1$ occupy consecutive positions on the horizontal line. Such permutations and reversions consist of the repeated application of nine transformation rules, called template matchings, to the PQ-trees [13].

Consider again graph G of Figure 4.4. Consider B_k for $k = 3$, as shown in Figure 4.6(a). Figure 4.6(b) shows the corresponding PQ-tree. This tree is transformed so that all occurrences of vertex 4 (i.e. $k + 1$) are consecutive. This is possible for each B_k if G is planar. This constitutes the main idea of the sequential planarity algorithms [13, 22] which are based on vertex addition.

The parallel PRAM algorithm [127] and the proposed CGM algorithm use a divide-and-conquer scheme, by combining or joining already embedded subgraphs into larger ones. In the following description, we use the multiple disjoint reduction operation (MDReduce) on a PQ-tree [16]. It performs the *reduce* operation for a set $\{A_1, \dots, A_k\}$ of subsets of S , where the A_i are disjoint. We also use the ROTATE(A ,

B, C) operation to make $A \cup B$ and $B \cup C$ contiguous (see Figure 4.8).

Figure 4.6: B_3 and the corresponding PQ-tree

Algorithm 16 Graph planarity testing

Input: A graph G with n vertices and $m = O(n)$ edges.

Output: Test if G is planar and, if so, report a planar embedding of G .

- (1) Compute the bi-connected components of G as described in [19].
- (2) Since a graph is planar if and only if all its bi-connected components are planar, we check each bi-connected component for planarity. Small components with less than $(\frac{n}{p})$ vertices can be stored in a single processor and can be tested by a sequential algorithm. For each larger component with $(n' \geq \frac{n}{p})$ vertices, a group of $p' = p \frac{n'}{n}$ processors is assigned to solve the problem. For the remainder, the algorithm only specifies the steps for *one* such group of processors, which handles one bi-connected component. To simplify exposition, we let n again refer to the

size of the component and p refer to the number of processors assigned to that component.

- (3) Compute the st -numbering of the bi-connected component as outlined in Section 4.4.2. Renumber all vertices according to the st -numbering with $s = 1$ and $t = n$.
- (4) Let $G^{(0)} = G$. For $i = 1 \cdots k = O(\log p)$ repeat:
 - (4.1) Select a set of joinable nodes defined as the output of Algorithm 17 and compute $G^{(i)}$ from $G^{(i-1)}$. See Algorithm 17 for details.
 - (4.2) Consider a set of joinable nodes; see Figure 4.7 for illustration and notation. Let $T(u_j)$ be the PQ -tree that is obtained from the previous iteration. From Klein and Reif [127], $B_j \neq \emptyset$ only for $j = 1$ and k . For each $j = 1 \cdots k$, if $B_j = \emptyset$, let $T'(u_j) = T(u_j)$. If $B_j \neq \emptyset$ (i.e. $j = 1$ or k), apply MDREDUCE [16] to $T(u_j)$ with respect to E_j and B_j , and then a rotation $ROTATE(E_j, B_j, F_j)$ as illustrated in Figure 4.8. Call the resulting tree $T'(u_j)$. Finally for $k = 0$, apply MDREDUCE [16] to $T(u_0)$ with respect to $E_1, \dots, E_k$ and call the resulting tree $T'(u_0)$.
 - (4.3) Join $T'(u_1), \dots, T'(u_k)$ with $T'(u_0)$. See Algorithm 18 for details.
 - (4.4) Apply MDREDUCE [16] to $T'(v)$ with respect to $out(v)$. If $B_1 \neq \emptyset$, perform $ROTATE(A, B_1, B_k \cup out(v))$. If $B_k \neq \emptyset$, perform $ROTATE(A \cup B_1, B_k, out(v))$. We call the resulting tree $T(v)$.
- (5) If any of the resulting PQ -trees is a null tree, STOP and report that G is not planar.
- (6) If $G^{(k)}$ is too large to be held in a single processor then STOP and report that G is not planar. This is because after k phases, the number of vertices has shrunk down to $O(\frac{n}{p})$ that is small enough to be held in a single processor, and we know that for a planar graph with n vertices, the number of edges cannot be larger than $3n - 6$, so if $G^{(k)}$ is planar, we would be able to send all the edges to one processor.
- (7) $G^{(k)}$ has $O(\frac{n}{p})$ vertices and $O(\frac{n}{p})$ edges. Compress $G^{(k)}$ into a single processor and apply a sequential planarity testing algorithm to $G^{(k)}$. G is planar if and

only if $G^{(k)}$ is planar. A planar embedding of G can be derived from the planar embedding of $G^{(k)}$ and the PQ-trees derived in Step 4, by a reverse processing of Steps 4.1 and 4.2.

— End of Algorithm —

Figure 4.7: A set of joinable nodes

We now discuss the details of how to implement Steps 4.1/4.3 on a CGM.

Algorithm 17 Details of Steps 4.1 in Algorithm 16.

(1) If $i \bmod 4 = 1$,

(1.1) For each node v other than s and t , define its parent $p(v)$ as follows: $p(v)$ is the highest st -numbered neighbor of v with st -number smaller than v . Using [19], compute the spanning tree $T^{(v-1)}$ of $G^{(v-1)}$ induced by the parent pointers and, for each node v , the distance from v to the root s of $T^{(v-1)}$. Identify each node as either *odd* or *even* based on its distance from s .

(1.2) Select a set of joinable nodes as follows: Contract all *even* nodes of $T^{(v-1)}$ into their *odd* parents.

(2) If $i \bmod 4 = 2$,

Figure 4.8: Illustration of the rotate operation

- (2.1) Re-use the spanning tree $T^{(v-2)}$ constructed in the previous stage (Note: We only need the even-odd labeling and parent pointers, and we assume that they were stored with the nodes and are available without re-computing the spanning tree.)
- (2.2) Select a set of joinable nodes as follows: Contract all *odd* nodes of $T^{(v-1)}$ into their *even* parents.
- (3) If $i \bmod 4 = 3$,
 - (3.1) For each node v other than s and t , define its parent $p(v)$ as follows: $p(v)$ is the lowest st -numbered neighbor of v with st -number larger than v . Using [19], compute the spanning tree $T^{(v-1)}$ of $G^{(v-1)}$ induced by the parent pointers and, for each node v , the distance from v to the root t of $T^{(v-1)}$. Identify each node as either *odd* or *even* based on its distance from t .
 - (3.2) Select a set of joinable nodes as follows: Contract all *even* nodes of $T^{(v-1)}$

into their *odd* parents.

(4) If $i \bmod 4 = 0$,

(4.1) Re-use the spanning tree $T^{(v-2)}$ constructed in the previous stage (Note: We only need the even-odd labeling and parent pointers, and we assume that they were stored with the nodes and are available without re-computing the spanning tree.)

(4.2) Select a set of joinable nodes as follows: Contract all *odd* nodes of $T^{(v-1)}$ into their *even* parents.

— End of Algorithm —

We now discuss the details of how to implement Step 4.2 on a CGM. Note that a contiguous set E of nodes in a PQ -tree T is *segregated* if $E = \text{leaves}(\text{lca}_T(E))$.

Algorithm 18 Details of Step 4.3 in Algorithm 16.

Input: A set of $k + 1$ PQ -trees $T_0, T_1, \dots, T_k$ with ground sets $S_0, S_1, \dots, S_k$, respectively, where $S_1, \dots, S_k$ are disjoint. Assume for $j = 1 \dots k$ that $E_j = S_0 \cap S_j$ is non-empty and contiguous in both T_0 and T_j , and that $D_j = S_j - E_j$ is also non-empty and contiguous in T_j .

Output: A PQ -tree representing the join of T_0 with $T_1, \dots, T_k$.

- (1) For each PQ -tree $T_1, \dots, T_k$, segregate T_0 and T_j with respect to E_j [16]. If $|E_j| = 1$, let $\hat{T}_j = T_j$ and skip to Step 4.
- (2) For each PQ -tree $T_1, \dots, T_k$, find two elements a_j and b_j of E_j that lie in different blocks of T_0 's partition of E_j and different blocks of T_j 's partition of E_j . This can be done by applying the prefix sum algorithm. We assume here that a_j precedes b_j in the frontier, otherwise swap a_j and b_j .
- (3) If b_j precedes a_j in the frontier of T_j then let $\hat{T}_j = T_j$. Otherwise, $\hat{T}_j$ is computed from T_j by flipping every node in T_j .
- (4) Segregate D_j in $\hat{T}_j$. Replace the subtree $T_0|E_j$ of T_0 by $\hat{T}_j|D_j$, letting z be the root of $\hat{T}_j|D_j$ in the resulting tree. If E_j is rigid in both T_0 and T_j , then rename z to be an R -node. If E_j is hinged in T_0 but z is an R -node, rename it to be a Q -node.

(5) Output the resulting PQ -tree.

— End of Algorithm —

Theorem 5 *Let G be a graph of size n . On a CGM with p processors and $O(\frac{n}{p})$ storage per processor, the problem of determining whether G is planar can be solved in $O(\log^2 p)$ communication rounds with $O(\frac{N}{p})$ local computation per round.*

Theorem 5 implies a BSP algorithm with local computation of $O(\frac{n}{p} \log^2 p)$ and communication cost of $O(g \frac{n}{p} \log^2 p)$.

We can calculate the speedup for this algorithm as follow. Let $T_s(n) = O(n)$ be the running time of the best sequential algorithm. Let $T_p(n, p) = O(\frac{n}{p} \log^2 p) + O(g \frac{n}{p} \log^2 p)$ be the running time of our algorithm. Let c, d and k be the constants associated to the different big-O notations. The speedup is therefore $s = \frac{T_s(n)}{T_p(n, p)} = \frac{\frac{cn}{p} \log^2 p + \frac{kgn}{p} \log^2 p}{\log^2 p} = \frac{p}{\log^2 p} (\frac{c}{d+kg}) = \frac{p}{g \log^2 p} (\frac{c}{k+\frac{d}{g}}) = \Omega(\frac{p}{g \log^2 p})$.

4.5 Summary of Results

The following table summarizes the result we got in this chapter. Note that n is the input size, p is the number of processors available.

Algorithm	Local Computation	Communication Rounds	Speedup
Bipartite Graph Recognition	$O(\frac{n}{p} \log p)$	$O(\log p)$	$\Omega(\frac{p}{g \log p})$
Convex Bipartite Graph Recognition	$O(\frac{n}{p} \log^2 p)$	$O(\log^2 p)$	$\Omega(\frac{p}{g \log^2 p})$
Planar Graph Recognition	$O(\frac{n}{p} \log^2 p)$	$O(\log^2 p)$	$\Omega(\frac{p}{g \log^2 p})$

Chapter 5

Maximum Matching on Graphs

5.1 Problem Overview

In this chapter, we study the problem of computing the maximum matching in specific graphs.

We will discuss the algorithms for finding the maximum matching for the following:

- Convex Bipartite Graphs
- (Free) Trees

The problem of finding a maximum matching in a bipartite graph or a convex bipartite graph is a classic and well studied problem [32, 60, 104, 140]. The case of convex bipartite graphs has several interesting applications as outlined in [60, 140]. A particularly interesting industrial application for matching parts with products was described in [140].

Recall that the vertices of a bipartite graph or a convex bipartite graph can be divided into two disjoint groups. Here we assume that the size of these two groups are n_A and n_B , respectively.

Sequential solutions for maximum matching in bipartite graphs and convex bipartite graphs with time complexities $O(n_A^{\frac{5}{2}})$ and $O(n_A + \alpha(n_B))$ were described in [104] and [140], respectively, where $\alpha(n_B)$ is a very slowly growing function related to the

inverse Ackermann function. A PRAM algorithm for maximum matching in convex bipartite graphs requiring $O(\log^2 n_A)$ time and $\frac{n_A}{2}$ processors was presented in [60].

Pawagi [166, 167] has presented a PRAM algorithm for finding a maximum weight matching in a tree of size n . His algorithm requires $O(\log^2 n)$ computation time and $O(n)$ processors.

5.2 Max. Matching in Convex Bipartite Graphs

In this section, we consider the maximum matching problem for convex bipartite graphs using the CGM/BSP models.

Before proceeding we note that finding a maximum matching in a convex bipartite graph can be transformed into the following problem:

Definition 12 *Given a set of intervals $I = \{I_1 \cdots I_{n_A}\}$, each of which represents an integer range, i.e. $I_i = (l_i, r_i)$, a maximum interval assignment consists of assigning, for a maximum number of intervals, one integer label to each interval such that the integer for each interval is within the interval's range and no two intervals are assigned the same integer.*

For a given convex bipartite graph $G = (A, B, E)$, let $I(G)$ be the set of $|A|$ intervals containing for each $a \in A$ an interval $I_i = (l_i, r_i)$ where l_i and r_i are the smallest and largest ranks, in B , of all element $b \in B$ with $(a, b) \in E$.

Observation 1 *The maximum matching problem for a convex bipartite graph G can be reduced to finding a maximum interval assignment in $I(G)$.*

Proof. In the matching M , every vertex $u_i \in A$ can be incident to at most one edge which leads to a vertex in B . Finding such an edge is equivalent to finding an integer within the range of the interval associated to I_i . $\square$

5.2.1 Algorithm Description

In Section 5.2.2, we first present an algorithm that solves a special case: all intervals start at the same left end point. This algorithm will be used in our solution for the general case which is presented in Section 5.2.3.

5.2.2 Special Case: All Intervals Start at the Same Point

Algorithm 19 All intervals have the same left end point $l_i = l, i = 1 \cdots n_A$.

Input: A set of n_A intervals $I = \{I_1 \cdots I_{n_A}\}$ with their associated left and right end points $I_i = (l_i, r_i), i = 1 \cdots n_A$, distributed over a p processor CGM with $\frac{n_A}{p}$ intervals per processor. Note that we assume $\frac{n_A}{p} \geq p$.

Output: A maximum interval assignment of I .

- (1) All intervals are sorted by their right end points using CGM parallel integer sort [23, 24, 85].
- (2) Each processor $P_i, i = 1, \cdots, p - 1$, determines the largest right end point, e_i , received in Step 1 and sends it to the next processor P_{i+1} .
- (3) Each processor $P_i, i = 2, \cdots, p$, sets $s_i = e_{i-1} + 1$. The first processor sets $s_1 = l$. We call (s_i, e_i) the controlled range of P_i . This is the range of integers that the processor can use to label intervals.
- (4) Each processor $P_i, i = 1, \cdots, p$, temporarily changes the left end points of its intervals to s_i and then solves the modified problem locally using a sequential algorithm (essentially sequential integer sort).
- (5) Each processor $P_i, i = 1, \cdots, p$, calculates how many labels in the controlled range are left unused (a_i), and how many intervals have not yet received an integer label (b_i). These two numbers (per processor) are broadcast to all processors (in one h -Relation). This is possible since $\frac{n}{p} \geq p$.
- (6) Based on the data received in the previous step, each processor can now calculate where to request labels from and where to send unused labels to. See Algorithm 20 for details.
- (7) Each processor selects unused labels and sends them to the processors that need them.

- (8) Upon receiving the needed labels, each processor can now assign them to the respective intervals.

— End of Algorithm —

Algorithm 20 Sequential computation for Step 6 in Algorithm 1, executed by processor P_k , $1 \leq k \leq p$.

Input: An array $a = (a_1, \dots, a_p)$ representing the numbers of free (unused) labels in the p processors. An array $b = (b_1, \dots, b_p)$ representing the number of intervals with no labels yet in each of the p processors.

Output: An array $get = (get_1, \dots, get_p)$ representing numbers of labels that are to be sent to processor P_k by the other processors. An array $give = (give_1, \dots, give_p)$ representing numbers of labels that are to be received from processor P_k by the other processors.

```

for ( $i \leftarrow 1$ ;  $i \leq p$ ;  $i++$ ) do
     $get_i \leftarrow 0$ ;
     $give_i \leftarrow 0$ ;
end for
 $i \leftarrow 1$ ;
 $j \leftarrow 2$ ;
while ( $(i \leq p) \& (j \leq p)$ ) do
    if ( $i = j$ ) then
         $j++$ ;
    end if
    while ( $(a_i \neq 0) \& (j \leq p)$ ) do
        if ( $a_i \geq b_j$ ) then
            if ( $P_k = i$ ) then
                 $get_j \leftarrow b_j$ ;
            end if
            if ( $P_k = j$ ) then
                 $get_i \leftarrow b_j$ ;
            end if
             $a_i \leftarrow a_i - b_j$ ;
        
```

```

     $b_j \leftarrow 0;$ 
     $j++;$ 
  else
    if ( $P_k = i$ ) then
       $get_j \leftarrow a_j;$ 
    end if
    if ( $P_k = j$ ) then
       $get_i \leftarrow a_j;$ 
    end if
     $b_j \leftarrow b_j - a_i;$ 
     $a_i \leftarrow 0;$ 
  end if
end while
 $i++;$ 
end while

```

— End of Algorithm —

Lemma 7 *The sequential time complexity of Algorithm 20 is $O(p)$.*

Proof. The variable i counts from 1 to $p-1$, and j counts from 2 to p . Since both variables are incremented independently, the total time complexity of the algorithm is $O(p)$. $\square$

Lemma 8 *The integer label assignment produced by Algorithm 19 is maximum.*

Proof. It is easy to see that the label assignment based on the order of the right end points is maximum. Let A_o be such an assignment. Let A_1 be the label assignment produced by Algorithm 19. Since A_o is maximum, $|A_o| \geq |A_1|$. We compare assignments A_1 and A_o . We first remove the intervals that are not assigned labels in both assignments. If the remaining intervals all have labels assigned in both assignments, then we have $|A_o| = |A_1|$ and the lemma follows.

If the k th interval (l'_k, r'_k) (with respect to the right end points) is assigned a label in A_o but not in A_1 , then the first $k - 1$ intervals must be assigned labels

$l, l+1, \dots, l+k-2$ in A_o and the k th interval must be assigned label $l+k-1$. This also means $r'_k \geq l+k-1$.

Now consider A_1 . If the first $k-1$ intervals are also assigned labels $l, l+1, \dots, l+k-2$, then label $l+k-1$ will be available for the k th interval. If some of the first $k-1$ intervals are not assigned labels in the range $l, l+1, \dots, l+k-2$, then there will be a “hole” in this range that can be used for the k th interval.

