Spring 1996

Visibility-Related Problems on Parallel Computational Models

Himabindu Gurla
Old Dominion University

VISIBILITY-RELATED PROBLEMS ON PARALLEL COMPUTATIONAL MODELS

by

Himabindu Gurla
B.E. (CS), July 1991, Osmania University, India

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements of the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY
April 1996

Approved by:

Stephan Olariu (Advisor)

James L. Schwing (Advisor)

Larry Wilson

Chester E. Grosch

Przemyslaw Bogacki

ABSTRACT

VISIBILITY-RELATED PROBLEMS ON PARALLEL COMPUTATIONAL MODELS.

Himabindu Gurla
Old Dominion University, 1996
Advisors: Drs. Stephan Olariu and James L. Schwing

Visibility-related problems find applications in seemingly unrelated and diverse fields such as computer graphics, scene analysis, robotics and VLSI design. While there are common threads running through these problems, most existing solutions do not exploit these commonalities. With this in mind, this thesis identifies these common threads and provides a unified approach to solve these problems and develops solutions that can be viewed as *template algorithms* for an abstract computational model. A template algorithm provides an architecture independent solution for a problem, from which solutions can be generated for diverse computational models. In particular, the template algorithms presented in this work lead to optimal solutions to various visibility-related problems on fine-grain mesh connected computers such as meshes with multiple broadcasting and reconfigurable meshes, and also on coarse-grain multicomputers.

Visibility-related problems studied in this thesis can be broadly classified into Object Visibility and Triangulation problems. To demonstrate the practical relevance of these algorithms, two of the fundamental template algorithms identified as powerful tools in almost every algorithm designed in this work were implemented on an IBM-SP2. The code was developed in the C language, using MPI, and can easily be ported to many commercially available parallel computers.

To my father

ACKNOWLEDGEMENTS

This work could not be completed without the help of many individuals, to whom I would like to express my appreciation. First and foremost, I would like to thank my advisors, Drs. Stephan Olariu and James Schwing, who have put a great deal of time and effort into the guidance of this work. I would like to thank Dr. Schwing for his help in conducting the experiments on IBM-SP2 at NASA langley.

Next, I would like to convey my sincere thanks to the other members of my dissertation committee, Drs. Larry Wilson, Chester Grosch and Przemyslaw Bogacki. Their expertise, thorough reviewing and valuable suggestions have also led to a greatly improved dissertation.

I wish to extend my appreciation to the faculty of the department, and my fellow students for providing a stimulating research environment.

I am grateful to my family for their encouragement and support. Finally, special thanks to my husband, Adarsh, for his understanding and patience during the several evenings and weekends required to complete this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

The design of optimal parallel algorithms is an art taking into consideration the challenges it poses to an algorithm designer. Two major challenges that are posed to the designer in providing parallel solutions to various problems are:

- To design the *fastest* algorithm for the particular model of computation under consideration,
- To develop *template algorithms* or *paradigms* that work in relatively many cases, possibly across diverse computational platforms.

Among the two, the first challenge is the relatively easier one to meet. This is obvious from the fact that there are few methods that work in relatively many cases and which are, therefore, worth becoming standard tools in the repertoire of every algorithm designer.

Geometric problems provide a fertile ground for challenging the designer of parallel algorithms. The solutions to these problems require the designer to make cautious decisions for each step of the algorithm, including mapping the input data to various processors of the parallel machine, balancing out the communication and

computation steps, while exploiting the inherent geometrical relations between the input items.

Ongoing research in the study of geometric problems is motivated by their significance in diverse applications in computer graphics, image processing and several other fields. Due to the real-time requirements of some of the applications in which geometric problems arise, the quest for faster and more efficient algorithms has made parallelism imperative.

Using these observations for motivation, this thesis will investigate the design of efficient, *time-optimal* algorithms for a subset of geometric problems, with the aim of developing architecture independent techniques that would serve as paradigms across diverse computational models. The paradigms will be specified as template algorithms designed for an abstract computational model. Implementing these template algorithms on a specific computational model requires the development of tools specific to that computational model. The computational models being studied are chosen from the opposite ends of the spectrum of the various parallel computational models, and are also practically relevant ones. Mesh-connected computers enhanced with various bus systems are studied among the fine-grain models. The coarse-grain multicomputer lying at the other end of the spectrum is the other computational model that is considered. A byproduct of this exercise of porting the template algorithms to these diverse computational models will be a rich collection of tools for each of the computational models that can be reused in other contexts.

The class of geometric problems that receives focus in this thesis are the visibility-related problems, involving visibility relations among objects in a plane. The basic concept in visibility problems is that two points p_1 and p_2 are mutually visible if the line segment p_1p_2 does not intersect any *forbidden-curve*. Visibility

is normally defined with respect to a viewpoint ω. One reason for choosing these problems stems from the variety of applications they have found in diverse fields such as computer graphics, scene analysis, robotics and VLSI design. Also, a review of the existing solutions to various members of this class of problems demonstrates that they do not follow a unified approach and there has been little or no emphasis on exploring the commonality between solutions. This thesis provides a unified look at these problems and, thus, identifies the common threads that run through these problems.

To set the stage for what follows, it is appropriate to introduce concepts concerning visibility problems. Let us begin with a brief survey on where and how visibility-related problems can be applied, which further lends emphasis to their significance across a wide variety of applications:

- In computer graphics, visibility from a point plays a crucial role in ray tracing and hidden-line elimination [39, 76].

- Visibility relations among objects are of significance in path planning and collision avoidance problems in robotics [54, 88, 89] where a navigational course for a mobile robot is sought in the presence of various obstacles.

- In VLSI design, visibility plays a fundamental role in the compaction process of integrated circuit design [53, 55, 58, 61, 77, 78, 82]. It is customary to formulate the compaction problem as a visibility problem involving a collection of iso-oriented, non-overlapping, rectangles in the plane.

The class of visibility-related problems explored in this thesis can be broadly classified into two categories:

- **Object Visibility:** This class of problems involves determining the visibility relations among a collection of objects such as line segments, rectangles, or disks in the plane.

- **Triangulations:** The class of triangulation problems involves partioning a planar region containing a sequence of forbidden subregions into triangles, without intersecting the forbidden subregions.

Visibility-related problems have been widely studied in both sequential and parallel settings. As the challenge to solve large and complex problems has constantly increased, achieving high performance by using large scale parallel machines became imperative. To effectively apply a high degree of parallelism to a single application, the problem data is spread across the processors. Each processor computes on behalf of one or a few data elements in the problem. This approach is called *data – level parallel* [30] and is effective for a broad range of computation-intensive applications including problems in vision geometry and image processing.

As the choice of computational platforms forms another important aspect of this thesis, let us briefly survey salient aspects of algorithm development in various parallel environments. In the parallel setting, much of the theoretical work done thus far has focussed on designing parallel algorithms for Parallel Random Access Machines (PRAM). The simple characteristics of PRAM make it suitable for theoretical results in evaluating the complexity of parallel algorithms, but only a small number of real architectures (some bus-based multiprocessors like Encore and Sequent) can be considered conceptually similar in design with the PRAM model.

Although any real machine can simulate the PRAM model, it is nevertheless true that algorithms designed for network-based models will better match the architectures of existing parallel machines like Intel Paragon, IBM SP2, Intel iPSC/860,

CM-5, MasPar MP-1 etc, where processors with local memories are interconnected through a high-speed network supporting message-based communication.

One of the goals of any algorithm designer is that the algorithms be practically relevant and be applicable to models of computation that are close to various commercially available parallel machines. With this in mind, among the fine-grain models of computation, mesh-connected computers enhanced with buses are studied in this thesis. In particular, mesh-connected computers enhanced with static and dynamically reconfigurable bus systems are considered, which are referred to as *meshes with multiple broadcasting,* and *reconfigurable meshes*, respectively.

The mesh-connected computer has emerged as one of the most widely investigated parallel models of computation. It provides a natural platform for solving a large number of problems in computer graphics, image processing, robotics, and VLSI design. In addition, due to its simple and regular interconnection topology, the mesh is well suited for VLSI implementation [12]. The large communication diameter being a bottle neck in the case of applications requiring *nonspatially organized* communications [40] where several hops have to be performed to complete data exchanges between nonadjacent processors, mesh-connected computers are enhanced by various bus systems. In particular, meshes with multiple broadcasting are mesh-connected computers where every row and every column of processors are connected to a bus, while the reconfigurable meshes are mesh-connected computers enhanced with dynamically reconfigurable bus systems.

Being of theoretical interest as well as commercially available, the mesh with multiple broadcasting has attracted a great deal of attention. In recent years, efficient algorithms to solve a number of computational problems on meshes with multiple broadcasting have been proposed in the literature. These include image

processing [48, 75], computational geometry [15, 18, 21, 47, 72, 73, 74], semigroup computations [10, 17, 26, 47], sorting [16], multiple-searching [21], and selection [19, 26, 47], among others.

At the same time, the huge demand for real-time computations in manufacturing, computer science, and the engineering community has motivated researchers to consider adding reconfigurable features to high-performance computers. Along this line of thought, a number of bus systems whose configuration can change, under program control, have been proposed in the literature. Examples include the *bus automaton* [81], the *reconfigurable mesh* [66], the GCN chip [84, 85], the *polymorphic torus* [50, 59], and the PPA architecture [60]. Among these, the reconfigurable mesh has emerged as a very attractive and versatile architecture. In recent years a number of efficient algorithms for problems ranging from sorting to computational geometry, image processing, and graph theory have been proposed on the reconfigurable mesh [13, 45, 52, 66, 68, 69, 70, 71, 90].

Another very interesting model of computation considered in this thesis is the coarse-grain multicomputer model. More recently, coarse-grain multicomputers are being considered to obtain solutions to various geometric problems. In theory, there are mapping methods to simulate fine-grain algorithms on coarse-grain machines, and it is claimed that this will not affect their asymptotic running time. In practice, the local computation and the interprocess communication have different contributions to the total running time and therefore changing the granularity of local processing may affect the scalability of the algorithms. It is obvious that there is a need to develop algorithms for the coarse-grain models of computation, with the aim of minimizing the computational time as well as the number of communication operations. The challenge is to reduce the computational time, by a factor propor-

tional to the number of processors, compared to the sequential computational time for the various algorithms without drastic increase in the cost of communication operations required to achieve that. Some progress in this direction has been made by Dehne,*et al.* [32], Devillers and Fabri [33], Atallah *et al.* [9], Hristescu [41], and others.

The work done on the coarse-grain multicomputers assumes a parallel model that is architecture independent, *communication round model.* In this model, n inputs are evenly distributed among p processors, $p \leq n$, each having local memory of size $O(\frac{n}{p})$. The processors communicate via an interconnection network in a communication round in which they specify the type of communication to occur. Algorithms are designed by specifying the local computation done within each processor between the communication rounds, and by specifying the type of communication performed in a communication round.

The organization of the remainder of this thesis is as follows: the following section of Chapter 1 discusses the state of the art for visibility-related problems on various computational models. Chapter 2 presents a detailed discussion of the diverse models of computation considered in this thesis, Chapter 3 discusses the object visibility problems in the context of an abstract computational model and presents solutions in the form of template algorithms, Chapters 4 and 5 discuss the porting of the template algorithms to fine-grain and coarse-grain models of computation respectively, Chapter 6 presents template algorithms for solving triangulation problems on the abstract computational model, Chapters 7 and 8 specify how these template algorithms are ported to fine-grain and coarse-grain computational models. Finally, Chapter 9 presents the experimental results on IBM-SP2 along with the concluding remarks.

1.2 STATE OF THE ART

Parallelism seems to hold the greatest promise for major reductions in computation time for various classes of geometric problems. The first look at parallel geometric algorithms dates back to 1950s and the modern approach to parallel computational geometry was pioneered by A. Chow in her Ph.D thesis [27]. For a survey of the first ten years of research in computational geometry the reader is referred to [3].

The early models of computation included *Perceptrons*, proposed in the late 1950's [80] and Cellular Automata [28]. The next generation of models considered are the interconnection networks including the linear arrays, meshes or two-dimensional arrays, several variations of meshes including the meshes with broadcast buses referred to as meshes with multiple broadcasting, and the meshes with reconfigurable buses. Tree networks, mesh-of-trees, pyramid networks, hypercube, cube-connected cycles, Butterfly, AKS Sorting network, Star and Pancakes are among the other network based models of computation which have been studied. On the other hand, shared memory models of computation were also studied and included parallel random access machines, scan model, broadcasting with selective reduction etc.

In particular, mesh-connected computers and enhanced mesh computers have been thoroughly investigated in the context of efficient algorithms for geometric problems as specified in the several references in the introduction. More recently, these problems are being looked at on coarse-grain multicomputers [9, 32, 33, 41].

Visibility problems include computation of visibility relations among objects in a plane from a view point, and determination of visibility pairs of line segments, the visibility polygon from a point inside a polygon, determination of a polygon visible in a direction. The problem of determination of visibility polygon has been solved in [31] using divide-and-conquer on a mesh of size $\sqrt{n} \times \sqrt{n}$ and runs in $O(\sqrt{n})$

time using $O(n)$ processors and in $O(n)$ time on a linear array [4] of size n. Given a view point w in the plane and an n-vertex polygonal chain, the portion of the chain visible from w can be determined in $O(\log n)$ time using $O(n/\log n)$ processors on a concurrent read exclusive write PRAM, referred to as CREW-PRAM [7].

Let us discuss the state-of-the-art for object visibility problems on various computational models. The segment visibility problem and its variants have attracted a good deal of attention in the literature. Given a set of n opaque non-intersecting line segments, the problem involves determining parts of the segments visible from a point w in the same plane. This problem has a sequential lower-bound of $\Omega(n \log n)$. A technique called *critical – point merging* is used in [5] to solve this problem in $O(\log n \log\log n)$ time, on CREW-PRAM with $O(n)$ processors, and this solution has been refined in [6] using cascading divide-and-conquer to run in $O(\log n)$ time. Another solution to this problem is discussed in [44] and has a running time of $O(\log n)$ in the CREW-PRAM model with n processors. These algorithms use the concept of *plane-sweep tree* of Atallah *et al.* [6]. The construction of the plane-sweep tree is nontrivial and uses the powerful technique of cascading divide-and-conquer. Yet another solution to the vertical segment visibility problem with the same time and processor complexity and using cascading divide-and-conquer has been reported in [24].

An algorithm to solve the vertical segment visibility on a linear array of size N is given in [8] and runs in $O(n \log n/\log N)$ time using $O(N)$ processors, where $N < n$. The problem has been solved on the hypercube with $O(n)$ processors [57] using multiway divide-and-conquer, and runs in $O(SORT(n))$ time. A randomized algorithm is given in [79] that solves the problem of determining which of a set of non-intersecting line segments are visible from $(0, \infty)$ by using *trapezoidal decomposition*

in $O(\log n)$ probablistic time on an $O(n)$ processor butterfly.

Another object visibility problem that has been studied in the literature and involves determination of visibility relations among a set of rectangles in the plane, is the construction of *dominance* and *visibility* graphs. Bhagavathi *et al* have a $O(\log n)$ time algorithm on EREW-PRAM model of computation using trapezoidal decomposition [20].

Another problem of interest is the visibility pair problem and is defined as follows. A pair of vertical line segments s_i and s_j form a *visibility pair* if there exists a horizontal line that intersects s_i and s_j and does not intersect any other segment lying between s_i and s_j. A sequential solution to the problem of finding visibility pairs of line segments in a set of vertical line segments runs in $O(n \log n)$ time [82] and that is the lower bound for the problem as well. Special cases of the problem exist which run in $O(n)$ time. There is a $O(\log n)$ time solution to the visibility pairs problem on a mesh of trees of size n^2 [53].

The problem of determining the *lower envelope* of non-intersecting line segments in the plane, which is nothing but the segment visibility problem with the view point at $(0, -\infty)$, is the only known object visibility problem studied in the coarse-grain models. Dehne *et al.* [32] have given a $O(\frac{n}{p} \log n + T_{Sort}(n,p))$ time algorithm for this problem on coarse-grain multicomputer model.

Let us now discuss the existing results for triangulation problems on various computational models. Triangulating a set S of n points in the plane has a sequential lower bound of $\Omega(n \log n)$ [78]. An algorithm is given in [25] that triangulates a set of n points in the plane on a linear array of size n in $O(n)$ time. Two more $O(\log n)$ time algorithms for triangulating point sets in parallel, on the CREW-PRAM with $O(n)$ processors are presented in [62, 91]. The algorithm in [91] is adapted to run

on an n-processor hypercube by MacKenzie and Stout [57] running in $O(SORT(n))$ time. An algorithm given in [36] triangulates a point set in arbitrary dimensions in $O(\log^2 n)$ time using $O(n/\log n)$ processors on a CREW-PRAM.

Recently, Nigam and Sahni [69] have proposed a constant time algorithm on reconfigurable meshes to triangulate a set of points in the plane. Their algorithm uses the well-known strategy of Wang and Tsin [91]. On coarse-grain models, only known parallel triangulation algorithm for a given set of points in the plane is the one presented by Hristescu [41], who has designed a $O(T_{Sort}(n,p))$ time algorithm on coarse-grain multicomputers.

CHAPTER 2

THE MODELS OF COMPUTATION

This chapter presents a detailed description of the diverse models of computation considered in this thesis. As stated in the introduction, the following two models of computation are considered in the context of fine-grain models, both belonging to the class of enhanced meshes:

- Mesh with multiple broadcasting, i.e, a mesh-connected computer enhanced with static buses,
- Reconfigurable mesh, which is also a mesh-connected computer enhanced with a dynamically reconfigurable bus system.

The other model of computation considered in this thesis lies at the other end of the spectrum of the parallel models of computation. It is a coarse-grain, communication-round model and is briefly described as follows:

- Coarse-grain multicomputer, consists of a number of state-of-the-art computers, communicating through an arbitrary interconnection network.

The organization of the chapter is as follows. Section 2.1 discusses the fine-grain models of interest. In particular, Subsection 2.1.1 discuss the architecture of a mesh with multiple broadcasting and Subsection 2.1.2 discusses the reconfigurable mesh. Finally, Section 2.2 discusses the coarse-grain multicomputer model in detail.

2.1 ENHANCED MESH-CONNECTED COMPUTERS

Being a natural platform for solving a large number of problems in computer graphics, image processing, robotics, and VLSI design, the mesh-connected computer has emerged as one of the most widely investigated parallel models of computation. As mentioned in the introduction, because of its simple and regular interconnection topology, the mesh is well suited for VLSI implementation [12]. However, the large diameter of the mesh does not deliver high performance in applications requiring nonspatially organized communications [40] where several hops have to be performed to complete data exchanges between nonadjacent processors.

To overcome this problem, the mesh architecture has been enhanced by various types of bus systems [22, 47, 50, 59, 81, 86]. Two popular architectures among the enhanced meshes are discussed in the following subsections.

2.1.1 MESHES WITH MULTIPLE BROADCASTING

Recently, a powerful architecture, referred to as a mesh with multiple broadcasting, has been obtained by adding one bus to every row and to every column of the mesh [47, 75]. The mesh with multiple broadcasting has proven to be feasible to implement in VLSI, and is used in the DAP family of computers [75].

A mesh with multiple broadcasting of size $M \times N$, referred to as a MMB, consists of MN identical processors positioned on a rectangular array overlaid with a bus system. In every row of the mesh the processors are connected to a horizontal bus. Similarly, in every column the processors are connected to a vertical bus as illustrated in Figure 2.1.

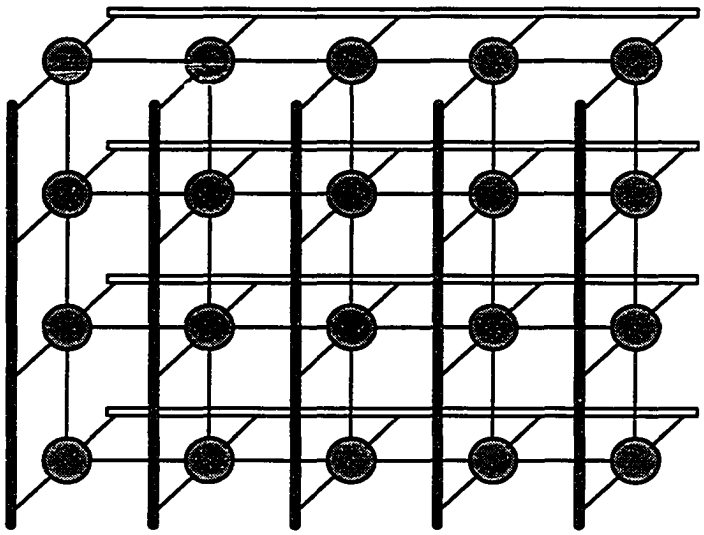

Figure 2.1: A mesh with multiple broadcasting of size 4×5

Processor $P(i,j)$ is located in row i and column j ($1 \leq i \leq M, 1 \leq j \leq N$), with $P(1,1)$ in the north-west corner of the mesh. Every processor $P(i,j)$ is connected to its four neighbors $P(i-1,j)$, $P(i+1,j)$, $P(i,j-1)$, $P(i,j+1)$, provided they exist. It is assumed that the mesh with multiple broadcasting operates in SIMD mode: in each time unit, the same instruction is broadcast to all processors, which execute it and wait for the next instruction. Each processor is assumed to know its own coordinates within the mesh and to have a constant number of registers of size $O(\log MN)$. In unit time, every processor performs some arithmetic or boolean operation, communicates with one of its neighbors using a local link, broadcasts a value on a bus, or reads a value from a specified bus. These operations involve handling at most $O(\log MN)$ bits of information.

For practical reasons, only one processor is allowed to broadcast on a given bus at any one time. However, all the processors on the bus can simultaneously read

the value being broadcast. In accord with other researchers [10, 22, 26, 47, 48, 50, 59, 75, 81], it is assumed that communications along buses take O(1) time. Although inexact, recent experiments with the DAP and the YUPPIE multiprocessor array systems seem to indicate that this is a reasonable working hypothesis [50, 59, 75].

2.1.2 RECONFIGURABLE MESHES

The huge demand for real-time computations in manufacturing, computer science, and the engineering community has motivated researchers to consider adding reconfigurable features to high-performance computers. Among the various architectures that emerged, the reconfigurable mesh has proved to be a very attractive and versatile platform.

A reconfigurable mesh, RMESH for short, of size $M \times N$ consists of MN identical SIMD processors positioned on a rectangular array with M rows and N columns. As in the MMB, it is assumed that every processor knows its own coordinates within the mesh: let $P(i,j)$ denote the processor placed in row i and column j, with $P(1,1)$ in the northwest corner of the mesh. Every processor $P(i,j)$ is connected to its four neighbors $P(i-1,j)$, $P(i+1,j)$, $P(i,j-1)$, and $P(i,j+1)$, provided they exist. It is assumed that the processors have a constant number of registers of $O(\log MN)$ bits and a very basic instruction set. Every processor has 4 ports denoted by N, S, E, and W (see Figure 2.2). Local connections between these ports can be established, under program control, creating a powerful bus system that changes dynamically to accommodate various computational needs. This computational model allows at most two connections involving distinct sets of ports to be set in each processor at any one time. For practical reasons, at any given time, only one processor can broadcast a value onto a bus, while all the processors on the

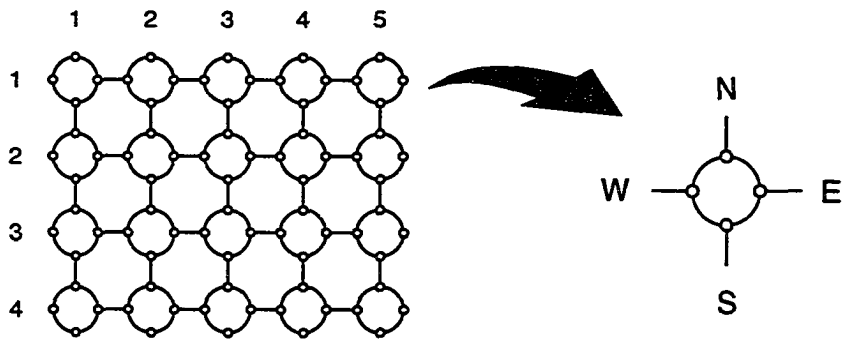

Figure 2.2: A reconfigurable mesh of size 4 × 5

bus can read the value on it simultaneously.

It is worth mentioning that at least two VLSI implementations have been performed to demonstrate the feasibility and benefits of the two-dimensional RMESH: one is the YUPPIE (Yorktown Ultra-Parallel Polymorphic Image Engine) chip [50, 59] and the other is the GCN (Gated-Connection Network) chip [84, 85]. These two implementations suggested that the broadcast delay, although not constant, is very small. For example, only 16 machine cycles are required to broadcast on a 10^6-processor YUPPIE. The GCN has further shortened the delay by adopting precharged circuits. Recently, it has been shown in [83] that the broadcast delay is even further reduced if the reconfigurable bus system is implemented using fiber optics as the underlying global bus system and electrically controlled directional coupler switches (ECS) [38] for connecting or disconnecting fibers. In the light of these experiments and in accord with other workers [1, 22, 50, 59, 66, 81, 84, 85] assume, as a working hypothesis, that communications along buses take O(1) time.

2.2 COARSE-GRAIN MULTICOMPUTERS

Most commercially-available parallel machines including Intel Paragon, IBM SP2, Intel iPSC/860, and CM-5 are coarse-grain where each processor has considerable processing power and local memory. This contrasts sharply with the O(1) memory registers per processor, traditionally assumed in *fine-grain* models. Another feature of commercially available parallel machines is that basic communication primitives (e.g., broadcasting, and routing) are usually available as system calls or as highly optimized utilities. By using these primitives, an applications programmer can design solutions in an architecture-independent setting without having to be familiar with the specific communication patterns of the problem being solved.

The model of computation considered in this thesis is a coarse-grain multicomputer, referred to as CGM(n,p), where p is the number of processors in the parallel machine, and n is the size of the instance of the problem that can be solved using this machine since each of the processors is assumed to have $O(\frac{n}{p})$ local memory. Unlike the fine-grain scenario where the processors are assumed to have O(1) memory words and limited processing capability, each processor in CGM(n,p) is assumed to have considerable processing power. The p processors of the CGM(n,p) are enumerated as $P_0, P_1, \ldots, P_{p-1}$ and each processor P_i is assumed to be aware of its identity i. These processors are connected through an arbitrary interconnection network and communicate using various communication primitives. They are assumed to be operating in SPMD (Single Program Multiple Data) mode, where all the them are executing the same program but on different data items in their local memories. This computational model represents the various commercially available parallel machines mentioned above.

The objective in designing solutions to various problems in this model is to

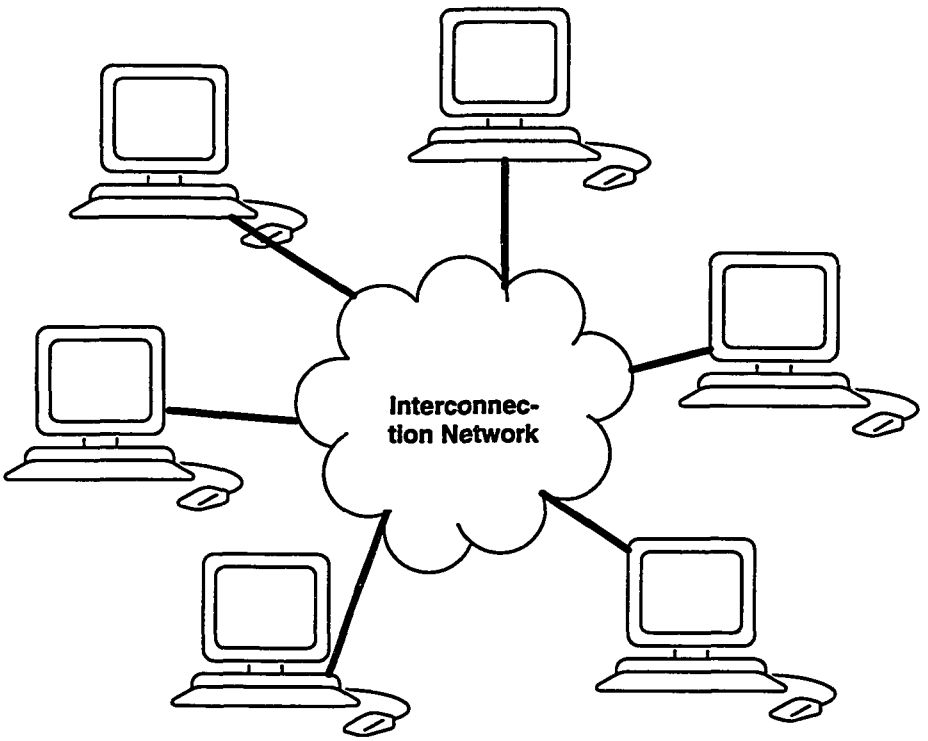

Figure 2.3: A coarse-grain multicomputer

design algorithms where the computational time of the algorithm for an input size of n is $O(\frac{f(n)}{p})$, where $\Omega(f(n))$ is the sequential lower-bound for the problem at hand. The running time of an algorithm is taken to be the sum of the total time spent on computation within any of the p processors and of the total time spent on interprocessor communication. Optimal solutions to various problems in this scenario would require the designer to reduce the computational time, keeping the number of communication rounds as low as possible.