Hence, for every interval with a label assigned in A_o , there is a label assigned to that interval in A_1 . Therefore, $|A_1| \geq |A_o|$. Thus, the label assignment obtained from Algorithm 19 is maximum. $\square$

Theorem 6 *Algorithm 19 finds a maximum label assignment for n_A intervals with the same left endpoints stored on a p processor CGM with $\frac{n_A}{p}$ local memory per processor, $\frac{n_A}{p} \geq p$, in $O(1)$ communication rounds with $O(\frac{n_A}{p})$ local computation.*

Proof. The correctness of the algorithm is shown in Lemma 8. The communication rounds required in each step of Algorithm 19 is $O(1)$. Hence the total number of communication rounds needed is $O(1)$. In each round, the total message size per processor is $O(\frac{n_A}{p})$. Step 1 uses CGM integer sort [23, 24, 85] which requires $O(1)$ communication rounds and $O(\frac{n_A}{p})$ local computation. The local computation of Step 6 is $O(p)$ (from Lemma 7). For all other steps, it is either $O(1)$ or $O(\frac{n_A}{p})$. Therefore, the total local computation per processor is $O(\frac{n_A}{p}) + O(p) = O(\frac{n_A}{p})$ since $\frac{n_A}{p} \geq p$. $\square$

5.2.3 The General Case

In order to solve the maximum interval assignment problem for arbitrary intervals, we now combine Algorithm 19 with a merge/unmerge scheme presented in [60].

Algorithm 21 General case where the intervals can have different left end points.

Input: A set of n intervals $I = \{I_1, \dots, I_{n_A}\}$ with their associated left and right end points $I_i = (l_i, r_i), i = 1 \dots n_A$, distributed over a p processor CGM with $\frac{n_A}{p}$ intervals per processor. Note that we assume that $\frac{n_A}{p} \geq p$.

Output: A maximum interval assignment of I .

- (1) All intervals are sorted by their left end points using CGM parallel integer sort [23, 24, 85]. (In case of a tie, intervals are compared by their right end points.)
- (2) Intervals with the same left end points are combined into groups. All groups that are stored completely within a processor are merged into a single group. Let γ be the number of groups. Note that γ is at most $2p + 1$.
- (3) Each group is assigned a *controlled range* (l_i, r_i) , $i = 1 \cdots \gamma$, where l_i is equal to the smallest left end point of that group and $r_i = l_{i+1} - 1$, $i = 1 \cdots \gamma - 1$. Let r_γ be the largest right end point of the intervals
- (4) Using a sequential algorithm, each processor solves the problem for interval groups that are completely within the processor. Remove the label of all those intervals that received a label outside their group's controlled range. Classify the intervals using one of the following three types: matchable (M) for all labelled intervals; to-be-determined (T) for all non-labelled intervals that extend beyond the rightmost label given by that processor; and unmatchable (U) for all remaining intervals. Remove all unmatchable intervals.
- (5) Using Algorithm 19, solve the problem for interval groups that cross a processor boundary (for each such group in isolation). Remove the label of all intervals that received a label outside their group's controlled range. Classify the intervals using one of the following three types: matchable (M) for all labelled intervals; to-be-determined (T) for all non-labelled intervals that extend beyond the rightmost label given by that processor; and unmatchable (U) for all remaining intervals. Remove all unmatchable intervals.
- (6) Merge the intervals in adjacent groups. Let M_L, T_L be the intervals from the left group, let M_R, T_R be the intervals from the right group, and let (l_L, r_L) and (l_R, r_R) be the controlled ranges of the left and right groups, respectively. Using Algorithm 19, solve the problem for $T_L \cup M_R$ over the range (l_R, r_R) . Let the resulting sets of matchable and to-be-determined intervals be M_n and T_n . For the combined group, set $M = M_L \cup M_n$ and $T = T_n \cup T_R$. The controlled range of the combined group is (l_L, r_R) .

- (7) Repeat Step 6 $O(\log \gamma)$ times until a single group is obtained.
- (8) Using a reverse process of the previous steps, using $O(\log \gamma)$ iterations, redistribute the intervals back into the γ groups obtained at the end of Step 3. In each splitting phase, let M be the group we are splitting and let M_L and M_R be the two corresponding components. Let V be the set of intervals that have a left end point smaller than the starting value of the controlled range of M . Note that V can be empty. Let W be the union of V and M_L . W should be distributed to M_L and the rest to M_R . However, the controlled range of M_L may not allow the entire W to be assigned to M_L . Hence, we assign only the first $\min(|W|, |R - L|)$ intervals to M_L and the remainder to M_R .
- (9) Step 8 is repeated until γ groups are obtained.
- (10) Using a sequential algorithm, each processor solves the problem for its interval groups over each group's controlled range. For groups that span over more than one processor, adjust the left end points to the starting values of the controlled range and then apply Algorithm 19.

— End of Algorithm —

Theorem 7 *Algorithm 21 solves the label assignment problem in $O(\log p)$ communication rounds and $O(T_s(\frac{n_A}{p}, \frac{n_B}{p}) + \frac{n_A}{p} \log p)$ local computation where $T_s(n_A, n_B)$ is the running time of the best sequential algorithm available for the same problem of size n .*

Proof. The correctness of the split/merge scheme follows from [60]. Step 6 is executed $O(\log \gamma)$ times. In Step 6, we invoke Algorithm 19 which requires $O(1)$ communication rounds. Since $\gamma \leq 2p + 1$, the total number of communication rounds is $O(\log p)$. The local computation time is dominated by the $O(1)$ executions of the sequential algorithm on each processor and the linear local time in each of the $O(\log \gamma)$ iterations. $\square$

Theorem 7 implies a BSP algorithm with local computation of $O(T_s(\frac{n_A}{p}, \frac{n_B}{p}) + \frac{n_A}{p} \log p)$ and communication cost of $O(g \frac{n_A}{p} \log p)$.

We can calculate the speedup for this algorithm as follow. Let $T_s(n_A, n_B) = O(n_A + \alpha(n_B))$ be the running time of the best sequential algorithm [140], and let

$T_p(n_A, n_B, p)$ be the running time of our algorithm. Since α is a very slowly growing function, for practical purpose, we can reduce T_s to $T_s(n_A, n_B) = O(n_A)$ and T_p to $T_p(n_A, n_B, p) = O(\frac{n_A}{p} + \frac{n_A}{p} \log p) + O(g \frac{n_A}{p} \log p)$. Let c , d and k be the constants associated to the different big-O notations. The speedup is therefore $s = \frac{T_s(n_A, n_B)}{T_p(n_A, n_B, p)} = \frac{cn_A}{d(\frac{n_A}{p} + \frac{n_A}{p} \log p) + kg \frac{n_A}{p} \log p} = \frac{p}{g \log p} (\frac{c}{\frac{d}{g \log p} + \frac{d}{g} + k}) = \Omega(\frac{p}{g \log p})$.

This theorem implies an algorithm for computing the maximum matching of a convex bipartite graph with the same number of communication rounds and local computation.

5.3 Maximum Weighted Tree Matching

In this section we consider the problem of finding a maximum weight matching in a tree. That is, given a tree $T = (V, E)$ and a weight $w_i \geq 0$ for each edge $e_i \in E$, we want to find a matching $M \subset E$ of T such that the total weight $w = \sum_{e_i \in M} w_i$ is maximized.

As mentioned before, Pawagi [166, 167] presented a PRAM algorithm for maximum weight tree matching. For a tree of size n , his algorithm requires $O(\log^2 n)$ computation time and $O(n)$ processors.

We will present a parallel CGM algorithm for maximum weight tree matching. The algorithm requires $O(\log^2 p)$ communication rounds and with $O(\frac{n}{p} \log p)$ local computation and $O(\frac{n}{p})$ memory per processor. It assumes that the local memory per processor, $\frac{n}{p}$, is larger than p^ϵ for some fixed $\epsilon > 0$. This assumption is true for all commercially available multiprocessors. Our results imply a BSP algorithm with $O(\log^2 p)$ supersteps, $O(g \log(p) \frac{n}{p})$ communication time, and $O(\log(p) \frac{n}{p})$ local computation time.

While our algorithm follows the general stucture of Pawagi's PRAM algorithm [166, 167], the implementation details are very different. For example, Pawagi's algorithm includes steps such as finding the centroid of a tree, re-rooting a tree to a given node, and finding a maximum gain alternating path of a tree. These steps are straight forward for the PRAM but nontrivial for the CGM. The main contribution of our work lies in the study of these individual steps and how they can be implemented

on a CGM. Our main result is a parallel maximum weight tree matching algorithm which is suitable to be executed on PC clusters. Our CGM algorithms for finding the centroid of a tree, re-rooting a tree and finding a maximum gain alternating path of a tree are also interesting in their own right, and probably useful for other coarse grained parallel graph algorithms.

5.3.1 Partitioning a Tree via its Centroid

We are using divide-and-conquer to solve the maximum weight matching problem. In order for divide-and-conquer to work efficiently, we partition the tree such that the subproblems are guaranteed to shrink by at least a constant factor.

In this section, we present an algorithm to do such partitioning efficiently.

Definition 13 *The centroid of a tree T is a vertex c that minimizes the size of the largest subtree in the forest generated by deletion of c from T .*

Note that for an n -vertex tree, the size of the largest subtree in the forest generated by the deletion of the centroid is less than or equal to $\frac{n}{2}$ [166, 167].

Algorithm 22 Centroid of a tree

Input: A tree T with n vertices and $n - 1$ edges, rooted at r . All nodes and edges are evenly distributed over a p -processor coarse grained multi-computer.

Output: A vertex c which is the centroid of T .

- (1) Compute an Euler tour of T [19]
- (2) Starting with the root r , using list ranking [19], calculate the distance of each vertex along the Euler tour. From these distances, we can easily calculate the size of the subtrees for each vertex. Among these subtrees, the subtree with maximum size for the vertex is selected.
- (3) Each processor determines which vertex has the smallest maximum subtree. The selected vertex is sent to Processor P_1 .
- (4) Processor P_1 selects and outputs the vertex that has the smallest maximum subtree.

— End of Algorithm —

Theorem 8 *Algorithm 22 calculates the centroid of a tree using $O(\log p)$ communication rounds, $O(\frac{n}{p} \log p)$ local computation, and $O(\frac{n}{p})$ storage per processor.*

Proof. The correctness of the algorithm follows immediately from the definition of the centroid. All steps are within the stated bounds. $\square$

5.3.2 Re-rooting a Tree

In this section, we describe a CGM algorithm to re-root a tree.

Algorithm 23 Re-rooting a tree

Input: A tree T with n vertices and $n - 1$ edges, rooted at r . Each vertex v in T has a pointer $p(v)$ to its parent. $p(r) = \emptyset$. A vertex $u \neq r$ in T (the new root).

Output: A tree T' that is topologically equivalent to T but rooted at u .

- (1) Find an Euler tour E of T . [19]
- (2) Perform list ranking along E [19]. Each edge e in T will receive two ranks, r_{e_1} and r_{e_2} . Assume $r_{e_1} < r_{e_2}$.
- (3) Let the ranks for the edge $(v, p(v))$ be r_1 and r_2 . Broadcast these two ranks to all processors.
- (4) Each processor for each edge e , compares r_{e_1}, r_{e_2} with r_1 and r_2 . Mark the edge e if $r_{e_1} \leq r_1$ and $r_{e_2} \geq r_2$. Note that all marked edges form a path from u to r .
- (5) Each processor for each vertex v checks if the edge $(v, p(v))$ is marked. If so, send v to $p(v)$.
- (6) Every vertex v' that received a new vertex v'' sets its parent $p(v) = v''$.
- (7) The vertex u sets its parent to $p(u) = \emptyset$.

— End of Algorithm —

Theorem 9 *Using a p processor CGM, Algorithm 23 re-roots an n -node tree in $O(\log p)$ communication rounds with $O(\frac{n}{p} \log p)$ local computation and $O(\frac{n}{p})$ memory per processor.*

Proof. The correctness of the algorithm lies on the properties of the Euler tour. As shown in Figure 5.1, the edges can be divided into three zones. The edges on the “left” of the path $(u, \dots, r)$ (zone 1 in Figure 5.1) will have $r_{e_1} \leq r_{e_2} \leq r_1$. The edges on the “right” of the path $(u, \dots, r)$ (zone 2 in Figure 5.1) will have $r_2 \leq r_{e_1} \leq r_{e_2}$. The edges “below” the path $(u, \dots, r)$ (zone 3 in Figure 5.1) will have $r_1 \leq r_{e_1} \leq r_{e_2} \leq r_2$. The edge on the path $(u, \dots, r)$ (and only the edges on the path) will have $r_{e_1} \leq r_1$ and $r_{e_2} \geq r_2$. Thus, Step 4 will mark all the edges on the path $(u, \dots, r)$, and Step 6 will reverse all the parent pointers along the path. Therefore, Algorithm 23 can re-root a tree.

Figure 5.1: Re-rooting a tree

Step 1 requires $O(1)$ communication rounds, $O(\frac{n}{p})$ local computation and local storage. Step 2 requires $O(\log p)$ communication rounds, $O(\frac{n}{p} \log p)$ local computation and $O(\frac{n}{p})$ local storage. Step 3 requires $O(1)$ communication rounds. Step 4 and 6 require no communication, $O(\frac{n}{p})$ local computation and local storage. Step 5 requires $O(1)$ communication rounds. Step 7 requires no communication and $O(1)$ local computation. $\square$

5.3.3 Maximum Gain Alternating Path

We start this section with some definitions and an observation.

Definition 14 Given a graph $G = (V, E)$ and a matching $M \subseteq E$, a node $v \in V$ is free if and only if $\forall u \in V, (u, v) \notin E$ or $(u, v) \notin M$.

Definition 15 A path $P = (v_{i_1}, v_{i_2}, \dots, v_{i_n} = v)$ in T is said to be an alternating path with respect to a matching M if every odd numbered edge on this path is in M and every even numbered edge is not in M or vice versa.

Definition 16 The gain of an alternating path P , denoted by $G(P)$, is the difference between the sum of the weights of the edges of P that are not in M and the sum of the weights of the edges of P that are in M .

An alternating path with positive gain can be used to obtain a new matching by removing from M all edges that are in P and adding the unmatched edges in P to M . Observe that the resulting matching is at least as large as the original one.

Algorithm 24 Maximum gain proper alternating path

Input: A weighted tree T rooted at r , a key k , and a maximum matching M in T .

Output: A vertex u in T such that the path from r to u forms a proper alternating path with maximum gain in T . All edges along the path $(r, \dots, u)$ are marked with the key k .

- (1) Let $P(v)$ be the parent of v , and w_v be the weight of the edge $(v, P(v))$. For each node v in the tree calculate $g(v)$ defined as follows:
 - if v is the root r , then $g(r) = 0$.
 - if $(v, P(v)) \in M$, then $g(v) = -w_v$.
 - if $(v, P(v)) \notin M$, then $g(v) = w_v$.
- (2) Except for the root r , each node v checks the values of $g(v)$ and $g(P(v))$. If both values are positive, delete the edge $(v, P(v))$.
- (3) Using list ranking [19], each node v determines its distance to the root r as well as the sum of $g(v)$ along the path $(r, \dots, g(v))$. (See proof of Theorem 10 for detail.) Let that sum be $G(v)$. Remove all nodes that are not able to reach the root r .
- (4) For each remaining node v , if v is not a leaf and $(v, P(v)) \notin M$, remove v .

- (5) Perform a partial sum with respect to $G(v)$ (using the maximum operator) on the nodes that are left from Step 4, and determine the node u that has the maximum gain.
- (6) Similar to Step (1) to (4) of Algorithm 23, mark all edges along the path $(r, \dots, u)$ with the supplied key k .

— End of Algorithm —

In Algorithm 24, if r is not free, then the edge $(r, u) \in M$ must be in the maximum alternating path. This is because M is maximum. An alternating path with positive gain can be used to obtain a new matching by removing from M all edges that are in P and adding the unmatched edges in P to M . Observe that the resulting matching is larger than the original one.

Theorem 10 *Algorithm 24 computes the maximum gain proper alternating path in $O(\log p)$ communication rounds with $O(\frac{n}{p} \log p)$ local computation and $O(\frac{n}{p})$ memory per processor.*

Proof. The correctness of the algorithm relies on whether we can find the correct gain $G(v)$ for each node v in Step 3. This can be achieved by using the Euler tour of the tree and then performing list ranking. More precisely, build an Euler tour by defining for each tree edge e_i between a node v_i and its parent $P(v_i)$ two directed edges $e_{i_1} = (P(v_i), v_i)$ and $e_{i_2} = (v_i, P(v_i))$. Assign $g(v_i)$ to e_{i_1} and $-g(v_i)$ to e_{i_2} . See Figure 5.2 for an illustration.

A partial sum along the Euler tour, using [19], will give the gain for each node. Thus, in Step 5, the resulting path will be the maximum gain alternating path.

Steps 1, 2 and 4 can be performed using $O(1)$ communication rounds, $O(\frac{n}{p})$ computation and storage. Step 3 can be performed using $O(\log p)$ communication rounds, $O(\frac{n}{p} \log p)$ computation and $O(\frac{n}{p})$ storage. Steps 5 and 6 can be executed using $O(\log p)$ communication rounds, $O(\frac{n}{p})$ computation and storage. $\square$

5.3.4 Maximum Weight Matching for Trees

Algorithm 25 Maximum weight tree matching

Figure 5.2: Gain computation through Euler tour list ranking

Input: A tree T with n vertices and $n - 1$ edges evenly distributed over a p -processor coarse grained multi-computer.

Output: The same tree T with all matched edges marked. All edges along the maximum gain path are marked. The maximum gain value and a boolean representing whether the root is free or not.