For the computational model to be practically relevant and the algorithms designed for this computational model to be portable across various computational platforms, including shared memory machines, the communication primitives as-

sumed to be available on the CGM(n, p) are the *collective communication* primitives defined by the Message Passing Interface Standard, referred to as MPI for short [67].

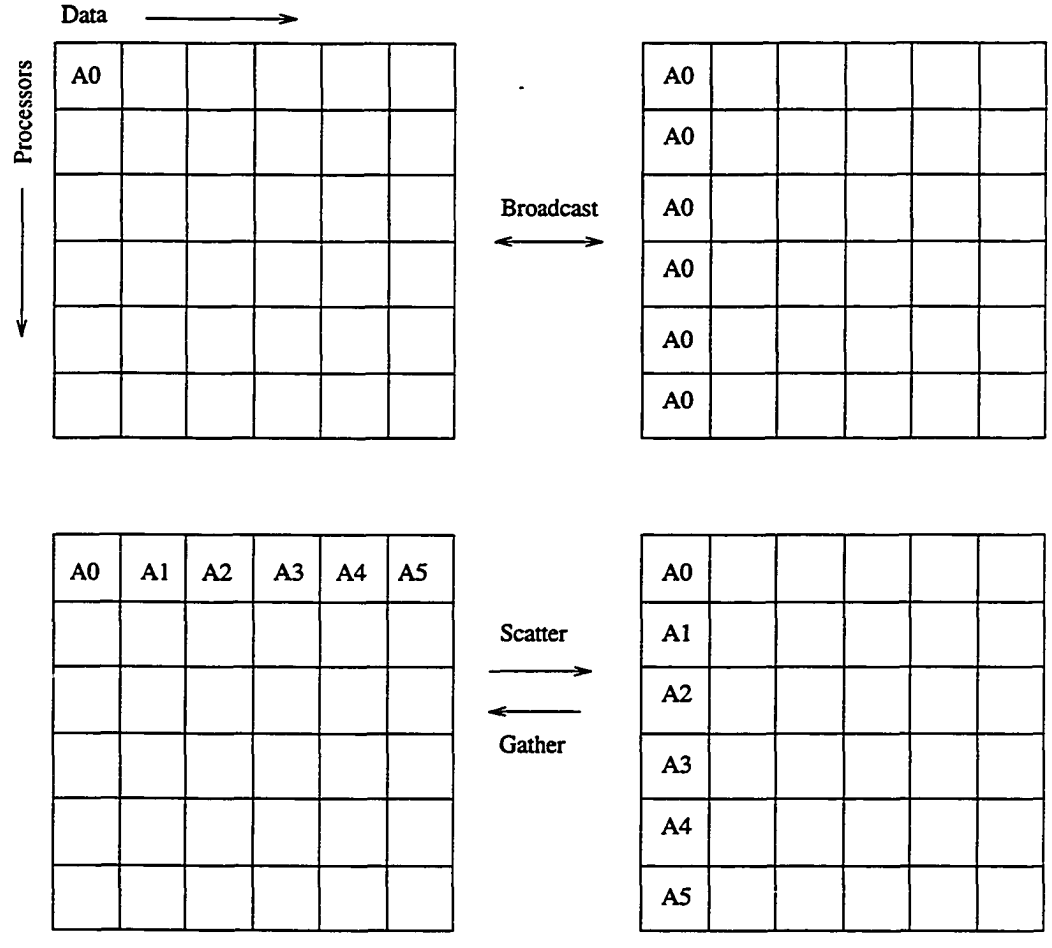

Figure 2.4: Illustration of broadcast, scatter/gather communication primitives

The MPI standardization is an effort involving more then 40 organizations around the world, with the aim of providing a widely used standard for writing message-passing programs and thus establishing a practical, portable, efficient, and flexible standard for message passing. The list of the collective communication primitives as defined by the MPI standard are as follows:

- Broadcast data from one processor, referred to as the root, across all the

processors. Refer to Figure 2.4, where processor P_0 broadcasts an item A_0 to all the processors in the CGM.

- Gather data from all processors to one processor. Refer to Figure 2.4, where the gather operation is illustrated. Every processor P_i stores data item A_i and after the gather operation, processor P_0 has items $A_0, A_1, \ldots, A_{p-1}$.

- Scatter data from one processor to all the processors. As illustrated in Figure 2.4, this data movement is just the reverse of the gather operation. Processor P_0 stores data items $A_0, A_1, \ldots, A_p$ and after the scatter operation, any processor P_i has the item A_i

- All-Gather is a variation of gather where all the processors receive the result of the gather operation and is illustrated in Figure 2.5. Initially, each processor P_i has an item A_i and after the all-gather operation, every P_i has a copy of the items $A_0, A_1, \ldots, A_{p-1}$.

- All-to-all involves Scatter/Gather data from all processors. This is also called complete exchange operation. This operation is clearly illustrated in Figure 2.5. Initially, every processor stores p items, where the first item is to be sent to processor P_0, second to processor P_1 and so on. After the all-to-all operation, every processor receives the p items, one from each of the processors (including itself).

- Global reduction operations such as sum, max, min or any other user-defined functions.

Note that, MPI extends the functionality of scatter, gather, all-gather and all-to-all operations by allowing a varying count of data from each processor. The

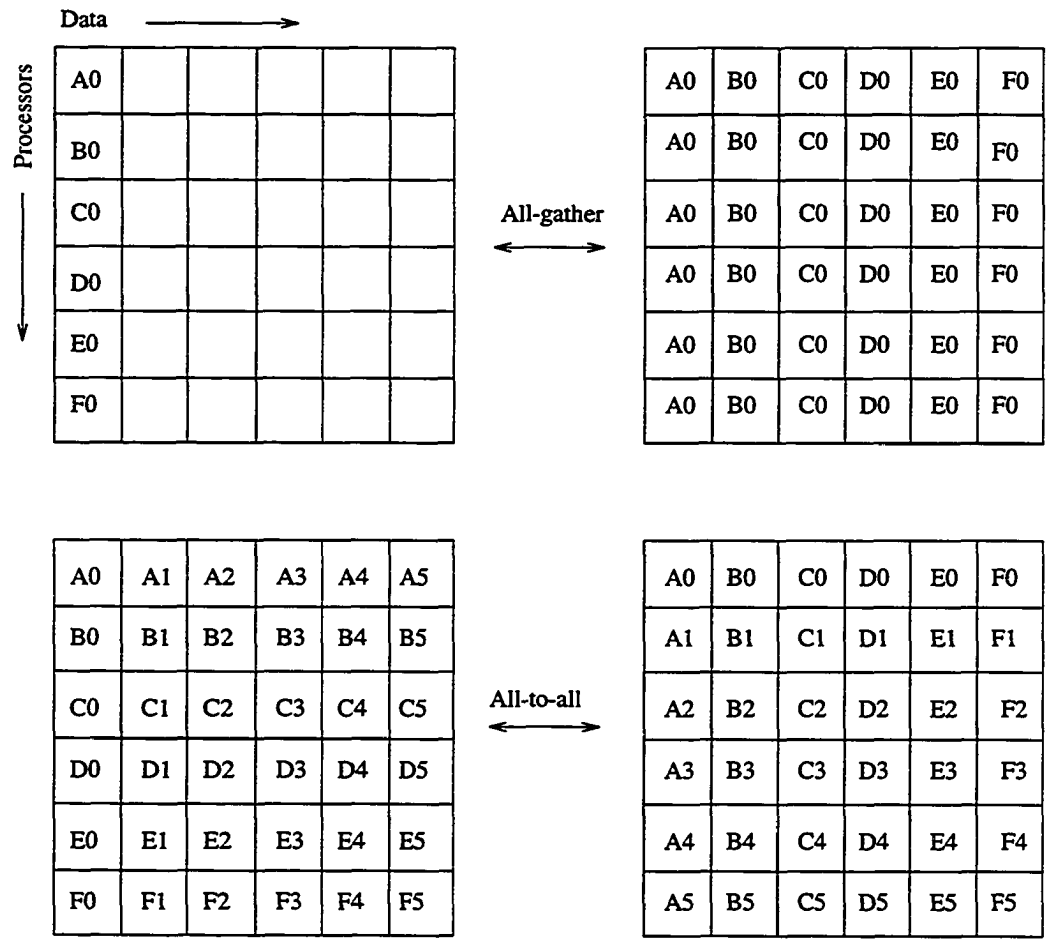

Figure 2.5: Illustration of all-gather and all-to-all communication primitives

processing among the p processors can be viewed as p processes running one per processor. MPI also provides primitives to divide the processes into various groups, each referred to as a *process group*. All the communication primitives can be applied within each of the process groups, in parallel. In the various algorithms designed on this model of computation, the time taken by any communication operation is denoted by $T_{operation}(N, p)$, where N is the number of data items involved in the communication operation, and p is number of processors in the process group.

Earlier work for geometric problems on Coarse-Grain Multicomputers has

been done by Dehne, *et al.* [32], Devillers and Fabri [33], Atallah et al [9], Hristescu [41], etc. The model of computation assumed by them is slightly different from the one considered in this thesis. They assume a different set of communication primitives like sorting, routing, etc. to be available for the various communication rounds. However, for the model to be practically relevant this work assumes that the communication primitives identified by the MPI standard are the only ones available.

CHAPTER 3

OBJECT VISIBILITY ON THE ABSTRACT MODEL

As mentioned in Chapter 1, a recurring problem in a number of contexts in computer graphics, VLSI design, and robot navigation involves computing the visibility of a set of objects in the plane from a distinguished point ω. In computer graphics, for example, visibility from a point plays a crucial role in ray tracing and hidden line elimination [39, 76]. The same problem arises in path planning and collision avoidance problems in robotics [54, 88, 89] where a navigational course for a mobile robot is sought in the presence of various obstacles. Yet another field where visibility plays a fundamental role is VLSI design, in the compaction process of integrated circuit design [53, 58, 61, 77, 78]. In this context, it is customary to formulate the compaction problem as a visibility problem involving a set of iso-oriented, non-overlapping, rectangles in the plane. For simplicity, the compaction process is often one-dimensional, i.e. the components are moved in the x-direction or y-direction only. Hence, it is convenient to abstract rectangles as vertical or horizontal line segments. In this context, the compaction is referred to as *stick* compaction and reduces to a special instance of the visibility problem of vertical line segments [53, 55, 82].

This chapter discusses architecture independent methodologies that provide solutions to the visibility problem for the following classes of objects: segments, disks, and iso-oriented rectangles in the plane. *Template algorithms* are designed for each of these problems for an abstract computational model, which can be ported to diverse models of computation discussed in Chapter 2. These template algorithms, in turn, are designed with emphasis on reusability of concepts developed and exploiting the existing tools.

The segment visibility problem turns out to be a very powerful tool in solving a host of object visibility problems. This problem can be described generically as follows: Given a point ω in the plane along with an ordered set $S = \{s_1, s_2, \ldots, s_n\}$ of non-intersecting line segments in the same plane, it is required to determine the portions of each segment s_i that is visible to an observer positioned at ω.

It will soon be evident that the segment visibility algorithm is a key ingredient in the determination of visibility relations among objects in the plane, such as a set of rectangles or disks. Other examples include determining the visibility pairs among a given set of vertical segments, and constructing the dominance and visibility graphs of a set of iso-oriented rectangles in the plane.

As mentioned earlier, the various template algorithms discussed in this chapter assume an abstract computational model, referred to as ACM, for short. The ACM is defined as follows:

An ACM(n, p, M) consists of p processors, each having O(M) memory, where $n \leq M * p$, (n is the size of the instance of the problem at hand). The p processors are assumed to be identical and are enumerated as $P_0, P_1, \ldots, P_{p-1}$. Each of the processors P_i $(0 \leq i \leq p-1)$ is assumed to know its identity i. All the processors communicate via an interconnection network. In addition, it is assumed that utilities

to perform the following operations are available:

• *Broadcasting*: Processor P_i $(0 \leq i \leq p-1)$ can inform every other processor in the ACM(n,p,M) about k $(1 \leq k \leq M)$ data items it stores. The time required to broadcast k items is $T_{Broadcast}(k,p,M)$.

• *Merging*: Given two sorted sequences of items $S_1 = < a_1, a_2, \ldots, a_r >$ and $S_2 = < b_1, b_2, \ldots, b_s >$, where $r+s=n$, stored at most M per processor in the first $\frac{n}{M}$ processors [1] of an ACM(n,p,M), the result of the merge operation gives a sequence $S=< c_1, c_2, \ldots, c_n >$ stored in the first $\frac{n}{M}$ processors so that processor P_i $(0 \leq i \leq \frac{n}{M}-1)$ stores the items $c_{i*M+1}, \ldots, c_{(i+1)*M}$. The time required to perform the merge operation is $T_{Merge}(n,p,M)$.

• *Sorting*: Given a sequence of items $S = < c_1, c_2, \ldots, c_n >$ from a totally ordered universe, stored M per processor among the first $\frac{n}{M}$ processors of an ACM(n,p,M), the sorting problem requires the determination of the corresponding sorted sequence enumerated as $q_1, q_2, \ldots, q_n$, such that processor P_i $(0 \leq i \leq \frac{n}{M}-1)$, stores the items $q_{i*M+1}, \ldots, q_{(i+1)*M}$. The time required to perform the sort operation is $T_{Sort}(n,p,M)$.

• *Compaction*: Consider a sequence of items $S = < a_1, a_2, \ldots, a_n >$ stored M items per processor, in the first $\frac{n}{M}$ processors of an ACM(n,p,M), with r $(1 \leq r \leq n)$ of the items *marked*. The marked items are enumerated as $B = < b_1, b_2, \ldots, b_r >$ and every marked a_i $(0 \leq i \leq n)$ knows its rank in the sequence B. The compaction operation asks to obtain the ordered sequence B, in order, in the first $O(\frac{r}{M})$ processors storing S, so that any processor P_i $(0 \leq i \leq \frac{r}{M}-1)$ stores items $b_{i*M+1}, \ldots, b_{(i+1)*M}$. The time required to perform this operation is $T_{Compact}(n,p,M)$.

Note that, in the various algorithms that follow, the ACM(n,p,M) may be viewed

[1] In this discussion, ceilings are implicitly assumed

as consisting of l independent ACM's given by ACM($\frac{n}{l}, p', M$) (where p' is at most $\frac{p}{l}$), whenever l identical subproblems are to be solved in each one of them in parallel.

In the following sections, let us discuss the various object visibility problems on the ACM(n, p, M). Section 3.1 discusses the template algorithms for endpoint and segment visibility problems, followed by Sections 3.2 and 3.3 which discuss the disk visibility and rectangle visibility algorithms, using the endpoint visibility algorithm as a basic ingredient. Finally, Section 3.4 discusses the template algorithm for dominance graphs, which in turn uses the algorithm for rectangle visibility as a basic tool.

3.1 ENDPOINT AND SEGMENT VISIBILITY

In this section, let us discuss the template algorithm for solving the endpoint and segment visibility problems for the abstract computational model. First, let us discuss the various terms used in the description of the algorithms that follow. Let ω be a distinguished point and let $S = s_1, s_2, \ldots, s_n$ be a set of non-intersecting line segments in the plane. The set S is said to be *well ordered* if for every i, j $(1 \leq i, j \leq n)$, $i < j$ guarantees that any ray that originates at ω and intersects both s_i and s_j, intersects s_i before s_j.

For an endpoint e of a line segment in S, let $e\omega$ denote the ray originating at e and directed towards ω. Similarly, let $e\overline{\omega}$ be the ray emanating from e, collinear with ω and away from ω. Let us first define the endpoint visibility problem (EV, for short) which is intimately related to segment visibility problem (SV, for short) mentioned earlier. Specifically, given a set S of well ordered line segments, the EV problem asks to determine, for every endpoint e of a segment in S, the closest segments (if any) intersected by the rays $e\omega$ and $e\overline{\omega}$. As an example, in Figure 3.1,

the closest segments intersected by the rays $f_3\omega$ and $f_3\overline{\omega}$ are s_1 and s_6, respectively.

To state the SV problem, define the *contour* of S from ω to be the ordered sequence of segment portions that are visible to an observer positioned at ω. The SV problem asks to compute the contour of S from ω. For an illustration refer to Figure 3.1 where the sequence of heavy lines, when traversed in increasing polar angle about ω, yields the contour of the set of segments.

The following discussion presents a solution to the EV and SV problems on an ACM(n, p, M). Consider an arbitrary set $S = \{s_1, s_2, \ldots, s_n\}$ of *well ordered* line segments, with every segment being specified by its two endpoints. The set S is assumed to be stored in the first $\frac{n}{M}$ processors, at most M segments per processor, of an ACM(n, p, M). Without loss of generality, assume that the viewpoint ω lies to the left of S (i.e. its x-coordinate is smaller than that of any endpoint of a segment in S). The endpoints are specified by their polar coordinates with ω as pole and the vertical ray from ω to $-\infty$ as polar axis. Also assume that the segments are in general position, with no two endpoints sharing the same polar angle. The reader will not fail to observe that these assumptions are made for convenience only and are, in fact, non-essential. For example, if ω does not lie to the left of S, the problem can be divided into two subproblems by splitting some of the segments into two parts, if necessary. The solutions of the two subproblems can be easily combined to yield the required solution.

Every line segment s_i in S has its endpoints denoted in increasing polar angle as f_i and l_i, standing for *first* and *last*, respectively. With a generic endpoint e_i of segment s_i associate the following variables:

- the identity of the segment to which it belongs (i.e. s_i);
- a bit indicating whether e_i is the first or last endpoint of s_i;

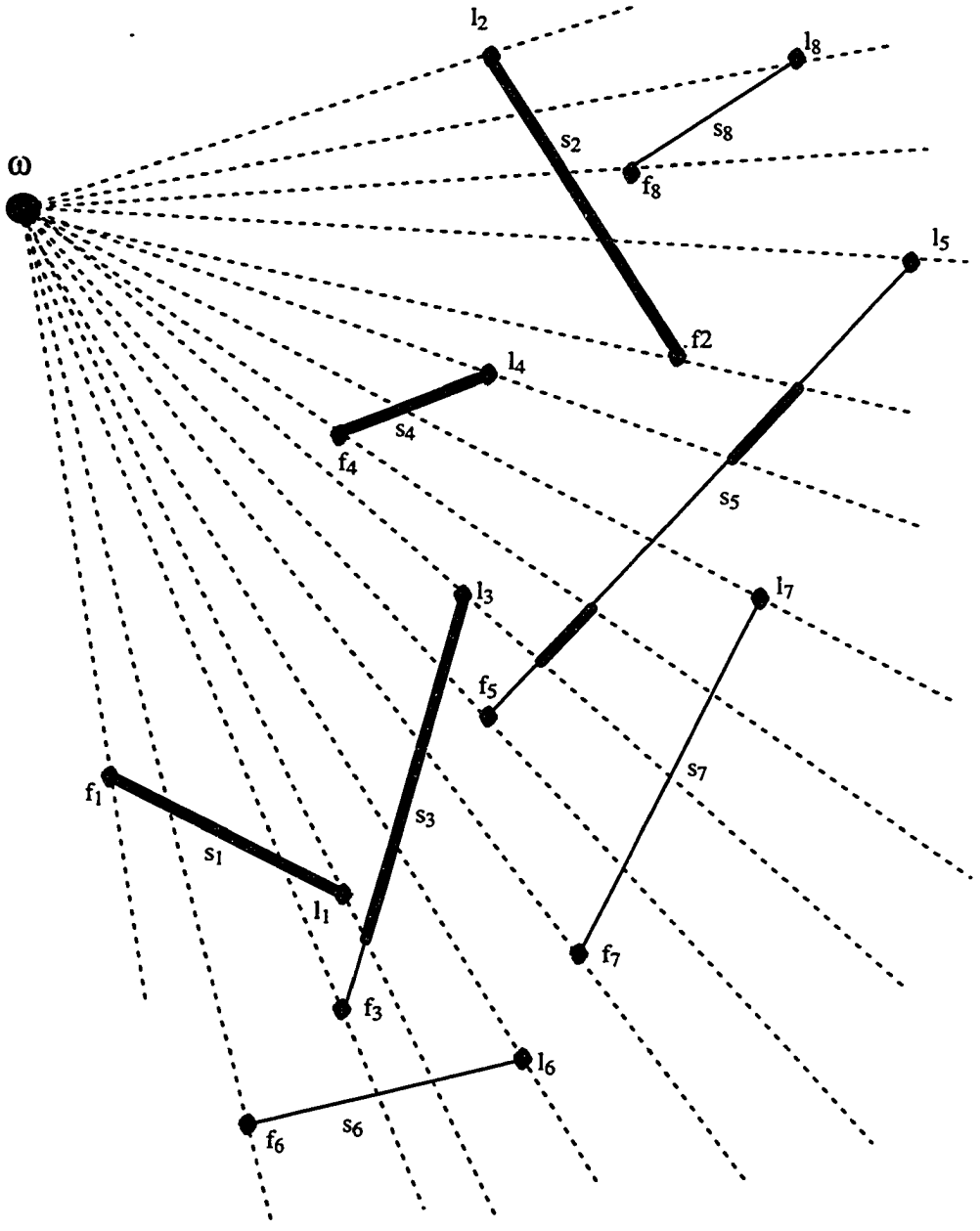

Figure 3.1: Illustrating the endpoint and segment visibility problems

- $t(e_i)$, the identity of the first segment, if any, that blocks the ray $e_i\omega$;
- $a(e_i)$, the identity of the first segment, if any, that blocks the ray $e_i\overline{\omega}$.

The notation $t(e_i)$ and $a(e_i)$ is meant to indicate directions *towards* and *away* from the viewpoint ω, respectively. At the beginning of the algorithm, $t(e_i)$ and $a(e_i)$, for every endpoint e_i, are initialized to 0. When the algorithm terminates, $t(e_i)$ and $a(e_i)$ will contain the desired solutions.

The algorithm begins by computing an *approximate* solution to the EV problem. This involves determining for each of the rays $e_i\omega$ and $e_i\overline{\omega}$ whether it is blocked by some segment in S, without specifying the identity of the segment. This approximate solution is then refined into an exact solution.

Let us proceed with a high-level description of the algorithm. Imagine planting a complete binary tree $\mathcal{T}$ on S, with the leaves corresponding, in left-to-right order, to the segments in S. Given an arbitrary node v of $\mathcal{T}$, let $L(v)$ stand for the set of leaf-descendants of v. Further assume that the nodes in $\mathcal{T}$ are numbered level after level in left-to-right order. For a generic endpoint e_i of segment s_i, let:

- t-blocked(e_i) stand for the identity of the first node in $\mathcal{T}$ on the path from the leaf storing the segment s_i to the root, at which it is known that the ray $e_i\omega$ is blocked by some segment in S;
- a-blocked(e_i) stand for the identity of the first node in $\mathcal{T}$ on the path from the leaf storing s_i to the root, at which it is known that the ray $e_i\overline{\omega}$ is blocked by some segment in S.

Both t-blocked(e_i) and a-blocked(e_i) are initialized to 0.

The algorithm proceeds in two stages. In the first stage, the tree $\mathcal{T}$ is traversed, in parallel, from the leaves to the root, computing for every endpoint e_i,

t-blocked(e_i) and a-blocked(e_i). In case t-blocked(e_i) is not 0, it is guaranteed that some segment in S blocks the ray $e_i\omega$. However, the identity of the blocking segment is not known at this stage. Similarly, if a-blocked(e_i) is not 0, then it is guaranteed that some segment in S blocks the ray $e_i\overline{\omega}$. As before, the identity of the blocking segment is unknown. In the second stage of the algorithm, the tree $\mathcal{T}$ is traversed again, from the leaves to the root, and in the process the information in t-blocked(e_i) and a-blocked(e_i) is refined into t(e_i) and a(e_i).

For convenience, the algorithm is viewed as a sequence of processing tasks involving the nodes of $\mathcal{T}$. A node v of $\mathcal{T}$ is said to be *processed* when the subproblem involving segments in $L(v)$ has been solved. Specifically, consider a generic node v of $\mathcal{T}$ with left and right children u and w, respectively. The following variables are associated with node v:

- $E(v)$, the sequence of endpoints of segments in $L(v)$ sorted by increasing polar angle;
- BT(v), the set of all endpoints e_i in $L(v)$ for which t-blocked(e_i)=v;
- BA(v), the set of all endpoints e_i in $L(v)$ for which a-blocked(e_i)=v;
- LC(v), the set of all endpoints e_i in $L(v)$ for which t-blocked(e_i)=0;
- RC(v), the set of all endpoints e_i in $L(v)$ for which a-blocked(e_i)=0.

The sets BT(v), BA(v) are initialized to the empty set. For a leaf α of $\mathcal{T}$, $E(\alpha)$, LC(α), and RC(α) contain the two endpoints of the corresponding segment in S, sorted by increasing polar angle.

The details of the template algorithm for the EV problem are as follows.

Template Algorithm 3.1:

The template algorithm takes as input the set S of ordered segments, and initializes the various data structures as specified above. The details of the Stage 1 and Stage 2 of the EV algorithm on the ACM follow.

Stage 1. This stage proceeds by processing the nodes of $\mathcal{T}$, level by level, beginning from the leaves of $\mathcal{T}$. Note that, all the nodes at a particular level of the tree $\mathcal{T}$ are processed in parallel.

Consider a generic node v in $\mathcal{T}$ with left and right children u and w, respectively. The tasks performed in the transition from u and w to v, is as follows:

Step 1. $E(v)$ is obtained by merging $E(u)$ and $E(w)$. Note that if $E(u)$ and $E(w)$ are stored in the same processor P_i, as in the case of the first $\log M$ levels of $\mathcal{T}$, the merge operation can be performed by P_i using the sequential merge algorithm in $O(N)$ time, where $N = |E(u)| + |E(w)|$. Note that, in the processing of the first $\log M$ levels of the tree $\mathcal{T}$, each processor P_i $(0 \leq i \leq \frac{n}{M} - 1)$, storing M segments, has to process $\frac{l*M}{n}$ nodes, where l is the number of nodes at that particular level of the tree. The processing of each of the nodes at a particular level of the tree is done sequentially by each P_i, in parallel, and takes $O(M)$ time. Thus, the processing of the first $\log M$ levels takes $O(M \log M)$ time. If $E(u)$ and $E(w)$ are distributed across several processors, for node v with the level greater than $\log M$, the processors storing every pair of sequences $E(u)$ and $E(w)$, for every v belonging to the same level, can be viewed as independent ACM's. Each independent ACM is infact an ACM(N, p', M), where p' is at most $\frac{p}{l}$, and l is the number of nodes at the same level as v. Thus the merge operations corresponding to l nodes at the same level of the tree can be carried out in each of the ACM(N, p', M), in parallel. This can be

accomplished in $T_{Merge}(N, p', M)$ time. Note that, $T_{Merge}(N, p', M)$ is bounded by $T_{Merge}(n, p, M)$.