- (1) Mark all vertices as free. If $p = 1$, solve the problem sequentially and return.
- (2) Compute the centroid c of T using Algorithm 22. Re-root T at c using Algorithm 23. Partition T into subtrees $T_1, T_2, \dots, T_k$, where k is the total in-degree of c . Each subtree T_i is assigned a unique key k_i .
- (3) For each subtree T_i , let n_i be the size of the subtree. Assign $\lceil \frac{n_i}{n} p \rceil$ processors to recursively solve the problem on the subtree.
- (4) Let e_i be the edge connecting c and T_i , and let its weight be w_i . Compute the value $q_i \leftarrow \begin{cases} w_i & \text{if root of } T_i \text{ is free} \\ w_i + \text{max. gain in } T_i & \text{otherwise} \end{cases}$
- (5) Select a j such that q_j is the maximum.
- (6) If $q_j \leq 0$ then mark c as free and go to Step 9.

- (7) If v_j is free then mark (c, v_j) as matched, mark c as not-free and go to Step 9.
- (8) Mark the edge (c, v_j) as matched. Invert the matching for all edges with key k_j . That is, for each edge e with key k_j , mark the edge matched if it was not matched originally and mark it unmatched if it was matched originally. Mark c as not-free.
- (9) Re-root T to r using Algorithm 23.

— End of Algorithm —

Theorem 11 *Algorithm 25 computes the maximum weight matching for the tree T using $O(\log^2 p)$ communication rounds, with $O(\frac{n}{p} \log p)$ local computation and $O(\frac{n}{p})$ memory per processor.*

Proof. The main step is Step 4 where we select an edge e_i such that adding e_i to the matching will result in a maximum increase in the total weight in the matching. Note that in Step 3, the matching in each subtree T_i has been maximized. Therefore, if the root of T_i is free, then by adding e_i to the matching, the total increase will be w_i . If the root of T_i is not free then, by adding e_i to the matching and flipping the edges along the maximum gain alternating path, the total increase of the weight will be w_i plus the maximum gain in T_i . At the end of the algorithm, the resulting matching will always be maximized. Therefore, Algorithm 25 is correct.

Step 2 will guarantee that the size of the maximum subtree will be at most one half of the original tree. Thus, after at most $O(\log p)$ recursions, each subtree will fit into one processor and we can apply the sequential algorithm to solve the subproblem. Step 1 will be executed once for each subtree, which will take no communication and use $O(\frac{n}{p})$ local computation and storage. Steps 2, 4 and 9 can be done using $O(\log p)$ communication rounds, $O(\frac{n}{p} \log p)$ computation and $O(\frac{n}{p})$ local storage. Steps 5 and 8 can be done using $O(1)$ communication rounds, $O(\frac{n}{p})$ computation and local storage. Steps 6 and 7 can be done without communication and with $O(\frac{n}{p})$ computation and local storage. Therefore, the total bounds for each recursion will be $O(\log p)$ communication rounds, $O(\frac{n}{p} \log p)$ computation and $O(\frac{n}{p})$ storage. Since the size of each subproblem is guaranteed to be at most half the size of the original problem, the theorem follows. $\square$

Theorem 11 implies a BSP algorithm with local computation of $O(\frac{n}{p} \log p)$ and communication cost of $O(g \frac{n}{p} \log^2 p)$.

We can calculate the speedup for this algorithm as follow. Let $T_s(n) = O(n)$ be the running time of the best sequential algorithm. Let $T_p(n, p) = O(\frac{n}{p} \log p) + O(g \frac{n}{p} \log^2 p)$ be the running time of our algorithm. Let c, d and k be the constants associated to the different big-O notations. The speedup is therefore $s = \frac{T_s(n)}{T_p(n, p)} = \frac{cn}{\frac{dn}{p} \log p + \frac{kgn}{p} \log^2 p} = \frac{p}{g \log^2 p} (\frac{c}{\frac{d}{g \log p} + k}) = \Omega(\frac{p}{g \log^2 p})$.

5.4 Summary of Results

The following table summarizes the result we got in this chapter. Note that n_A and n_B are the input sizes for the two groups in the convex bipartite graph matching problem, n is the input size in the tree matching problem. Also p is the number of processors available. Finally $T_s(n_A, n_B)$ is the performance of the best sequential convex bipartite graph matching algorithm available.

Algorithm	Local Computation	Communication Rounds	Speedup
Maximum Matching in Convex Bipartite Graphs	$O(T_s(\frac{n_A}{p}, \frac{n_B}{p}) + \frac{n_A}{p} \log p)$	$O(\log p)$	$\Omega(\frac{p}{g \log p})$
Maximum Matching in Trees	$O(\frac{n}{p} \log p)$	$O(\log^2 p)$	$\Omega(\frac{p}{g \log^2 p})$

Chapter 6

Next Element Search

6.1 Problem Overview

The next element search problem is a well known problem in computational geometry and has many applications [1]. Given a set of n non-intersecting line segments $s_1, \dots, s_n$ and a direction D_{next} (without loss of generality we can assume that D_{next} is the direction of the positive Y -axis), the next element search problem consists of finding for each query point q_i of a set of m query points $q_1, \dots, q_m$ the line segment s_{j_i} first intersected by the ray starting at q_i in direction D_{next} ; see Figure 6.1. A sequential solution requires $O(n \log n)$ time and $O(n)$ space for pre-processing the segments as well as $O(\log n)$ time for each query. Therefore the total query time excluding pre-processing is $O(m \log n)$. If we assume $m = O(n)$, then the total query time including pre-processing is $O(n \log n)$ [169].

In this chapter, we present a parallel algorithm for solving the next element search problem on a coarse grained multicomputer (CGM). The algorithm requires $O(1)$ communication rounds, $O(\frac{n}{p} \log n)$ local computation, and needs $O(\frac{n}{p} \log p)$ memory per processor, assuming $\frac{n}{p} \geq p$. A simplified version for axis parallel segments requires only $O(\frac{n}{p})$ memory per processor.

The next element search algorithm presented here implies immediate solutions for the point location, trapezoidal decomposition and triangulation problems.

As in a previous paper by Devillers and Fabri [62], our algorithm is based on a

Figure 6.1: Illustration of next element search

distributed implementation of segment trees which are of size $O(n \log n)$. This chapter improves on [62] in several ways:

- It studies the more general next element search problem which also solves, e.g., planar point location.
- The algorithms we provide require only $O(\frac{n}{p} \log n)$ local computation instead of $O(\log p * \frac{n}{p} \log n)$.
- These algorithms also require less on memory storage: $O(\frac{n}{p} \log p)$ instead of $O(\frac{n}{p} \log n)$.

It is also interesting to observe that the method presented in this chapter needs less than $O(n \log n)$ total memory (over all processors) because it eliminates some of the segment tree catalogs (while maintaining the same next element search functionality).

6.2 Segment Tree Definition

A well known method for solving the next element search problem is to apply a segment tree [9, 147, 169]. Let $s_i^{(x)}$ and $q_i^{(x)}$ be the projection of the line segment s_i and query point q_i , respectively onto the x -axis, and let $(x_1, x_2, \dots, x_{2n})$ be the sorted sequence of the projections of the $2n$ endpoints of $s_1, \dots, s_n$ onto the x -axis. The segment tree $T(S) = (V_s, E_s)$ is a complete binary tree with leaves $x_1, x_2, \dots, x_{2n}$. For every node v of $T(S)$ an interval $xrange(v)$ is defined as follows:

- if v is a leaf x_i , then $xrange(v) = [x_i, x_{i+1})$, where $[x_{2n}, x_{2n+1}) = [x_{2n}, x_{2n}]$.
- if v is an internal node, then $xrange(v)$ is the union of all intervals $xrange(v')$ such that v' is a leaf of the subtree of $T(S)$ rooted at v .

With every node v of the segment tree $T(S)$ there is associated a catalog $C(v) \subseteq S$ defined as follows:

- $C(v) = \{s \in S \mid xrange(v) \subseteq s(x) \text{ and not } (xrange(\text{parent of } v) \subseteq s(x))\}$.

Note that each line segment can occur in $O(\log n)$ catalogs. The size of the segment tree $T(S)$, denoted $|T(S)|$, is equal to the number of nodes and edges of $T(S)$ plus the total size of all catalogs. Therefore $|T(S)| = O(n \log n)$. Storing the segment tree with all of its catalogs requires $O(n \log n)$ space. Also note that the sum of the lengths of all catalogs of all nodes with the same level (height) is $O(n)$ [147]. For the remainder, define $xrange(T(S)) = xrange(r)$ where r is the root of $T(S)$. Also define $xrange(s)$ and $xrange(q)$ to be $s_i^{(x)}$ and $q_i^{(x)}$, respectively.

6.3 Parallel Segment Tree Construction

In this section we will show how to construct a distributed representation of a segment tree $T(S)$, called a *parallel segment tree*, for a set of n line segments on a CGM such that the resulting data structure can be efficiently used to process next element search queries in parallel. The approach will be to partition the segment tree (without associated catalogs) into substructures of size $O(\frac{n}{p})$ such that no processor stores more than $O(1)$ such substructures; see Figure 6.2. The catalogs for nodes in tree T_0 will be partitioned into lists of size $\frac{n \log p}{p}$ and distributed such that no processor stores more than $O(1)$ such lists; see Figure 6.3. The catalogs for nodes in subtrees $T_1, \dots, T_p$ will not be built explicitly. Instead, the algorithm will use standard sequential plane sweep (using a standard binary search tree) [169]. We first describe our distributed segment tree representation and then give an algorithm that efficiently constructs such a segment tree on a CGM.

Figure 6.2: Decomposition of the segment tree

A parallel segment tree $T(S)$ (without catalogs) is a complete tree with n leaves which consists of $\log n$ levels where the level of a node v is defined recursively as follows: $level(v) = 1$, if v is the root, and $level(v) = level(parent(v)) + 1$, otherwise. Let $rank(v)$ denote the rank of the vertex v in the left to right ordering of the vertices with the same level as v . For $v \in V$, let $subtree(v) \in 1 \dots p$ be defined as follows: $subtree(v) = p$, if $1 \leq level(v) \leq \log p - 1$ and $subtree(v) = rank(v')$ otherwise, where v' is the ancestor of v such that $level(v') = \log p$. 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 $E_i = \{(v, v') \in E \mid subtree(v) = subtree(v') = i\}$. Let $S(T_i)$ be the set of segments of S with an endpoint in $xrange(T_i)$.

In our distributed representation of a segment tree $T(S)$, processor P_i ($1 \leq i \leq p$) stores $S(T_i)$, the tree T_0 without catalogs but with the values $xrange(v)$ for each $v \in V_0$, and the list l_i which is a portion, or all, of the catalog of a node v of T_0 .

Observation 2 *If a line segment contains $xrange(T_i)$, then it is not contained in any catalog of T_i (except for possibly the root).*

Observation 3 $S(T_i)$ is of size $O(\frac{n}{p})$.

A consequence of Observation 3 is that each subset $S(T_i)$, $1 \leq i \leq p$, can be stored in the memory of a single processor. The tree T_0 consists of $O(p)$ nodes and catalogs whose combined size is $O(n \log p)$. Therefore T_0 is too big to be stored on a single

processor. Instead, each processor will store a copy of T_0 (without catalogs) and a list l_i which is a portion, or all, of the catalog of a node v of T_0 . Let L denote the list formed by concatenating the catalogs associated with nodes of T_0 , where catalogs are ordered by level and then rank in level and all catalogs are padded to be of a length evenly divisible by $\frac{n \log p}{p}$. The list l_i consisting of elements $\frac{in}{p} \dots \frac{(i+1)n}{p}$ from L will be stored on processor P_i ; see Figure 6.3.

Figure 6.3: The tree T_0 with catalogs partitioned into lists l_i

Observation 4 Since T_0 has height $\log p$ and a line segment can appear in at most 2 catalogs of T_0 at the same level, the total size of list L is $O(n \log p)$.

Definition 17 Given two line segments s_1 and s_2 . We define a relation " $\leq$ " as follows. If $xrange(s_1) \cap xrange(s_2) = \emptyset$ then $s_1 \leq s_2$ if and only if the y -coordinate of the left endpoint of s_1 is smaller or equal than the y -coordinate of the left endpoint of s_2 . If $xrange(s_1) \cap xrange(s_2) \neq \emptyset$ then $s_1 \leq s_2$ if and only if s_1 is below s_2 .

Algorithm 26 Parallel segment tree construction

Input: Processor P_i ($1 \leq i \leq p$) stores a subset S_i of $\frac{n}{p}$ elements of S .

Output: Processor P_i ($1 \leq i \leq p$) stores $S(T_i)$, the tree T_0 without catalogs but with the values $xrange(v)$ for each $v \in V_0$, and the list l_i which is a portion, or all, of the catalog of a node v of T_0 .

- (0) Sort S globally with respect to the relation " $\leq$ " in Definition 17. For each $s \in S$ calculate its rank $y(s)$ in S with respect to this ordering and store it with s .
- (1) Create for each $s \in S$ two copies, one for each endpoint, and sort the set by x -coordinate such that each processor P_i contains a subset S_i of size $O(\frac{n}{p})$. Now, processor P_i stores $S(T_i)$ and $xrange(T_i)$. From Observation 3, $|S(T_i)| = O(\frac{n}{p})$.
- (2) Use an all-to-all broadcast to distribute all $xrange(T_i)$ ($1 \leq i \leq p$) to all processors. Each processor computes T_0 without any catalogs but with $xrange(v)$ values for all $v \in V_0$.
- (3) Processor P_i computes the catalogs of T_0 with respect to S_i only. We refer to this reduced version of T_0 as $T_{0,i}$. Note that $|T_{0,i}| = O(\frac{n}{p} \log p)$.
- (4) Assume that all nodes of T_0 have a unique index. Consider a line segment s in the catalog of the node v in T_0 with index $j(s)$. For each such line segment s , define a key $\lambda(s)$ obtained by concatenating the bits of $j(s)$ and $y(s)$, in that order. Using a global integer sort, all line segments in the catalogs of all $T_{0,i}$, $1 \leq i \leq p$, are sorted with respect to key $\lambda(s)$ in such a way that no processor stores two line segments with different j values. The latter can be achieved by using $2p$ virtual processors.

— End of Algorithm —

Theorem 12 *Algorithm 26 constructs a parallel segment tree on a CGM using $O(\frac{n}{p} \log p)$ memory per processor, $O(1)$ communication rounds and $O(\frac{n}{p} \log n)$ local computation.*

Proof. The memory bound follows from Observations 3 and 4. The algorithm uses a constant number of the basic communication operations of the CGM model and, hence, $O(1)$ communication rounds. We now prove that the algorithm requires $O(\frac{n}{p} \log n)$ local computation. We observe that the local computation is dominated by the sorting steps. There are two types of sorting operations used: (1) global sort on $O(\frac{n}{p})$ data per processor in Steps 0 and 1, and (2) global *integer* sort on $O(\frac{n}{p} \log p)$ data per processor in Step 4. Since $O(\frac{n}{p} \log n) + O(\frac{n}{p} \log p) = O(\frac{n}{p} \log n)$, the claim follows. $\square$

6.4 Parallel Query Processing

Given a segment tree $T(S)$ and a query point $q \in Q$, the next element of q in S can be determined by a simple search in $T(S)$ from the root of $T(S)$ to the leaf v whose $xrange$ contains q (see e.g. [169] for details).

Recall that, at the end of Algorithm 26, processor P_i ($1 \leq i \leq p$) stores $S(T_i)$, the tree T_0 without catalogs but with the values $xrange(v)$ for each $v \in V_0$, and a list l_i which is a portion, or all, of the catalog of a node v of T_0 . Let $first_i$ and $last_i$ refer to the first and last element of l_i , respectively (see Figure 6.3).

The following algorithm uses the parallel segment tree to answer all queries in parallel. Each individual query is first “routed” through T_0 and then through the respective T_i . In T_0 , the tree structure is used to schedule the computation. However, the catalog lookups are reduced to sequential next element search problems. For the subtrees T_i , a load balancing scheme is used to ensure equal distribution of work. In each T_i , all search processes are reduced to a single sequential next element search problem.

Algorithm 27 Parallel query processing

Input: A parallel segment tree $T(S)$ as produced by Algorithm 26 and a set Q of n queries, where each processor P_i stores a subset Q_i of size $\frac{n}{p}$.

Output: Each Processor P_i ($1 \leq i \leq p$) stores for each $q \in Q_i$ its next element $s \in S$.

- (1) Using an all-to-all broadcast, send all $first_i$ and $last_i$ to all processors P_i . Recall that every processor P_i ($1 \leq i \leq p$) stores T_0 without catalogs but with values $xrange(\cdot)$.
- (2) Using the $O(p)$ line segments received in Step 1, processor P_i computes for each $q \in Q_i$ the sublists l_j ($1 \leq j \leq p$) that have to be searched in order to process q on T_0 . The problem reduces to solving for each node v of T_0 a next element search problem for a certain number X_v of line segments and a certain number Y_v of queries. Note that each such problem requires $O((X_v + Y_v) \log(X_v + Y_v))$ computation. Since $\sum_v X_v = p$ and $\sum_v Y_v = \frac{n}{p}$, it follows that the total local time is $O(\frac{n}{p} \log n)$.

- (3) Using global sort and partial sum operations, determine for each sublist l_i the number of queries, $g(l_i)$, that have to be located in l_i . Let $k(l_i) = \lceil \frac{g(l_i)p}{n} \rceil$.
- (4) Create $k(l_i)$ copies of l_i ($1 \leq i \leq p$). Note that this requires $2p$ virtual processors. Broadcast the new addresses of the sublists l_i .
- (5) Each processor P_i makes $\log p$ copies of its query set Q_i and routes the queries to the respective sublists using integer sort.
- (6) The queries are processed on the sublists to which they were sent in Step 5, and the $\log p$ results for each query are collected in a single processor by using global integer sort.
- (7) Determine for each T_i the number, $a(T_i)$, of queries whose search path includes the root of T_i ($1 \leq i \leq p$). This can be computed by using global integer sort and partial sum operations. Let $b(T_i) = \lceil \frac{a(T_i)}{p} \rceil$.
- (8) Create $b(T_i)$ copies of $S(T_i)$. Note that this requires $2p$ virtual processors. Using integer sort, route $\frac{n}{p}$ queries to each processor such that a processor storing $S(T_i)$ receives $\frac{n}{p}$ queries whose search path contains the root of T_i .
- (9) Each processor processes the queries for its subtree T_i ($1 \leq i \leq p$) by applying plane sweep [169].
- (10) Combine the results of Step 9 with those obtained in Step 6, using integer sort (by query ID).