After the merge operation, for every endpoint e_i in the sorted sequence $E(u)$, let pred$(e_i, E(w))$ and succ$(e_i, E(w))$ stand for the *predecessor* and *successor* in $E(w)$, that is, the endpoints that precede and succeed e_i in $E(w)$, respectively. For an endpoint e_i in $E(w)$ the predecessor and successor pred$(e_i, E(u))$ and succ$(e_i, E(u))$ in $E(u)$ are defined analogously.

Step 2. Next, t-blocked(e_i) and a-blocked(e_i) are computed. The well ordering of the segments in S guarantees that if an endpoint e_i in $E(u)$ has t-blocked(e_i)=0 just prior to processing v, then t-blocked(e_i)=0 holds after v has been processed. Similarly, if the endpoint e_i in $E(w)$ has a-blocked(e_i)=0 just prior to processing v, then a-blocked(e_i)=0 after v has been processed. Now, let e_i be an endpoint in $E(u)$ with a-blocked(e_i)=0. Write e_j=pred$(e_i, E(w))$ and e_k=succ$(e_i, E(w))$. After v has been processed, a-blocked(e_i)=0 only if e_k and e_j belong to different segments and t-blocked(e_j), a-blocked(e_j), t-blocked(e_k), and a-blocked(e_k) are all 0's. Otherwise, a-blocked(e_i) is set to v. Similarly, let e_i be an endpoint in $E(w)$ with t-blocked(e_i)=0, and write e_j=pred$(e_i, E(w))$ and e_k=succ$(e_i, E(w))$. Now t-blocked(e_i)=0 after processing v, only if e_k and e_j belong to different segments and t-blocked(e_j), a-blocked(e_j), t-blocked(e_k), a-blocked(e_k) are all 0's. Otherwise, t-blocked(e_i) is set to v. This can be accomplished in O(M) time for each level of the tree. The correctness of this assignment is guaranteed by the following result.

Lemma 3.1.

(a) Let e_i be an endpoint in $E(u)$ with a-blocked(e_i)=0. If, in the transition from u and w to v, a-blocked(e_i)=v, then the ray $e_i\overline{\omega}$ intersects some segment in $L(w)$.

(b) Let e_i be an endpoint in $E(w)$ with t-blocked(e_i)=0. If, in the transition from

u and w to v, t-blocked(e_i)=v, then the ray $e_i\omega$ intersects some segment in $L(u)$.

Proof. The proof is by induction on the level of v in $\mathcal{T}$. The statement is vacuously true at the leaves of $\mathcal{T}$ which are at level 0. Assume that both (a) and (b) hold for u and w, and suppose that in the transition from u and w to v, a-blocked(e_i)=v for some endpoint e_i in $E(u)$. As above, write e_j=pred(e_i, $E(w)$) and e_k=succ(e_i, $E(w)$). Since a-blocked(e_i)=v, one of the following cases must have occurred:

Case 1. e_j and e_k belong to the same segment.

Let s_p be the segment in $S(w)$ with endpoints e_j and e_k. Since S is well ordered, $i < p$ and, consequently, s_p blocks the ray $e_i\overline{\omega}$, as claimed.

Case 2. a-blocked(e_j)$\neq$ 0 or a-blocked(e_k)$\neq$ 0.

Consider the case a-blocked(e_k)$\neq$ 0, the other following by a mirror argument. By the induction hypothesis, a-blocked(e_k)$\neq$ 0 guarantees the existence of a segment s_q in $S(w)$ that blocks the ray $e_k\overline{\omega}$. Since S is well ordered, $i < q$. Furthermore, since e_j and e_k are consecutive in $E(w)$, the first endpoint of s_q cannot occur between e_j and e_k and, therefore, s_q blocks the ray $e_i\overline{\omega}$.

Case 3. t-blocked(e_j)$\neq$ 0 or t-blocked(e_k)$\neq$ 0.

Consider the case t-blocked(e_j)$\neq$ 0, the other following by a mirror argument. By the induction hypothesis, t-blocked(e_j)$\neq$ 0 guarantees the existence of a segment s_p in $S(w)$ that blocks the ray $e_j\omega$. The fact that S is well ordered guarantees that $i < p$. Since e_j and e_k are consecutive in $E(w)$, the last endpoint of s_p cannot occur between e_j and e_k and, therefore, s_p blocks the ray $e_i\overline{\omega}$.

This completes the proof of (a). The proof of (b) is similar. □

By virtue of Lemma 3.1, when *root*($\mathcal{T}$), the root of $\mathcal{T}$, is reached at the end of Stage 1, all the endpoints e_i having t-blocked(e_i)=0 know that the ray $e_i\omega$ is blocked by no segment in $\mathcal{S}$. All the endpoints e_i with a-blocked(e_i)=0 set a(e_i) = $+\infty$. The

running time of the Stage 1 is bounded by $O(M \log M)+O(\log p T_{Merge}(n,p,M))$ time.

Stage 2. As in Stage 1, the computation in Stage 2 proceeds by processing the nodes of the tree $\mathcal{T}$, level after level, beginning from the leaves. Again, all the nodes at the same level of tree are processed in parallel by viewing the ACM as consisting of several independent ACM's. The main goal of this stage is to use the information obtained in Stage 1 to compute the actual values of $t(e_i)$ and $a(e_i)$, for every endpoint e_i. A key role in the computation specific to this stage is played by the sets $BT(v)$, $BA(v)$, $LC(v)$, and $RC(v)$ defined in the preamble to the template algorithm.

For all nodes v of $\mathcal{T}$, determine $BT(v)$ and $BA(v)$ from the information obtained in Stage 1. Note that, $LC(v)$ contains a sorted sequence of endpoints e_i in $E(v)$ whose t-blocked(e_i)=0, after node v in $\mathcal{T}$ has been processed. Put differently, Lemma 3.1 guarantees that $LC(v)$ contains all the endpoints in $E(v)$ for which the ray $e_i\omega$ is blocked by no segment in $L(v)$. For this reason, and since ω lies to the left of S, $LC(v)$ is referred to as the *left contour* at v. It is important to note that the left contour $LC(v)$ provides a partial solution to the segment visibility problem. The set $RC(v)$ is defined similarly and will be referred to as the *right contour* at v.

Consider again a generic node v in $\mathcal{T}$ with left and right children u and w, respectively. The sets $RC(u)$, $RC(w)$, $LC(u)$, and $LC(w)$ are updated into $RC(v)$ and $LC(v)$ in the transition from u and w to v, as follows.

With $\cup$ standing for the set-merge,

$$RC(v) = (RC(w) \cup RC(u)) - BA(v) \tag{3.1}$$

and

$$LC(v) = (LC(w) \cup LC(u)) - BT(v). \tag{3.2}$$

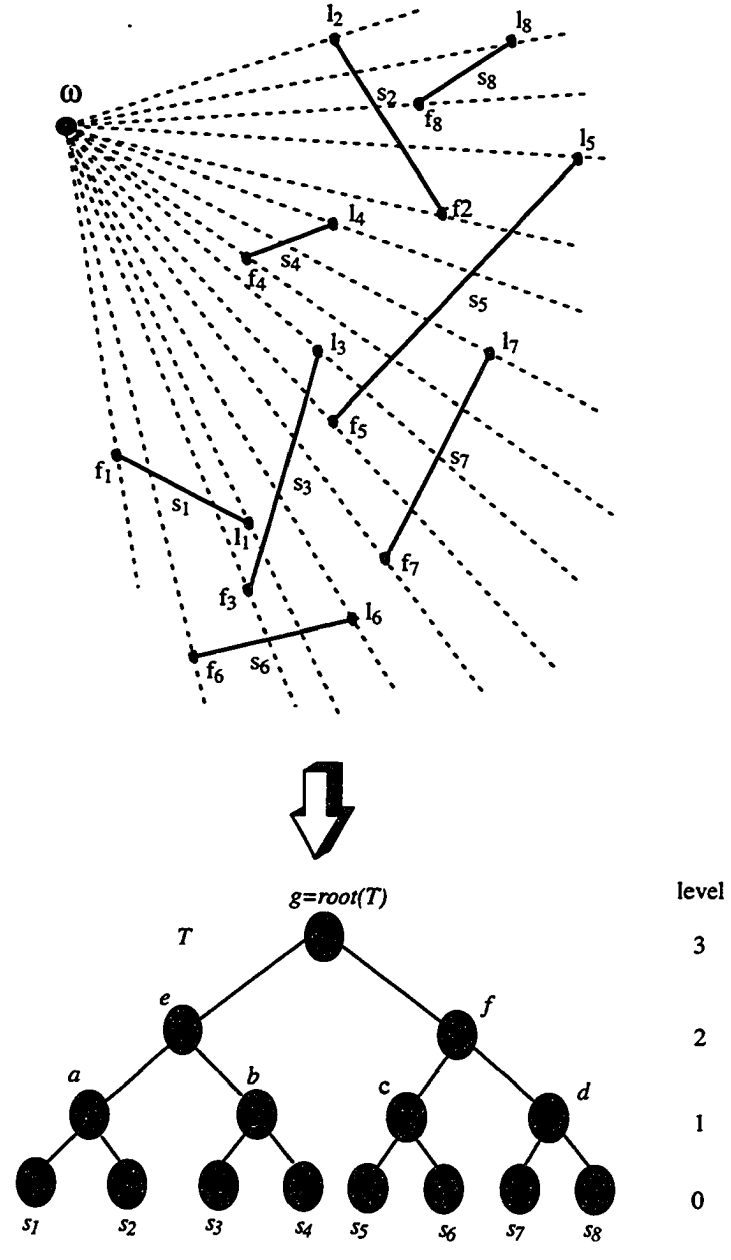

Figure 3.2: The set of segments in Figure 3.1 and the associated binary tree

The determination of the sequence RC(v) in equation 3.1 from RC(u), RC(w), and BA(v) is explained below. Begin by merging RC(u) and RC(w) into a sequence $E'(v)$. This operation takes $O(M \log M)$ time for the first $\log M$ levels of the tree $\mathcal{T}$. For the rest of the levels of the tree having l nodes to process, just as in Stage 1, the ACM(n, p, M) can be viewed as l independent ACM's given by ACM(N, p', M) where $N = |\text{RC}(u)| + |\text{RC}(w)|$, and p' is at most $\frac{p}{l}$. From $E'(v)$, delete those endpoints e_i that have a-blocked(e_i)=v time, i.e, the sequence BA(v), thus giving RC(v) corresponding to the unblocked endpoints in $E'(v)$. Compact the endpoints in RC(v) in each ACM(N, p', M) in $T_{Compact}(N, p', M)$ time. The computation of LC(v) in equation 3.2 is perfectly similar.

Consider, again, the processing that takes place in the Stage 2 of the algorithm, in the transition from u and w to v. Having computed the sets RC(u), RC(w), LC(u), and LC(w), the values of t(e_i) and a(e_i) for all endpoints in BA(v) and BT(v) are determined. For this purpose, RC(u) and BT(v) are merged.

In the process of merging, every endpoint e_i in BT(v) determines the identity of two endpoints e_j and e_k such that e_j=pred(e_i,RC(u)) and e_k=succ(e_i,RC(u)). The value of t(e_i) is set as follows:

- in case e_j and e_k are endpoints of the same segment s_p, then t(e_i)=s_p;
- if both e_j and e_k are last endpoints, then t(e_i) is set to the segment s_p whose last endpoint is e_k;
- if both e_j and e_k are first endpoints, then t(e_i) is set to the segment s_p whose first endpoint is e_j;
- if e_k is a first endpoint and e_j is a last endpoint then t(e_i)=t(e_j)=t(e_k).

The correctness of this assignment follows by an easy inductive argument. The cor-

rect value of a(e_i) for every endpoint e_i in BA(u) is computed similarly.

Stage 2 takes $O(M \log M)+O(\log p T_{Merge}(n,p,M))$ $+O(\log p T_{Compact}(n,p,M))$ time on the ACM(n,p,M). Thus, the following result is obtained.

Theorem 3.2. The EV problem for a set S of n ordered segments, stored M per processor in the first $\frac{n}{M}$ processors of an ACM(n,p,M), can be solved in $T_{EV}(n,p,M) =$ $O(M \log M)+O(\log p T_{Merge}(n,p,M))+O(\log p T_{compact}(n,p,M))$ time. □

It is important to note that from the information in LC($root(\mathcal{T})$) at the end of Stage 2, along with t(e_i) and a(e_i), the contour of S from ω can be computed as follows. Let LC($root(\mathcal{T})$) contain the endpoints $e_1, e_2, \ldots, e_m$ sorted in increasing polar angle. For every i $(2 \leq i \leq m)$:

- if e_{i-1} and e_i belong to the same segment s_p in S, then s_p belongs to the contour;
- if e_{i-1} is a last endpoint and e_i is a first endpoint, then with s_p standing for the common value of a(e_{i-1}) and a(e_i), the portion of s_p between the rays $e_{i-1}\overline{\omega}$ and $e_i\overline{\omega}$ belongs to the contour;
- if both e_{i-1} and e_i are first endpoints, then with s_p standing for the segment whose first endpoint is e_{i-1}, the portion of s_p between e_{i-1} and the ray $e_i\overline{\omega}$ belongs to the contour;
- if both e_{i-1} and e_i are last endpoints, then with s_p standing for the segment whose last endpoint is e_i, the portion of s_p between the ray $e_{i-1}\overline{\omega}$ and e_i belongs to the contour.

Consequently, the algorithm just described also solves the SV problem. Thus, the following result is obtained.

Table 3.1: Illustrating Stage 1 of the algorithm

level	0		1		2		3	
val.	t-blkd	a-blkd	t-blkd	a-blkd	t-blkd	a-blkd	t-blkd	a-blkd
f_1	0	0	0	0	0	0	0	0
l_1	0	0	0	0	0	e	0	e
f_2	0	0	0	0	0	0	0	g
l_2	0	0	0	0	0	0	0	0
f_3	0	0	0	0	e	0	e	g
l_3	0	0	0	0	0	0	0	g
f_4	0	0	0	0	0	0	0	g
l_4	0	0	0	0	0	0	0	g
f_5	0	0	0	0	0	f	g	f
l_5	0	0	0	0	0	0	g	0
f_6	0	0	0	0	0	0	g	0
l_6	0	0	0	0	0	0	g	0
f_7	0	0	0	0	0	0	g	0
l_7	0	0	0	0	f	0	f	0
f_8	0	0	0	0	0	0	g	0
l_8	0	0	0	0	0	0	g	0

Table 3.2: Illustrating Stage 2 of the algorithm

NODE	BT	BA	LC	RC
a	ϕ	ϕ	$f_1\ l_1\ f_2\ l_2$	$f_1\ l_1\ f_2\ l_2$
b	ϕ	ϕ	$f_3\ l_3\ f_4\ l_4$	$f_3\ l_3\ f_4\ l_4$
c	ϕ	ϕ	$f_6\ l_6\ f_5\ l_5$	$f_6\ l_6\ f_5\ l_5$
d	ϕ	ϕ	$f_7\ l_7\ f_8\ l_8$	$f_7\ l_7\ f_8\ l_8$
e	f_3	l_1	$f_1\ l_1\ l_3\ f_4\ l_4\ f_2\ l_2$	$f_1\ f_3\ l_3\ f_4\ l_4\ f_2\ l_2$
f	l_7	f_5	$f_6\ l_6\ f_7\ f_5\ l_5\ f_8\ l_8$	$f_6\ l_6\ f_7\ l_7\ l_5\ f_8\ l_8$
g	$f_5\ l_5\ f_6\ l_6\ f_7\ f_8\ l_8$	$f_2\ f_3\ l_3\ f_4\ l_4$	$f_1\ l_1\ l_3\ f_4\ l_4\ f_2\ l_2$	$f_6\ l_6\ f_7\ l_7\ l_5\ f_8\ l_8\ l_2$

Theorem 3.3. The SV problem for a set S of n ordered segments stored in the first $\frac{n}{M}$ processors, at most M per processor on an ACM(n,p,M), can be solved in $T_{SV}(n,p,M)$=O$(M \log M)$+O$(\log pT_{Merge}(n,p,M))$ +O$(\log pT_{Compact}(n,p,M))$ time. □

A complete worked example based on the set of segments featured in Figure 3.1 is presented for the reader's benefit. Figure 3.2 shows the set of input segments along with the binary tree $\mathcal{T}$ that guides the algorithm. The various data items computed in Stage 1 are summarized in Table 3.1. The results of Stage 2 are captured, in succinct form, in Tables 3.2 and 3.3. Specifically, the solution to the endpoint visibility problem is contained in Table 3.3.

Table 3.3: The solution to the endpoint visibility problem

level →	0		1		2		3	
Values of	t	a	t	a	t	a	t	a
f_1	$-\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$
l_1	$-\infty$	0	$-\infty$	0	$-\infty$	s_3	$-\infty$	s_3
f_2	$-\infty$	0	$-\infty$	0	$-\infty$	0	$-\infty$	s_5
l_2	$-\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$	$-\infty$	$+\infty$
f_3	0	0	0	0	s_1	0	s_1	s_6
l_3	$-\infty$	0	$-\infty$	0	$-\infty$	0	$-\infty$	s_5
f_4	$-\infty$	0	$-\infty$	0	$-\infty$	0	$-\infty$	s_5
l_4	$-\infty$	0	$-\infty$	0	$-\infty$	0	$-\infty$	s_5
f_5	0	0	0	0	0	s_7	s_3	s_7
l_5	0	$+\infty$	0	$+\infty$	0	$+\infty$	s_2	$+\infty$
f_6	0	$+\infty$	0	$+\infty$	0	$+\infty$	s_1	$+\infty$
l_6	0	$+\infty$	0	$+\infty$	0	$+\infty$	s_3	$+\infty$
f_7	0	$+\infty$	0	$+\infty$	0	$+\infty$	s_3	$+\infty$
l_7	0	$+\infty$	0	$+\infty$	s_5	$+\infty$	s_5	$+\infty$
f_8	0	$+\infty$	0	$+\infty$	0	$+\infty$	s_2	$+\infty$
l_8	0	$+\infty$	0	$+\infty$	0	$+\infty$	s_2	$+\infty$

3.2 DISK VISIBILITY

Given a set $D = \{d_1, d_2, \ldots, d_n\}$ of n non-overlapping opaque disks and a viewpoint ω in the plane, the disk visibility problem (DV, for short) involves determining the portion of each disk $d_i \in D$, that is visible to an observer positioned at ω. The DV problem finds applications to path planning in robotics where a mobile robot must navigate amidst a set of planar obstacles. It is customary to consider, in a first approximation, that all these obstacles are circular (i.e. disks). In this setup, the robot is shrunk to a point while the disks are augmented using Minkowski sums [49, 54], reducing the navigational problem to an instance of the DV problem.

The purpose of this section is to present an architecture independent methodology to solve the DV problem, which leads to optimal solutions to this problem in diverse computation models. As in the case of SV problem, the template algorithm for the DV problem assumes the ACM model of computation and the discussion on porting the template algorithms to various computational models is described in Chapters 4 and 5.

Consider an arbitrary set $D = \{d_1, d_2, \ldots, d_n\}$ of disks stored M per processor among the first $\frac{n}{M}$ of the p processors of an ACM(n, p, M), so that any processor P_i $(0 \leq i \leq \frac{n}{M} - 1)$ stores the subset of disks, $d_{i*M+1}, \ldots, d_{(i+1)*M}$. For simplicity, it is assumed that ω lies to the left of D, that is, all the disks lie in the right half-plane determined by the vertical ray from ω to $-\infty$.

The details of the algorithm is as follows:

Template Algorithm 3.2:

As a preprocessing step, inform all the processors storing the input about the view point ω, and this is accomplished by broadcasting the value ω to all the processors storing the input. This can be performed in $T_{Broadcast}(1, p, M)$ time.

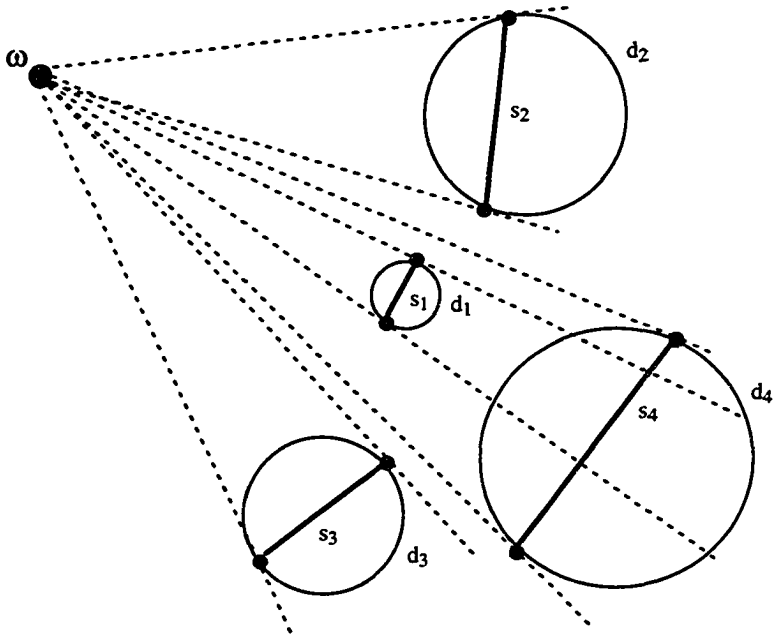

Figure 3.3: Illustrating the disk visibility problem

Step 1. Every P_i $(0 \leq i \leq \frac{n}{M})$, storing M disks $d_{i*M+1}, \ldots, d_{(i+1)*M}$, determines the tangents to each one of them, from the viewpoint ω. The length of these tangents, i.e. the distance between ω and the tangency points, is also determined. This requires $O(M)$ computation time.

Step 2. With every disk d_i associate the line segment s_i obtained by joining the corresponding tangency points. For an illustration, refer to Figure 3.3. Next, sort the s_i's by increasing distance of their endpoints to ω. This is done in $T_{Sort}(n, p, M)$ time. Without loss of generality, let $S = \{s_1, s_2, \ldots, s_n\}$ be the set of these segments in sorted order.

Lemma 3.4. The sorted sequence S is well ordered.

Proof. Suppose not. This implies that there exist subscripts i, j with $i < j$ and some ray δ originating at ω that intersects s_j before intersecting s_i. Let d_i and d_j

be the two corresponding disks and let δ_1 and δ_2 be the supporting rays to d_j from ω. Let a and b be the points where δ_1 and δ_2 meet d_j.

Consider the circle C centered at ω and of radius the length $|\overline{\omega a}| = |\overline{\omega b}|$ of the segments ωa and ωb. Let A stand for the planar region defined as the intersection of C with the half-plane determined by the line collinear with a and b that does not contain ω. Let O_j be the center of d_j. Simple geometric considerations guarantee that A lies entirely within the triangle determined by a, b, and O_j, which in turn, lies completely within d_j.

Observe that the ray δ that intersects both s_i and s_j must lie in the wedge determined by δ_1 and δ_2. Since δ intersects s_j before s_i, it follows that at least one of the endpoints of s_i lies in A. This, however, contradicts the assumption that the disks do not intersect. □

Step 3. Lemma 3.4 guarantees that SV algorithm developed in the Section 3.1 can be applied to the set of segments S. Once the visible portions of the segments are determined, the portions of the disks visible from ω can be trivially computed. This step requires $O(M) + T_{SV}(n, p, M)$ time. Thus, the following result is obtained.

Theorem 3.5. The DV problem for a set S of n non-overlapping disks in the plane, stored M per processor in the first $\frac{n}{M}$ processors of an ACM(n, p, M), can be solved in $T_{DV}(n, p, M) = O(T_{SV}(n, p, M)) + O(T_{Sort}(n, p, M))$ time. □

3.3 RECTANGLE VISIBILITY

Given a set $R = \{R_1, R_2, \ldots, R_n\}$ of n iso-oriented, non-overlapping, opaque rectangles in the plane and a viewpoint ω, the rectangle visibility problem (RV, for short) involves determining the portions of each rectangle that are visible to an observer positioned at ω. The RV problem finds applications to computer graphics, digital

geometry, collision avoidance, VLSI design, and image processing [76, 77, 88].

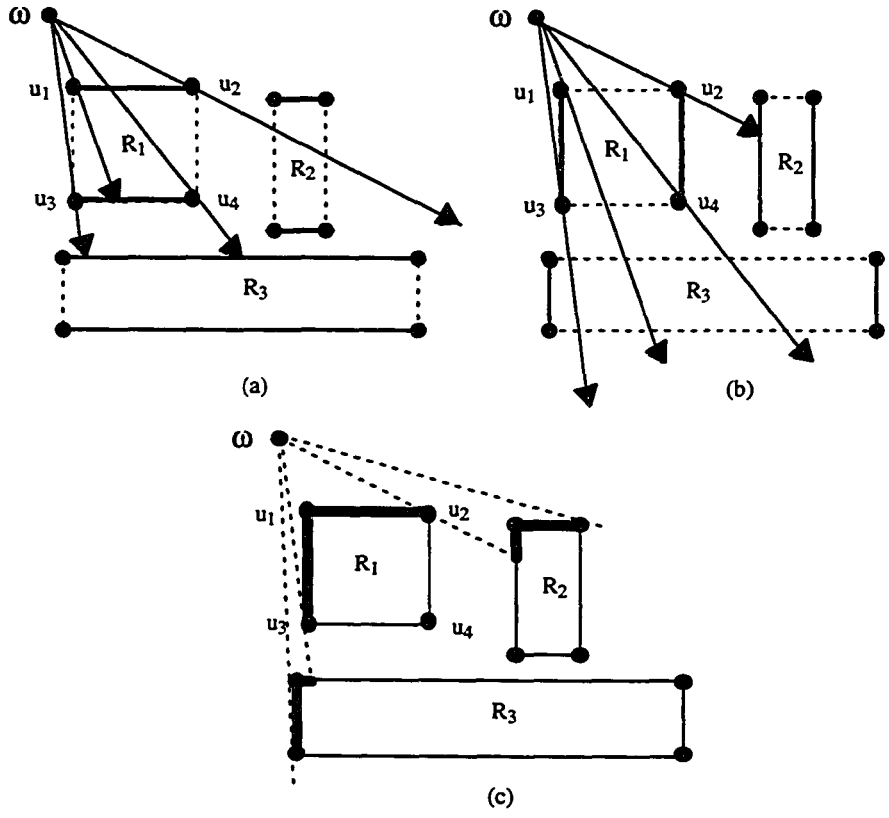

Figure 3.4: Illustrating the rectangle visibility problem

The purpose of this section is to present a template algorithm to solve the RV problem on an ACM(n, p, M). Consider a set $R = \{R_1, R_2, \ldots, R_n\}$ of iso-oriented, non-overlapping, rectangles stored at most M per processor, in the first $\frac{n}{M}$ processors of the ACM(n, p, M). For simplicity, assume that the viewpoint ω lies to the left of R, i.e. that all the rectangles lie in the right half-plane determined by the vertical ray from ω to $-\infty$. Each rectangle R_i is specified by its bottom-left and top-right corners, from which the four sides of the rectangle referred to as *top*, *bottom*, *left* and *right* edges, can be trivially determined. The algorithm to solve the RV problem is described below.

Template Algorithm 3.3:

Step 1. Solve the instance of the EV problem obtained by considering the top and bottom edges of every rectangle $R_i \in R$. Begin by sorting these top and bottom edges by increasing y-coordinate. It is an easy observation that the sorted set of these segments is well ordered and so the EV algorithm applies. Thus, this step can be accomplished in $T_{EV}(n,p,M)$+O($T_{Sort}(n,p,M)$) time.

Step 2. The above process is repeated for the left and right edges of every rectangle $R_i \in R$. Now, every generic corner e_i of rectangle r_i has four solutions: a1(e_i), t1(e_i), a2(e_i), and t2(e_i) obtained in Step 1 and Step 2, respectively. A corner e_i is *marked* if t1(e_i)=t2(e_i)=0. Now, every marked corner e_i combines the information stored in a1(e_i) and a2(e_i) by selecting, among them, the segment closer to e_i along the ray $e_i\overline{\omega}$. If in the process e_i discovers that the closer of a1(e_i) and a2(e_i) is an edge that belongs to its own rectangle, then e_i becomes unmarked. This step can be accomplished in O(M)+$T_{EV}(n,p,M)$+O($T_{Sort}(n,p,M)$) time.