— End of Algorithm —

Theorem 13 *Algorithm 27 solves the next element search problem for n line segments on a CGM with $O(\frac{n}{p} \log p)$ memory per processor using $O(1)$ communication rounds and $O(\frac{n}{p} \log n)$ local computation.*

Proof. The memory bound follows from Theorem 12. The algorithm uses a constant number of the basic communication operations of the CGM model and, hence, $O(1)$ communication rounds. We now prove that the algorithm requires $O(\frac{n}{p} \log n)$ local computation. We observe that the local computation is dominated by sorting and sequential plane sweep. There are two types of sorting operations used: (1) global sort on $O(\frac{n}{p})$ data per processor in Step 3, and (2) global *integer* sort on $O(\frac{n}{p} \log p)$ data per processor in Steps 5, 6, 7, 8, and 10. Since sequential plane sweep requires

$O(\frac{n}{p} \log n)$ steps [169]. We have $O(\frac{n}{p} \log n) + O(\frac{n}{p} \log p) = O(\frac{n}{p} \log n)$, and the claim follows. $\square$

Theorem 13 implies a BSP algorithm with local computation of $O(\frac{n}{p} \log n)$ and communication cost of $O(g \frac{n}{p})$.

We can calculate the speedup for this algorithm as follow. Let $T_s(n) = O(n \log n)$ be the running time of the best sequential algorithm. Let $T_p(n, p) = O(\frac{n}{p} \log n) + O(g \frac{n}{p})$ be the running time of our algorithm. Let c, d and k be the constants associated to the different big-O notations. The speedup is therefore $s = \frac{T_s(n)}{T_p(n, p)} = \frac{cn}{\frac{dn}{p} \log n + \frac{kgn}{p}} = \frac{p}{g} (\frac{c}{d+k}) = \Omega(\frac{p}{g})$.

6.5 A Simplified Algorithm for Axis Parallel Line Segments

If we limit the segments to be axis parallel (i.e. they are all horizontal), we can reduce the space requirement to $O(\frac{n}{p})$ per processor by applying the lower envelope algorithm presented in [58]. During the algorithm, queries will be handled like line segments of zero length.

Algorithm 28 Next element search for axis parallel line segments

Input: A set S of n axis parallel line segments and a set Q of n queries, where each processor P_i stores $\frac{n}{p}$ line segments and queries, respectively.

Output: The next element in S for each query point $q \in Q$, where each processor stores $\frac{n}{p}$ next element results.

- (1) Sort $S \cup Q$ by increasing y -coordinate. Each processor P_i solves, both, the next element search problem and lower envelope problem sequentially for the set $S_i \cup Q_i$ it received in the sort. Let Q' be the set of all queries whose next element has not yet been found, and let S' be the union of all lower envelopes.
- (2) Sort $S' \cup Q'$ by x -coordinate of the right endpoints and the x -coordinates of the queries, respectively.
- (3) Let $l_1, \dots, l_{p-1}$ be the vertical lines that separate the sorted segments in the p different processors. Perform an all-to-all broadcast where processor P_i sends l_i

to all other processors.

- (4) Perform a personalized all-to-all broadcast, where processor P_i sends segment $s \in S'$ to processor P_j if and only if s intersects the vertical line l_j .
- (5) Each processor P_i solves locally the next element search problem for its subset of Q' and the line segments of S' received in Steps 2 and 4.

— End of Algorithm —

Theorem 14 *Algorithm 28 solves the next element search problem for n axis parallel line segments on a GCM with $O(\frac{n}{p})$ memory per processor using $O(1)$ communication rounds and $O(\frac{n}{p} \log n)$ local computation, assuming $\frac{n}{p} \geq p$.*

Proof. The correctness follows from the fact that for each $q \in Q_i$, its next element is either in S_i or in the lower envelope of an S_j with $j > i$. The algorithm uses a constant number of the basic communication operations of the CGM model and duplicates no data. $\square$

With the same argument as in the general case, the speedup of this algorithm can be calculated as $\Omega(\frac{p}{g})$.

6.6 Applications

We have presented a BSP like coarse grained parallel algorithm for the next element search problem which requires $O(1)$ h -Relations ($h = O(\frac{n}{p})$), $O(\frac{n}{p} \log p)$ memory per processor and $O(\frac{n}{p} \log n)$ local computation, assuming $\frac{n}{p} \geq p$. An important advantage of our model is that it gives a very good indication of the running time observed in an actual implementation. As demonstrated in [56–58], coarse grained parallel algorithms with $O(1)$ communication rounds are also portable across very different parallel platforms. Therefore, we expect that our algorithm presented here will also run well on other parallel machines.

Next element search can be used to solve many other geometric problems. Some of the more important examples include the following:

1. **Planar subdivision search problem.** Given a plane graph $G = (V, E)$ with vertex coordinates, and a set of n query points q_i ($1 \leq i \leq n$), find for each query point q_i , the face of G containing q_i .
2. **Trapezoidal map problem.** Given a set of segments in the plane, decompose the plane into a set of trapezoids based on the arrangement of the segments.
3. **Triangulation problem for a simple polygon.** Partition the interior of a simple polygon into a set of triangles.

The above three problems can be reduced to $O(1)$ next element search problems (obvious for 1 and 2; see [189] for 3). Hence, Theorems 12 applies to these problems as well and we obtain:

Corollary 1 *The planar subdivision search problem, trapezoidal map problem, and triangulation problem for a simple polygon can be solved on a CGM with $O(\frac{n}{p} \log p)$ memory per processor in $O(1)$ communication rounds and $O(\frac{n}{p} \log n)$ local computation, assuming $\frac{n}{p} \geq p$.*

6.7 Summary of Results

The following table summarizes the result we got in this chapter. Note that n is the input size, p is the number of processors available.

Algorithm	Local Computation	Communication Rounds	Speedup	Storage Requirement
General Next Element Search	$O(\frac{n}{p} \log n)$	$O(1)$	$\Omega(\frac{p}{g})$	$O(\frac{n}{p} \log p)$
Next Element Search for Axis Parallel Segments	$O(\frac{n}{p} \log n)$	$O(1)$	$\Omega(\frac{p}{g})$	$O(\frac{n}{p})$

Chapter 7

Coarse Grained Sorting

7.1 Problem Overview

Goodrich [85] presented a deterministic sorting algorithm for the BSP [185] and the closely related CGM model [56, 58]. Given $O(n)$ data items stored on a p processor CGM/BSP, $O(\frac{n}{p})$ data items per processor, these items can be sorted in $O(\frac{\log n}{\log(h+1)})$ communication rounds (h -Relations), for $h = O(\frac{n}{p})$, with $O(\frac{n \log n}{p})$ local computation, using $O(\frac{n}{p})$ memory per processor. For $\frac{n}{p} \geq p^\epsilon$, $\epsilon > 0$, $O(\frac{\log n}{\log(h+1)}) = O(1)$. That is, for this case, the algorithm requires $O(1)$ communication rounds.

We are interested in the problem of sorting n integers in the range $1, \dots, n^c$, for fixed constant c , stored on a p processor CGM/BSP, $\frac{n}{p}$ data items per processor. The sort algorithm in [85] is based on Cole's merge sort [33]. The $O(\frac{n \log n}{p})$ local computation in [85] is due to a constant number of local sorts. Hence, by applying radix sort for the integer case, it is easy to obtain $O(\frac{n}{p})$ local computation without increasing the number of communication rounds.

In this chapter we observe that for $\frac{n}{p} \geq p$, which is usually the case in practice, there exists a very simple, deterministic, optimal CGM/BSP integer sorting algorithm with 24 communication rounds (6 $\frac{n}{p}$ -Relations and 18 p -relations), $O(\frac{n}{p})$ memory per processor and $O(\frac{n}{p})$ local computation.

7.2 The Algorithm

We first present a CGM/BSP integer sorting algorithm for $\frac{n}{p} \geq p^2$, which serves as our base case, and then extend it to the case $\frac{n}{p} \geq p$. The integer sorting algorithm for $\frac{n}{p} \geq p^2$, described next, follows the well known *deterministic sample sort* method combined with *radix sort* for the sequential sorting steps. We are making the algorithm descriptions fairly detailed in order to allow an analysis that includes estimates of constant factors.

Algorithm 29 Sorting n integers on a p processors CGM/BSP with $\frac{n}{p} \geq p^2$.

Input: n integers in the range $1, \dots, n^c$, for fixed constant c , stored on a p processor

CGM/BSP, $\frac{n}{p}$ integers per processor. $\frac{n}{p} \geq p^2$.

Output: The integers are permuted into sorted order.

- (1) Each processor sorts locally its $\frac{n}{p}$ integers, using radix sort.
- (2) Each processor selects from its locally sorted integers a sample of p integers with ranks $i\frac{n}{p^2}$, $0 \leq i \leq p-1$. We refer to these selected integers as *local samples*. All local samples are sent to processor P_1 . (Note that P_1 is to receive in total p^2 integers. This is possible since $\frac{n}{p} \geq p^2$.)
- (3) P_1 sorts the p^2 local samples received in Step 2 (using radix sort) and selects a sample of p integers with ranks ip , $0 \leq i \leq p-1$. We refer to these selected integers as the *global samples*. The p global samples are then broadcast to all processors.
- (4) Based on the received global samples, each processor P_i partitions its $\frac{n}{p}$ integers into p buckets $B_{i,1}, \dots, B_{i,p}$ where $B_{i,j}$ are the local integers with value between the $(j-1)$ -th and j -th global sample.
- (5) In one (combined) h -Relation, every processor P_i , $1 \leq i \leq p$, sends $B_{i,j}$ to processor P_j , $1 \leq j \leq p$. Let R_j be the set of integers received by processor P_j , $1 \leq j \leq p$, and let $r_i = |R_i|$.
- (6) Every processor P_i , $1 \leq i \leq p$, locally sorts R_i using radix sort.
- (7) A global “balancing shift” operation which distributes all integers evenly among the processors without changing their order is performed as follows: Every processor P_i , $1 \leq i \leq p$, sends r_i to P_1 . Processor P_1 calculates for each P_j an array

A_j of p numbers indicating how many of its integers have to be moved to the respective processors. In one h -Relation, every A_j is sent to P_j , $1 \leq i \leq p$. The “balancing shift” is then performed in a subsequent single h -Relation according to the A_j values.

— End of Algorithm —

Theorem 15 *Algorithm 29 sorts n integers in the range $1, \dots, n^c$, for fixed constant c , stored on a p processor CGM/BSP, $\frac{n}{p}$ integers per processor, $\frac{n}{p} \geq p^2$, using 6 communication rounds ($2 \frac{n}{p}$ -Relations and $4 p^2$ -Relations), $O(\frac{n}{p})$ local memory per processor, and $O(\frac{n}{p})$ local computation.*

Proof. Correctness: The algorithm follows the well known deterministic sample sort method combined with radix sort for the sequential sorting steps. Let $S_{i,j}$ be the set of integers on P_i , at the end of Step 1, with rank between $j \frac{n}{p^2}$ and $(j+1) \frac{n}{p^2}$, $0 \leq j \leq p-1$. The main observation is that each R_k contains at most $3p$ sets $S_{i,j}$. Complexity: The local memory and local computation are bounded by the sequential local radix sorts. For the communication, we observe that Steps 2 and 3 require one p^2 -Relation, each, Step 5 requires one $\frac{n}{p}$ -Relation, and Step 7 requires two p^2 -Relations and one $\frac{n}{p}$ -Relation. $\square$

We now describe the algorithm for the case $\frac{n}{p} \geq p$. The basic idea is to partition the p processors into $\sqrt{p}$ groups $G_1, \dots, G_{\sqrt{p}}$ of $\sqrt{p}$ processors, each. The main operation consists of permuting the integers such that all integers stored in G_i are smaller than all integers stored in G_j for all $i < j$. We can then apply Algorithm 29 to each group. Again, we are making the description of Algorithm 30 fairly detailed in order to allow an analysis that includes estimates of constant factors.

Algorithm 30 Sorting n integers on a p processors machine when $\frac{n}{p} \geq p$.

Input: n integers in the range $1, \dots, n^c$, for fixed constant c , stored on a p processor CGM/BSP, $\frac{n}{p}$ integers per processor. $\frac{n}{p} \geq p$.

Output: The integers are permuted into sorted order.

- (1) Group the p processors into $\sqrt{p}$ groups $G_1, \dots, G_{\sqrt{p}}$ of $\sqrt{p}$ processors, each. For each group, apply Algorithm 29 to sort the integers within the group.

- (2) Each processor sends its smallest integer to processor P_1 . We will refer to these items as the *local minima*. (Note that P_1 is to receive p local minima in total, and that this is possible since $\frac{n}{p} \geq p$.)
- (3) P_1 sequentially sorts all local minima and selects the $\sqrt{p}$ integers with rank $i \frac{n}{p\sqrt{p}}$, $0 \leq i \leq \sqrt{p}$, referred to as *global splitters*. These $\sqrt{p}$ global splitters are broadcast to all processors (using 2 p -Relations).
- (4) Based on the received global splitters, each processor P_i partitions its $\frac{n}{p}$ integers into $\sqrt{p}$ buckets $B'_{i,1}, \dots, B'_{i,\sqrt{p}}$ where $B'_{i,j}$ contains the local integers with value between the $(j-1)$ -th and j -th global splitter.
- (5) Every processor P_i , $1 \leq i \leq p$, sends $B'_{i,j}$ to a processor in group G_j , $1 \leq j \leq \sqrt{p}$. Let R'_j be the set of integers received by processors in group G_j , $1 \leq j \leq \sqrt{p}$, and let $r'_j = |R'_j|$.

The routing schedule for this operation is determined as follows: First, within each group G_i the size, $t_{i,j}$, of each integer set sent to group G_j , $1 \leq j \leq \sqrt{p}$, is computed. All $t_{i,j}$, $1 \leq i, j \leq \sqrt{p}$, are sent to one *leading* processor per group. Furthermore, within each group, the sizes of all $B'_{i,j}$ are also sent to the leading processor. Each leading processor can then compute a routing schedule for its group and broadcast it to the processors in its group.

- (6) Each group G_j , $1 \leq j \leq \sqrt{p}$, sorts R'_j using Algorithm 29.
- (7) A global “balancing shift” operation which distributes all integers evenly among the processors without changing their order is performed analogous to Step 6 of Algorithm 29 but with a “two-phase” scheme analogous to the routing schedule computation in Step 5 of Algorithm 30.

— End of Algorithm —

Theorem 16 *Algorithm 30 sorts n integers in the range $1, \dots, n^c$, for fixed constant c , stored on a p processor CGM/BSP, $\frac{n}{p}$ integers per processor, $\frac{n}{p} \geq p$, using 24 communication rounds (6 $\frac{n}{p}$ -Relations and 18 p -Relations), $O(\frac{n}{p})$ local memory per processor, and $O(\frac{n}{p})$ local computation.*

Proof. We first show the correctness of the algorithm. For each group G_i of processors, we have $n' = \frac{n}{\sqrt{p}}$ integers and $p' = \sqrt{p}$ processors. Therefore, $\frac{n'}{p'} = \frac{(\frac{n}{\sqrt{p}})}{\sqrt{p}} = \frac{n}{p} \geq$

$p = p'^2$, and Algorithm 29 is applicable for each group. Let S'_i be the set of integers stored at P_i after Step 1. The second main observation is that each R'_j contains at most $3\sqrt{p}$ sets S'_i .

Now we show the complexity of the algorithm. The local memory and local computation are bounded by the sequential local radix sorts. For the communication, we observe that Step 1 requires $2 \frac{n}{p}$ -Relations and 4 p -Relations, Step 2 requires 1 p -Relation, Step 3 requires 2 p -Relations, Step 5 requires 1 $\frac{n}{p}$ -Relation and 3 p -Relations, Step 6 requires 2 $\frac{n}{p}$ -Relations and 4 p -Relations, and Step 7 requires 1 $\frac{n}{p}$ -Relations and 3 p -Relations. $\square$

Theorem 16 implies a BSP algorithm with local computation of $O(\frac{n}{p})$ and communication cost of $O(g\frac{n}{p})$.

We can calculate the speedup for this algorithm as follow. Let $T_s(n) = O(n)$ be the running time of the best sequential algorithm. Let $T_p(n, p) = O(\frac{n}{p}) + O(g\frac{n}{p})$ be the running time of our algorithm. Let c, d and k be the constants associated to the different big-O notations. The speedup is therefore $s = \frac{T_s(n)}{T_p(n, p)} = \frac{cn}{\frac{dn}{p} + \frac{kgn}{p}} = \frac{p}{g}(\frac{c}{d+k}) = \Omega(\frac{p}{g})$.

7.3 Generalization

We observe that the algorithms presented in the previous section are not restricted to sorting integers. Of course, if the items to be sorted are not integers, then we cannot apply the integer sort algorithm in the local sorting steps. We will need a more general sorting algorithm such as heapsort or merge sort. In this case, the other parts of the algorithms are not affected. The only consequence is the time complexity now changes to $O(\frac{n}{p} \log n)$ instead of $O(\frac{n}{p})$.

Similar to the last section, we can calculate the speedup of this algorithm as $s = \frac{T_s(n)}{T_p(n, p)} = \frac{cn \log n}{\frac{dn \log n}{p} + \frac{kgn}{p}} = \frac{p}{g}(\frac{c}{d + \frac{k}{\log n}}) = \Omega(\frac{p}{g})$.

7.4 Summary of Results

The following table summarizes the result we got in this chapter. Note that n is the input size, p is the number of processors available, and $T_s(n)$ is the performance of the best sequential algorithm available.

Algorithm	Local Computation	Communication Rounds	Speedup
Parallel Sorting	$O(\frac{T_s(n)}{p})$	$O(1)$	$\Omega(\frac{p}{g})$

Chapter 8

CGMlib and CGMgraph

8.1 Introduction

In this chapter, we present *CGMgraph*, the first integrated library of CGM methods for various graph problems including list ranking, Euler tour, connected components, spanning forest, and bipartite graph recognition. Our library also includes a library *CGMlib* of basic CGM tools that are necessary for parallel graph methods as well as many other CGM algorithms: sorting, prefix sum, one to all broadcast, all to one gather, *h*-Relation, all to all broadcast, array balancing, and CGM partitioning. In comparison with [130], *CGMgraph* implements both a randomized as well as a deterministic list ranking method. Our experimental results for randomized list ranking are similar to the ones reported in [130]. Our implementations of deterministic list ranking, Euler tour, connected components, spanning forest, and bipartite graph recognition are, to our knowledge, the first efficient implementations for PC clusters.