Step 3. Finally, after sorting the remaining marked corners by increasing polar angle, the contour of the set of rectangles can be determined as in the case of SV problem. This step takes O($T_{Sort}(n,p,M)$) time. Thus, the following result is obtained.

Theorem 3.6. The RV problem for a set S of n iso-oriented, non-overlapping rectangles in the plane, stored M per processor in the first $\frac{n}{M}$ processors of an ACM(n,p,M), is solved in $T_{RV}(n,p,M)$=O($T_{EV}(n,p,M)$)+O($T_{Sort}(n,p,M)$) time. □

For an illustration, the reader is referred to Figure 3.4. For every rectangle R_i ($1 \leq i \leq 3$), let t_i, b_i, l_i, and r_i stand for the top, bottom, left, and right edges of R_i, respectively.

- Step 1 is depicted in Figure 3.4(a). At the end of this step, the solutions corresponding to the corners of R_1 are as follows: a1(u_1)=b_1, a1(u_2)=+∞, a1(u_3)=t_3, a1(u_4)=t_3, t1(u_1)=0, t1(u_2)=0, t1(u_3)=0, and t1(u_4)=t_1.

- Step 2 is depicted in Figure 3.4(b). At the end of this step, the solutions corresponding to the corners of R_1 are as follows: a2(u_1)=+∞, a2(u_2)=l_2, a2(u_3)=+∞, a2(u_4)=+∞, t2(u_1)=0, t2(u_2)=0, t2(u_3)=0, and t2(u_4)=0.

- After Step 2, only the corners u_1, u_2, and u_3 are marked. Of these, u_1 detects that the closer segment along the ray $u_1\overline{\omega}$ is b_1, and so u_1 becomes unmarked. The resulting contour is featured in Figure 3.4(c).

3.4 DOMINANCE GRAPH

Consider a set $R = \{R_1, R_2, \ldots, R_n\}$ of n non-overlapping iso-oriented rectangles in the plane. A rectangle R_i is said to be *above* rectangle R_j if there are points in R_i and R_j sharing the same x-coordinate, with the points in R_i having larger y-coordinates. A rectangle R_i is *directly above* R_j if R_i is above R_j and no rectangle R_k is such that R_i is above R_k and R_k is above R_j. The *dominance graph* of the set R is a directed graph $\vec{D}$ whose vertices correspond to the rectangles in R with two vertices u and v in $\vec{D}$ linked by a directed edge (u, v) whenever the rectangle corresponding to v is directly above the rectangle corresponding to u (see Figure 3.5). The dominance graph problem (DG, for short), involves computing the dominance graph of a given set of non-overlapping rectangles in the plane.

The purpose of this section is to describe a template algorithm for the DG problem on an ACM(n, p, M). Consider an arbitrary instance of size n of the DG problem stored in the first $\frac{n}{M}$ of the p processors in the ACM(n, p, M), with each

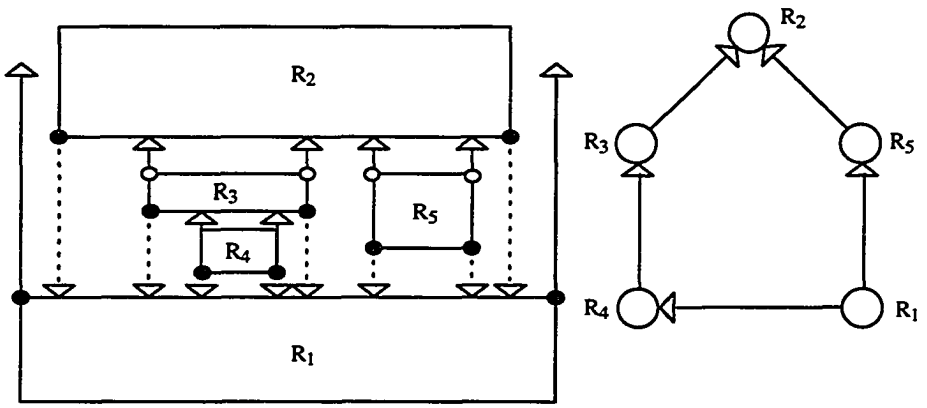

Figure 3.5: A set of rectangles and its dominance graph

processor storing at most M rectangles. Assume that the rectangles are specified by their bottom-left and top-right corners. For every i $(1 \leq i \leq n)$, the top edge t_i and the bottom edge, b_i of rectangle R_i can be trivially computed.

Template Algorithm 3.4:

Step 1. The rectangles are sorted by the x-coordinate of their bottom left corners. For convenience, continue to refer to the resulting sequence as $R = \{R_1, R_2, \ldots, R_n\}$. For each rectangle R_i $(1 \leq i \leq n)$, i is said to be the identity of R_i. This step can be accomplished in $T_{Sort}(n, p, M)$ time.

Step 2. Next, solve the instance of the EV problem consisting of the set of top and bottom edges of rectangles, with the viewpoint ω at $(0, -\infty)$. For each b_i, compute the segments visible in the negative y-direction. Similarly, for each t_i compute the segments visible in the positive y-direction. This can be accomplished in $O(T_{EV}(n, p, M))$ time.

Step 3. With each endpoint associate a 4-tuple (L, U, x, TB), whose semantics are as follows: for each endpoint of a top segment, L is assigned the identity of its own rectangle and U is assigned the identity of the rectangle visible in the positive

y-direction (-1 if undefined). Similarly, for each endpoint of a bottom segment, U is assigned the identity of its own rectangle and L is assigned the identity of the rectangle visible in the negative y-direction (-1 if undefined). In both cases, TB is a bit indicating whether the endpoint belongs to a top or bottom segment, and x is the x-coordinate of the endpoint. Sort the set of tuples first by L and then by x. This is accomplished in $O(T_{Sort}(n, p, M))$ time.

Step 4. Now, consider the tuples (L_1, U_1, x_1, TB_1) and (L_2, U_2, x_2, TB_2) adjacent to each other in the sorted sequence. If $L_1 = L_2$ and $U_1 = U_2$ then record an edge in $\vec{D}$, from the rectangle corresponding to L_1 to the rectangle corresponding to U_1. Each edge is stored as (L_1, U_1). After sorting the resulting ordered pairs, the dominance graph can be constructed trivially. This step is also accomplished in $O(T_{Sort}(n, p, M))$ time.

In order to prove the correctness of this algorithm, it must be shown that the algorithm reports all directly above relations and no others. Consider first the situation where R_i is directly above R_j. A number of cases occur. For illustration, let us consider the case where both bottom endpoints of R_i report R_j as visible. The proofs of all the other cases are similar. Since both bottom endpoints report R_j as visible, both will set $U = i$ and $L = j$. Due to the assumption that R_i is directly above R_j, no other tuples can appear between these in the sorted sequence. Thus, the algorithm will report an edge in the dominance graph corresponding to these rectangles.

Next, consider the case where R_i is not directly above R_j. Let us distinguish between the following two cases.

Case 1. R_i is not above R_j. In this case R_i does not have any tuple containing the identity of R_j, so the edge between R_i and R_j cannot be reported.

Case 2. R_i is above R_j and there exists a rectangle R_k such that R_i is above R_k and R_k is above R_j. In this case the tuples containing information about R_i and R_j cannot occur consecutively. Again, the edge between R_i and R_j cannot be reported. This completes the proof of correctness. Thus, the following result is obtained.

Theorem 3.7. The DG problem for a set of n iso-oriented, non-overlapping rectangles in the plane, stored M rectangles per processor in the first $\frac{n}{M}$ processors of an ACM(n, p, M), can be solved in $T_{DG}(n, p, M)$=O$(T_{EV}(n, p, M))$+O$(T_{Sort}(n, p, M))$ time. □

CHAPTER 4

OBJECT VISIBILITY ON ENHANCED MESHES

The objective of this chapter is to present a detailed discussion on how the template algorithms designed for the class of object visibility problems on the abstract computational model are ported to the MMB and the RMESH.

In particular, Section 4.1 discusses the various tools designed for the MMB, Section 4.2 discusses the porting of template algorithms discussed in Chapter 3 to give time-optimal algorithms on the MMB, Section 4.3 discusses the tools for the RMESH and, finally, Section 4.4 discusses the O(1) time algorithms for object visibility problems on the RMESH, obtained by applying the template algorithms.

An MMB or RMESH of size $n \times n$ can be mapped to the abstract computational model ACM(n, p, M) as follows: Each processor of the MMB has O(1) memory registers. The n^2 processors of the MMB correspond to the n^2 processors of the ACM$(n, n^2, 1)$. A processor of the mesh, referred to as $P(i, j)$, where i is the row number and j is the column number to which the processor belongs, corresponds to the processor $P_{(i-1)n+j-1}$ in the ACM$(n, n^2, 1)$. The input for the various algorithms is assumed to be stored in the first row of the mesh, corresponding to the first n/M (here, $M = 1$) processors of the ACM$(n, n^2, 1)$.

4.1 TOOLS FOR THE MMB

Template algorithms for the object visibility problems, when ported to the MMB, yield time-optimal algorithms. Thus, in order to prove the time-optimality of each of these algorithms for this model of computation, the corresponding lower bound argument is also discussed. To port the various template algorithms to the MMB, there is a need to first discuss how the various operations assumed by the ACM are implemented on the MMB. These tools can then be applied to the template algorithm to obtain the required solutions.

Let us discuss how the various tools that are assumed by the ACM(n, p, M) are implemented on the MMB of size $n \times n$.

• *Broadcasting* : Processor $P(i, j)$ can broadcast the item it holds to every other processor in the MMB in O(1) time using the row and column buses. Thus, the broadcast operation can be performed on the MMB in O(1) time per data item.

• *Merging* : Recently, Olariu *et al.* [72] have proposed an O(1) time algorithm to merge two sorted sequences of total length n stored in one row of a MMB of size $n \times n$.

Here are the details of the algorithm for merging two sorted sequences $S_1 =< a_1, a_2, \ldots, a_r >$ and $S_2 =< b_1, b_2, \ldots, b_s >$, with $r + s = n$, stored in the first row of a MMB of size $n \times n$, with $P(1, i)$ holding a_i $(1 \leq i \leq r)$ and $P(1, r + i)$ holding b_i $(1 \leq i \leq s)$. To begin, using vertical buses, the first row is replicated in all rows of the MMB. Next, in every row i $(1 \leq i \leq r)$, processor $P(i, i)$ broadcasts a_i horizontally on the corresponding row bus. It is easy to see that for every i, a unique processor $P(i, r + j)$ $(1 \leq j \leq s)$, will find that $b_{j-1} < a_i \leq b_j$ (b_0 is taken to be $-\infty$). Clearly, this unique processor can now use the horizontal bus to broadcast j back to $P(i, i)$. In turn, $P(i, i)$ has enough information to compute the position

of a_i in S. In exactly the same way, the position of every b_j in S can be computed in O(1) time. Finally, a simple data movement sends every element to its final destination in the first row of the MMB.

Proposition 4.1. Two sorted sequences $S_1 =< a_1, a_2, \ldots, a_r >$ and $S_2 =< b_1, b_2, \ldots, b_s >$, with $r + s = n$, stored in the first row of a MMB of size $n \times n$, with $P(1,i)$ holding a_i $(1 \leq i \leq r)$ and $P(1, r+i)$ holding b_i $(1 \leq i \leq s)$, can be merged into a sorted sequence S in O(1) time. □

• *Sorting* : Proposition 4.1 is the main stepping stone for a time-optimal sorting algorithm developed in [72]. This algorithm implements the well-known strategy of sorting by merging. Here is a brief sketch of the data movement operations performed in the sorting algorithm of [72]. First, the input sequence is divided into a left subsequence containing the first $\lceil \frac{n}{2} \rceil$ items and a right subsequence containing the remaining $\lfloor \frac{n}{2} \rfloor$ items. Further, imagine dividing the original MMB into four equal submeshes of size $\frac{n}{2} \times \frac{n}{2}$. Note that for computational purposes, the north-west and south-east submeshes can be treated as independent MMB's.

In preparation for sorting, the right subsequence is broadcast to the first row of the south-eastern submesh. The algorithm then proceeds to recursively sort the data in each submesh. The resulting sorted subsequences are merged using the process described in Proposition 4.1. It is easy to see that the overall running time of this simple algorithm is $O(\log n)$.

Proposition 4.2. An n-element sequence of items from a totally ordered universe stored one item per processor in the first row of a MMB of size $n \times n$ can be sorted in $O(\log n)$ time. Furthermore, this is time-optimal. □

• *Compaction*: The details of a data movement that allows to compact a sequence by eliminating some of its elements is as follows. Supposing that the processors in

the first row of the MMB store a sequence $< a_1, a_2, \ldots, a_n >$ of items with some of the items marked. Assume further that every marked item knows its rank among the marked items. The aim is to obtain an ordered subsequence consisting of the marked elements stored, in order, in the leftmost positions of the first row of the MMB. This task can be performed as follows. Suppose that a_i is the k-th marked element in the sequence; processor $P(1,i)$ will broadcast a_i vertically to processor $P(k,i)$ which, in turn, will broadcast a_i horizontally to $P(k,k)$. Finally, $P(k,k)$ will broadcast a_i vertically to $P(1,k)$, as desired. Consequently, the following result is obtained.

Lemma 4.3. Consider a sequence $< a_1, a_2, \ldots, a_n >$ of items stored in the first row of a MMB of size $n \times n$, one item per processor, with some of the items marked. If every marked item knows its rank among the marked items, then an ordered subsequence consisting of the marked elements stored in order in the leftmost positions of the first row of the MMB can be obtained in O(1) time. □

4.2 OBJECT VISIBILITY ALGORITHMS ON THE MMB

This section involves a discussion on how the template algorithms for the class of object visibility problems discussed in Chapter 3 are instantiated in the context of the MMB using the tools developed in the Section 4.1.

4.2.1 ENDPOINT AND SEGMENT VISIBILITY

The purpose of this subsection is to demonstrate that the template algorithm 3.1 to solve SV and EV can be ported to the MMB to yield time-optimal solutions. Let us first discuss time lower bounds for the SV and the EV problems on the MMB. In

fact, the time lower bound also holds for the CREW-PRAM.

Let us briefly recall the definitions of the EV and SV problems. Given a set S of well ordered line segments, the EV problem asks to determine, for every endpoint e of a segment in S, the closest segments (if any) intersected by the rays $e\omega$ and $e\overline{\omega}$, in the directions towards and away from the view point ω respectively. The SV problem asks to compute the contour of S from ω i.e, the portions of the segments that are visible to an observer placed at ω.

The following discussion presents an $\Omega(\log n)$ lower bound for EV problem on the CREW-PRAM by reducing OR to EV. The well-known OR problem, given a sequence of n bits $b_1, b_2, \ldots, b_n$, asks for computing their logical OR. The following fundamental result of Cook *et al.* [29] that will be used in all the time lower bound arguments in this chapter and also in Chapter 7.

Proposition 4.4. The time lower bound for computing the OR of n bits on the CREW-PRAM is $\Omega(\log n)$ no matter how many processors and memory cells are used. □

In addition, the lower bound arguments rely on the following result of Lin *et al.* [52].

Proposition 4.5. Any computation that takes $O(t(n))$ computational steps on an n-processor MMB can be performed in $O(t(n))$ computational steps on an n-processor CREW-PRAM with $O(n)$ extra memory. □

It is important to note that Proposition 4.5 guarantees that if $T_M(n)$ is the execution time of an algorithm for solving a given problem on an n-processor MMB, then there exists a CREW-PRAM algorithm to solve the same problem in $T_P(n) = T_M(n)$ time using n processors and $O(n)$ extra memory. In other words, *too fast* an algorithm on the MMB implies *too fast* an algorithm for the CREW-PRAM. This

observation is exploited in [52] to transfer known time lower bounds for the PRAM to the MMB.

Let $b_1, b_2, \ldots, b_n$ be an arbitrary input to the OR problem. Now consider any algorithm that correctly solves the EV problem with ω at $(-\infty, 0)$ and with input z_0, z_1, z_2, ..., z_{n+1}, where z_i is the vertical segment with endpoints $bottom(z_i) = (i, 0)$ and $top(z_i) = (i, 3)$ in case $b_i = 1$, and the segment with endpoints $bottom(z_i) = (i, 0)$ and $top(z_i) = (i, 1)$ if $b_i = 0$. To complete the construction, we let z_0 and z_{n+1} be the segments with endpoints $bottom(z_0) = (0, 0)$ and $top(z_0) = (0, 2)$, and $bottom(z_{n+1}) = (n + 1, 0)$ and $top(z_{n+1}) = (n + 1, 3)$, respectively. The construction guarantees that the resulting set of segments is well ordered. Clearly, the answer to the OR problem is 0 if, and only if, the ray $top(z_0)\overline{\omega}$ encounters the segment z_{n+1}. The conclusion follows by Proposition 4.4.

Lemma 4.6. The task of solving the EV problem for a set of n well ordered line segments in the plane has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, no matter how many processors and memory cells are used. □

Lemma 4.6 and Proposition 4.5 combined, imply the following result.

Corollary 4.7. The task of solving the EV problem for a set of n well ordered line segments in the plane has a time lower bound of $\Omega(\log n)$ on the MMB of size $n \times n$.

□

It is now shown that the same lower bound applies to the SV problem. As before, this is achieved by reducing OR to SV. Let b_1, b_2, ..., b_n be an arbitrary input to the OR problem. Now consider any algorithm that correctly solves the SV problem with input z_1, z_2, ..., z_{n+1}, where z_i is the vertical segment with endpoints $(i, 0)$ and $(i, 1)$ in case $b_i = 1$, and the (degenerate) segment with endpoints $(i, 0)$ and $(i, 0)$ if $b_i = 0$. To complete the construction, let z_{n+1} be the segment with endpoints

$(n+1,0)$ and $(n+1,1)$ and place the viewpoint ω at $(0,1)$. The construction guarantees that the resulting set of segments is well ordered. Clearly, the answer to the OR problem is 0 if, and only if, the entire segment z_{n+1} is visible from ω. The conclusion follows by Proposition 4.4. Thus, the following result is obtained.

Lemma 4.8. The task of solving the SV problem for a set of n well ordered line segments in the plane has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, no matter how many processors and memory cells are used. □

Lemma 4.8 and Proposition 4.5 combined, imply the following result.

Corollary 4.9. The task of solving the segment visibility problem for a set of n well ordered line segments in the plane has a time lower bound of $\Omega(\log n)$ on the MMB of size $n \times n$. □

The next goal is to show that the time lower bounds of Corollaries 4.7 and 4.9 are tight, by devising an algorithm that solves an arbitrary instance of size n of the EV and SV problems in $O(\log n)$ time on a MMB of size $n \times n$. Consider an arbitrary set $S = \{s_1, s_2, \ldots, s_n\}$ of well ordered line segments, with every segment being specified by its endpoints. The set S is assumed to be stored, one segment per processor, in the first row of a MMB of size $n \times n$.

The terminology and data structures used in this algorithm are identical to that used by the template algorithm 3.1. Let us briefly discuss how the two stages of the template algorithm proceed, each involving processing the nodes of an abstract tree $\mathcal{T}$.

Stage 1. Consider a generic node v in $\mathcal{T}$ with left and right children u and w, respectively. Let $E(v)$ be the sequence of endpoints in segments $L(v)$ (set of leaf descendents of v). First, $E(v)$ is obtained by merging $E(u)$ and $E(w)$. By Proposition 4.1, this task is carried out in $O(1)$ time. Note that in the process of merging

$E(u)$ and $E(w)$ into $E(v)$, every endpoint e_i updates its predecessor and successor information in O(1) time. Updating t-blocked(e_i) and a-blocked(e_i) for endpoints $e_i \in E(u)$ or $e_i \in E(w)$ is thus accomplished in O(1) time. Since the processing of each level of $\mathcal{T}$ takes at most O(1) time, the over all running time of Stage 1 is $O(\log n)$.

Stage 2. As mentioned in the template algorithm, the main goal of this stage is to use the information obtained in Stage 1 to compute the actual values of t(e_i) and a(e_i) for every endpoint e_i.

Begin by sorting the endpoints of segments in S separately, first by a-blocked(e_i) and then by t-blocked(e_i). By Proposition 4.2 this operation can be performed in $O(\log n)$ time. As a result, the two sorted sequences are obtained: in the first one, all the endpoints that have the value a-blocked(e_i)=v occur consecutively, and will be referred to as BA(v). In the second one, all the endpoints that have the value t-blocked(e_i)=v occur consecutively, and will be denoted by BT(v). Both BT(v) and BA(v) feature endpoints sorted in increasing polar angle: this can be easily achieved by using two keys for sorting and the complexity will not be affected.

Equations 3.1 and 3.2 can be applied to obtain RC(v) and LC(v). Merge RC(u) and RC(w) into a sequence $E'(v)$, and again this can be accomplished in O(1) time. Next, delete the endpoints e_i from $E'(v)$ that have a-blocked(e_i)=v, and the items to be deleted are determined by merging $E'(v)$ with the sequence BA(v) that is readily available by virtue of the sorting step described above. Again, by Proposition 4.1, the merging operation runs in O(1) time. Every endpoint e_i whose a-blocked(e_i) value is 0 after node v has been processed, computes its rank in RC(v). Now, Lemma 4.3 guarantees that a compacted version of RC(v) can be obtained in

O(1) time. The computation of LC(v) is perfectly similar.

To determine the values of t(e_i) and a(e_i), merge RC(u) with BT(v) and LC(w) with BA(v) and the values of t(e_i) and a(e_i) for every endpoint in BT(v) and BA(v), respectively, can be determined in O(1) time. Thus the following result is obtained.

Theorem 4.10. An arbitrary n-segment instance of the EV problem can be solved in O($\log n$) time on a MMB of size $n \times n$. Furthermore, this is time-optimal. □

As mentioned in Chapter 3, the contours can be trivially computed from the solution to the EV problem, thus the following result is obtained.

Theorem 4.11. An arbitrary n-segment instance of the SV problem can be solved in O($\log n$) time on a MMB of size $n \times n$. Furthermore, this is time-optimal. □

4.2.2 DISK VISIBILITY

The purpose of this subsection is to show that the template algorithm 3.2 leads to a time-optimal solution to the DV problem when ported to the MMB. Recall the definition of the DV problem discussed in the Chapter 3: Given a set $D = \{d_1, d_2, \ldots, d_n\}$ of n non-overlapping opaque disks and a viewpoint ω in the plane, the DV problem involves determining the portions of each disk that are visible to an observer positioned at ω.

First, a $\Omega(\log n)$ lower bound is presented for DV problem on the CREW-PRAM model by reducing OR to DV. Let b_1, b_2, ..., b_n be an arbitrary input to the OR problem. Now, consider any algorithm that correctly solves the DV problem with ω at $(-\infty, 0)$ and with input $d_1, d_2, \ldots, d_{n+1}$, where d_i $(1 \leq i \leq n)$ is the disk of unit radius, centered at $(i, -1)$ if $b_i = 0$, and centered at $(i, 1)$ if $b_i = 1$. To complete the construction, add the disk d_{n+1} of unit radius centered at $(n+1, 1)$. This construction guarantees that the solution to OR is 0 if and only if d_{n+1} is visible

from ω. The conclusion follows by Proposition 4.4.

Lemma 4.12. The task of solving the disk visibility problem for a set of n disks in the plane has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, no matter how many processors and memory cells are used. □

Lemma 4.12 and Proposition 4.5 combined, imply the following result.

Corollary 4.13. The task of solving the disk visibility problem for a set of n disks in the plane has a time lower bound of $\Omega(\log n)$ on the MMB of size $n \times n$. □

Now, let us confirm that the running time of the DV algorithm for input size of n, obtained by applying template algorithm 3.2 to an MMB of size $n \times n$ is time-optimal i.e, had a running time of $O(\log n)$. Assume that an arbitrary set $D = \{d_1, d_2 \ldots, d_n\}$ of disks is stored, one disk per processor, in the first row of the MMB. The other assumptions about the position of the view point and the disks as well as the terminology is as described in the template algorithm 3.2.

In $O(1)$ time, the viewpoint ω is broadcast in the first row of the MMB and each processor holding a disk can determine the tangents to the disk from ω, as well as the length of these tangents. As described in the template algorithm, with every disk d_i associate the line segment s_i obtained by joining the corresponding tangency points. Sort the s_i's by increasing distance of their endpoints to ω. By Proposition 4.2, this can be done in $O(\log n)$ time. Apply the SV algorithm developed in the Subsection 4.2.1 to the sequence of sorted segments and this can be accomplished in $O(\log n)$ time. Once the visible portions of the segments are determined, the portions of the disks visible from ω can be trivially computed in $O(1)$ time. Thus, the following result is obtained.

Theorem 4.14. The DV problem for a set of n disks can be solved in $O(\log n)$ time on a MMB of size $n \times n$. Furthermore, this is time-optimal. □

4.2.3 RECTANGLE VISIBILITY

The purpose of this subsection is to show that the template algorithm 3.3 for the RV problem, when ported to the MMB, results in a time-optimal algorithm. First, let us establish an $\Omega(\log n)$ lower bound for the RV problem on the CREW-PRAM model by reducing the OR problem to RV. Let b_1, b_2, ..., b_n be an arbitrary input to the OR problem. Now consider any algorithm that correctly solves the instance of the RV problem with ω at $(-\infty, 0)$ and with input $R_1, R_2, \ldots, R_{n+1}$, where R_i $(1 \leq i \leq n)$ is the rectangle with top-left corner at $(i, 2)$ and bottom-right corner at $(i + 0.5, 0)$ in case $b_i = 1$, and with top-left corner at $(i, 1)$ and bottom right corner at $(i + 0.5, 0)$ otherwise. To complete the construction, add the rectangle R_{n+1} with with top-left and bottom-right corners at $(n+1, 2)$ and $(n+1.5, 0)$. This construction guarantees that the solution to OR is 0 if and only if R_{n+1} is visible from ω. The conclusion follows by Proposition 4.4. The following result is thus obtained.

Lemma 4.15. The task of solving the RV problem for a set of n iso-oriented rectangles in the plane has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, no matter how many processors and memory cells are used. □

Lemma 4.15 and Proposition 4.5 combined, imply the following result.

Corollary 4.16. The task of solving the RV problem for a set of n iso-oriented rectangles in the plane has a time lower bound of $\Omega(\log n)$ on the MMB of size $n \times n$. □

Now, let us discuss the porting of template algorithm 3.3 to the MMB and confirm that the resulting algorithm is time-optimal, i.e, it has a running time of $O(\log n)$. Consider a set $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ of iso-oriented, non-overlapping, rectangles stored one per processor in the first row of a MMB of size $n \times n$. Sort the

top and bottom edges by increasing y-coordinate, and apply the EV algorithm for the resulting set of ordered segments. This can be done in $O(\log n)$ time. Repeat the same for the vertical segments of every rectangle.

As described in the template algorithm, every generic corner e_i of rectangle r_i has four solutions: a1(e_i), t1(e_i), a2(e_i), and t2(e_i). A corner e_i is *marked* if t1(e_i)=t2(e_i)=0. Now, every marked corner e_i combines the information stored in a1(e_i) and a2(e_i) by selecting the segment closer to e_i along the ray $e_i\overline{\omega}$. If in the process e_i discovers that the closer of a1(e_i) and a2(e_i) is an edge that belongs to its own rectangle, then e_i becomes unmarked. Sort the remaining marked corners by increasing polar angle, and the contour of the set of rectangles can now be computed as specified in the template algorithm. The following result is thus obtained.

Theorem 4.17. An arbitrary instance of size n of the RV problem can be solved in $O(\log n)$ time on a MMB. Furthermore, this is time-optimal. □

4.2.4 DOMINANCE GRAPH

This subsection discusses the DG problem in the context of MMB's where the template algorithm 3.4, can be ported to obtain a time-optimal solution to the problem.