CGMgraph and *CGMlib* are based on the CGM/BSP model and are optimized for PC clusters. In the experimental part of this chapter, we show the performance of our methods on three different cluster architectures: *THOG*, *CGM1* and *ULTRA*. The *THOG* cluster is a gigabit connected high performance cluster, *CGM1* is a smaller cluster connected via fast ethernet, and *ULTRA* is a network of workstations. Our experiments show that our library provides good relative parallel speedup and scalability on all three platforms. The communication overhead is, in most cases, small and

does not grow significantly with an increasing number of processors. This is a very important feature of CGM algorithms which makes them very efficient in practice. The communication overhead is, in most cases, dominated by the local computation time, and the latter is proportional to the problem size divided by the number of processors. Therefore, the total measured wall clock time is also proportional to the problem size divided by the number of processors, i.e., the relative speedup is close to linear.

Both the *CGMlib* and *CGMgraph* libraries are freely available for download at <http://www.scs.carleton.ca/~cgm>, together with a library installation script. The aims of these libraries are to make efficient parallel graph methods for processor clusters available to a wider community of researchers who can utilize them as building blocks for other parallel programming projects [26].

8.2 Library Overview and Experimental Setup

Figure 8.1 illustrates the general use of *CGMlib* and *CGMgraph*, and Figure 8.2 shows the class hierarchy of the main classes in *CGMlib* and *CGMgraph*. Note that all classes in *CGMgraph*, except *EulerNode*, are independent. Both libraries require an underlying communication library such as MPI or PVM. *CGMlib* provides a class *Comm* which interfaces with the underlying communication library. It provides an interface for all communication operations used by *CGMlib* and *CGMgraph* and thereby hides the details of the communication library from the user. At this moment, an MPI implementation of *Comm* is available as part of *CGMlib*. The aim of our design is to provide portability of *CGMlib* and *CGMgraph* accross different architectures and communication libraries. For any other communication library, an implementation of *Comm* is sufficient to port the entire *CGMlib* and *CGMgraph*. As shown in Figure 8.1, *CGMlib* provides basic CGM functionality (sorting, prefix sum, etc.) and *CGMgraph* uses *CGMlib* in order to implement graph algorithms. The design separates the two libraries so that other CGM algorithms (in addition to the CGM graph library provided) can also make use of the *CGMlib* methods.

The performance of our library was evaluated on three parallel platforms: *THOG*,

Figure 8.1: General use of *CGMlib* and *CGMgraph*

CGM1 and *ULTRA*. The *THOG* cluster is located in the High Performance Computing Virtual Laboratory (HPCVL) at Carleton University. This cluster consists of 64 nodes, each with two Xeon processors. The nodes are of two different generations, with processors at 1.7 or 2.0GHz, 1.0 or 1.5GB RAM, and 60GB disk storage. The nodes are interconnected via a Cisco 6509 switch using Gigabit ethernet. The operating system is Linux Red Hat 7.1 together with LAM-MPI version 6.5.6. The *CGM1* cluster is located in the Carleton/Dalhousie CGMlab. This cluster consists 32 processor nodes. Each node has two Xeon processors at 1.8GHz with 1.0GB RAM and 80GB disk storage. The nodes are interconnected via 100Mb Switched Fast Ethernet. The operating system is Linux Red Hat 7.1 together with LAM-MPI version 6.5.6. The *ULTRA* platform is a network of workstations located in the Carleton Univ. School of Computer Science graduate laboratory. The network consists of 10 Sun Sparc Ultra 10 workstations. The processor speed is 440MHz, Each processor has 256MB RAM. The nodes are interconnected via 100Mb Switched Fast Ethernet. The operating system is Sun OS 5.7 and LAM-MPI version 6.3.2.

For the remainder, p denotes the number of processors, m denotes the total memory size of the parallel machine, $\frac{m}{p}$ is the memory per processor, and n denotes the total input data size. All times reported in the remainder of this paper are wall clock times in seconds. Communication times reported are the combined wall clock times

Figure 8.2: Hierarchy of main classes in CGMlib and CGMgraph

spent by the MPI method calls. The reported computation times are the difference between the total wall clock times and the communication times. The input data sets for our tests consist of randomly created test data. Unfortunately, different test data sizes had to be chosen for the different platforms because of the smaller memory capacity of *CGM1* and in particular *ULTRA*. For each experiment, we measured the total wall clock time as well as the wall clock time for the communication and the computation portion of the experiment. Each data point in the diagrams presented in the remainder of this paper represents the average of three experiments for *CGMgraph* and ten experiments for *CGMlib*.

During the development of *CGMlib* and *CGMgraph*, we observed irregular performance for large input data sizes. Investigation showed that this was caused by

the C++/Linux memory management. When our code releases unused memory, the memory management system attempts to defragment the memory. This defragmentation occurs at unpredictable times and can lead to performance degradation. To remedy the problem, we provide our own memory management for the class `ObjList` where arrays are allocated manually and memory is managed by our code. Our experience has shown that this can bring very substantial performance improvements.

8.3 CGMlib: Basic Infrastructure and Utilities

8.3.1 CGM Communication Operations

The basic library, called *CGMlib*, provides basic functionality for CGM communication. An interface, `Comm`, defines the basic communication operations such as

- `oneToAllBCast (int source, CommObjectList &data)`: Broadcast the list `data` from processor number `source` to all processors.
- `allToOneGather (int target, CommObjectList &data)`: Execute a gather operation on the lists `data` from every processor to processor number `target`.
- `hRelation (CommObjectList &data, int *ns)`: Perform an *h*-Relation on the lists `data` using the integer array `ns` to indicate for each processor which list objects are to be sent to which processor.
- `allToAllBCast (CommObjectList &data)`: Every processor broadcasts its list `data` to all other processors.
- `arrayBalancing (CommObjectList &data, int expectedN = -1)`: Shift the list elements between the lists `data` such that every processor will contain `expectedN` elements. `expectedN = -1` indicates that every processor receives the same number of elements.
- `partitionCGM (int groupId)`: Partition the CGM into groups indicated by `groupId`. All subsequent communication operations, such as the ones listed above, operate within the respective processor's group only.

- `unPartitionCGM ()`: Undo the previous partition operation.

All communication operations in *CGMlib* send and receive data in the form of lists of type `CommObjectList`. A `CommObjectList` is a list of `CommObject` elements. The `CommObject` interface defines the operations which every object that is to be sent/received has to support. A class `SimpleCommObject` is provided which puts an appropriate wrapper around any C++ object (as long as it does not contain pointers). A class `BasicCommObject` is provided which puts an appropriate wrapper around the basic C++ data types (int, float, etc.).

The above operations are defined for a CGM with local memory $\frac{m}{p} \geq O(p)$. For larger local memory with $\frac{m}{p} \geq O(p^2)$, simpler versions `allToAllBCast2 (...)` and `arrayBalancing2 (...)` for all-to-all broadcast and array balancing, respectively, are available.

The library is designed such that everything is built on the `Comm` interface. This allows different implementations of `Comm` to support different platforms. For the current *CGMlib*, we provide an MPI implementation `MPIComm`. In the future it is possible to add, for example, a PVM implementation of `Comm` which will enable the entire *CGMlib* and *CGMgraph* to run on a PVM platform as well.

8.3.2 CGM Utilities

CGM Parallel Prefix Sums

Definition 18 *Given a list of n items $A = \{a_1, a_2, \dots, a_n\}$ and an associative operator $\otimes$, the prefix sum operation over A and $\otimes$ is to calculate another list of n items $B = \{b_1, b_2, \dots, b_n\}$ such that $b_i = a_1 \otimes a_2 \otimes \dots \otimes b_i$.*

A CGM implementation of the parallel prefix sum algorithm is available. The algorithm is simple. The method `calculatePrefixSum (CommObjectList &result, CommObjectList &data)` performs the following steps: Each processor calculates a local prefix sum. The local total sum values are sent to one processor via a `allToOneGather`. That processor calculates a prefix sum of the total sums and sends the results back to the other processors via an `hRelation` operation. Finally, each

processor adjusts its local prefix sum values. The operator used for the prefix sum is flexible and is set via the class constructor.

CGM Parallel Sorting

A method `sort (CommObjectList &data)` is provided which sorts the union of the lists `data` from all processors. The comparison operator used for the sort is flexible and is set via the class constructor. We use the deterministic parallel sample sort methods in [23, 24] and [175]. Both methods require a local sequential sort method for which the default in *CGMlib* is Heapsort. However, any other local sequential sort method can be assigned via the class constructor. While [175] requires $\frac{m}{p} \geq O(p^2)$, the algorithm in [23, 24] requires only $\frac{m}{p} \geq O(p)$ but is slightly more complex. Depending on the local memory available, the user can select between the two sorting methods via a method `setScalability`. In theory, any fixed scalability can be achieved via the algorithm in [85]. However, this method is very complex and incurs large overheads. Therefore, the method in [85] is not implemented in this library.

CGM Request System

In some CGM algorithms, it is necessary for processors to obtain information from other processors. This can be done by sending requests. In the CGM model, the requests are all handled jointly via some *h*-Relation and sort operations. The *CGMlib* provides methods `sendRequests (...)` and `sendResponses (...)` for routing the requests from their senders to their destinations and returning the responses to the senders, respectively.

Other CGM Utilities

Various utilities are provided. The most important is the class `CGMTimers`. We provide six timers within `CGMTimers` which measure computation time, communication time, and total time, both in wall clock time and CPU ticks. Another utility provided is a parallel random number generator. The main purpose is to make sure that all processors use different seeds for the pseudo random number computation.

8.3.3 Performance Evaluation

We have measured the performance of our communication operations and utilities on *THOG*, *CGM1* and *ULTRA*. In the following, we present the results for the *h*-Relation, prefix sum and parallel sort operations. For each operation, we measured the performance on *THOG* with $n = 5,000,000$ and $n = 10,000,000$, on *CGM1* with $n = 5,000,000$, and on *ULTRA* with $n = 100,000$. The data consists of randomly created integers. For the *h*-Relation, the array `ns` is a different array of random integers for each processor.

Figure 8.3 shows the performance of our *h*-Relation implementation on *THOG*, *CGM1* and *ULTRA*. Figure 8.4 shows the performance of our prefix sum implementation, and Figure 8.5 shows the performance of our parallel sort implementation. For $n = 5,000,000$, *THOG* is about twice as fast as *CGM1*. Both machines have comparable processors, and the computation times are indeed similar. However, *THOG* has a much more powerful switch which results in much faster communication times. For our *h*-Relation implementation, we observe that the curves for the total wall clock times are in all four diagrams similar to $\frac{1}{p}$ for $p \leq 10$ but then become more “flat”. This is particularly pronounced for *THOG* where we have a larger number of processors available. It appears to be caused by the communication times. The computation portion always appear to be similar to $\frac{1}{p}$ but the communication times appear to go “flat” for $p > 10$. This is caused by the fact that the total volume of data communicated through the switch is the same, regardless of p . For small p , sending the data to the switch through more ports appears to bring some improvement in speed but this effect disappears for larger p . This is an important observation. The communication time for an *h*-Relation operation can essentially be seen as a function of the total data volume only, independent of the number of processors.

For our prefix sum implementation, we observe that all experiments show a close to zero communication time, except for some noise on *THOG*. The prefix sum method communicates only very few data items. The total wall clock time and computation time curves in all four diagrams are similar to $\frac{1}{p}$.

For our parallel sort implementation, we observe a small fixed communication time, essentially independent of p . This is easily explained by the fact that the

parallel sort uses of a fixed number of h -Relation operations, independent of p . Most of the total wall clock time is spent on local computation which consists mainly of local sorts of $\frac{n}{p}$ data. Therefore, the curves for the local computation and the total parallel wall clock time are similar to $\frac{1}{p}$.

Figure 8.3: Performance of the h -Relation implementation

8.4 CGM_{graph}: Parallel List Ranking and Euler Tour

CGM_{graph} provides a list ranking method `rankTheList (ObjList <Node> &nodelist, bool randomized, Comm *comm)` which implements a randomized method as well as a deterministic method depending on the boolean `randomized = TRUE/FALSE`, respectively. The input to the list ranking method is a linear linked list where each

Figure 8.4: Performance of the prefix sum implementation

node has four integer fields: `index` is a non-negative integer index. The indices need not to be consecutive. Field `next` is a pointer that points to the next node in the linked list. The pointer is stored as the index of the next node. Field `rank` is an integer to be filled by the `rankTheList (...)` method to represent the rank of the node. Field `tail` is a pointer to be filled by the `rankTheList (...)` method to represent the last node in the linked list.

`CGMgraph` also provides a method to compute the Euler tour traversal of a forest. The method is `getEulerTour (ObjList<Vertex> &r, ObjList<Vertex> &v, ObjList<Edge> &e, ObjList<EulerNode> &eulerNodes, Comm *comm)`.

The forest is represented by a list of vertices, a list of edges and a list of roots. The input to the Euler tour method is a forest which is stored as follows: `r` is a set of vertices that represents the roots of the trees, `v` is the input set of vertices, `e` is the input set of edges, and `eulerNodes` is the output data of the method.

Figure 8.5: Performance of the parallel sort implementation

8.4.1 Randomized List Ranking

We implemented a simplified version of the *randomized* list ranking method described in [59]. The following is an outline of our method which proceeds in three phases. Phase 1: A randomized method is used to approximate a p -sample, which is defined as a subset of $\frac{n}{p}$ *selected* nodes with the property that two consecutive selected nodes in the linked list have a distance of $O(p^k)$, $k = O(1)$, with respect to the linked list. The method simply selects $\frac{n}{p}$ random nodes. Clearly, the average distance between two selected nodes is p . However, what is more important is the maximum distance between two selected nodes which is, with high probability, bounded by $O(p \log n)$. Phase 2: Using simulated pointer jumping [173], implemented via the CGM Request System (Section 8.3.2), every list node obtains a pointer to the next selected node in the list, together with the distance to that node. All selected nodes are then

compressed into one processor and sequential weighted list ranking is applied where the weights are the distances obtained in the previous step. Phase 3: All other (non-selected) nodes determine their rank by obtaining the rank of the next selected node (as determined in the previous step) and adding to it the distance obtained through the simulated pointer jumping. The total number of h -Relation operations for all three phases is, with high probability between $O(\log p)$ and $O(\log p + \log \log n)$. For practical purposes, $\log \log n$ can be assumed to be bounded by a constant (approx. 4 for our data sets).

8.4.2 Deterministic List Ranking

We also implemented the *deterministic* list ranking method described in [20]. The deterministic algorithm is much more involved than the randomized method. The following is a brief outline of the algorithm. For more details, see [20]. The basic structure of the deterministic method is the same as in the randomized case. However, a different approach is used for determining the selected nodes in Phase 1. The approach consists of a combination of a CGM adaptation of deterministic coin tossing [173] and various merge steps. At the end of this phase, the set of selected nodes has the property that every node has a distance of at most $O(p^2)$ from the next selected node in the list. This guarantees that the entire algorithm requires between $c \log p$ and $2c \log p$ h -Relation operations in total. The drawback of this method is that it is more involved and incurs considerably larger constant factors in the running times.

8.4.3 Euler Tour

The Euler tour traversal of a tree is simple, once a list ranking method is available. An Euler tour traversal in linked list format is generated by converting each tree edge into two directed edges. The edges have to be linked together in the appropriate way to avoid self loops. See [20] for more details. Then, list ranking is applied to this linked list.

8.4.4 Performance Evaluation

We have measured the performance of our randomized and deterministic list ranking methods on *THOG*, *CGM1* and *ULTRA*. In the following, we present the results of our experiment. For each operation, we measured the performance on *THOG* with $n = 10,000,000$, on *CGM1* with $n = 5,000,000$, and on *ULTRA* with $n = 100,000$.

Figure 8.6 shows the performance of the randomized list ranking algorithm on *THOG*, *CGM1* and *ULTRA*. We observe that for all three machines, the communication time is only a small portion of the total time and appears to be fixed even with growing p . The number of rounds measured in our experiments was usually between 6 and 8, with small fluctuations. Recall that each round has a fixed communication time (except for small p), as discussed in Section 8.3.3 and that the randomized list ranking requires between $O(\log p)$ and $O(\log p + \log \log n)$ h -Relation operations. With $\log p$ in the range $[1, 5]$ and $\log \log n$ approx. 4, we expect between 5 and 9 h -Relation operations. What is somewhat surprising at first is that, for our experiments, the number of h -Relation operations stayed always in the $[6, 8]$ “middle range”. However, even if it would not, the fluctuation in number of h -Relation operations would not be large. The experiments show that the $\log p$ factor is growing so slowly in practice that it has little influence on the measured communication time which is essentially independent of p . In summary, since the communication time is only a small fixed value and the computation time is dominating and similar to $\frac{1}{p}$, the entire measured wall clock time is similar to $\frac{1}{p}$.

Figure 8.7 shows the performance of the deterministic list ranking algorithm. Again, we observe that for all three machines, the communication time is a small, essentially fixed, portion of the total time (though somewhat larger than in Figure 8.6). The deterministic list ranking requires between $c \log p$ and $2c \log p$ h -Relation operations. With $\log p$ in the range $[1, 5]$, we expect between c and $10c$ h -Relation operations. Since the deterministic algorithm is more involved and incurs larger constants, c may be around 10 which would imply a range of $[10, 100]$ for the number of h -Relation operations. We measured usually around 20 h -Relation operations. The number is fairly stable, independent of p , which shows again that $\log p$ has little influence on the measured communication time. The small increases for $p = 4, 8, 16$

are due to the fact that the number of h -Relation operations grows with $\lfloor \log p \rfloor$, which gets incremented by 1 when p reaches a power of 2. In summary, since the communication time is only a small fixed value and the computation time is dominating and similar to $\frac{1}{p}$, the entire measured wall clock time is similar to $\frac{1}{p}$.