First, the lower bound of $\Omega(\log n)$ is established for the DG problem on both the CREW-PRAM and the MMB. As usual, this is done by reducing the OR problem to DG. Let $b_1, b_2, \ldots, b_n$ be an arbitrary input to the OR problem. Based on this sequence, construct an instance $\mathcal{R} = \{R_0, R_1, \ldots, R_n\}$ of the DG problem as follows:

- the bottom-left and the top-right corners of R_0 are $(0, -1)$ and $(n, -0.75)$;
- if $b_i = 0$, then the bottom-left and the top-right corners of R_i are $(n + i - 0.75, 0)$ and $(n + i - 0.25, 1)$;
- if $b_i = 1$, then the bottom-left and the top-right corners of R_i are $(i - 0.75, 0)$ and $(i - 0.25, 1)$.

Clearly this construction takes O(1) time. It is easy to verify that the solution to the OR problem is 0 if, and only if, the out-degree of the vertex corresponding to R_0 is 0.

The conclusion follows by Proposition 4.4. Thus, the following result is obtained.

Lemma 4.18. The DG problem for a set of n non-overlapping iso-oriented rectangles in the plane has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, no matter how many processors and memory cells are used. □

Lemma 4.18 and Proposition 4.5 combined, imply the following result.

Corollary 4.19. The DG problem for a set of n non-overlapping iso-oriented rectangles in the plane has a time lower bound of $\Omega(\log n)$ on the MMB of size $n \times n$. □

Consider an arbitrary instance of size n of the DG problem stored in the first row of a MMB of size $n \times n$. Sort the rectangles sorted by the x-coordinate of their bottom left corners in $O(\log n)$ time. Let the sorted sequence be $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$. Solve the instance of the EV problem consisting of the set of top and bottom edges of rectangles, with the viewpoint ω at $(0, -\infty)$. By virtue of Theorem 4.10, this step can be performed in $O(\log n)$ time. As in the template algorithm, with each endpoint associate a 4-tuple (L, U, x, TB). For each endpoint of a top segment, sort the set of tuples first by L and then by x. This step takes $O(\log n)$ time. Now, consider the tuples (L_1, U_1, x_1, TB_1) and (L_2, U_2, x_2, TB_2) in adjacent processors. If $L_1 = L_2$ and $U_1 = U_2$ then record an edge in $\vec{D}$, from the rectangle corresponding to L_1 to the rectangle corresponding to U_1. Each edge is stored as (L_1, U_1). After sorting the resulting ordered pairs, the dominance graph can be constructed trivially. This leads to the following result.

Theorem 4.20. Given a set $\mathcal{R}$ of n rectangles stored in the first row of the MMB of size $n \times n$, the DG problem can be solved in $O(\log n)$ time. Furthermore, this is time-optimal. □

4.3 TOOLS FOR THE RMESH

This section discusses the tools required to solve the object visibility problems in the context of the RMESH. The various template algorithms discussed in Chapter 3 can be applied to obtain O(1) time solutions to the object visibility problems using the collect of tools discussed in this section. However, the EV/SV problem is solved independent of the template algorithm and the power of dynamically reconfigurable bus system can be exploited to obtain a much simpler, O(1) time solution.

The purpose of this section is to discuss how the various operations assumed by the ACM are implemented on a RMESH. The operations or tools are then applied to the various template algorithms discussed in Chapter 3 to obtain O(1) time solutions to the various object visibility problems.

• *Broadcasting* : Processor $P(i,j)$ can broadcast the item it holds to every other processor in the mesh in O(1) time by configuring the bus appropriately. Thus, the broadcast operation can be performed on the RMESH in O(1) time per item.

• *Merging* : Recently, Olariu *et al.* [70] have proposed the following result.

Proposition 4.21. Let $S_1 = < a_1, a_2, \ldots, a_r >$ and $S_2 = < b_1, b_2, \ldots, b_s >$, with $r + s = n$, be sorted sequences stored in the first row of a RMESH of size $n \times n$, with $P(1,i)$ holding a_i $(1 \leq i \leq r)$ and $P(1, r+i)$ holding b_i $(1 \leq i \leq s)$. The two sequences can be merged into a sorted sequence in O(1) time. □

• *Sorting* : Recently, Lin *et al.* [51], Jang and Prasanna [46], and Nigam and Sahni [68] have shown that an n-element sequence of items chosen from a totally ordered

universe can be sorted in O(1) time on a RMESH of size $n \times n$. Furthermore, this result achieves the VLSI lower bound for the problem.

Proposition 4.22. An n-element sequence from a totally ordered universe can be sorted in O(1) time on a RMESH of size $n \times n$. □

4.4 OBJECT VISIBILITY ALGORITHMS ON THE RMESH

This section provides O(1) time algorithms for the various object visibility problems on the RMESH by applying the template algorithms from Chapter 3 can be applied for the DV, RV and DG problems. However, the solution to the SV/EV problem is much simpler because of the powerful bus system available.

4.4.1 ENDPOINT AND SEGMENT VISIBILITY

This subsection presents a single algorithm that implements EV and SV problems in O(1) time on the RMESH. The powerful bus system of this parallel machine, makes it unnecessary to use the tree-fashioned computation described in the template algorithm. The details of the algorithm for the RMESH is as follows:

Consider a set of n segments stored, one segment per processor, in the first row of a RMESH, $\mathcal{M}$, of size $n \times n$ such that $P(1,i)$ stores s_i. The idea of the algorithm is to dedicate row i of $\mathcal{M}$ to segment s_i. For this purpose, after having established vertical buses in all columns of the mesh, mandate the processors in the first row to broadcast the segment they hold on the bus in their own column, thus replicating S in all rows of $\mathcal{M}$. Next, in every row of the mesh the processors connect their ports E and W. Let e be a generic endpoint of s_i. To determine $l(e)$, processor $P(i,i)$ broadcasts e westbound on the horizontal bus in row i. Every

processor P(i,j) $(j < i)$ checks whether the ray e^- intersects the s_j. If so, P(i,j) disconnects the horizontal bus and broadcasts the identity of s_j eastbound from its port E. Since the segments are well ordered, the information (if any) received by $P(i,i)$ from its port W is precisely $l(e)$. In case no information is received, $l(e)$ is set to $-\infty$. Thus,, the following result is obtained.

Theorem 4.23. Given a set S of n well ordered segments in the plane, stored in the first row of a RMESH of size $n \times n$, the corresponding instance of the EV problem can be solved in O(1) time. □

Once the solution to EV problem is obtained, the solution to the SV problem can be obtained in O(1) time. Thus, the following result is obtained.

Theorem 4.24. Given a set S of n well ordered segments in the plane, stored in the first row of a RMESH of size $n \times n$, the corresponding instance of the SV problem can be solved in O(1) time. □

4.4.2 DISK VISIBILITY

In this subsection, the template algorithm for DV problem presented in Section 3.3 of Chapter 3 is instantiated in the context of the RMESH to obtain an O(1) time solution. Consider a set of n non-overlapping disks in the plane, $D = \{d_1, d_2, \ldots, d_n\}$, stored one disk per processor in the first row of the RMESH of size $n \times n$. As in the template algorithm 3.2, each processor in the first row of the mesh, determines the tangents to the disk it stores, from the viewpoint ω. The length of these tangents, i.e. the distance between ω and the tangency points, is also determined. This would require O(1) computation time. As before, with every disk d_i associate the line segment s_i obtained by joining the corresponding tangency points. Next, sort the s_i's by increasing distance of their endpoints to ω. This is done in O(1) time, by virtue of Proposition 4.22. Let S=$s_1, s_2, \ldots, s_n$ be the set of these segments in sorted order

and the SV algorithm can be applied to S. Once the visible portions of the segments are determined, the portions of the disks visible from ω can be trivially computed. This step would require O(1) time. Thus the following result is obtained.

Theorem 4.25. Given a set D of n non-intersecting disks in the plane, stored in the first row of a RMESH of size $n \times n$, the corresponding instance of the DV problem can be solved in O(1) time. □

4.4.3 RECTANGLE VISIBILITY

In this subsection, the template algorithm 3.3 for RV problem is applied to obtain a O(1) solution to the problem on the RMESH. Consider a set $\mathcal{R} = \{R_1, R_2 \ldots, R_n\}$ of n non-overlapping, opaque rectangles in the plane with edges parallel to the axes, stored one rectangle per processor in the first row of a RMESH $\mathcal{M}$ of size $n \times n$. Sort the top and bottom edges of the rectangles in $\mathcal{R}$ by increasing y-coordinate, and apply the EV algorithm to the resulting sequence of well ordered segments. Repeat the same for the top and bottom edges, after sorting them in increasing order of their x-coordinates. Combine the solutions obtained above as described in template algorithm 3.3. This can be accomplished in O(1) by virtue of Proposition 4.22 and Theorem 4.23. Thus, the following result is obtained.

Theorem 4.26. Given a set $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$, of n iso-oriented, non-overlapping rectangles stored one per processor on a RMESH of size $n \times n$, the corresponding instance of the RV problem can be solved in O(1) time. □

4.4.4 DOMINANCE GRAPH

In this subsection, let us discuss the O(1) time solution to the DG problem on the RMESH obtained by porting the template algorithm 3.4.

Consider an arbitrary instance of size n of the DG problem stored one rectangle per processor in the first row of the RMESH of size $n \times n$. The rectangles are

sorted by the x-coordinate of their bottom left corners. For convenience, continue to refer to the resulting sequence as $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$. Next, solve the instance of the EV problem consisting of the set of top and bottom edges of rectangles, with the viewpoint ω at $(0, -\infty)$. This can be accomplished in O(1) time, by virtue of Theorem 4.23. With each endpoint associate a 4-tuple (L, U, x, TB) as described in the template algorithm. Sort the set of tuples first by L and then by x. This is accomplished in O(1) time, as stated in Proposition 4.22. Now, consider the tuples (L_1, U_1, x_1, TB_1) and (L_2, U_2, x_2, TB_2) adjacent to each other in the sorted sequence. If $L_1 = L_2$ and $U_1 = U_2$ then record an edge in $\vec{D}$, from the rectangle corresponding to L_1 to the rectangle corresponding to U_1. Each edge is stored as (L_1, U_1). After sorting the resulting ordered pairs, the dominance graph can be constructed trivially. This step is also accomplished in O(1) time, by virtue of Proposition 4.22. Thus, the following result is obtained.

Theorem 4.27. The DG problem for a set of n iso-oriented, non-overlapping rectangles in the plane can be solved in O(1) time on a RMESH of size $n \times n$. □

CHAPTER 5

OBJECT VISIBILITY ON COARSE-GRAIN MULTICOMPUTERS

The objective of this chapter is to present a detailed discussion on how the template algorithms, designed for the class of object visibility problems on the abstract computational model, are ported to coarse-grain multicomputers. In particular, Section 5.1 discusses the various tools developed for coarse-grain multicomputers, and Section 5.2 discusses the porting of the template algorithms for object visibility problems for this model.

Recall that a coarse-grain multicomputer, referred to as CGM(n,p), consists of p processors, each having $O(\frac{n}{p})$ local memory. The p processors, enumerated as $P_0, P_1, \ldots, P_{p-1}$, are assumed to be connected through an arbitrary interconnection network and communicate using various communication primitives as described in Chapter 2.

In this model, an algorithm is said to be computationally optimal whenever the computational time of the algorithm is $O(\frac{f(n)}{p}))$, where $\Omega(f(n))$ is the sequential lower bound for the problem. However, since the communication across various processors is an expensive operation, the objective in designing solutions to various problems in this model is to minimize the number of communication rounds, while

keeping the amount of computation as low as possible. The running time of the algorithm is the sum of the total time spent on computation by of the p processors and the total time spent on interprocess communication.

The CGM(n,p) can be be viewed as an ACM$(n,p,\frac{n}{p})$, where the p processors of the CGM correspond to the p processors of the ACM, each of them having O(M)=O($\frac{n}{p}$) local memory. In the various algorithms every processor, P_i $(0 \leq i \leq p-1)$, of the CGM(n,p) is assumed to store $\frac{n}{p}$ of the input items. The CGM(n,p) can be viewed as independent CGMs by dividing the p processors into disjoint process groups as mentioned in Chapter 2.

5.1 TOOLS

In purpose of this section is to devise a variety of tools that are useful in porting the template algorithms to the CGM(n,p). The various operations assumed by the ACM in Chapter 3 are implemented on the CGM as follows:

• *Broadcasting* : The broadcast operation assumed by the ACM can be implemented using the broadcast primitive available, in $T_{Broadcast}(k,p)$ time, where k $(1 \leq k \leq \frac{n}{p})$ is the number of data items to be broadcast.

• *Merging* : The merge operation is performed on the CGM(n,p) as described in Subsection 5.1.2.

• *Sorting* : The sort operation is performed on the CGM(n,p) as described in Subsection 5.1.3.

• *Compaction* : The compaction operation is performed as specified in Subsection 5.1.4.

Before discussing the implementation details of these basic tools, a dynamic load balancing scheme is discussed in Subsection 5.1.1. This scheme plays a very crucial

role in the design of basic tools such as merging and sorting.

5.1.1 DYNAMIC LOAD BALANCING

Several problems on the CGM(n,p) can be classified as problems that require dynamic balancing of the load on the various processors depending on the particular instance of the input. The situation in which this scheme is needed is described as below.

Given the following input:

- A sequence $S = < s_1, s_2, \ldots, s_n >$ of n items stored $\frac{n}{p}$ per processor in a CGM(n,p), where any processor P_i stores the subsequence of items $S_i = < s_{(i*\frac{n}{p})+1}, \ldots, s_{i*\frac{n}{p}} >$. Every item $s_i \in S$ is associated with a *solution*, depending on the problem to which the dynamic load balancing scheme is being applied. Thus, it is required to determine the *solution* to every $s_j \in S$.

- A sequence $D = < d_1, d_2, \ldots, d_n >$ of n elements stored $\frac{n}{p}$ per processor in a CGM(n,p), where each processor P_i stores a subsequence of items $D_i = < d_{(i*\frac{n}{p})+1}, \ldots, d_{i*\frac{n}{p}} >$. Each D_i is referred to as a *pocket*. The solution to each $s_j \in S$ is determined by exactly one pocket $D_i < i < \frac{n}{p}$.

- A sequence $B = < b_1, b_2, \ldots, b_n >$ of n elements stored $\frac{n}{p}$ per processor in a CGM(n,p), where each processor P_i stores the subsequence of items $B_i = < b_{(i*\frac{n}{p})+1}, \ldots, b_{i*\frac{n}{p}} >$. Every element $b_j \in B$, is the subscript of the pocket D_{b_j} which determines the solution to the item $s_j \in S$.

Thus, every processor P_i is given B_i, the sequence corresponding to the pocket to which each $s_j \in S_i$ belongs, and has to determine the solution to every s_j. For every item $s_j \in S_i$ with $b_j = i$, the solution can be determined sequentially within the processor. However, if b_j is not equal to i, there is a need to send every such s_j to

the processor storing the pocket D_{b_j}.

Let N_i be the number of items $s_j \in S$, such that $b_j = i$. In general, the value of N_i $(0 \leq i \leq p-1)$ may vary from 0 to $O(n)$ depending on the particular instance of the input. Since, a processor has at most $O(\frac{n}{p})$ memory, atmost $O(\frac{n}{p})$ items with $b_j = i$ can be sent to the processor storing D_i, at one time. This motivates the need to schedule the movement of the every $s_j \in S$, in order to determine its solution. In this section, the dynamic load balancing scheme provides a solution to this scheduling problem. The various steps involved in obtaining the solution of every s_j, using the dynamic load balancing scheme, is discussed below:

Step 1. The purpose of this step is to determine N_i for every pocket D_i. Every processor P_l $(0 \leq l \leq p-1)$ determines the number C_{lk} of items $s_j \in S_l$ such that $b_j = k$. This takes $O(\frac{n}{p})$ computation time. Next, every P_l obtains information about $C_{0l}, C_{1l}, \ldots, C_{(p-1)l}$ from processors $P_0, P_1, \ldots, P_{p-1}$ respectively. This step takes $T_{Alltoall}(p,p)$ time where each processor P_m sends the values $C_{m0}, C_{m1}, \ldots, C_{m(p-1)}$ to processors $P_0, P_1, \ldots, P_{p-1}$, respectively. Upon receiving $C_{0l}, C_{1l}, \ldots, C_{(p-1)l}$ from every processor, P_l determines their sum in $O(p)$ time, to obtain the value N_l. The p items $N_0, N_1, \ldots, N_{p-1}$ are replicated in each of p processors using an all-gather operation. This step takes a communication time of $T_{Allgather}(p,p)$.

Let $c * \frac{n}{p}$ (where c is an integer constant greater than or equal to 2) be a value that is known to every P_i. Now, a pocket D_k is said to be *sparse* if N_k is less than or equal to $c * \frac{n}{p}$; otherwise D_k is said to be *dense*. In $O(\frac{n}{p})$ time, every P_i $(0 \leq i \leq p-1)$ determines for every $b_j \in B_i$, whether D_{b_j} is a dense pocket or not.

Step 2. The aim of this step is to obtain the solution of every item $s_j \in S$ where pocket D_{b_j} is sparse.

Let every P_i send $s_j \in S_i$, to processor P_{b_j}, storing the pocket D_{b_j}, where pocket D_{b_j} is sparse. This can be accomplished by performing an all-to-all communication operation. Note that, any processor P_i would receive at most $O(\frac{n}{p})$ items. This step would take $T_{Alltoall}(n,p)$ time for the communication operation. The solution to every item s_j that is sent to the processor storing the pocket containing its solution, can now be determined sequentially in each of processors P_i storing a sparse pocket. Let the time taken for this computation be $O(f(\frac{n}{p}))$. The solutions can be sent back by performing a reverse data movement to the one performed earlier in $T_{Alltoall}(n,p)$ time.

Step 3. Finally, let us determine the solution to every $s_j \in S$, where pocket D_{b_j} is dense. In order to ensure that atmost $O(\frac{n}{p})$ such s_j's are moved to any processor, there is a need to make copies of every dense pocket D_k. This is accomplished as follows.

Let n_d be the number of dense pockets. Determine the number of copies that each dense pocket D_k should have, and is given by $\mathcal{N}_k = \frac{N_k}{c*\frac{n}{p}}$.

Observation 5.1. The total number of copies of all the dense pockets D_k's given by $\mathcal{N}_0+\mathcal{N}_1+\ldots+\mathcal{N}_{n_d-1}$ is no more than $\frac{p}{2}$. □

Let the n_d dense pockets be enumerated as $D_{m_1}, D_{m_2}, \ldots, D_{m_{n_d}}$ in increasing order of their subscripts. Similarly, let the $p - n_d$ sparse pockets be enumerated as $D_{q_1}, D_{q_2}, \ldots, D_{q_{p-n_d}}$ in increasing order of their subscripts. Since, the sparse pockets are already processed, the processors storing them are *marked* as available to hold copies of the dense pockets. Let the marked processors be enumerated as $P_{q_1}, P_{q_2}, \ldots, P_{q_{p-n_d}}$. Let every processor P_i, such that D_i is a dense pocket, retain a copy of pocket D_i. Now, the rest of the copies of each of the dense pockets are scheduled among the marked processors $P_{q_1}, P_{q_2}, \ldots, P_{q_{p-n_d}}$. The scheduling of the

copies is done as follows. The copies of D_{m_1} are assigned to the first $\mathcal{N}_{m_1} - 1$ marked processors. The copies of D_{m_2} are assigned the next $\mathcal{N}_{m_2} - 1$ processors, and so on.

Now, each of the processors that should be storing the copy of the a dense pocket D_k, including P_k, join a process group. Note that, there are exactly n_d process groups. Now, in a broadcast operation in each of the process groups, every processor P_l can obtain the copy of the dense pocket it is to store. Note that this operation can be performed using an all-to-all communication operation which takes $T_{Alltoall}(n,p)$ time.

Since there may be several copies of a dense pocket D_k, each processor P_i needs to determine to which copy it has to send its items s_j with $b_j = k$. This can be accomplished as follows: for each dense pocket D_k, the processor P_k is aware of $C_{0k}, C_{1k}, \ldots, C_{(p-1)k}$, and performs a prefix sum on this sequence giving the sequence $Q_{0k}, Q_{1k}, \ldots, Q_{(p-1)k}$. Every Q_{ik} is sent to processor P_i. This could also be performed in one all-to-all communication operation, in $T_{Alltoall}(p^2,p)$ time. Note that, at this stage, every processor P_i has information to determine to which processors each of the unsolved items $s_j \in S_i$ is to be sent.

Now, move the unsolved items $s_j \in S_i$ from every processor P_i to the processor containing the copy of dense pocket D_k determined in the previous step. The solution to each one of them is then determined in $O(f(\frac{n}{p}))$ time and sent back to the corresponding processor. Thus, the required dynamic load balancing operation is accomplished and the solutions for every $s_j \in S$ is determined.

Lemma 5.2. An instance of size n of a problem applying the dynamic load balancing scheme can be solved in $O(\frac{n}{p})+O(f(\frac{n}{p}))$ computational time, where function f depends on the particular problem, and a communication time of $O(T_{Alltoall}(n,p))$.

□

5.1.2 MERGING

In this subsection, the solution to the merge problem on a CGM(n,p) is presented. This solution uses the dynamic load balancing scheme discussed in Subsection 5.1.1. The computation time of the algorithm is $O(\frac{n}{p})$, and since the sequential lower bound of the merge problem is $\Omega(n)$, this algorithm is computationally time-optimal.

Let $S_1 = < a_1, a_2, \ldots, a_{\frac{n}{2}} >$ and $S_2 = < b_1, b_2, \ldots, b_{\frac{n}{2}} >$, be two sorted sequences of $\frac{n}{2}$ items each. Let S_1 be stored in processors $P_0, P_1, \ldots, P_{\frac{p}{2}-1}$ of the CGM(n,p), $\frac{n}{p}$ per processor. Similarly, let S_2 be stored in $P_{\frac{p}{2}}, P_{\frac{p}{2}+1}, \ldots, P_{p-1}$, $\frac{n}{p}$ per processor. Any P_i $(0 \leq i \leq \frac{p}{2}-1)$ stores items $S_{i1} = < a_{i*\frac{n}{p}+1}, \ldots, a_{(i+1)*\frac{n}{p}} >$ belonging to S_1. Similarly, any P_i $(\frac{p}{2} \leq i \leq p-1)$ stores items $S_{i2} = < b_{(i-\frac{p}{2})*\frac{n}{p}+1}, \ldots, b_{(i-\frac{p}{2}+1)*\frac{n}{p}} >$ belonging to S_2. The two sequences S_1 and S_2 are to be merged into a sorted sequence $S = < c_1, c_2, \ldots, c_n >$, so that any processor P_i stores items $< c_{i*\frac{n}{p}+1}, \ldots, c_{(i+1)*\frac{n}{p}} >$ in the sorted sequence. Define the *rank* of an item e in any sorted sequence $Q = < q_1, q_2, \ldots, q_r >$ as the number of items in the sequence Q that are less than the item e, and is denoted as $rank(e, Q)$. In order to merge the sequences S_1 and S_2, determine $rank(a_i, S)$ for every $a_i \in S$ and $rank(b_j, S)$ for every $b_j \in S_2$. First, determine the $rank(a_i, S_2)$ for every $a_i \in S_1$. The sum of $rank(a_i, S_2)$ and $rank(a_i, S_1)$ given by i, gives the value of $rank(a_i, S)$. Similarly, $rank(b_j, S_1)$ and $rank(b_j, S_2)$ is to be determined for every $b_j \in S_2$, to obtain the value of $rank(b_j, S)$. This can be accomplished as described in the following steps.

Step 1. Let every processor P_m $(0 \leq m \leq \frac{p}{2}-1)$ set the value of the $rank(a_i, S_1)$ to i, for every $a_i \in S_{m1}$. Similarly, let every processor P_m $(\frac{p}{2} \leq m \leq (p-1))$ set the value of the $rank(b_j, S_2)$ to j, for every $b_j \in S_{m2}$. This can be accomplished in $O(\frac{n}{p})$ time.

Step 2. Every processor P_m determines the *largest* item it holds, and that is re-

ferred to as the *sample* item l_m. Since the sequence of items stored by any P_m are already sorted, the value of l_m can be obtained in O(1) time. Now, perform an all-gather operation so that every processor has a copy of the sequence of sample items $L = < l_0, l_1, \ldots, l_{p-1} >$. This can be accomplished in $T_{Allgather}(p, p)$.

In every P_m $(0 \leq m \leq \frac{p}{2} - 1)$, perform the following computation in parallel. Determine the pocket for every $a_i \in S_{m1}$, where pocket for any a_i is determined as follows. Given the sequence of sample items $L = < l_0, l_1, \ldots, l_{p-1} >$, a_i finds its rank in $L_2 = < l_{\frac{p}{2}}, \ldots, l_{(p-1)} >$ (L_2 is determined from L). The value $rank(a_i, L_2)$ corresponds to the pocket of a_i. Similarly, in every P_m $(\frac{n}{2} \leq m \leq (p-1))$, perform the following computation in parallel. Determine the pocket for every $b_j \in S_{m2}$, where pocket for any b_j is determined as follows. Given the sequence of sample items $L = < l_0, l_1, \ldots, l_{p-1} >$, b_j finds its rank in $L_1 = < l_0, \ldots, l_{\frac{p}{2}-1} >$ (L_1 is determined from L). The value $rank(b_j, L_1)$ gives the pocket of b_j.

Observation 5.3. The value of $rank(a_i, S_{k2})$, where k is the pocket of a_i, gives the rank of a_i in the sorted list S_2 as $rank(a_i, S_2)$=$rank(a_i, S_{k2})$+$(k - \frac{p}{2}) * \frac{n}{p}$. Similarly, the value of $rank(b_j, S_{k1})$, where k is the pocket of b_j, gives the rank of b_j in the sorted list S_1 as $rank(b_j, S_1)$=$rank(b_j, S_{k1})$+$(k * \frac{n}{p})$. □

Now, each of the items $a_i \in S_1$ with pocket k, has to calculate $rank(a_i, S_{k2})$, in order to determine $rank(a_i, S)$. Also, each item $b_j \in S_2$ with pocket k, has to calculate $rank(b_j, S_{k1})$. In the worst case, it is possible that all the a_i's have the same pocket and all the b_j's have the same pocket. Thus, there is a need to apply the dynamic load balancing scheme.

Step 3. The load balancing scheme is applied to determine the $rank(a_i, S_{k2})$ for every $a_i \in S_1$ and $rank(b_j, S_{k1})$ for every $b_j \in S_2$. This can be performed as described in Subsection 5.1.1 in $O(\frac{n}{p})$ computational time and $O(T_{Alltoall}(n, p))$ communication

time. Now, determine the rank of every $a_i \in S_1$, in the sorted sequence S as $rank(a_i, S_1) + rank(a_i, S_2)$. Equivalent computation is performed for every item $b_j \in S_2$.

Step 4. Once every item $a_i \in S_1$ and $b_j \in S_2$ determines its rank in S, denoted as $rank(a_i, S)$ and $rank(b_j, S)$, respectively, the destination processor for each item a_i is determined as $\lfloor \frac{rank(a_i,S)}{\frac{n}{p}} \rfloor$ and for b_j as $\lfloor \frac{rank(b_j,S)}{\frac{n}{p}} \rfloor$, respectively. This is accomplished in $O(\frac{n}{p})$ time. In one all-to-all communication operation, the items can be moved to their final positions giving the sorted sequence S. This step requires $T_{Alltoall}(n,p)$ communication time. Thus the following result is obtained.

Lemma 5.4. Consider two sorted sequences, $S_1 = < a_1, a_2, \ldots, a_{\frac{n}{2}} >$, $S_2 = < b_1, b_2, \ldots, b_{\frac{n}{2}} >$, stored $\frac{n}{p}$ per processor, with S_1 stored in processors $P_0, P_1, \ldots, P_{\frac{p}{2}-1}$ and S_2 in processors $P_{\frac{p}{2}}, P_{\frac{p}{2}+1}, \ldots, P_{p-1}$, of a CGM$(n,p)$. The two sequences can be merged in $O(\frac{n}{p})$ computational time, and $O(T_{Alltoall}(n,p))$ communication time. □

5.1.3 SORTING

Lemma 5.4 is the main stepping stone for the sorting algorithm developed in this section. This algorithm implements the well-known strategy of sorting by merging. The computational time of the algorithm is $O(\frac{n \log n}{p})$ and since the sequential lower bound for sorting is $\Omega(n \log n)$, this algorithm is computationally time-optimal.

Let $S = < a_1, a_2, \ldots, a_n >$ be a sequence of n items from a totally ordered universe, stored $O(\frac{n}{p})$ per processor on a CGM(n,p), where any processor P_i stores the items $a_{(i*\frac{n}{p})+1}, \ldots, a_{i*\frac{n}{p}}$. The sorting problem requires the sequence S to be sorted in a specified order and the resulting sequence of items $< b_1, b_2, \ldots, b_n >$, are stored $\frac{n}{p}$ per processor so that any processor P_i stores the items, $< b_{(i*\frac{n}{p})+1}, \ldots, b_{i*\frac{n}{p}} >$ The details of the algorithm are as follows:

First, the input sequence is divided into a left subsequence containing the first $\frac{n}{2}$

items and a right subsequence containing the remaining $\frac{n}{2}$ items. Further, imagine dividing the original CGM(n,p) into two independent machines, CGM$(\frac{n}{2},\frac{p}{2})$. This can be accomplished by dividing the p processors into two process groups having $\frac{p}{2}$ processors each.

The algorithm proceeds to recursively sort the data in each of the two CGM's. The resulting sorted subsequences are merged using the algorithm described in Subsection 5.1.2. The recursion terminates when each of the CGM's is a CGM$(\frac{n}{p},1)$, and the data items can be sorted using the sequential algorithm running in $O(\frac{n\log n}{p})$ time. It is easy to see that the overall running time of this simple algorithm is $O(\frac{n\log n}{p})$ computation time and $O(\log p T_{Alltoall}(n,p))$ communication time.

Lemma 5.5. Given a sequence $S = < a_1, a_2, \ldots, a_n >$ of n items from a totally ordered universe, stored $O(\frac{n}{p})$ per processor on a CGM(n,p), sorting of the sequence can be accomplished in $O(\frac{n\log n}{p})$ computation time and $O(\log p T_{Alltoall}(n,p))$ communication time. □

5.1.4 COMPACTION

The compaction operation involves a sequence of items $S = < a_1, a_2, \ldots, a_n >$ stored $\frac{n}{p}$ items per processor, in the p processors of an CGM(n,p), with r $(1 \leq r \leq n)$, items *marked*. The marked items are enumerated as $B = < b_1, b_2, \ldots, b_r >$ and every a_i $(0 \leq i \leq n)$ knows its *rank* in the sequence B. The result of the compaction operation is to obtain the ordered sequence B, in order, in the first $O(\lceil \frac{r}{\frac{n}{p}} \rceil)$ processors storing S, so that any processor P_i $(0 \leq i \leq \lceil \frac{r}{\frac{n}{p}} \rceil)$ stores items $b_{i*\frac{n}{p}+1}, \ldots, b_{(i+1)*\frac{n}{p}}$. This data movement operation can be accomplished by determining the destination processors for each of the marked items as $\lfloor \frac{rank}{\frac{n}{p}} \rfloor$ in $O(\frac{n}{p})$ computational time, followed by an all-to-all operation to move the marked items to their destination processors. This can be accomplished in $T_{Alltoall}(n,p)$ time.

Thus the following result is obtained.

Lemma 5.6. Consider a sequence $S = < a_1, a_2, \ldots, a_n >$ of items stored $\frac{n}{p}$ per processor in the p processors of a CGM(n,p), with r of the items marked. The marked items can be compacted to the first $\lceil \frac{r}{\frac{n}{p}} \rceil$ processors of the CGM(n,p) in $O(\frac{n}{p})$ computation time and $O(T_{Alltoall}(n,p))$ communication time. □

5.2 OBJECT VISIBILITY ALGORITHMS

This section presents a brief discussion on how the template algorithms for the various object visibility problems discussed in Chapter 3 are ported to the CGM(n,p).

5.2.1 ENDPOINT AND SEGMENT VISIBILITY

The purpose of this subsection is to show that the template algorithm 3.1 to solve SV and EV can be ported to the CGM(n,p) using the various tools developed in Section 5.1. The computational time of the resulting algorithm is $O(\frac{n \log n}{p})$. Since the sequential lower bounds to these problems is $\Omega(n \log n)$, this algorithm is computationally time-optimal.

Consider an arbitrary set S of n vertical line segments with every segment being specified by its top and bottom endpoints. The set S is assumed to be stored, $\frac{n}{p}$ segments per processor, in a CGM(n,p), where any processor P_i stores segments $S_i = \{s_{i*\frac{n}{p}+1}, \ldots, s_{(i+1)*\frac{n}{p}}\}$.

The various assumptions and the terminology is identical to what is described in the template algorithm. Let us discuss the porting of the two stages of the template algorithm on the CGM(n,p).

Stage 1. Consider a generic node v in the abstract tree $\mathcal{T}$ with left and right children u and w, respectively. $E(v)$ is obtained by merging $E(u)$ and $E(w)$. If the level of v is less than or equal to $\log \frac{n}{p}$, the merging of $E(u)$ and $E(w)$, for every node at that

level of the tree is carried out using the sequential algorithm running in $O(|E(u)| + |E(w)|)$ time. As noted in the template algorithms, for the first $O(\log \frac{n}{p})$ levels the merging can be accomplished in $O(\frac{n}{p} \log \frac{n}{p})$ time. For the nodes at level greater than $\log \frac{n}{p}$, the merging of $E(u)$ and $E(w)$ is accomplished by applying the merge algorithm discussed in the Subsection 5.1.2. The task of determining t-blocked(e_i) and a-blocked(e_i) are performed exactly as mentioned in the template algorithm and requires $O(\frac{n}{p})$ computational time. Stage 1 takes $O(\frac{n \log n}{p})$ computational time and $O(\log p T_{Alltoall}(n,p))$ communication time.

Stage 2. The values of RC(v) and LC(v) are computed as specified in equations 3.1 and 3.2. Merge RC(u) and RC(w) into a list $E'(v)$, and from $E'(v)$ delete those endpoints e_i that have a-blocked(e_i)=v and thus determine the endpoints in RC(v) and rank them. Obtain a compacted version of RC(v) applying the compaction operation in $O(\frac{n}{p})$+ $O(T_{Alltoall}(n,p))$ time. The computation of LC(v) is perfectly similar. Again, the determination of the values of t(e_i) and a(e_i) for all endpoints in BA(v) and BT(v), can be accomplished using the merge operation, exactly as described in the template algorithm. Stage 2 takes $O(\frac{n \log n}{p})$ computational time and $O(\log p T_{Alltoall}(n,p))$ communication time. Thus the following result is obtained.

Theorem 5.7. An arbitrary n-segment instance of the EV problem can be solved in $O(\frac{n \log n}{p})$ computational time and $O(\log p T_{Alltoall}(n,p))$ communication time, on a CGM(n,p). □

As mentioned in the template algorithm, the contours can be trivially computed from the solution to the EV problem, thus the following result is obtained.

Theorem 5.8. An arbitrary n-segment instance of the SV problem can be solved in $O(\frac{n \log n}{p})$ computational time and $O(\log p T_{Alltoall}(n,p))$ communication time, on a CGM(n,p). □

5.2.2 DISK VISIBILITY

Assume that an arbitrary set $D = \{d_1, d_2, \ldots, d_n\}$ of disks is stored, $\frac{n}{p}$ disks per processor, in a CGM(n,p), so that any processor P_i $(0 \leq i \leq p-1)$ stores the disks $D_i = \{d_{i*\frac{n}{p}+1}, \ldots, d_{(i+1)*\frac{n}{p}}\}$. The other assumptions about the position of the view point and the disks is as described in the template algorithm 3.2.

Each processor determines the tangents to the disks it stores from ω, as well as the length of these tangents, i.e. the distance between ω and the tangency points, in $O(\frac{n}{p})$ computational time. As before, with every disk d_i associate the line segment s_i obtained by joining the corresponding tangency points, sort the segments and obtain the solution to SV problem. This can be done in $O(\frac{n \log n}{p})$ computational time and $O(\log p T_{Alltoall}(n,p))$ communication time, by virtue of Lemma 5.5 and Theorem 5.8. Once the visible portions of the segments are determined, the portions of the disks visible from ω can be trivially computed in $O(\frac{n}{p})$ time. Thus, the following result is obtained.

Theorem 5.9. The DV problem for a set of n disks can be solved in $O(\frac{n \log n}{p})$ computational time and $O(\log p T_{Alltoall}(n,p))$ communication time, on a CGM(n,p).

□

5.2.3 RECTANGLE VISIBILITY

The purpose of this subsection is to show how the template algorithm 3.3 for the RV algorithm, is ported to the CGM(n,p). Consider a set $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ of iso-oriented, non-overlapping, rectangles stored $\frac{n}{p}$ per processor, in a CGM(n,p), so that any processor P_i stores the rectangles $R_{i*\frac{n}{p}+1}, \ldots, R_{(i+1)*\frac{n}{p}}$.

Solve the instance of the EV problem obtained by considering the top and bottom edges of every rectangle in $\mathcal{R}$. Repeat the same for the vertical segments of every rectangle. This can again be performed in $O(\frac{n \log n}{p})$ computational time and

$O(\log p T_{Alltoall}(n,p))$ communication time by virtue of Theorem 5.7.

As described in the template algorithm, every generic corner e_i of rectangle r_i has four solutions: $a1(e_i)$, $t1(e_i)$, $a2(e_i)$, and $t2(e_i)$. The solution to the RV problem can be obtained from this information as described in the template algorithm. The following result is thus obtained.

Theorem 5.10. An arbitrary instance of size n of the RV problem can be solved in $O(\frac{n \log n}{p})$ computational time and $O(\log p T_{Alltoall}(n,p))$ communication time, on a CGM(n,p). □

5.2.4 DOMINANCE GRAPH

In this subsection, let us discuss how the template algorithm 3.4 can be applied to the CGM(n,p) to obtain computationally optimal algorithm for the dominance graph problem.

Consider an arbitrary instance of size n of the DG problem stored $\frac{n}{p}$ per processor on a CGM(n,p). Sort the rectangles by the x-coordinate of their bottom left corners. Solve the instance of the EV problem consisting of the set of top and bottom edges of rectangles, with the viewpoint ω at $(0,-\infty)$ in $O(\frac{n \log n}{p})$ computational time and $O(\log p T_{Alltoall}(n,p))$ communication time. As in the template algorithm 3.4, with each endpoint associate a 4-tuple (L,U,x,TB). Sort the set of tuples first by L and then by x as discussed in Subsection 5.1.3. This can be accomplished in $O(\frac{n \log n}{p})$ computational time and $O(\log p T_{Alltoall}(n,p))$ communication time as stated in Lemma 5.5. Consider the tuples (L_1,U_1,x_1,TB_1) and (L_2,U_2,x_2,TB_2) that are adjacent in the sorted sequence. If $L_1 = L_2$ and $U_1 = U_2$ then record an edge in $\vec{D}$, from the rectangle corresponding to L_1 to the one corresponding to U_1. Each edge is stored as (L_1,U_1). After sorting the resulting ordered pairs, the dominance graph can be constructed trivially. This leads to the following result.

Theorem 5.11. An arbitrary instance of size n of the DG problem can be solved in $O(\frac{n \log n}{p})$ computational time and $O(\log p T_{Alltoall}(n,p))$ communication time, on a CGM(n,p). □

CHAPTER 6

TRIANGULATION ON THE ABSTRACT MODEL

One of the natural problems that arises in a number of seemingly unrelated areas in manufacturing, robotics, CAD, VLSI design, and pattern recognition involves partitioning a planar region of interest into simple subregions, typically triangles. The motivation for doing so is that the restriction of the original problem to a triangular subregion is often more tractable and, furthermore, once the problem is solved for each of the triangles in the partition, the overall solution is obtained by a *conquer* process.

Such a situation occurs, for example, in pattern recognition and computational morphology where one desires to infer properties of a region by averaging a certain objective function over the triangles in the partition [88]. The same problem appears in unstructured multigrid strategies [23] that are being used to speed up the convergence of computationally intensive PDE solution schemes. Here, the domain is discretized and decomposed into triangular subregions in order to meet stability requirements. Yet another example is provided by motion planning in robotics where, in an unknown terrain, a robot builds a navigational plan by combining a number of simpler courses each through a triangular region [49]. As is often the case,

the terrain contains natural obstacles that must be excluded from the triangulation.

More generally, one is interested in the following problem: given a planar region along with a sequence of forbidden subregions, partition the given region into triangular subregions, none of which intersects the forbidden subregions. The instance of this generic problem where the region of interest is implicitly specified by the convex hull of a set of points with no forbidden subregions is commonly referred to as the *triangulation* problem. Instances of the generic problem featuring forbidden subregions of some sort are typically referred to as *constrained triangulations.* Being of practical relevance and of theoretical interest triangulation problems have been extensively studied in the literature. For an excellent discussion the reader is referred to [88] where many of the above applications are summarized.

This chapter, discusses architecture independent methodologies to solve various triangulation problems. Template algorithms are designed for these problems for an abstract computational model, which can be ported to the diverse models of computation discussed in Chapter 2.

As described in Chapter 3, an ACM(n, p, M) consists of p processors having O(M) memory each, so that $n \leq M * p$, where n is the size of the instance of the problem at hand. The p processors are assumed to be identical and are enumerated as $P_0, P_1, \ldots, P_{p-1}$ and each of the processors P_i $(0 \leq i \leq p-1)$ is assumed to know its identity i. All the processors communicate via an interconnection network. In addition to the operations assumed to be available on the ACM(n, p, M) in Chapter 3, it is assumed that the following are available:

• *All Nearest Larger Values*: The all nearest larger values problem (ANLV, for short) is defined as follows. Given a sequence of n real numbers $< a_1, a_2, \ldots, a_n >$, stored at most M per processor in the first $\frac{n}{M}$ processors of an ACM(n, p, M), for

each a_i $(1 \leq i \leq n)$, find the nearest element to its left and the nearest element to its right (if any) that is larger than a_i. The time to solve the ANLV on an ACM(n, p, M) is given by $T_{ANLV}(n, p, M)$.

• *Convex Hull*: The convex hull of a set of planar points is the smallest convex set containing the given set. Given a set of n points in the plane, stored at most M per processor in the first $\frac{n}{M}$ processors of an ACM(n, p, M), the time to compute the convex hull is given by $T_{Convexhull}(n, p, M)$.

In the various algorithms, the ACM(n, p, M), can be viewed as l independent ACM's, each solving subproblems of sizes $N_1, N_2, \ldots, N_l$, respectively (where $N_1 + N_2 + \ldots + N_l <= n$). A subproblem i of size N_i is solved on an ACM(N_i, p', M) (p' is at most $\frac{pN_i}{n}$).

Before presenting the triangulation algorithms, let us discuss the terminology used in the various template algorithms for the triangulation problems.

Specifying an n-vertex polygon P in the plane amounts to enumerating its vertices in clockwise order as $v_1, v_2, \ldots, v_n$ $(n \geq 3)$. Here $v_i v_{i+1}$ $(1 \leq i \leq n-1)$ and $v_n v_1$ define the edges of P. This representation is also known as the *vertex* representation of P. Note that the vertex representation of a polygon can be easily converted into an *edge* representation: namely, P is represented by a sequence $e_1, e_2, \ldots, e_n$ of edges, with e_i $(1 \leq i \leq n-1)$ having v_i and v_{i+1} as its endpoints, and e_n having v_n and v_1 as its endpoints.

A polygon P is termed *simple* if no two of its non-consecutive edges intersect. Recall that well known Jordan Curve Theorem guarantees that a simple polygon partitions the plane into two disjoint regions, the *interior* (bounded) and the *exterior* (unbounded) that are separated by the polygon. A simple polygon is convex if its interior is a convex set. In particular, the convex hull of a set of points is a convex

polygon. A polygon P is said to be monotone in some direction δ if any normal to δ intersects P in at most two points as illustrated in Figure 6.1.

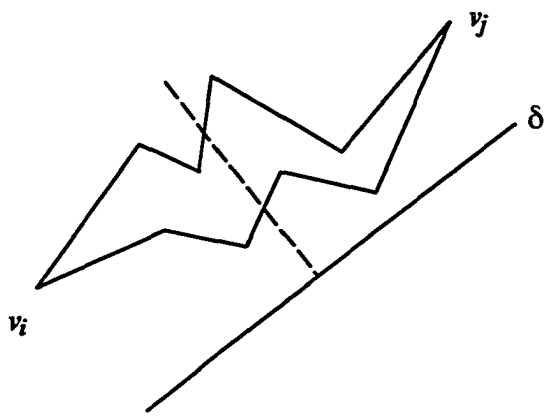

Figure 6.1: A monotone polygon in the direction δ

Let v_i and v_j be the first and last vertices of P in the direction δ. These two vertices partition P into two polygonal chains monotone with respect to δ. A monotone polygon is termed *special* if one of these chains reduced to a single edge, termed the *base* edge. Refer to Figure 6.2 for an illustration. As it turns out, special monotone polygons have interesting properties that will be exploited in a number of contexts.

In the following sections, let us discuss the various triangulation algorithms on the ACM(n, p, M), assumed to be equipped with the powerful tools to solve ANLV and convex hull problems, in addition to the tools discussed in Chapter 3.

In Section 6.1, the triangulation of special monotone polygons is discussed, which in turn is a powerful tool to solve several triangulation problems. Section 6.2 discusses the problem of triangulating a set of points in the plane using the triangulation of monotone polygons as a basic building block. Section 6.3 discusses

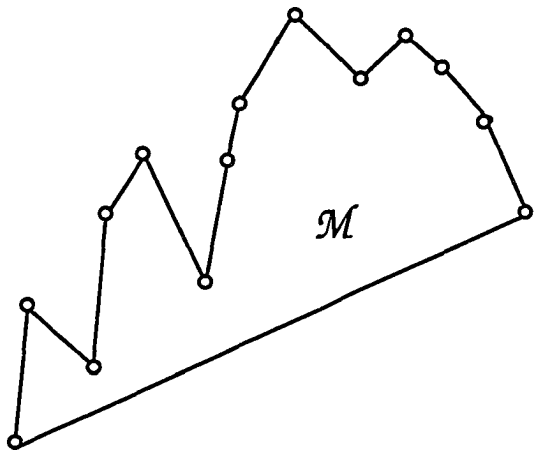

Figure 6.2: A special monotone polygon

the triangulation of a convex region in the presence of a convex forbidden region. Sections 6.4 and 6.5 discuss two other cases of constrained triangulations where the forbidden regions are specified as a set of rectangles and ordered segments, respectively.

6.1 SPECIAL MONOTONE POLYGONS

In this section, let us discuss an algorithm for triangulating a *special* monotone polygon. This algorithm turns out to be very handy tool in providing solutions to the triangulation of a set of points in the plane and to the constrained triangulations.

Consider a special monotone polygon $\mathcal{M} = v_1, v_2, \ldots, v_n$ in the plane with its vertices specified in clockwise order and with v_1v_n denoting the base edge. Assume that the interior of the polygon lies in the positive half-plane determined by the line v_1v_n. The vertices of the polygon are assumed to be stored at most M vertices per processor among the first $\frac{n}{M}$ processors of an ACM(n, p, M). The polygonal chain $v_1, v_2, \ldots, v_n$ is termed the *monotone chain.* Further subdivide the monotone chain into (sub)chains monotone in the y-direction. Such chains are termed *ascending* and *descending.* Now, let us discuss the template algorithm.

Template Algorithm 6.1:

The details of the various steps involved in triangulating the special monotone polygon $\mathcal{M}$ are as follows:

Step 1. By checking its neighbors, every vertex v_i of $\mathcal{M}$ determines whether it belongs to an ascending or descending chain. Vertices achieving local minima in the y-direction are treated as part of both ascending and descending chains. Assuming that every vertex stores the information about its neighbors, this step can be accomplished in O(M) time.

Step 2. With each vertex $v_i = (x_i, y_i)$ of $\mathcal{M}$ associate an element $s_i = y_i$ and solve the resulting instance of the ANLV problem. This can be accomplished in $T_{ANLV}(n, p, M)$ time. Let $l(v_i) = s_j$, where $l(v_i)$ is the solution to ANLV for s_i to its left. Similarly, let $r(v_i) = s_k$, where $r(v_i)$ is the solution to the right.

For a vertex v_i on an ascending (resp. descending) chain of $\mathcal{M}$ the vertex v_j is said to be a *match* if s_j is a solution obtained in Step 2 and v_j belongs to a descending (resp. ascending) chain.

Step 3. Every vertex v_i that has identified (at least) a match v_j adds the diagonal v_iv_j to the triangulation and records the resulting triangle. This takes O(M) time.

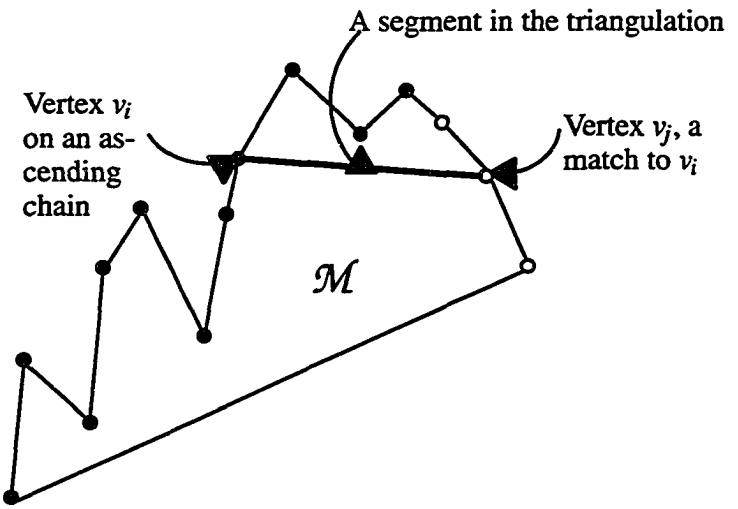

Figure 6.3: Illustrating Step 3 of the triangulation of a special monotone polygon

Step 4. The following vertices mark themselves:

- v_1 and v_n;
- vertices that have identified no match;
- vertices achieving local minima in the y-direction that have found only one match.

It is important to note that in case the base edge v_1v_n is horizontal, only v_1 and v_n are marked. Step 4 is accomplished in $O(M)$ time.

Step 5. Let $v_1 = v_{i_1}, v_{i_2}, \ldots, v_{i_r} = v_n$ be the sequence of marked vertices enumerated by increasing x-coordinate and let $\mathcal{M}'$ be the monotone polygon determined by these marked vertices. Rotate $\mathcal{M}'$ so that v_1v_n becomes parallel to the x-axis and repeat Steps 2 to 4. This step takes another $O(M)+O(T_{ANLV}(n,p,M))$ time.

Various steps of the algorithm are illustrated in Figures 6.3, 6.4 and 6.5. The diagonals to be added are determined by finding a *match* for each of the vertices

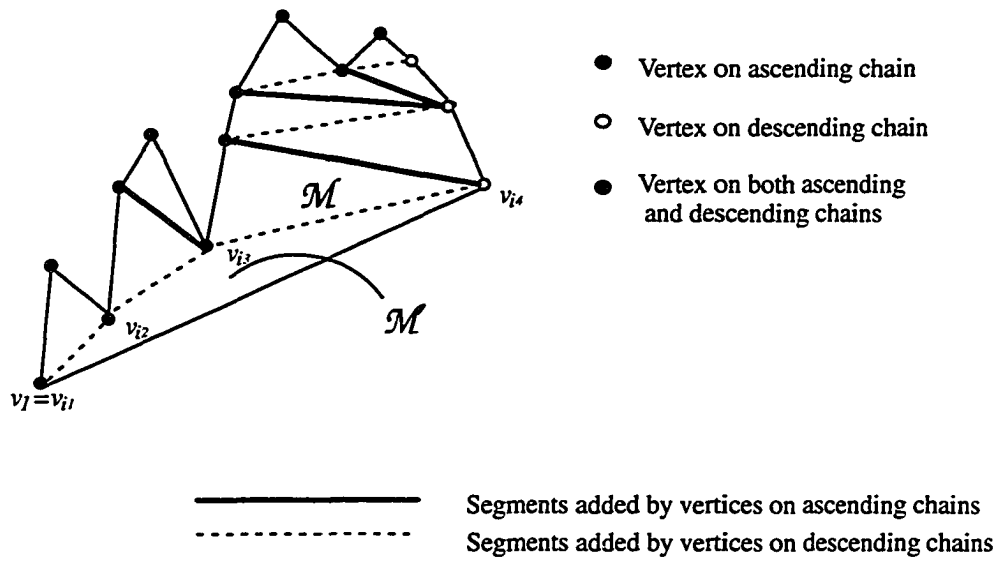

Figure 6.4: Illustrating the special monotone polygon after Step 4

as shown in Figure 6.3. Figure 6.4 shows $\mathcal{M}$ after the diagonals are added in Step 3. The vertices marked in Step 4 are $v_{i_1}, \ldots, v_{i_4}$. Notice that at the end of Step 4, the only part of the original polygon that is not triangulated is *bounded* by the marked vertices. Figure 6.5 shows the entire polygon triangulated. It is easy to see that after having rotated the edge v_1v_n, the solution $l(v_{i_2}) = s_n$, confirming that the diagonal $v_{i_2}v_n$ (i.e. v_3v_n) will be added to the triangulation. The correctness and the time complexity of the algorithm are established by the following result.

Theorem 6.1. The problem of triangulating an n-vertex special monotone polygon, stored M vertices per processor among the first $\frac{n}{M}$ processors of a ACM(n, p, M), can be solved in $T_{Monotone}(n, p, M)$=O(M)+O$(T_{ANLV}(n, p, M))$ time.

Proof. In order to show that the triangulation is done correctly, it is enough to prove that the diagonals added in Step 3 do not intersect and that when the algorithm terminates there are no polygons with more than three sides left.

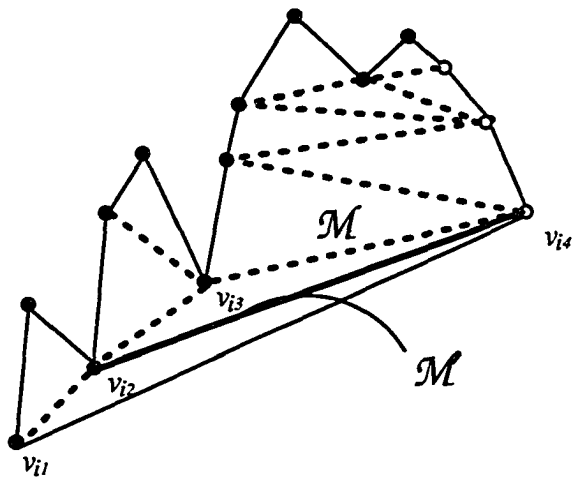

Figure 6.5: The triangulated special monotone polygon

Let v_i belong to an ascending chain and let v_k be a match found in Step 2. By definition, v_k belongs to a descending chain and v_k has a lower y-coordinate than v_i. The diagonal v_iv_k is added in Step 3. If some other diagonal v_pv_q, added in Step 3, intersects v_iv_k then, exactly one of v_p and v_q lies on the monotone chain from v_i to v_k. Assume, without loss of generality, that v_p does. But now, either $r(v_i) = s_p$ in case the y-coordinate of v_p is lower than that of v_i, or $l(v_p) = s_{i'}$ and $r(v_p) = s_{k'}$, otherwise, with $v_{i'}$ and $v_{k'}$ lying between v_i and v_k. Both scenarios lead to a contradiction.