Figure 8.8 shows the performance of the Euler tour algorithm on *THOG*, *CGM1* and *ULTRA*. Our implementation uses the deterministic list ranking method for the Euler tour computation. Not surprisingly, the performance is essentially the same as for deterministic list ranking. Due to the fact that all tree edges need to be duplicated, the data size increases by a factor of three (original plus two copies). This is the reason why we could execute the Euler tour method on *THOG* for $n = 10,000,000$ only with $p \geq 10$.

Figure 8.6: Performance of the randomized list ranking algorithm

Figure 8.7: Performance of the deterministic list ranking algorithm

8.5 CGM_{graph}: Parallel Connected Components and Spanning Forest

CGM_{graph} provides a method called `findConnectedComponents` (Graph &g, Comm *comm) for the computation of the connected components of a graph. It also has a method `findSpanningForest` (Graph &g, ObjList<Vertex> &spanningRoots, ObjList<Edge> &spanningTreeEdges, Comm *comm) for the calculation of the spanning forest of a graph. The input to the above two methods is a graph represented as a list of vertices and a list of edges.

Figure 8.8: Performance of the Euler tour algorithm

8.5.1 Outline of Method

We implemented the connected component method described in [20]. The method also provides immediately a spanning forest of the given graph. The following is a brief outline of the algorithm. For more details, see [20]. The method consists of two phases. Phase 1: Using the CGM Request System (Section 8.3.2), $\log p$ iterations of the standard PRAM algorithm for connected component computation [173] are simulated on the CGM. Each round of the PRAM algorithm creates “super vertices” by combining adjacent vertices (or super vertices). Each round reduces the number of vertices or super vertices by at least a constant factor. Therefore, after $\log p$ iterations, the number of (super) vertices is reduced to $O(\frac{n}{p})$. Phase 2: Each processor obtains a copy of the $O(\frac{n}{p})$ (super) vertices and stores a subset of the edges (which could be still more than $\frac{n}{p}$ in total). Each processor first builds a spanning forest for its subset of

edges and then $\log p$ “binary merge” steps are performed in which pairs of processors merge their two spanning forests into one spanning forest. After $\log p$ merging steps, we obtain one single spanning forest with at most $O(\frac{n}{p})$ edges.

8.5.2 Performance Evaluation

We have measured the performance of the connected components and spanning forest algorithms on *THOG*, *CGM1* and *ULTRA*. In the following, we present the results of our experiment. For each operation, we measured the performance on *THOG* with $n = 10,000,000$, on *CGM1* with $n = 5,000,000$, and on *ULTRA* with $n = 100,000$.

Figure 8.9: Performance of the connected components algorithm

Figure 8.9 shows the performance of the connected components algorithm on *THOG*, *CGM1* and *ULTRA*. Figure 8.10 shows the performance of the spanning forest algorithm on *THOG*, *CGM1* and *ULTRA*. The only difference between the two

Figure 8.10: Performance of the spanning forest algorithm

methods is that the spanning forest algorithm has to create the spanning forests after the connected components have been identified. Therefore, the times shown in Figures 8.9 and 8.10 are very similar. (Note, the second graph in Figure 8.10 starts only at $p = 2$.) Again, we observe that for all three machines, the communication time is a small, essentially fixed, portion of the total time. The connected component method uses deterministic list ranking. It requires $c \log p$ h -Relation operations with $\log p$ in the range $[1, 5]$. The communication time observed is fairly stable, independent of p , which shows that the $\log p$ factor has little influence on the measured communication time. The entire measured wall clock time is dominated by the computation time and similar to $\frac{1}{p}$.

8.6 CGM_{graph}: Parallel Bipartite Graph Recognition

CGM_{graph} provides a method `isBipartiteGraph (Graph &g, Comm *comm)` for recognizing whether a graph is a bipartite graph. The input is a graph represented as a list of vertices and a list of edges.

8.6.1 Outline of Method

We implemented the bipartite graph recognition algorithm described in [17]. The following is a brief outline of the algorithm. For more details, see [17]. The method proceeds in four steps. Step 1: A spanning forest of the given graph is computed using the method in Section 8.5. Step 2: Using the method in Section 8.4, for each spanning tree in the spanning forest, its Euler tour is computed and for each node, the distance to the root of its spanning tree is determined. Step 3: The nodes are categorized into two groups, the nodes with an odd distance to their root and the nodes with an even distance to their root. Step 4: If there is any edge whose two vertices do not belong to different groups, then the graph is not bipartite; otherwise the graph is bipartite.

8.6.2 Performance Evaluation

We have measured the performance of the bipartite graph recognition algorithm on *THOG*, *CGM1* and *ULTRA*. We measured the performance on *THOG* with $n = 5,000,000$ and $n = 10,000,000$, on *CGM1* with $n = 5,000,000$, and on *ULTRA* with $n = 100,000$.

Figure 8.11 shows the performance of the bipartite graph recognition algorithm on *THOG*, *CGM1* and *ULTRA*. The results mirror the fact that the algorithm is essentially a combination of Euler tour traversal and spanning forest computation. The curves are similar to the former but the amount of communication time is now larger, representing the sum of the two. This effect is particularly strong on *ULTRA* which has the weakest network. Here, the $\log p$ in the number of communication

rounds actually leads to a steadily increasing communication time which, for $p = 9$ starts to dominate the computation time. However, for *THOG* and *CGM1*, the effect is much smaller. For these machines, the communication time is still essentially fixed over the entire range of values of p . The computation time is similar to $\frac{1}{p}$ and determines the shape of the curves for the entire wall clock time. The computation and communication times become equal for larger p but only because of the decrease in computation time.

Figure 8.11: Performance of the bipartite graph recognition algorithm

8.7 Summary

In this chapter, we presented *CGMgraph*, the first integrated library of parallel graph methods for PC clusters solving deterministic list ranking, Euler tour, connected components, spanning forest, and bipartite graph recognition. We also presented *CGMlib*, a library of basic CGM tools such as sorting, prefix sum, one to all broadcast, all to one gather, h -Relation, all to all broadcast, array balancing, and CGM partitioning. In our experiments, we demonstrated the performance of our methods on three different architectures and showed that our library provides good relative parallel speedup and scalability on all three platforms.

Chapter 9

Conclusion

We are interested in developing efficient and practical parallel CGM graph algorithms. This is basically a new area that not many researchers have touched on before. We are particularly interested in graph recognition and maximum matching problems.

We have presented CGM algorithms to recognize bipartite graphs, convex bipartite graphs, and planar graphs, and for finding maximum matching in convex bipartite graphs and trees. We have also presented CGM algorithms for solving the next element search problem and for parallel sorting. Section 9.1 summarizes the performance of these algorithms.

We have also discussed *CGMgraph* and *CGMlib*. *CGMgraph* is the first integrated library of parallel graph methods for PC clusters to solve the deterministic list ranking, Euler tour, connected components, spanning forest, and bipartite graph recognition problems. *CGMgraph* is based on *CGMlib*, a library of basic CGM tools such as sorting, prefix sum, one to all broadcast, all to one gather, *h*-Relation, all to all broadcast, array balancing, and CGM partitioning. In our experiments, we demonstrated the performance of our methods on three different machines and showed that our library provides good relative parallel speedup and scalability on all three platforms.

9.1 Summary of CGM/BSP Algorithms

The following table summarizes the performance of our CGM/BSP algorithms:

CGM Algorithm	Local Computation	Communication Rounds	Speedup
Bipartite Graph Recognition	$O(\frac{n}{p} \log p)$	$O(\log p)$	$\Omega(\frac{p}{g \log p})$
Convex Bipartite Graph Recognition	$O(\frac{n}{p} \log^2 p)$	$O(\log^2 p)$	$\Omega(\frac{p}{g \log^2 p})$
Planar Graph Recognition	$O(\frac{n}{p} \log^2 p)$	$O(\log^2 p)$	$\Omega(\frac{p}{g \log^2 p})$
Maximum Matching in Convex Bipartite Graphs	$O(T_s(\frac{n_A}{p}, \frac{n_B}{p}) + \frac{n_A}{p} \log p)$	$O(\log p)$	$\Omega(\frac{p}{g \log p})$
Maximum Matching in Trees	$O(\frac{n}{p} \log p)$	$O(\log^2 p)$	$\Omega(\frac{p}{g \log^2 p})$
Next Element Search	$O(\frac{n}{p} \log n)$	$O(1)$	$\Omega(\frac{p}{g})$
Parallel Sorting	$O(\frac{T_s(n)}{p})$	$O(1)$	$\Omega(\frac{p}{g})$

By Lemma 1, our CGM algorithms can also be implemented on BSP with the following performance:

BSP Algorithm	Local Computation	Communication Cost	Speedup
Bipartite Graph Recognition	$O(\frac{n}{p} \log p)$	$O(g \frac{n}{p} \log p)$	$\Omega(\frac{p}{g \log p})$
Convex Bipartite Graph Recognition	$O(\frac{n}{p} \log^2 p)$	$O(g \frac{n}{p} \log^2 p)$	$\Omega(\frac{p}{g \log^2 p})$
Planar Graph Recognition	$O(\frac{n}{p} \log^2 p)$	$O(g \frac{n}{p} \log^2 p)$	$\Omega(\frac{p}{g \log^2 p})$
Maximum Matching in Convex Bipartite Graphs	$O(T_s(\frac{n_A}{p}, \frac{n_B}{p}) + \frac{n_A}{p} \log p)$	$O(g \frac{n_A + n_B}{p} \log p)$	$\Omega(\frac{p}{g \log p})$
Matching in Trees	$O(\frac{n}{p} \log p)$	$O(g \frac{n}{p} \log^2 p)$	$\Omega(\frac{p}{g \log^2 p})$
General Next Element Search	$O(\frac{n}{p} \log n)$	$O(g \frac{n}{p})$	$\Omega(\frac{p}{g})$
Parallel Sorting	$O(\frac{T_s(n)}{p})$	$O(g \frac{n}{p})$	$\Omega(\frac{p}{g})$

9.2 Future Work

Graph theory is a broad topic in computer science. There are many books and articles devoted to graph theory. Much effort has been put into the study of sequential and PRAM graph algorithms. Coarse grained graph algorithms have started to attract scholars' interests recently. There are still many areas in the field of CGM graph algorithms to be studied. Here are some of the possible areas:

- Matching in general graphs
- Graph coloring
- Perfect eliminating order
- Independent sets
- Graph isomorphism
- Graph partitioning

Matching in General Graphs

Israeli and Shiloach [106] presented a PRAM algorithm for finding maximal matching in general graphs with m edges and n vertices in $O(\log^3 n)$ time using $O(m)$ processors.

Maximum matching in a graph $G = (V, E)$ is a matching of G of which the size (number of edges) is maximum. Karloff [113] showed that maximum matching can be found using a randomized PRAM algorithm in $O(\log^2 n)$ time using $O(n^2 M(n))$ processors. Here $M(n)$ is the number of arithmetic operations required to multiply two $n \times n$ matrices. The current bound for $M(n)$ is $O(n^{2.396})$.

Perfect matching in a graph $G = (V, E)$ is a matching of G of which each vertex is incident to exactly one edge. Lev, Pippenger and Valiant [133] presented a PRAM algorithm for finding perfect matching in a Δ -degree bipartite graph in $O(\log^2 n \log \Delta)$ time using $O(m)$ processors.

For general graphs, Mulmuley, Vazirani and Vazirani [154] presented a PRAM algorithm for finding perfect matching in $O(\log^2 n)$ time using $O(n^{3.5}m)$ processors.

Galil and Pan [71] showed that perfect matching can be found using a randomized PRAM algorithm in $O(\log^3 n)$ time using $O(nM(n))$ processors.

Our goal is to design efficient CGM algorithms for general graph matching that use only $O(\log p)$ communication rounds.

Graph Coloring

The problem of graph coloring is to assign colors to edges of a graph, such that all edges incident to the same vertex have different colors. This problem is very much related to the graph matching problems. A generic graph coloring algorithm can be described as follows:

- find a maximum matching;
- assign a (unique) color to all the matched edges;
- remove all the matched edges;
- remove all the dangling vertices (vertices that are not incident to any edge);
- recursive solve the problem on the remaining graph.

The coloring problem has another restriction that we want to minimize the number of colors. Obviously, a graph of degree Δ will need at least Δ colors. Very often, this is not achievable, and we have to settle for $(1 + \epsilon)\Delta$ where ϵ is a small positive real number $\epsilon > 0$. Many algorithms concentrate on getting ϵ as small as possible. Panconesi and Srinivasan [163] presented a randomized algorithm for coloring an n -vertex graph of maximum Δ degrees using at most $(1.6 + \epsilon)\Delta + 0.4 \log^{2+\delta} n$ colors with high probability for any fixed $\delta, \epsilon > 0$.

The method outlined above cannot be easily parallelized. Our goal is to develop efficient coarse grained parallel algorithms for graph coloring that require at most $O(\log^2 p)$ communication rounds.

Perfect Eliminating Order

The perfect eliminating order (peo) is a very useful tool for chordal graphs. The peo was discovered by Fulkerson and Gross [69]. Dirac [67] had proved that every chordal graph has a simplicial node (a node of which the neighbors form a clique.) Fulkerson

and Gross observed that since every induced subgraph of a chordal graph is also a chordal graph, deletion of a vertex and its incident edges will result in a chordal graph. Consequently, they proposed an “elimination” process: repeatedly find a simplicial node and delete it until all nodes have been deleted or no remaining node is simplicial. According to Dirac’s theorem, this process will delete every node of a chordal graph. In fact, Fulkerson and Gross showed that a graph is chordal if and only if this process deletes every node. Therefore, the peo process can be used to determine if a graph is chordal. Klein [123] presented a PRAM algorithm for finding the peo in $O(\log^2 n)$ time using $O(n)$ processors.

Our goal is to design coarse grained parallel algorithms for finding peo. The running time for the CGM algorithms should be no more than $O(\frac{n \log^2 n}{p})$ and there should be at most $O(\log^2 p)$ communication rounds.

Independent Sets

An independent set of a graph is a set of vertices in which no two vertices are joined by an edge. An independent set is maximal if it is not properly contained in another independent set. A maximum independent set is the maximal independent set that has the largest cardinality. Dhall, Liang, and Lakshmivarahan [63] presented an algorithm for finding maximum independent set for chordal graphs in $O(n^2)$ time using $O(n^2)$ space.

Our goal is to design coarse grained algorithms for finding maximum independent sets. The running time for the CGM algorithms should be no more than $O(\frac{n^2}{p})$ and there should be at most $O(\log^2 p)$ communication rounds.

Graph Isomorphism

Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, if there exists a function f such that f maps one-to-one from V_1 to V_2 , and $(u, v) \in E_1$ if and only if $(f(u), f(v)) \in E_2$, then G_1 and G_2 are said to be isomorphic. JáJá and Kosaraju [109] presented a parallel algorithm that solves the planar graph isomorphism problem in $O(\log^2 n)$ time using $O(n^2)$ processors. Later, Gazit [73] presented a deterministic parallel

algorithm for planar graphs isomorphism problem in $O(\log^3 n)$ time using $O(\frac{n^{1.5}}{\log^2 n})$ processors.

Our goal is to design coarse grained algorithms for recognizing planar graph isomorphism. The running time for the CGM algorithms should be no more than $O(\frac{n^{1.5}}{p})$ and there should be at most $O(\log^2 p)$ communication rounds.

Graph Partitioning

The problem of graph partitioning is to partition a graph into subgraphs by removing some edges. This is an NP-complete problem [10]. However, graph partitioning is an important problem, and therefore, heuristic methods have been developed to solve this problem. The following are some of the heuristics that have been used:

- genetic algorithms [10]
- hill climbing [10]
- by node separators [141]
- spectral method [91]
- linear programming [162]

Our goal is to parallelize some of these heuristics or to find some new heuristics that is more suitable for parallel algorithms.

Implementation

Both *CGMlib* and *CGMgraph* are currently in beta state. Despite extensive work on performance tuning, there are still many possibilities for fine-tuning the code in order to obtain further improved performance. Of course, adding more parallel graph algorithm implementations to *CGMgraph* is an important task for the near future. Other possible extensions include porting *CGMlib* and *CGMgraph* to other communication libraries, e.g. PVM and OpenMP. We also plan to integrate *CGMlib* and *CGMgraph* with other libraries, in particular the LEDA library [136].

Bibliography

- [1] S. Akl and K. Lyons. *Parallel Computational Geometry*. Prentice Hall, 1993.
- [2] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.
- [3] Farid Alizadeh. A sublinear-time randomized parallel algorithm for the maximum clique problem in perfect graphs. In Alok Aggarwal, editor, *Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '90)*, pages 188–194, San Francisco, CA, USA, January 1990. SIAM.
- [4] M. Allen and B. Wilkinson. *Parallel Programming - Techniques and applications using networked workstations and parallel computers*. Prentice Hall, 1999.
- [5] R. Anderson and L. Snyder. A comparison of shared and nonshared memory models of computation. In *IEEE*, volume 79(4), pages 480 – 487, 1993.
- [6] M. C. Andrews, M. J. Atallah, D. Z. Chen, and D. T. Lee. Parallel algorithms for maximum matching in interval graphs. In *Proceedings of the 9th International Symposium on Parallel Processing (IPPS'95)*, pages 84–92, Santa Barbara, CA, USA, April 1995. IEEE Computer Society Press.
- [7] F. Bauernöppel, E. Kranakis, D. Krizane, A. Maheshwari, J.-R. Sack, and J. Urrutia. An improved maximum matching algorithm in a permutation graph. Technical Report TR-95-12, Carleton University, 1995.
- [8] F. Bauernöppel, E. Kranakis, D. Krizane, A. Maheshwari, J.-R. Sack, and J. Urrutia. Planar stage graphs: Characterizations and applications. *Theoretical Computer Science*, 175:239–255, 1997.

- [9] J. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, 29(7):571–576, 1980.
- [10] P. Bessire and E. G. Talbi. A parallel genetic algorithm for the graph partitioning problem. In *Proceedings of the 1991 International conference on Supercomputing*, pages 312 – 320, 1991.
- [11] I. S. Bhandari, C. M. Krishna, and D. P. Siewiorek. A parallel algorithm for coloring graphs. In *8th International Conference on Distributed Computing Systems*, pages 326–333, San Jose, Ca., USA, June 1988. IEEE Computer Society Press.
- [12] Gianfranco Bilardi, Kieran T. Herley, Andrea Pietracaprina, Geppino Pucci, and Paul Spirakis. Bsp vs logp. In *Proceedings of the 8th annual ACM symposium on Parallel algorithms and architectures*, pages 25–32, 1996.
- [13] K. Booth and G Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *Journal of Computer and System Sciences*, 13:335–379, 1976.
- [14] P. Bose, A. Chan, F. Dehne, and M. Latzel. Coarse grained parallel maximum matching in convex bipartite graphs. In *Proceeding of 13th International Parallel Processing Symposium (IPPS’99)*, pages 125–129, 1999.
- [15] T. N. Bui and C. Jones. Parallel algorithms for partitioning simple classes of graphs. In Pen-Chung Yew, editor, *Proceedings of the 1990 International Conference on Parallel Processing. Volume 3: Algorithms and Architectures*, pages 150–153, Urbana-Champaign, IL, August 1990. Pennsylvania State University Press.
- [16] E. Cceres, A. Chan, F. Dehne, and G. Prencipe. Coarse grained parallel algorithms for detecting convex bipartite graphs. Technical report, School of Computer Science, Carleton University, 1999.
- [17] E. Caceres, A. Chan, F. Dehne, and G. Prencipe. Coarse grained parallel algorithms for detecting convex bipartite graphs. In *Proceeding 26th Workshop on Graph-Theoretic Concepts in Computer Science (WG 2000)*, pages 83–94, 2000.
- [18] E. Caceres, A. Chan, F. Dehne, and S. Song. Coarse grained parallel graph planarity testing. In *Proceeding International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’2000)*, pages 1589–1595, 2000.

- [19] E. Cáceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S.W. Song. Efficient parallel graph algorithms for coarse grained multicomputers and bsp. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Proceedings ICALP '97 - 24th International Colloquium on Automata, Languages, and Programming*, volume 1256 of *Lecture Notes in Computer Science*, pages 390–400. Springer Verlag, 1997.
- [20] E. Caceres, F. Dehne, A. Ferreira, A. Roncato, and S. W. Song. Efficient parallel graph algorithms for coarse grained multicomputers and bsp. *Algorithmica*, 33(2):183–200, 2002.
- [21] Yuval Caspi and Eliezer Dekel. Edge coloring series parallel graphs. *Journal of Algorithms*, 18(2):296–321, 1995.
- [22] I. Cederbaum, S. Even, and A. Lempel. *An Algorithm for Planarity Testing of Graphs*, pages 215–232. Theory of Graphs: International Symposium: Rome, July, 1966. Gordon and Breach, New York, 1967.
- [23] A. Chan and F. Dehne. A note on coarse grained parallel integer sorting. Technical Report TR-98-06, School of Computer Science, Carleton University, 1998. Available at URL <http://www.scs.carleton.ca/>.
- [24] A. Chan and F. Dehne. A note on coarse grained parallel integer sorting. In *Proceedings of 13th Annual International Symposium on High Performance Computers (HPCS'99)*, pages 261–267, 1999.
- [25] A. Chan and F. Dehne. A coarse grained parallel algorithm for maximum weight matching in trees. In *Proceedings of 12th IASTED International Conference Parallel and Distributed Computing and Systems (PCDS 2000)*, pages 134–138, 2000.
- [26] A. Chan and F. Dehne. CGMgraph/CGMlib: Implementing and testing CGM graph algorithms on pc clusters. 2003, submitted.
- [27] A. Chan, F. Dehne, and A. Rau-Chaplin. Coarse grained parallel geometric search. *Journal of Parallel and Distributed Computing*, 57(2):224–236, 1999.
- [28] Danny Z. Chen and D. T. Lee. Solving the all-pair shortest path problem on interval and circular-arc graphs. In Howard Jay Siegel, editor, *Proceedings of the 8th International Symposium on Parallel Processing*, pages 224–228, Cancún, Mexico, April 1994. IEEE Computer Society Press.

- [29] L. Chen. Graph isomorphism and identification matrices: Parallel algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):308 ff, March 1996.
- [30] L. Chen and Y. Yesha. Parallel recognition of the consecutive ones property with applications. *Journal of Algorithms*, 12(3):375–392, 1991.
- [31] B. S. Chlebus, K. Diks, and T. Radzik. Testing isomorphism of outerplanar graphs in parallel. In M. P. Chytil, L. Janiga, and V. Koubek, editors, *Proceedings of the Conference on Mathematical Foundations of Computer Science*, volume 324 of *LNCS*, pages 220–230, Carlsbad, Czechoslovakia, August 1988. Springer.
- [32] F. Clover. Maximum matching in convex bipartite graphs. *Naval Res. Logist. Quarterly*, 14:313–316, 1967.
- [33] R. Cole. Parallel merge sort. *SIAM Journal of Computing*, 17(4):770–785, 1988.
- [34] Richard Cole and Ofer Zajicek. An asynchronous parallel algorithm for undirected graph connectivity. *Journal of Algorithms*, 18(1):50–97, 1995.
- [35] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: Towards a realistic model of parallel computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.
- [36] D.E. Culler, R.M. Karp, D. Patterson, A. Sahay, E.E. Santos, K.E. Schauser, R. Subramonian, and T. von Eicken. Logp: A practical model of parallel computation. *Communications of the ACM*, 39(11):78 – 85, November 1996.
- [37] E. Dahlhaus and M. Karpinski. A fast parallel algorithm for computing all maximal cliques in a graph and the related problems. In J. H. Reif, editor, *Proceedings of the 3rd Aegean Workshop on VLSI Algorithms and Architectures*, volume 319 of *LNCS*, pages 139–144, Corfu, Greece, June-July 1988. Springer.
- [38] Elias Dahlhaus, Peter Hajnal, and Marek Karpinski. Optimal parallel algorithm for the hamiltonian cycle problem on dense graphs. Technical Report TR-89-042, ICSI, 1989.
- [39] Elias Dahlhaus, Péter Hajnal, and Marek Karpinski. On the parallel complexity of Hamiltonian cycle and matching problem on dense graphs. *Journal of Algorithms*, 15(3):367–384, 1993.

- [40] Elias Dahlhaus and Marek Karpinski. On the parallel complexity of matching for chordal and path graphs. Technical Report 8512 - CS, Institut für Informatik der Universität Bonn, 1987.
- [41] Elias Dahlhaus and Marek Karpinski. Parallel complexity for matching restricted to degree defined graph classes. Technical Report 859 - CS, Institut für Informatik der Universität Bonn, 1987.
- [42] Elias Dahlhaus and Marek Karpinski. Parallel construction of perfect matchings and hamiltonian cycles on dense graphs. Technical Report 8518 - CS, Institut für Informatik der Universität Bonn, 1987.
- [43] Elias Dahlhaus and Marek Karpinski. Parallel construction of perfect matchings and Hamiltonian cycles on dense graphs. *Theoretical Computer Science*, 61(2-3):121-136, 1988.
- [44] Elias Dahlhaus and Marek Karpinski. An efficient parallel algorithm for the minimal elimination ordering (meo) of an arbitrary graph. Technical Report TR-89-024, ICSI, 1989.
- [45] Elias Dahlhaus and Marek Karpinski. An efficient parallel algorithm for the minimal elimination ordering (MEO) of an arbitrary graph. *Theoretical Computer Science*, 134(2):493-528, 1994.
- [46] Elias Dahlhaus and Marek Karpinski. On the computational complexity of matching on chordal and strongly chordal graphs. Technical Report TR-94-043, ICSI, 1994.
- [47] Elias Dahlhaus and Marek Karpinski. The parallel and sequential complexity of matching on chordal and strongly chordal graphs. Technical Report 85129 - CS, Institut für Informatik der Universität Bonn, 1995.
- [48] Elias Dahlhaus, Marek Karpinski, and Andrzej Lingas. A parallel algorithm for maximum matching in planar graphs. Technical Report TR-89-018, ICSI, 1989.
- [49] Elias Dahlhaus, Hajnal Péter, and Marek Karpinski. Optimal parallel algorithm for the hamiltonian cycle problem on dense graphs. Technical Report 8527 - CS, Institut für Informatik der Universität Bonn, 1988.
- [50] Fabrizio d'Amore, Paolo G. Franciosa, and Alberto Marchetti-Spaccamela. More general parallel tree contraction : register allocation and broadcasting in a tree. In *Intern.*

- Workshop on Graph-theoretic concepts in computer science (WG-22) : Cadenabbia, Italy, June 12-14, 1996; proceedings*, volume LNCS 1197, pages 126–140S. Springer, 1996.
- [51] S. K. Das and C. C. Y. Chen. A new parallel algorithm for breadth-first search on interval graphs. In Viktor K. Prasanna and Larry H. Canter, editors, *Proceedings of the 6th International Parallel Processing Symposium*, pages 150–155, Beverly Hills, CA, March 1992. IEEE Computer Society Press.
 - [52] S. Dascal and U. Vishkin. Experiments with list ranking on explicit multi-threaded (xmt) instruction parallelism. In *3rd Workshop on Algorithms Engineering (WAE-99)*, volume 1668 of *Lecture Notes in Computer Science*, page 43 ff, 1999.
 - [53] Ali Dasdan and Cevdet Aykanat. Efficient multipleway graph partitioning algorithms. In D. H. Bailey, P. E. Bjørstad, J. E. Gilbert, M. V. Mascagni, R. S. Schreiber, H. D. Simon, V. J. Torczon, and L. T. Watson, editors, *Proceedings of the 27th Conference on Parallel Processing for Scientific Computing*, pages 637–638, San Francisco, CA, USA, February 1995. SIAM Press.
 - [54] N. Dean, T. Hsu, and V. Ramachandran. Parallel implementation of algorithms for finding connected components in graphs. In *AMS/DIMACS Parallel Implementation Challenge Workshop III*, 1997.
 - [55] F. Dehne. Guest editor’s introduction, Special issue on coarse grained parallel algorithms. *Algorithmica*, 24(3/4):173–176, 1999.
 - [56] F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. Kokhar. A randomized parallel 3d convex hull algorithm for coarse grained multicomputers. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures (SPAA ’95)*, pages 27–33, 1995.
 - [57] F. Dehne, A. Fabri, and C. Kenyon. Scalable and architecture independent parallel geometric algorithms with high probability optimal time. In *Proceedings of 6th IEEE Symposium on Parallel and Distributed Processing*, pages 586–593, 1994.
 - [58] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric algorithms for coarse grained multicomputers. In *Proceedings of ACM 9th Annual Computational Geometry*, pages 298–307, 1993.
 - [59] F. Dehne and S. W. Song. Randomized parallel list ranking for distributed memory multiprocessors. In *Asian Computer Science Conference (ASIAN ’96)*, volume 1179 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 1996.

- [60] E. Dekel and S. Sahni. A parallel matching algorithm for convex bipartite graphs. In *International Conference on parallel Processing*, pages 178–184, 1982.
- [61] Anders Dessmark, Carsten Dorgerloh, Andrzej Lingas, and Jürgen Wirtgen. Ultrafast randomized parallel construction- and approximation algorithms for spanning forests in dense graphs. Technical Report 85180 - CS, Institut für Informatik der Universität Bonn, 1997.
- [62] O. Devillers and A. Fabri. Scalable algorithms for bichromatic line segment intersection problems on coarse grained multicomputers. *International Journal of Computational Geometry and Applications*, 6:487–506, 1996.
- [63] S.K. Dhall, S. Lakshmivarahn, and Y.D. Liang. On the problem of finding all maximum weight independent sets in interval and circular-arc graphs. In *Proceedings of the 1991 Symposium on Applied Computing*, pages 465 – 470, 1991.
- [64] R. Diestel. *Graph Theory*. Springer-Verlag, 1997.
- [65] K. Diks. A fast parallel algorithm for six-colouring of planar graphs. In B. Rován J. Gruska and J. Wiedermann, editors, *Proceedings of the 12th Symposium on Mathematical Foundations of Computer Science*, volume 233 of *LNCS*, pages 273–282, Bratislava, Czechoslovakia, August 1986. Springer.
- [66] K. Diks, T. Hagerup, and W. Rytter. Optimal parallel algorithms for the recognition and colouring outerplanar graphs. In A. Kreczmar and G. Mirkowska, editors, *Proceedings of the Conference on Mathematical Foundations of Computer Science*, volume 379 of *LNCS*, pages 207–217. Springer, August–September 1989.
- [67] G.A. Dirac. On rigid circuit graphs. In *Abh. Math. Sem.*, pages 71 – 76, 1961.
- [68] A. Edenbrandt. Chordal graph recognition is in nc. *Information Processing Letters*, 24, 1987.
- [69] D Fulkerson and O. Gross. Incidence matrices and interval graphs. In *Pac. J. math*, volume 15, pages 835 – 855, 1965.
- [70] Martin Fúrer and Balaji Raghavachari. An efficient parallel algorithm for finding Hamiltonian cycles in dense directed graphs. *Journal of Algorithms*, 18(2):203–220, 1995.

- [71] Z. Galil and V. Pan. Improved processor bounds for combinatorial problems in RNC. *Combinatorica*, 8:189–200, 1988.
- [72] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. In *27th Annual Symposium on Foundations of Computer Science*, pages 492–501, Toronto, CAN, October 1986. IEEE Computer Society Press.
- [73] H. Gazit. A deterministic parallel algorithm for planar graphs isomorphism. In IEEE, editor, *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 723–732, San Juan, Porto Rico, October 1991. IEEE Computer Society Press.
- [74] H. Gazit. An optimal randomized parallel algorithm for finding connected components in a graph. *SIAM J. Computing*, 20, 1991.
- [75] H. Gazit and G. L. Miller. A parallel algorithm for finding a separator in planar graphs. In Ashok K. Chandra, editor, *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 238–248, Los Angeles, CA, October 1987. IEEE Computer Society Press.
- [76] H. Gazit and J. Reif. A randomized parallel algorithm for planar graph isomorphism. In ACM-SIGACT; ACM-SIGARCH, editor, *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 210–219, Island of Crete, Greece, July 1990. ACM Press.
- [77] A. Gibbons and W. Rytter. Fast parallel algorithms for optimal edge-colouring of some tree-structured graphs. In O. B. Lupanov L. Budach, R. G. Bukharajev, editor, *Proceedings of the International Conference on Fundamentals of Computation Theory (FCT '87)*, volume 278 of *LNCS*, pages 155–162, Kazan, USSR, June 1987. Springer.
- [78] P. B. Gibbons, Y. Matias, and V. Ramachandran. The queue-read queue-write asynchronous PRAM model. *Theoretical Computer Science*, 196(1–2):3–29, 1998.
- [79] Phillip B. Gibbons. What good are shared-memory models? In *ICPP Workshop*, pages 103–114, 1996.
- [80] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. Can shared-memory model serve as a bridging model for parallel computation? In *ACM Symposium on Parallel Algorithms and Architectures*, pages 72–83, 1997.

- [81] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. The queue-read queue-write PRAM model: Accounting for contention in parallel algorithms. *SIAM Journal on Computing*, 28(2):733–769, 1999.
- [82] J. R. Gilbert and E. Zmijewski. A parallel graph partitioning algorithm for a message-passing multiprocessor. In E. N. Houstis, T. S. Papatheodorou, and C. D. Polychronopoulos, editors, *Proceedings of the 1st International Conference on Supercomputing*, volume 297 of *LNCS*, pages 498–513, Athens, Greece, June 1987. Springer.
- [83] Andrew V. Goldberg, Serge A. Plotkin, and Pravin M. Vaidya. Sublinear-time parallel algorithms for matching and related problems. *Journal of Algorithms*, 14(2):180–213, 1993.
- [84] M. Goldberg and T. Spencer. A new parallel algorithm for the maximal independent set problem. *SIAM J. Computing*, 18, 1989.
- [85] M. Goodrich. Communication efficient parallel sorting. In *ACM Symposium on Theory of Computing (STOC)*, 1996.
- [86] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison Wesley, 2003.
- [87] Brian Grayson, Michael Dahlin, and Vijaya Ramachandran. Emulations between qsm, bsp and logp: A framework for general-purpose parallel algorithm design. Technical Report TR98-22, CS dept., Univ. of Texas at Austin, 11 1998.
- [88] E. F. Grove. Connected components and the interval graph. In *Proceedings of the 4th Annual Symposium on Parallel Algorithms and Architectures*, pages 382–391, San Diego, CA, USA, June 1992. ACM Press.
- [89] L. K. Grover. Fast parallel algorithms for bipartite matching. In E. Balas, G. Cornuéjols, and R. Kannan, editors, *Proceedings of the 2nd Integer Programming and Combinatorial Optimization Conference*, pages 367–384, Pittsburgh, PA, May 1992. Carnegie Mellon University.
- [90] S. Guattery and G. Miller. A contraction procedure for planar directed graphs. In *Proceedings of the 4th Annual Symposium on Parallel Algorithms and Architectures*, pages 431–441, San Diego, CA, USA, June 1992. ACM Press.