Let $v_1 = v_{i_1}, v_{i_2}, \ldots, v_{i_r} = v_n$ be the sequence of marked vertices obtained in Step 4, enumerated by increasing x-coordinate. Let A be the portion of the monotone chain between two adjacent marked vertices v_{i_j} and $v_{i_{j+1}}$.

It can be claimed that the interior of A is triangulated. The proof involves a simple counting argument. Let m be the total number of vertices between v_{i_j} and $v_{i_{j+1}}$. Let p be the number of local maxima in the y-direction in A. It follows that

the number of local minima is $p-1$. Every vertex *internal* to A that is not a local maximum or a local minimum adds exactly one diagonal in Step 3. Further, vertices that are local maxima add no edges, while vertices that are local minima add two edges. Thus, the total number of edges added to A in Step 3 is $m-2-2p+1+2(p-1) = m-3$. As shown before, these internal diagonals are non-intersecting, and thus A is triangulated, as claimed.

Finally, let $\mathcal{M}'$ be the polygon determined by the marked vertices. To complete the proof, it is necessary to show that when the algorithm terminates $\mathcal{M}'$ is triangulated. It is clear that $\mathcal{M}'$ is monotone in the x-direction and that $\mathcal{M}'$ is special. Observe that, $\mathcal{M}'$ has much stronger properties.

Observation 6.2. $\mathcal{M}'$ is monotone in both x and y direction.

(First, assume that v_1 has a lower y-coordinate than v_n. Now, if $\mathcal{M}'$ fails to be monotone in the y-direction, then there must exist two vertices $v_{i_p}= (x_{i_p}, y_{i_p})$ and $v_{i_q}= (x_{i_q}, y_{i_q})$ in $\mathcal{M}'$ such that $x_{i_p} < x_{i_q}$ and $y_{i_p} > y_{i_q}$. However, this leads to a contradiction: both horizontal rays to the right and to the left originating at v_{i_p} must find a solution in Step 2 and so v_{i_p} cannot possibly be marked. The case where v_n has a lower y-coordinate than v_1 is similar.)

Observation 6.3. $\mathcal{M}'$ is monotone with respect to the direction of the edge v_1v_n.

(Follows immediately from the definition of $\mathcal{M}'$ and Observation 6.2.)

Now, consider what happens when $\mathcal{M}'$ is rotated as to make the edge v_1v_n parallel to the x-axis. By Observations 6.2 and 6.3, $\mathcal{M}'$ is a special polygon monotone in the new x-direction. Therefore, after applying Steps 2–4 above, the only marked vertices of $\mathcal{M}'$ are v_1 and v_n and so, by the above argument, the triangulation of the original polygon $\mathcal{M}$ is complete. This establishes the correctness of the algorithm. □

6.2 SET OF POINTS

The purpose of this section is to present a template algorithm to triangulate a set of points in the plane. The algorithm to triangulate special monotone polygons, discussed in Section 6.1, plays a very significant role in providing the solution to this problem.

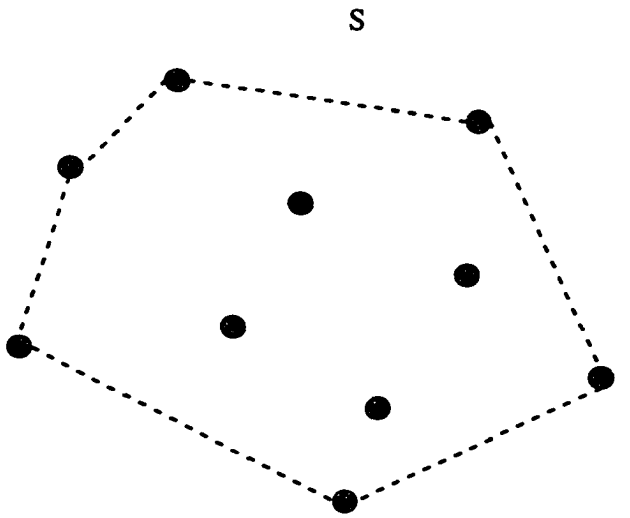

Figure 6.6: Edges of the convex hull of S included in the triangulation

Consider a set S of n points in the plane stored in the first $\frac{n}{M}$ processors of an ACM(n, p, M), at most M per processor.

Template Algorithm 6.2:

Step 1. Compute the convex hull of S, in $T_{Convexhull}(n, p, M)$ time. Note that all the edges of the convex hull will be part of the desired triangulation (see Figure 6.6).

Step 2. Next, in $T_{Sort}(n, p, M)$ time, sort the points in S in increasing order of their x-coordinates and add a diagonal between adjacent points in the sorted sequence.

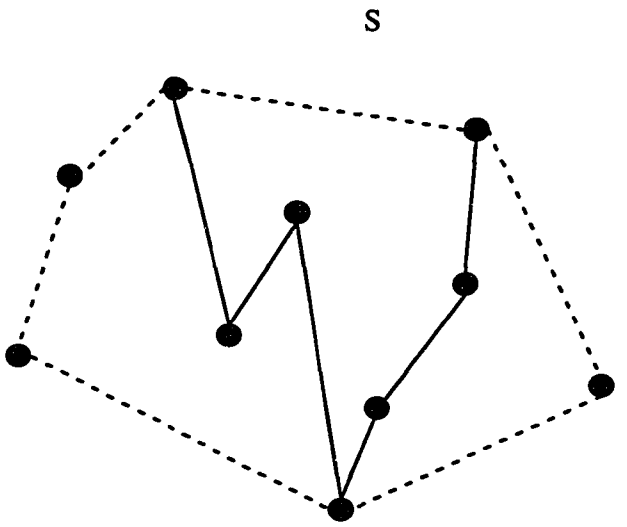

Figure 6.7: Diagonals added in Step 2 of the algorithm

Step 3. Referring to Figure 6.7, observe that the diagonals added in Step 2 divide the entire region within the hull into special monotone polygons having the convex hull edges as base edges. Consider the lower hull with l edges, and let $N_1, N_2, \ldots, N_l$ be the number of vertices in the monotone polygons with each of the l lower hull edges as the base edges. Consider all the monotone polygons having at most M vertices, such that all the vertices are stored in one processor. All such monotone polygons can be triangulated in $O(M)$ time, in each of the processors sequentially. The remaining monotone polygons are triangulated independently, in parallel, using the algorithm for triangulating a special monotone polygon described in Section 6.1, where a polygon i with N_i vertices is solved on an ACM(N_i, p', M) (p' is at most $\frac{pN_i}{n}$). The same can be repeated for the special monotone polygons with the base edge on the upper hull. Thus, the convex hull of S is triangulated as illustrated

in Figure 6.8. The above steps can be performed in at most $O(T_{Monotone}(n, p, M))$ time.

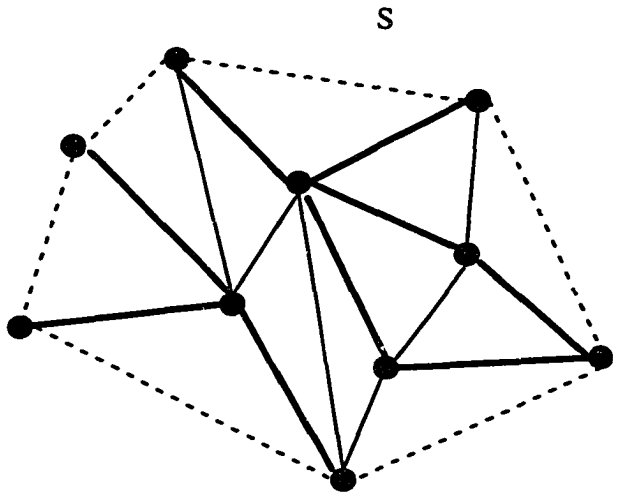

Figure 6.8: S is triangulated after Step 3

Consequently, the following result is obtained.

Theorem 6.4. An arbitrary set S of n points in the plane, stored M points per processor in the first $\frac{n}{M}$ processors of an ACM(n, p, M), can be triangulated in $O(T_{Convexhull}(n, p, M))+O(T_{Sort}(n, p, M))+O(T_{Monotone}(n, p, M))+O(M)$ time. □

6.3 CONVEX REGIONS WITH ONE CONVEX HOLE

In this section, let us discuss the template algorithm for the triangulation of a convex region with a convex hole. Let $C = c_1, c_2, \ldots, c_n$ be a convex region of the plane and $H = h_1, h_2, \ldots, h_m$ be a convex hole within C. In many applications in

computer graphics [76], computer-aided manufacturing and CAD [37], it is necessary to triangulate the region $C \setminus H$. The task at hand can be perceived as a constrained triangulation of C. For an illustration refer to Figure 6.9.

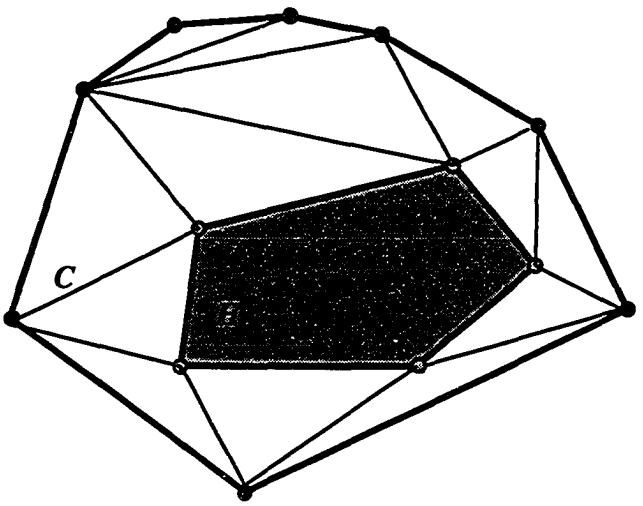

Figure 6.9: Triangulating a convex region with a convex hole

Note that, the algorithm for triangulating a convex region with a convex hole will be a key ingredient in the constrained triangulation algorithms discussed in the Section 6.4.

Let C be stored $2M$ vertices per processor among the first $\frac{n}{2M}$ processors of the ACM(n, p, M) and H be stored $2M$ vertices per processor in the next $\frac{m}{2M}$ processors of the ACM. The triangulation algorithm proceeds as follows.

Template Algorithm 6.3:

Step 1. Determine an arbitrary point w interior to H and in $T_{Broadcast}(1, p, M)$ time broadcast its value to the first $\frac{n+m}{2M}$ processors of the ACM(n, p, M). Convert

the vertices of C and H to polar coordinates having ω as pole and the positive x-direction as polar axis. This can be accomplished in $O(M)$ time.

Since ω is interior to C and H, convexity guarantees that the vertices of both C and H occur in sorted order about ω.

Step 2. The two sorted sequences corresponding to vertices of C and H, are merged in $O(T_{Merge}(n, p, M))$ time. Let $B = b_1, b_2, \ldots, b_{n+m}$ be the resulting sequence and is sorted by polar angle.

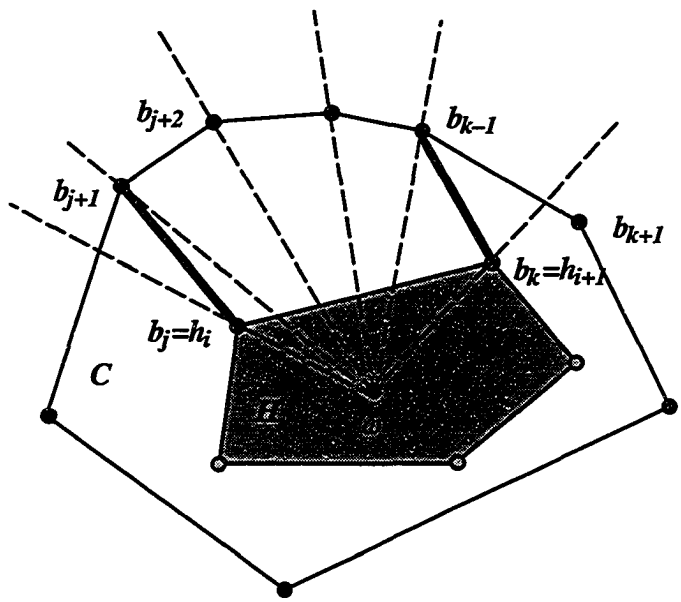

Figure 6.10: Illustrating Case 1

In the process of triangulating $C \setminus H$ let us distinguish the following two cases.

Case 1. Consider the subsequences of B having the following form. For some i ($1 \leq i \leq m$) $h_i = b_j$ and $h_{i+1} = b_k$ with $j+1 < k$. Each of these subsequences corresponds to a polygon which can be triangulated as described below. Referring to Figure 6.10, note that in this case, the line segment $b_{j+1}b_{k-1}$ lies in the wedge determined by

h_i, h_{i+1} and ω. Furthermore, the polygon $b_{j+1}, b_{j+2}, \ldots, b_{k-1}$ is convex. It is clear that this polygon can be triangulated in by simply adding all the possible diagonals originating at b_{j+1}.

Case 2. Again, consider the subsequences of B having the following form: for some i $(1 \leq i \leq n)$ $c_i = b_j$ and $c_{i+1} = b_k$ with $j + 1 < k$. Let us show the triangulation of the polygon with vertices $c_i = b_j, b_k, b_{k-1}, b_{k-2}, \ldots, b_{j+1}$. Let us make the following simple observation that follows immediately by the convexity of H.

Observation 6.5. Let t $(j + 1 \leq t \leq k - 1)$ be such that c_i is visible from vertex b_t. Then c_i is visible for every vertex h_s with $j + 1 \leq s \leq t$. □

Observation 6.6. Every vertex b_t $(j + 1 \leq t \leq k - 1)$ on H is visible from either c_i or c_{i+1}. □

Referring to Figure 6.11, let b_r be the vertex among $b_{j+1}, b_{j+2}, \ldots, b_{k-1}$ with the smallest Euclidian distance to the line segment $c_i c_{i+1}$. Clearly, b_r is visible from both c_i and c_{i+1}. Now the conclusion follows from Observation 6.5.

Observations 6.5 and 6.6 justify the following approach to triangulating the polygon $c_i = b_j$, b_k, b_{k-1}, b_{k-2}, ..., b_{j+1}. First, determine the vertex b_r by determining the vertex achieving the minimum euclidean distance to the line segment $c_i c_{i+1}$. Add to the triangulation all the edges $c_i h_s$ with $j + 1 \leq s \leq r$ and all the edges $c_{i+1} h_u$ with $r \leq u \leq k - 1$.

Step 3. In this step, subsequences in B corresponding to Case 1 and Case 2 described above, are identified and each of the corresponding polygons is triangulated. The details are as follows: assume that the sequence b_1, b_2, ..., b_{n+m} is stored $2M$ per processor in the first $\frac{n+m}{2M}$ processors of an ACM(n, p, M). Let us solve the polygons determined by subsequences belonging to Case 1. First, determine all pairs h_i, h_{i+1} that bound the subsequences of the form in Case 1. Note that there are at

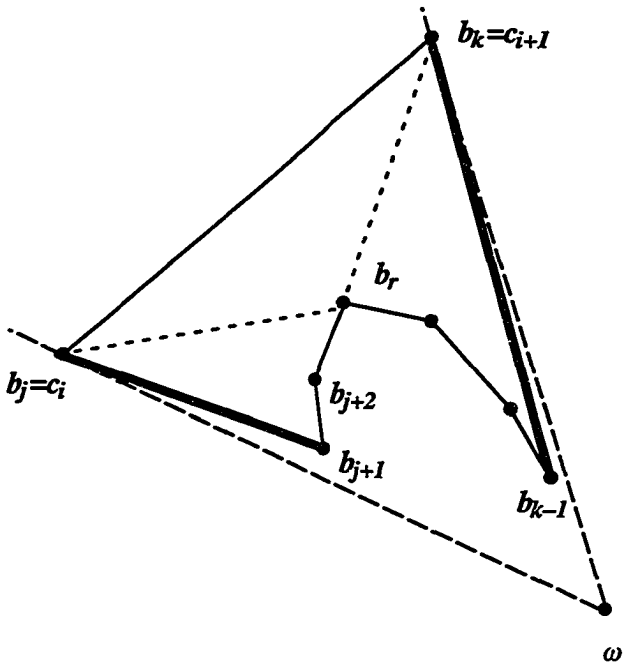

Figure 6.11: Illustrating Case 2

most m such pairs and all the vertices of C that lie between each pair h_i, h_{i+1} are said to belong to H_i. Every H_i having less than M vertices, with all the vertices stored locally in a processor P_j of the ACM, can be solved sequentially in $O(M)$ time on every such P_j. Every H_i that is not stored in any one processor, can be processed in parallel on independent ACMs as follows. Broadcast b_{j+1} and add diagonals from every vertex in H_i to b_{j+1}, as described in Case 1. Next, all pairs c_j, c_{j+1} as in Case 2 above are detected and all vertices of H lying between them are said to belong to a subsequence C_j. Every C_j can be processed in parallel on an independent ACM as follows. Determine the vertex in C_j, belonging to H, achieving the minimum euclidean distance from c_ic_{i+1}, and add the diagonals as described in Case 2. The running time of this step is bounded by $O(M)+O(T_{Broadcast}(1,p,M))$ time.

Theorem 6.7. Let C be an n-vertex convex region, stored at most $2M$ vertices per processor among the first $\frac{n}{2M}$ processors of an ACM(n, p, M), and let H be an m-vertex convex hole ($m \in O(n)$) within C, also stored $2M$ vertices per processor in the next $\frac{m}{2M}$ processors of the ACM(n, p, M). The planar region $C \setminus H$ can be triangulated in $T_{Convexhole}(n, p, M)$=O(M)+O($T_{Broadcast}(1, p, M)$) time. □

6.4 CONVEX REGIONS WITH RECTANGULAR HOLES

This section discusses a particular case of constrained triangulation problems involving rectangular forbidden regions within a convex region to be triangulated. The template algorithm for this problem for the ACM(n, p, M) is developed and uses as building blocks the algorithms for the triangulation of special monotone polygons and the triangulation of convex region with convex holes.

Let $C = c_1, c_2, \ldots, c_n$ be a convex region containing n rectangular holes specified by a set $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ of iso-oriented, non-overlapping rectangles. The task at hand is to triangulate $C \setminus \mathcal{R}$. The required triangulation can be obtained in two phases after determining the convex hull of the set $\mathcal{R}$ of rectangles. Let C' be the convex hull of $\mathcal{R}$. In the first phase of the algorithm $C \setminus C'$ is triangulated and in the second phase C' is triangulated. The details of the template algorithm are as follows:

Template Algorithm 6.4:

Step 1. The task of computing the convex hull of $\mathcal{R}$ is a particular instance of the convex hull problem and can be solved in $T_{Convexhull}(n, p, M)$ time. Now, the triangulation of the region $C \setminus C'$ can be done in $T_{Convexhole}(n, p, M)$ time.

Thus, the focus is now on the second phase where the problem reduces to triangulating C'. Let $tr(R_i)$, $tl(R_i)$, $br(R_i)$, and $bl(R_i)$ stand for top-right, top-left, bottom-right, and bottom-left corners of R_i, respectively. Refer to the left vertical edge of R_i as $left(R_i)$ and the right vertical edge as $right(R_i)$. For convenience, each rectangle R_i is given the identity i. To the given set $\mathcal{R}$ of rectangles, add two rectangles R_0 and R_{n+1} with $bl(R_0) = (x_{min} - 1, y_{min} - 1 - \epsilon)$, $tr(R_0) = (x_{min} - \epsilon, y_{max} + 1 + \epsilon)$ and $bl(R_{n+1}) = x_{max} + \epsilon, y_{min} - 1 - \epsilon)$, $tr(R_{n+1}) = (x_{max} + 1, y_{max} + 1 + \epsilon)$, where x_{max}, x_{min} and y_{max}, y_{min} are the maximum and minimum values among the coordinates of the endpoints of the rectangles in x and y directions and $\epsilon > 0$ is a small constant (see Figure 6.12).

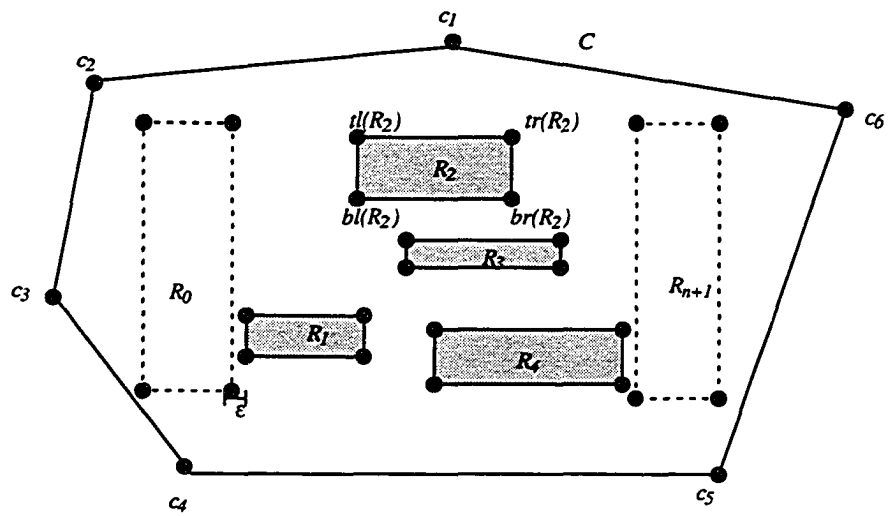

Figure 6.12: Illustrating the convex region C with rectangular holes

Step 2. Solve the rectangle visibility for the set $R_0, R_1, \ldots, R_{n+1}$. This can be done in $T_{RV}(n, p, M)$ time (see Figure 6.13).

Step 3. Associate with each corner point of rectangle R_i an information packet

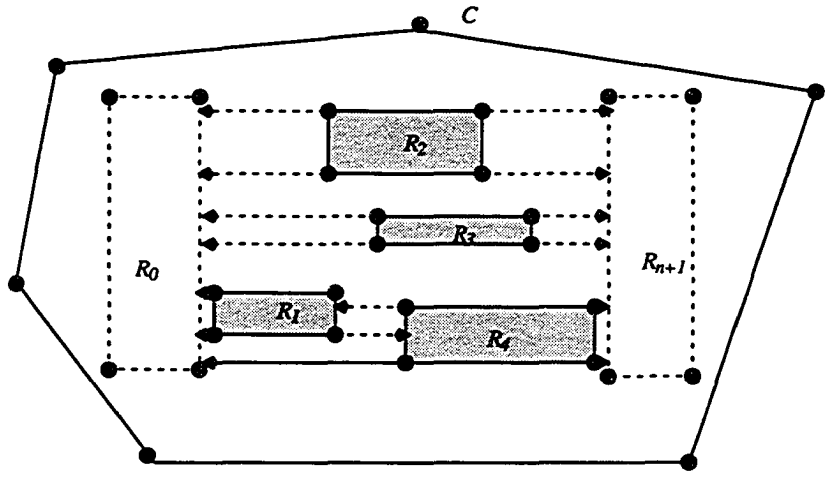

Figure 6.13: Determining the rectangle visibility for R

containing its coordinates and two numbers u and v. For endpoints of $left(R_i)$, u is set to its identity i and v is set to the identity of the rectangle visible in the negative x-direction. Similarly, for endpoints of $right(R_i)$, v is set to the identity of R_i and u is set to the identity of the rectangle visible in the positive x-direction. Sort the information packets, first on the u value and then on the y-coordinate. Clearly, this step requires $O(T_{Sort}(n, p, M))$ time.

Notice that after the sort, for every $left(R_i)$ the identities of R_j, with $r(e) = left(R_i)$ where e is an endpoint of R_j, will occur in consecutive positions. A diagonal connecting two corner points belonging to R_p and R_q is added to the triangulation if p and q occur in adjacent positions corresponding to some $left(R_k)$ (see Figure 6.14). Note that this determination takes O(1) time.

For any $left(R_i)$, the sequence of diagonals, including the rectangle edges between them, is called the *closest* contour of $left(R_i)$ and denoted by $CL(R_i)$.

The above process is repeated for $right(R_i), (\ 0 \leq i < n + 1)$ and for any

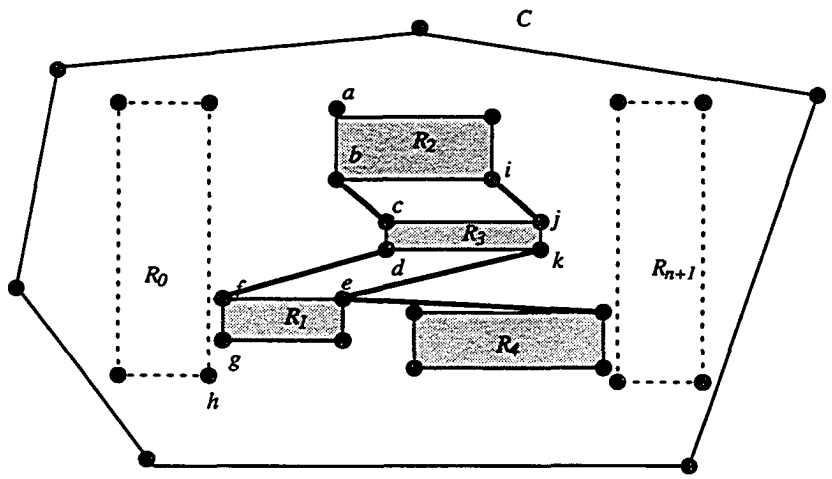

Figure 6.14: Illustrating the computation of closest contours

$right(R_i)$, the closest contour $CR(R_i)$ is computed similarly. Consider the partitioning of C' after the addition of the diagonals. The various pieces of partitions belong to one of the following types:

- the rectangles (R_i's);
- the special monotone polygons formed by the left and right edges of various rectangles with their closest contours;
- the remaining regions referred to as *special trapezoids.*

Step 4. All the special trapezoids can be identified as follows. Consider two rectangles R_p and R_q such that $r(br(R_p)) = r(tr(R_q))$ and $l(bl(R_p)) = l(tl(R_q))$. The region joining $br(R_p)$ with $tr(R_q)$ and $bl(R_p)$ with $tl(R_q)$ is a special trapezoid and can be triangulated by adding a diagonal (see Figure 6.15). Also, the special monotone polygons can be identified and triangulated in independent ACM's in $T_{Monotone}(n, p, M)$ time. Thus, C' is triangulated.

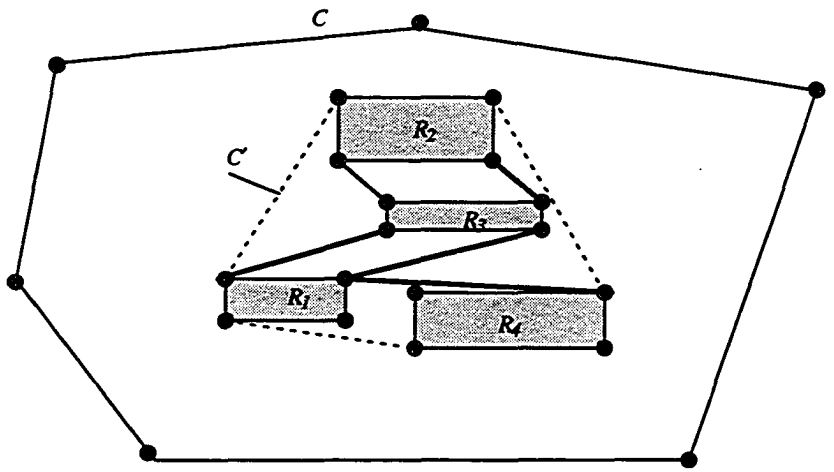

Figure 6.15: Illustrating the partitioning of C' after Step 3

Thus the following result is obtained.

Theorem 6.8. Triangulation of the convex hull of a given set of n iso-oriented rectangular holes can be done in $T_{RV}(n,p,M)+T_{Monotone}(n,p,M)+T_{Sort}(n,p,M)+O(M)$ time on an ACM(n,p,M).

Proof. The running time of the algorithm is obvious from the time taken by each of the steps. To prove the correctness it suffices to show that every point interior to the convex region determined by R is within a triangle.

Consider a point q within the convex hull. Let R_l and R_r be the two rectangles hit by q^- and q^+, respectively. Note that, R_l and R_r always exist because of the rectangles R_0 and R_{n+1} appended by us.