- [91] Stephen Guattery and Gary L. Miller. On the performance of spectral graph partitioning methods. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, pages 233 – 242, 1995.
- [92] Frank Guillaume, Tillmann Seidel, and Jop F. Sibeyn. Practical parallel list ranking. *Journal of Parallel and Distributed Computing*, 56(2):156–180, 1999.
- [93] T. Hagerup. Planar depth-first search in $o(\log)$ parallel time. *SIAM J. Computing*, 19, 1990.
- [94] T. Hagerup, M. Chrobak, and K. Diks. Parallel 5-colouring of planar graphs. In Thomas Ottmann, editor, *Proceedings of the 14th International Colloquium on Automata, Languages, and Programming*, volume 267 of *LNCS*, pages 304–313, Karlsruhe, FRG, July 1987. Springer.
- [95] T. Hagerup, M. Chrobak, and K. Diks. Optimal parallel 5-colouring of planar graphs. *SIAM J. Computing*, 18, 1989.
- [96] Torben Hagerup. Parallel 5-colouring of planar graphs. Technical Report 10/86, Universität des Saarlandes, Saarbrücken, 1986.
- [97] Torben Hagerup. Optimal parallel computation of minimum spanning forests in planar graphs. Technical Report 11/1990, Oldenbourg-Verl., Saarbrücken, 1990.
- [98] Torben Hagerup, Marek Chrobak, and Krzysztof Diks. Optimal parallel 5-colouring of planar graphs. *SIAM Journal on Computing*, 18(2):288–300, 1989.
- [99] F. Harary. *Graph Theory*. Addison Wesley, 1969.
- [100] X. He. Efficient parallel and sequential algorithms for 4-coloring perfect planar graphs. *Algorithmica*, 5, 1990.
- [101] X. He and Y. Yeshua. Parallel recognition and decomposition of two terminal series parallel graphs. *Information and Computation*, 75:15–38, 1987.
- [102] Xin He. A nearly optimal parallel algorithm for constructing maximal independent set in planar graphs. *Theoretical Computer Science*, 61(1):33–47, 1988.
- [103] J. Hopcroft and R. Tarjan. An efficient algorithm for graph planarity. *Journal of Associate computing Mach...*, 21:549–328, 1974.

- [104] J. Hopcroft and R. Karp. An $n^{\frac{5}{2}}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal in Computing*, 2:225–231, 1973.
- [105] O. H. Ibarra and Q. Zheng. On the shortest path problems for permutation graphs. In V.K. Prasanna, editor, *Proceedings of the 7th International Parallel Processing Symposium*, pages 198–204, Newport Beach, CA, April 1993. IEEE Computer Society Press.
- [106] A. Israeli and Y Shiloach. An improved parallel algorithm for maximal matching. *Information Processing Letter*, 22(2):57–60, 1986.
- [107] Kazuo Iwama and Yahiko Kambayashi. A simpler parallel algorithm for graph connectivity. *Journal of Algorithms*, 16(2):190–217, 1994.
- [108] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
- [109] J. JáJá and S.R. Kosaraju. Parallel algorithms for planar graph isomorphism and related problems. *IEEE Transactions on Circuits and Systems*, 35(3):304 – 311, 1988.
- [110] J. JáJá and J. Simon. Parallel algorithms in graph theory: Planarity testing. *SIAM Journal of Computing*, 11(2):313–328, 1982.
- [111] Ming Kao, Marek Karpinski, and Andrzej Lingas. Nearly optimal parallel algorithm for maximum matching in planar graphs. Technical Report 8549 - CS, Institut für Informatik der Universität Bonn, 1990.
- [112] T. W. Kao and S. J. Horng. Computing k-vertex connectivity on an interval graph. In Jagdish Chandra, editor, *Proceedings of the 23rd International Conference on Parallel Processing. Volume 3: Algorithms and Applications*, pages 218–221. CRC Press, August 1994.
- [113] H. Karloff. A randomized parallel algorithm for the odd set cover problem. *Combinatorica*, 6:387–391, 1986.
- [114] R. M. Karp and A. Wigderson. A fast parallel algorithm for the maximal indepent set problem. *Journal of the ACM*, 32, 1985.
- [115] Richard M. Karp, Abhijit Sahay, Eunice E. Santos, and Klaus E. Schauser. Optimal broadcast and summation in the logp model. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 142–153, 1993.

- [116] Marek Karpinski and Wojciech Rytter. *Fast parallel algorithms for graph matching problems*, volume 9 of *Oxford lecture series in mathematics and its applications*. Clarendon Press, 1998.
- [117] G. Karypis and V. Kumar. Multilevel graph partitioning schemes. In *International Conference on Parallel Processing*, pages 113–122, University of Illinois at Urbana-Champaign, Ill., USA, August 1995. CRC Press.
- [118] Z. Kedem, K. Palem, G. Pantziou, P. Spirakis, and C. Zaroliagis. Fast parallel algorithms for coloring random graphs. In Gunther Schmidt and Rudolf Berghammer, editors, *Proceedings on Graph-Theoretic Concepts in Computer Science (WG '91)*, volume 570 of *LNCS*, pages 135–147, Fischbachau, GER, June 1991. Springer.
- [119] Pierre Kelsen. Fast parallel matching in expander graphs. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 293–299, Velen, FRG, June 1993. ACM Press.
- [120] Pierre Kelsen. An optimal parallel algorithm for maximal matching. In *Information Processing Letters*, volume 52, pages 223–228, Amsterdam, 1994. Elsevier.
- [121] Myung-Ho Kim, Chang-Sung Jeong, and Myung-Soo Kim. An optimal parallel matching algorithm for a convex bipartite graph on a mesh-connected computer. In Howard Jay Siegel, editor, *Proceedings of the 8th International Symposium on Parallel Processing*, pages 229–233, Cancún, Mexico, April 1994. IEEE Computer Society Press.
- [122] P. Klein. An efficient parallel algorithm for planarity. Master's thesis, MIT, 1986.
- [123] P. Klein. Efficient parallel algorithms for chordal graphs. In *29th Annual Symposium on Foundations of Computer Science*, pages 150 – 161, 1988.
- [124] P. Klein. *Efficient Parallel Algorithms for Planar, Chordal, and Interval Graphs*. PhD thesis, MIT, 1988.
- [125] P. Klein. Efficient parallel algorithms for chordal graphs. In *Proceedings of 29th Symposium on the Foundations of Computer Science (FOCS)*, pages 150–161, 1989.
- [126] P. Klein. *Parallel Algorithms for Chordal Graphs*, pages 341–407. Synthesis of Parallel Algorithms. Morgan Kaufmann Publishers, 1993.

- [127] P. Klein and J. Reif. An efficient parallel algorithm for planarity. *Journal of Computer and System Sciences*, 37:190–246, 1988.
- [128] Ludek Kucera. Coloring k-colorable graphs in constant expected parallel time. Technical Report 93-110, Max-Planck-Institut für Informatik, 1993.
- [129] S. Kumar, S. Das, and R. Biswas. Graph partitioning for parallel applications in heterogeneous grid environments. In *16th International Parallel and Distributed Processing Symposium*, pages 66–66. IEEE, 2002.
- [130] Isabelle Gurin Lassous, Jens Gustedt, and Michel Morvan. Feasability, portability, predictability and efficiency : Four ambitious goals for the design and implementation of parallel coarse grained graph algorithms. Technical Report RR-3885, INRIA, <http://www.inria.fr/rrrt/rr-3885.html>.
- [131] C. H. Lee, M. Kim, and C. J. Park. An efficient k-way graph partitioning algorithm for task allocation in parallel computing systems. In P. A. Ng, C. V. Ramamoorthy, L. C. Seifert, and R. T. Yeh, editors, *Proceedings of the First International Conference on Systems Integration*, pages 748–751, Morristown, NJ, April 1990. IEEE Computer Society Press.
- [132] F. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann, 1992.
- [133] A. Lev, N. Pippenger, and L. Valiant. A fast parallel algorithm for routing in permutation networks. *IEEE Transactions on Computers*, C30(2):93–100, 1981.
- [134] C. Levcopoulos, A. Lingas, and O. Petersson. Optimal parallel algorithms for testing isomorphism of trees and outerplanar graphs. In *Foundations of Software Technology and Theoretical Computer Science.*, pages 204–214, Bangalore, India, December 1990. Springer.
- [135] A. Levitin. *Introduction to The Design and Analysis of Algorithms*. Addison Wesley, 2003.
- [136] LEDA library. <http://www.algorithmic-solutions.com/>.
- [137] A. Lingas. Subgraph isomorphism for connected graphs of bounded valence and bounded separator is in NC. In *International Conference on Parallel Processing, Vol. 3: Algorithms and Applications*, pages 304–307, University Park, Pa., USA, August 1988. The Pennsylvania State University.

- [138] A. Lingas and A. Proskurowski. Fast parallel algorithms for the subgraph homeomorphism and the subgraph isomorphism problem for classes of planar graphs. In Kesav V. Nori, editor, *Proceedings of the 7th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 287 of *LNCS*, pages 79–94, Pune, India, December 1987. Springer.
- [139] Andrzej Lingas and Andrzej Proskurowski. On parallel complexity of the subgraph homeomorphism and the subgraph isomorphism problem for classes of planar graphs. *Theoretical Computer Science*, 68(2):155–173, 1989.
- [140] W. Lipski and F. Preparata. Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems. *Acta Informatica*, 15:329–346, 1981.
- [141] Joseph W. H. Liu. A graph partitioning algorithm by node separators. *ACM Transactions Mathematical Software*, 15(3):198 – 219, September 1989.
- [142] S. Lor, H. Shen, and P. Maheshwari. Divide-and-conquer minimal-cut bisectioning of task graphs. In *Proceedings of the 1st International Conference on Massively Parallel Computing Systems*, pages 155–162, Ischia, Italy, May 1994. IEEE Computer Society Press.
- [143] E. M. Luks. Parallel algorithms for permutation groups and graph isomorphism. In *27th Annual Symposium on Foundations of Computer Science*, pages 292–302, Toronto, CAN, October 1986. IEEE Computer Society Press.
- [144] Y. D. Lyuu and E. Schenfeld. Parallel graph contraction with applications to a reconfigurable parallel architecture. In Jagdish Chandra, editor, *Proceedings of the 23rd International Conference on Parallel Processing. Volume 3: Algorithms and Applications*, pages 258–265. CRC Press, August 1994.
- [145] Philip D. MacKenzie and Quentin F. Stout. Optimal parallel construction of hamiltonian cycles and spanning trees in random graphs. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 224–229, Velen, FRG, June 1993. ACM Press.
- [146] Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search (eds) and st-numbering in graphs. *Theoretical Computer Science*, 47:277–298, 1986.
- [147] K. Mehlhorn. *Multi-Dimensional searching and Computational Geometry*. Number 3 in Data Structures and Algorithms. Spring Verlag, 1984.

- [148] R. Merris. *Graph Theory*. John Wiley & Sons, 2001.
- [149] R. Michelson, L. Smith, E. Williams, and B. Yantis. Parallel graph reduction on a supercomputer: A status report. In Robert M. Keller Joseph H. Fasel, editor, *Proceedings of a Workshop on Graph Reduction*, volume 279 of *LNCS*, pages 114–118, Sante Fé, NM, September-October 1986. Springer.
- [150] R. Milikowski and W. G. Vree. The g-line: A distributed processor for graph reduction. In Emile H. L. Aarts, Jan van Leeuwen, and Martin Rem, editors, *Proceedings of Parallel Architectures and Languages Europe (PARLE '91)*. Vol. I: *Parallel Architectures and Algorithms*, volume 505 of *LNCS*, pages 119–136, Eindhoven, NET, June 1991. Springer.
- [151] G. L. Miller and J. H. Reif. Parallel tree contraction. part 2: Further applications. *SIAM J. Computing*, 20, 1991.
- [152] P. Mitra. Efficient parallel shortest path algorithms for k-chordal graphs. In Viktor K. Prasanna and Larry H. Canter, editors, *Proceedings of the 6th International Parallel Processing Symposium*, pages 88–94, Beverly Hills, CA, March 1992. IEEE Computer Society Press.
- [153] A. Moitra and R. C. Johnson. A parallel algorithm for maximum matching on interval graphs. In *International Conference on Parallel Processing, Vol.3: Algorithms and Applications*, pages 114–120. The Pennsylvania State University, August 1989.
- [154] K. Mulmuley, U. Vazirani, and V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7:105–113, 1987.
- [155] Yoshinori Nakanishi and Kuniaki Uehara. Parallel organization algorithm for graph matching and subgraph isomorphism detection. In Setsuo Arikawa and Hiroshi Motoda, editors, *Proceedings of the 1st International Conference on Discovery Science (DS-98)*, volume 1532 of *LNAI*, pages 407–408, Fukuoka, Japan, December 1998. Springer.
- [156] J. Naor. A fast parallel coloring of planar graphs with five colors. *Information Processing Letters*, 25, 1987.
- [157] D. Nassimi. A parallel MSF algorithm for planar graphs on a mesh and applications to image processing. In V.K. Prasanna, editor, *Proceedings of the 7th International Parallel Processing Symposium*, pages 205–212, Newport Beach, CA, April 1993. IEEE Computer Society Press.

- [158] K. S. Natarajan. Graph-partitioning on shared-memory multiprocessor systems. In *International Conference on Parallel Processing, Vol.3: Algorithms and Applications*, pages 120–124, Austin, Tex., USA, August 1991. CRC Press.
- [159] M. Nigam, S. Sahni, and B. Krishnamurthy. Embedding hamiltonians and hypercubes in star interconnection graphs. In Pen-Chung Yew, editor, *Proceedings of the 1990 International Conference on Parallel Processing. Volume 3: Algorithms and Architectures*, pages 340–343, Urbana-Champaign, IL, August 1990. Pennsylvania State University Press.
- [160] T. Nishizeki. *Planar Graphs: Theory and Algorithms*. North Holland, 1988.
- [161] O. A. Ogunyode. Parallel algorithms for approximate edge coloring of simple graphs. In Kesav V. Nori, editor, *Proceedings of the 7th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 287 of *LNCS*, pages 207–225, Pune, India, December 1987. Springer.
- [162] Chao-Wei Ou and Sanjay Ranko. Parallel incremental graph partitioning using linear programming. In *Proceedings of the Conference on Supercomputing*, pages 458–467, Washington, DC, USA, November 1994. IEEE Computer Society Press.
- [163] Alessandro Panconesi and Aravind Srinivasan. Fast randomized algorithms for distributed edge coloring. In *Proceedings of the eleventh annual ACM symposium on Principles of distributed computing*, pages 251–262, 1992.
- [164] G. E. Pantziou, P. G. Spirakis, and C. D. Zaroliagis. Efficient parallel algorithms for shortest paths in planar graphs. In *SWAT 90: 2nd Scandinavian Workshop on Algorithm Theory, Bergen, Sweden, July 1990. Proceedings*, pages 288–300. Springer, July 1990.
- [165] B. Parhami. *Introduction to Parallel Processing, Algorithms and Architectures*. Plenum Press, 1998.
- [166] S. Pawagi. Parallel algorithm for maximum weight matching in trees. In *International Conference on Parallel Processing*, pages 204–206, University Park, Penn State University, Pa., USA, August 1987. Pennsylvania State Univ. Press.
- [167] S. Pawagi. Parallel algorithm for maximum weight matching in trees. Technical Report Report 4, Department of Computer Science, SUNY at Stony Brook, 1987.

- [168] C. A. Phillips. Parallel graph contraction. In ACM-SIGACT; ACM-SIGARCH, editor, *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 148–157, Santa Fe, NM, June 1989. ACM Press.
- [169] F. Preparata and M. Shamos. *Computational Geometry - an Introduction*. Springer Verlag, 1985.
- [170] V. Ramachandran. A general purpose shared-memory model for parallel computation. *Algorithms for Parallel Processing*, 105(?):1–17, 1999.
- [171] V. Ramachandran and J. Reif. An optimal parallel algorithm for planarity. *Journal of Algorithms*, 1992.
- [172] Margaret Reid-Miller. List ranking and list scan on the cray c-90. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 104–113, 1994.
- [173] J. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan and Kaufman Publishers, 1993.
- [174] Berry Schoenmakers. A new algorithm for the recognition of series parallel graphs. Technical Report CS-R9504, CWI, 1995.
- [175] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992.
- [176] Jop F. Sibeyn. List ranking on meshes. *Acta Informatica*, 35(7):543–566, 1998.
- [177] L. Snyder. Type architectures, shared memory and the corollary of modest potential. *Annu. Rev. Comput. Sci.*, 1:289 – 317, 1986.
- [178] M. A. Sridhar and S. Goyal. Efficient parallel computation of hamilton paths and circuits in interval graphs. In *International Conference on Parallel Processing, Vol.3: Algorithms and Applications*, pages 83–90, Austin, Tex., USA, August 1991. CRC Press.
- [179] R. Sridhar and S. S. Iyengar. Fast parallel algorithms for recognizing strongly chordal, ptolemaic, and block graphs. In Pen-Chung Yew, editor, *Proceedings of the 1990 International Conference on Parallel Processing. Volume 3: Algorithms and Architectures*, pages 141–144, Urbana-Champaign, IL, August 1990. Pennsylvania State University Press.

- [180] I. A. Stewart. Colouring perfect planar graphs in parallel. In J. van Leeuwen, editor, *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 344 of *LNCS*, pages 268–275, Amsterdam, The Netherlands, June 1988. Springer.
- [181] M. Swamy and K. Thulasiraman. *Graphs: Theory and Algorithms*. John Wiley, 1992.
- [182] T. Takaoka. An efficient parallel algorithm for the all pairs shortest path problem. In J. van Leeuwen, editor, *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 344 of *LNCS*, pages 276–287, Amsterdam, The Netherlands, June 1988. Springer.
- [183] Alexander Y. Tetelbaum. Estimation of the graph partitioning for a hierarchical system. In D. H. Bailey, P. E. Bjørstad, J. E. Gilbert, M. V. Mascagni, R. S. Schreiber, H. D. Simon, V. J. Torczon, and L. T. Watson, editors, *Proceedings of the 27th Conference on Parallel Processing for Scientific Computing*, pages 635–636, San Francisco, CA, USA, February 1995. SIAM Press.
- [184] Tucker. Matrix characterization of circular-arc graphs. *Pacific Journal of Mathematics*, 39(2):535–545, 1971.
- [185] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), August 1990.
- [186] Jan van Leeuwen. Coloring k -colorable graphs in constant expected parallel time. In *Graph-Theoretic Concepts in Computer Science : 19th International Workshop, WG '93, Utrecht, The Netherlands, June 16-18, 1993; proceedings*, volume LNCS 790, pages 167–176, Berlin, 1994. Springer.
- [187] W. G. Vree. Implementation of parallel graph reduction by explicit annotation and program transformation. In *Mathematical Foundations of Computer Science 1990*, pages 135–151, Banska Bystrica, Czechoslovakia, August 1990. Springer.
- [188] M. W. Wu and M. C. Loui. An efficient distributed algorithm for maximum matching in general graphs. *Algorithmica*, 5, 1990.
- [189] C. Yap. Parallel triangulation of a polygon in two calls to the trapezoidal map. *Algorithmica*, 3:279–288, 1988.