Observe that, if $CR(R_l)$ is ϕ then $q \in CL(R_r)$. Similarly, if $CL(R_r)$ is ϕ then $q \in CR(R_l)$. If $CR(R_l) = \phi$ then $bl(R_r) < br(R_l) < tr(R_l) < tl(R_r)$. To see that this is true, assume $bl(R_r) > br(R_l)$. Since, $CR(R_l)$ is empty, $br(R_r)$ cannot be blocked by R_l. This implies that there exists some rectangle R_x blocking the horizontal ray towards negative x-direction from $br(R_r)$. Obviously, the top edge of

R_x lies below q and $tl(R_x)$ cannot be blocked by $left(R_l)$. By repeating the above argument there should exist a rectangle below q that has $left(R_l)$ as its solution, contradicting the assumption that $CR(R_l)$ is empty. Other cases can be argued similarly. Thus, the horizontal strip (see Figure 6.17) determined by the horizontal rays from $tr(R_l)$ and $br(R_l)$ blocked by $right(R_r)$ contains no other rectangle and q is in $CL(R_r)$. Similarly if $CL(R_r)$ is ϕ, q lies in $CR(R_l)$.

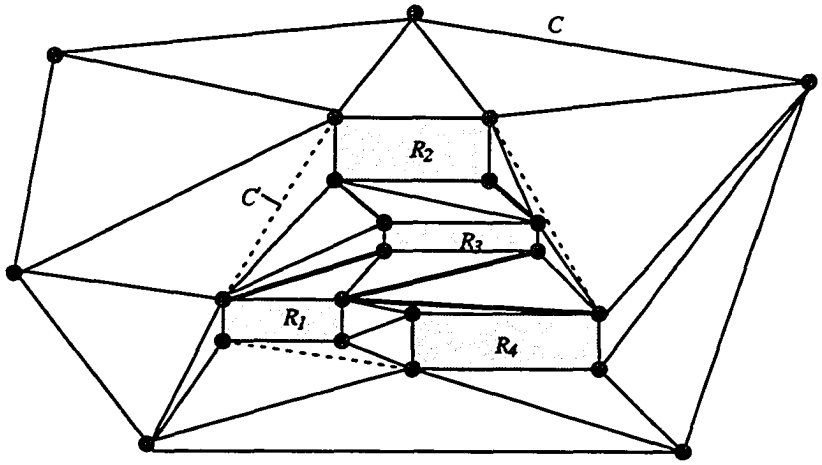

Figure 6.16: Illustrating the triangulation after Step 4

The only other case left is when both $CR(R_l)$ and $CL(R_r)$ exist. In this case, consider the rectangles R_a and R_b above and below q respectively, having the closest y-coordinates. At least one of R_a and R_b is guaranteed to exist because of the assumption that both the contours $CL(R_r)$ and $CR(R_l)$ exist. Note that, the bottom edge of R_a should be above q and the top edge of R_b below q. As shown in the Figure 6.17, let e be the diagonal of the triangulation joining $bl(R_a)$ with $tl(R_b)$ and e' be the one joining $br(R_a)$ with $tr(R_b)$. Since, $e \in CR(R_l)$ and $e' \in CL(R_r)$, q belongs to either of the contours or the special trapezoid bounded by R_a, R_b with e and e'. Since each of these regions is triangulated, it is guaranteed that every point

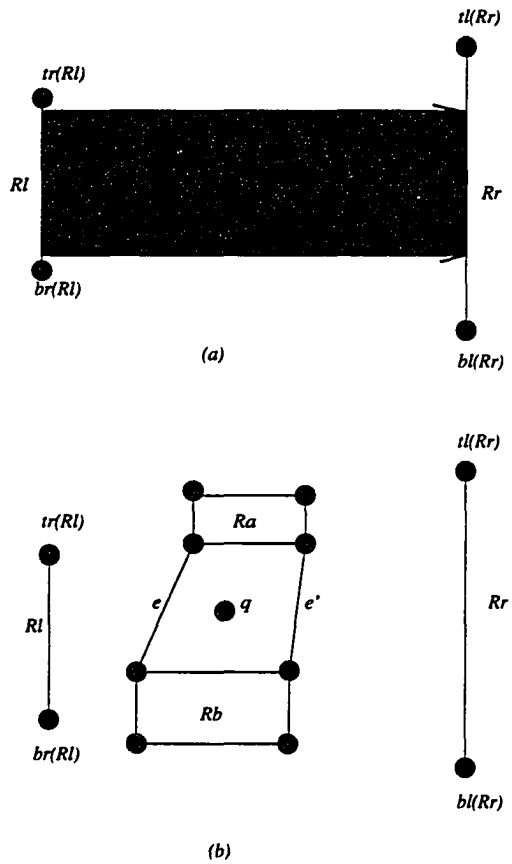

Figure 6.17: Illustrating the proof of Theorem 6.8

within the convex region belongs to some resulting triangle. □

Once $C \setminus C'$ is triangulated, the problem at hand is solved as illustrated in Figure 6.16. Thus, the following result is obtained.

Theorem 6.9. Triangulation of a convex region, of size n, with n iso-oriented rectangular holes can be done in $O(T_{RV}(n,p,M))+ O(T_{Monotone}(n,p,M))+O(M)+ O(T_{Convexhull}(n,p,M))+O(T_{Convexhole}(n,p,M))$ time on an ACM(n,p,M). □

6.5 CONVEX REGION WITH ORDERED SEGMENTS

In this section let us discuss another variation of the constrained triangulation problem where a convex region containing an ordered set of line segments is to be triangulated, including the various segments in the triangulation.

The problem is stated as follows: given a set of n well ordered segments $S = \{s_1, s_2, \ldots, s_n\}$ contained in a convex region C with n vertices, it is required to determine the triangulation of C including the given segments.

Assume that the set S is stored M segments per processor in the first $\frac{n}{M}$ processors of an ACM(n, p, M), where a processor P_i $(0 \leq i \leq \frac{n}{M} - 1)$ stores the seqments $s_{i*M+1}, \ldots, s_{(i+1)M}$. Add two segments s_0 and s_{n+1} to S as illustrated in Figure 6.18. Also, C is stored M vertices per processor in the first $\frac{n}{M}$ processors of an ACM(n, p, M).

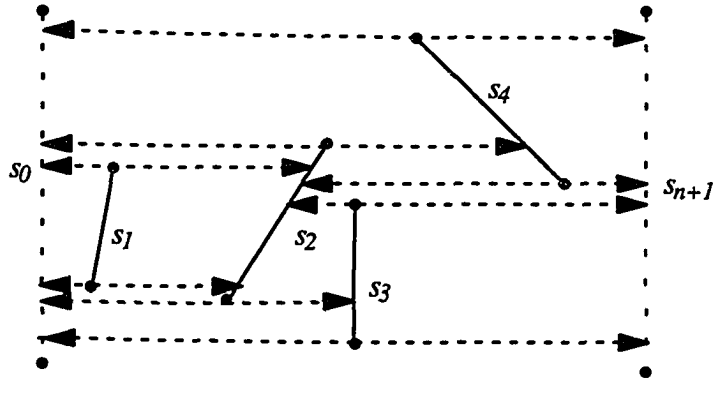

Figure 6.18: Illustrating the solutions to EV in Step 2 of triangulation of segments

The approach to this problem is similar to the triangulation in presence of rectangular forbidden regions.

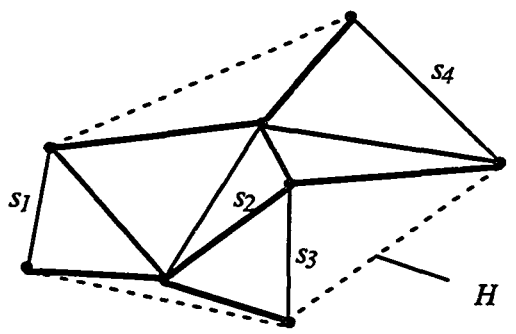

Figure 6.19: Illustrating the convex hull H after Step 4

Template Algorithm 6.5:

Step 1. Determine the convex hull H of S in $O(T_{convexhull}(n,p,M))$ time.

Step 2. Triangulate $C \setminus H$ in $O(T_{Convexhole}(n,p,M))$ time by applying template algorithm 6.4.

Step 3. In order to triangulate H, solve EV problem for S in $T_{EV}(n,p,M)$ time. The solution to the EV problem for the segments in Figure 6.17 is illustrated in Figure 6.18. The definition of closest left contour $CL(s_i)$, and the closest right contour $CR(s_i)$ for each of the segments is identical to that for the rectangles in Section 6.4. For every segment s_i compute $CL(s_i)$, and $CR(s_i)$. Observe that in this case there will be no special trapezoids. The convex hull of the segments is divided into several special monotone polygons.

Step 4. Triangulate all the special monotone polygons in parallel, as described in Section 6.4. This is accomplished in $O(T_{Monotone}(n,p,M))$ time.

Theorem 6.10. The problem of triangulating a convex region, of size n, containing a set of n ordered segments $S = s_1, s_2, \ldots, s_n$ stored M per processor among the first $\frac{n}{M}$ processors of an ACM(n,p,M) is solved in $O(T_{Convexhull}(n,p,M))+O(T_{EV}(n,p,M))+O(T_{Monotone}(n,p,M))$ time. □

CHAPTER 7

TRIANGULATION ON ENHANCED MESHES

In this chapter, let us discuss how the template algorithms for the triangulation problems discussed for the abstract computational model, ACM(n, p, M), in Chapter 6, are ported to enhanced meshes. Not surprisingly, porting the template algorithms to the RMESH results in O(1) time solutions to the various triangulation problems, thus proving for another time that the power of reconfigurability of the bus system can be exploited to design very fast algorithms.

The organization of the chapter is as follows. Section 7.1 discusses the tools needed to port the template algorithms from Chapter 6 to the MMB. Next, Section 7.2 discusses the triangulation algorithms on the MMB. Section 7.3 discusses the various tools for the RMESH and finally Section 7.4 presents the O(1) time triangulation algorithms for the RMESH.

7.1 TOOLS FOR THE MMB

In this section, let us discuss the implementation of the various tools that are needed to port the template algorithms to the MMB.

• *ANLV*: Given an arbitrary sequence of real numbers $< a_1, a_2, \ldots, a_n >$, stored one per processor in the first row of an mesh with multiple broadcasting of size

$n \times n$, associate with every a_i a vertical line segment s_i with endpoints $(i, -\infty)$ and (i, a_i). Assume that the viewpoint ω lies at $(-\infty, 0)$. It is easy to confirm that the resulting set S of vertical line segments is well ordered, and the EV algorithm discussed in Section 4 can be applied to solve the visibility relations between the segments. Clearly, for every endpoint (i, a_i) the solution corresponds to the nearest line segment that is blocking a horizontal ray emanating from (i, a_i) to the left and to the right. This translates immediately into a solution to the ANLV, as desired. Consequently, the following result is obtained.

Lemma 7.1. An arbitrary instance of size n of the all nearest larger values problem stored in the first row of the MMB of size $n \times n$ can be solved in $O(\log n)$ time. □

• *Convex hull*: Quite recently, Olariu *et al.* [72] have proposed a time-optimal algorithm to compute the convex hull of a set of points in the plane, on the MMB. More precisely, they proved the following result.

Proposition 7.2. The convex hull of an n-element set of points in the plane, stored one item per processor in one row or one column of the MMB of size $n \times n$ can be computed in $O(\log n)$ time. Furthermore, this is time-optimal. □

7.2 TRIANGULATION ON THE MMB

In this section, let us discuss the various triangulations in the context of the MMB, which are instantiations of the template algorithms discussed in Chapter 6.

7.2.1 TRIANGULATING A SPECIAL MONOTONE POLYGON

In this subsection, let us discuss how the template algorithm 6.1 to triangulate a special monotone polygon is ported to the MMB.

Let $\mathcal{M} = v_1, v_2, \ldots, v_n$ be an n-vertex special monotone polygon with its vertices specified in clockwise order and with v_1v_n denoting the base edge. The vertices of the polygon are assumed to be stored in the first row of a mesh with multiple broadcasting of size $n \times n$, one vertex per processor. The details of the various steps involved in triangulating the special monotone polygon $\mathcal{M}$ are identical to the template algorithm and can be ported to an MMB as follows. Every vertex v_i of $\mathcal{M}$ determines whether it belongs to an ascending or descending chain. This can be performed in O(1) time. As in the template algorithm, each vertex $v_i = (x_i, y_i)$ of $\mathcal{M}$ is associated with an element $s_i = y_i$ and solve the resulting instance of the ANLV problem. Every vertex v_i that has identified (at least) a match v_j adds the diagonal v_iv_j to the triangulation. This can be accomplished in $O(\log n)$ time by virtue of Lemma 7.1. Mark the vertices as specified in Step 4 of the template algorithm 6.1. Let $v_1 = v_{i_1}, v_{i_2}, \ldots, v_{i_r} = v_n$ be the sequence of marked vertices enumerated by increasing x-coordinate and let $\mathcal{M}'$ be the monotone polygon determined by these marked vertices. Rotate $\mathcal{M}'$ so that v_1v_n becomes parallel to the x-axis and repeat the above process. This can again be accomplished in $O(\log n)$ time. Thus the following result is obtained.

Theorem 7.3. The problem of triangulating an n-vertex special monotone polygon can be solved in $O(\log n)$ time on the MMB of size $n \times n$. □

7.2.2 TRIANGULATING A SET OF POINTS

This subsection discusses the solution to the problem of triangulating a given set S of n points in the plane obtained by porting the template algorithm 6.2 to the MMB. Furthermore, this algorithm is found to be time-optimal on the MMB.

Let us begin by showing that for both the CREW-PRAM and the mesh with multiple broadcasting, the task of triangulating a set of n points in the plane has a time lower bound of $\Omega(\log n)$.

The stated time lower bound can be derived by reducing the OR problem to triangulation. Let $b_1, b_2, \ldots, b_n$ be an arbitrary input to OR. Construct a set $\{p_0, p_1, \ldots, p_{n+1}\}$ of points in the plane by setting for every i $(1 \leq i \leq n)$, $p_i = (i, 0)$ if $b_i = 0$, and by setting $p_i = (i, 1)$ if $b_i = 1$. To complete the construction, add the points $p_0 = (0, 1)$ and $p_{n+1} = (n + 1, 1)$. Now, the solution to the OR problem is 0 if, and only if, the segment $p_0 p_{n+1}$ belongs to the triangulation. The conclusion follows by Proposition 4.4.

Lemma 7.4. The problem of triangulating a set of n points in the plane has a time lower bound of $\Omega(\log n)$ on the CREW-PRAM, no matter how many processors and memory cells are used. □

Now Lemma 7.4 and Proposition 4.5 combined, imply the following result.

Corollary 7.5. The problem of triangulating a set of n points in the plane has a time lower bound of $\Omega(\log n)$ on a mesh with multiple broadcasting of size $n \times n$. □

Now, let us confirm that the application of template algorithm 6.2 results in a time-optimal algorithm to the triangulation problem on the MMB. Begin by computing the convex hull of S, and by Proposition 7.2 this task can be performed in $O(\log n)$ time. Next, sort all the points in S by their x coordinates. By virtue of Proposition 4.6, this task can be performed in $O(\log n)$ time. Further, join every point with

its immediate neighbor in the sequence sorted by x. All the convex hull edges and the edges drawn between two adjacent points are included in the triangulation. As noted in the template algorithm, the chain determined by joining adjacent points in the sorted sequence divides the entire region within the hull into special monotone polygons. Each of these polygons with a base edge on the lower hull can be triangulated independently in parallel using the algorithm described in Subsection 7.2.1. The same can be repeated for the polygons with a base edge belonging to the upper hull. Now, Theorem 7.3 guarantees that each of the above steps can be performed in $O(\log n)$ time and thus the triangulation can be computed in $O(\log n)$ time. The time-optimality of the algorithm is guaranteed by Corollary 7.5. Thus, the following result is obtained.

Theorem 7.6. The problem of triangulating a set S of n points in the plane can be done in $O(\log n)$ time on a mesh with multiple broadcasting. Furthermore, this is time-optimal. □

7.2.3 TRIANGULATING A CONVEX HULL WITH A CONVEX HOLE

In this subsection, let us discuss how the triangulation of convex region with a convex hole is implemented on the MMB, which is in fact an adaptation of the template algorithm 6.3 to the MMB.

Let C be stored at most two vertices per processor in the first $\frac{n}{2}$ processors, in the first row of the MMB and H be stored at most two vertices per processor in the next $\frac{m}{2}$ processors in the first row of the MMB of size $n \times n$. Begin by chosing an arbitrary point interior to H and convert the vertices of C and H to polar coordinates having ω as pole and the positive x-direction as polar axis. Since ω is interior to C and H, convexity guarantees that the vertices of both C and

H occur in sorted order about ω. Next, these two sorted sequence are merged in O(1) time as described in Proposition 4.1, and let $b_1, b_2, \ldots, b_{n+m}$ be the resulting sequence sorted by polar angle.

Identify the Case 1 sequences and Case 2 sequences as in the template algorithm. All the polygons corresponding to the Case 1 sequences can be solved in parallel by replicating the first row in all the rows of the mesh and solving a subsequence per row. Case 2 items can be solved similarly. This can be accomplished in O(1) time. Thus the following result is obtained.

Theorem 7.7. Let C be an n-vertex convex region and let H be an m-vertex convex hole ($m \in O(n)$) within C. Assuming that C and H are stored in one row or column of a mesh with multiple broadcasting of size $n \times n$, the planar region $C \setminus H$ can be triangulated in O(1) time. □

7.2.4 TRIANGULATING A CONVEX REGION WITH RECTANGULAR HOLES

This subsection discusses the implentation of the template algorithm 6.4 to the MMB, to solve the problem of triangulating a convex region with rectangular forbidden regions.

Let $C = c_1, c_2, \ldots, c_n$ be a convex region containing n rectangular holes specified by a set $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ of rectangles with their sides parallel to the axes. The task at hand is to triangulate $C \setminus \mathcal{R}$. Let C' be the convex hull of the set $\mathcal{R}$ of rectangles. Triangulate $C \setminus C'$, using the algorithm discussed in Subsection 7.2.3. Now to triangulate C', as in the template algorithm, add two rectangles R_0 and R_{n+1}, to the given set $\mathcal{R}$ of rectangles. Solve the rectangle visibility for the set $R_0, R_1, \ldots, R_{n+1}$. This can be done in $O(\log n)$ time as stated in Theorem 4.17. Associate with each corner point of rectangle R_i an information packet containing its

coordinates and two numbers u and v, as specified in the template algorithm. Sort the information packets, first on the u value and then on the y-coordinate. Clearly, this step requires $O(\log n)$ time. Determine the closest contours, and identify the special trapezoids and special monotone polygons. The special trapezoids can be trivially triangulated in O(1) time. Also, the special monotone polygons can be identified and triangulated in independent submeshes of the original mesh in $O(\log n)$ time as stated in Theorem 7.3.

Theorem 7.8. Triangulation of the convex region, of size n, containing a given set of n iso-oriented rectangular holes can be done in $O(\log n)$ time on a mesh with multiple broadcasting of size $n \times n$. □

7.2.5 TRIANGULATING A CONVEX REGION WITH ORDERED SEGMENTS

In this subsection let us discuss triangulation problem where a convex region containing an ordered set of line segments is to be triangulated, including the various segments in the triangulation.

Consider a set of n well ordered segments $S = \{s_1, s_2, \ldots, s_n\}$ in the plane enclosed in a convex region C. C is stored one vertex per processor in the first row of the MMB and S is stored one segment per processor in the first row of the MMB. As described in the template algorithm, determine the convex hull H of the endpoints of S. Triangulate $C \setminus H$ in O(1) time, as described in Subsection 7.2.3. H can be triangulated as described in template algorithm 6.5 after applying the EV algorithm to S and determining the closest contours. By virtue of Theorem 4.10 and Theorem 7.3, H can be triangulated in $O(\log n)$ time. Thus, the following result is obtained.

Theorem 7.9. The problem of triangulating a convex region, of size n, containing a set of n ordered segments $S = s_1, s_2, \ldots, s_n$ stored one per processor in the first row of the MMB, can be done in $O(\log n)$ time. □

7.3 TOOLS FOR THE RMESH

The purpose of this section is to discuss a number of data movement techniques for the RMESH that will be instrumental in the instantiation of the template algorithms to the RMESH.

In addition to the various tools discussed in Chapter 4, the following tools are needed for the various triangulation algorithms.

• *ANLV*: Given the solution to the SV problem, ANLV problem can be solved in O(1) time. Thus the following result is stated.

Lemma 7.10. The ANLV problem of an n element set can be determined in O(1) time on a RMESH of size $n \times n$. □

• *Convex hull*: Quite recently, Olariu *et al.* [71], Wang and Chen [90], and Nigam and Sahni [69] have proposed a O(1) time algorithm to compute the convex hull of a set of points in the plane. More precisely, they all proved the following result.

Proposition 7.11. The convex hull of an n-element set of points in the plane, stored one item per processor in one row or one column of a RMESH of size $n \times n$ can be computed in O(1) time. □

7.4 TRIANGULATION ON THE RMESH

In this section, the template algorithms for the various triangulation problems are ported to the RMESH, giving O(1) time solutions.

7.4.1 TRIANGULATING A SPECIAL MONOTONE POLYGON

In this subsection, template algorithm 6.1 to triangulate special monotone polygons is implemented on the RMESH. Consider a special monotone polygon, $\mathcal{M} = v_1, v_2, \ldots, v_n$, specified in clockwise order and with v_1v_n denoting the base edge. The vertices of the polygon are assumed to be stored in the first row of a RMESH $\mathcal{M}$ of size $n \times n$, one vertex per processor. The details of the various steps involved in triangulating the special monotone polygon $\mathcal{M}$ are spelled out as follows: By checking its neighbors, every vertex v_i of $\mathcal{M}$ determines whether it belongs to an ascending or descending chain, in O(1) time. Each vertex $v_i = (x_i, y_i)$ of $\mathcal{M}$ is associated with a element y_i and solve the resulting instance of ANLV problem. By virtue of Lemma 7.10, this can be accomplished in O(1) time. As in the template algorithm, every vertex v_i that has identified (at least) a match v_j adds the diagonal v_iv_j to the triangulation and records the resulting triangle in O(1) time. Mark the vertices as in Step 4 of the template algorithm. Let $v_1 = v_{i_1}, v_{i_2}, \ldots, v_{i_r} = v_n$ be the sequence of marked vertices enumerated by increasing x-coordinate and let $\mathcal{M}'$ be the monotone polygon determined by these marked vertices. Rotate $\mathcal{M}'$ so that v_1v_n becomes parallel to the x-axis and triangulate it by repeating the above process. The following result is thus obtained.

Theorem 7.12. The problem of triangulating an n-vertex special monotone polygon stored in the first row of a RMESH size $n \times n$ can be solved in O(1) time. □

7.4.2 TRIANGULATING A SET OF POINTS

The purpose of this subsection is to demonstrate a O(1) time triangulation algorithm for points in the plane. Template algorithm 6.2 is instantiated in the context of the RMESH to achieve this.

Specifically, consider a set S of n points in the plane stored in the first row of a RMESH of size $n \times n$, one point per processor. Computing the convex hull of the S. This computation takes O(1) time as stated in Proposition 7.11. Note that all the edges of the convex hull will be part of the desired triangulation. Next, sort the points in S in increasing order of their x-coordinates and add a diagonal between adjacent points in the sorted sequence, which divide the region within the convex hull into several monotone polygons as stated in the template algorithm. This is accomplished in O(1) time, as stated in Proposition 4.22. Each of these polygons with the base edge on the lower hull can be triangulated independently, in parallel, using the algorithm for triangulating a special monotone polygon described in Subsection 7.4.1. The same can be repeated for the polygons with an edge on the upper hull. Theorem 7.12 guarantees that the above step can be performed in O(1) time. Consequently, the following result is obtained.

Theorem 7.13. An arbitrary set S of n points in the plane, stored on point per processor in the first row of a RMESH of size $n \times n$, can be triangulated in O(1) time. □

7.4.3 TRIANGULATING A CONVEX REGION WITH ONE CONVEX HOLE

This subsection discusses how the problem of triangulation a convex region with a convex hole is implemented on the RMESH, based on the template algorithm 6.3.

Let $C = c_1, c_2, \ldots, c_n$ be a convex region of the plane and $H = h_1, h_2, \ldots, h_m$ be a convex hole within C. Let both C and H be stored one vertex per processor in the first row of a RMESH $\mathcal{M}$ of size $n \times n$. As in the template algorithm, choose an arbitrary point interior to H and convert the vertices of C and H to polar coordinates having ω as pole and the positive x-direction as polar axis, and merge

the vertices of C and H. This can be done in O(1) time as specified in Proposition 4.21, and let $b_1, b_2, \ldots, b_{n+m}$ be the resulting sequence sorted by polar angle.

Consider the sequence b_1, b_2, ..., b_{n+m} is stored in order by the processors in the first row of the mesh, at most two vertices per processor. Identification and triangulation of the polygons corresponding to Case 1 and Case 2 subsequences detailed in the template algorithm is identical to the way it is implemented on the MMB and is accomplished in O(1) time. Thus the following result is obtained.

Theorem 7.14. Let C be an n-vertex convex region and let H be an m-vertex convex hole ($m \in O(n)$) within C. Assuming that C and H are stored in one row or column of a RMESH of size $n \times n$, the planar region $C \setminus H$ can be triangulated in O(1) time. □

7.4.4 TRIANGULATING A CONVEX REGION WITH RECTANGULAR HOLES

In this subsection, the template algorithm 6.4 to triangulate a convex region in the presence of rectangular holes is ported to a O(1) time algorithm on the RMESH.

Let $C = c_1, c_2, \ldots, c_n$ be a convex region containing n rectangular holes specified by a set $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ of rectangles with their sides parallel to the axes. Convex hull C' of $\mathcal{R}$ can be determined in O(1) time by Proposition 7.11. Triangulate $C \setminus C'$ using the algorithm discussed in Subsection 7.4.3 and this takes O(1) time by virtue of Theorem 7.14. As in the template algorithm, to the given set of rectangles add two rectangles R_0 and R_{n+1}. Solve the rectangle visibility for the set $R_0, R_1, \ldots, R_{n+1}$. This can be done in O(1) time as stated in Theorem 4.26. Associate with each corner point of rectangle R_i an information packet containing its coordinates and two numbers u and v, as in the template algorithm. Sort the information packets, first on the u value and then on the y-coordinate. Clearly, this

step requires O(1) time. The closest left and right contours can be identified in O(1) time. Now in another O(1) time the special trapezoids can be identified and triangulated by adding appropriate diagonals. Also, the special monotone polygons can be identified and triangulated in independent submeshes of the original mesh in O(1) time as stated in Theorem 7.12. Thus, the following result is obtained.

Theorem 7.15. Triangulation of a convex region, of size n, containing a given set of n iso-oriented rectangular holes can be done in O(1) time on a RMESH of size $n \times n$. □

7.4.5 TRIANGULATING A CONVEX REGION WITH ORDERED SEGMENTS

Consider a set of n well ordered segments $S = s_1, s_2, \ldots, s_n$ contained in a convex region C of n vertices. The segments in S are stored one per processor in the first row of the mesh. Similarly, the vertices of C are stores one vertex per processor in the first row of the mesh. The approach to this problem is similar to the triangulation in the presence of rectangular holes and the details are omitted. Thus, the following result is obtained.

Theorem 7.16. The problem of triangulating a convex region, of size n, containing a set of n ordered segments $S = s_1, s_2, \ldots, s_n$ stored one per processor can be done in O(1) time on a RMESH of size $n \times n$. □

CHAPTER 8

TRIANGULATION ON COARSE-GRAIN MULTICOMPUTERS

In this chapter, let us develop some very powerful tools for the coarse-grain multicomputers, in addition to the ones developed in Chapter 5, and use them to port the various template algorithms for the triangulation problems to coarse-grain multicomputers. The computation time of the resulting algorithms is found to be optimal.

The organization of the chapter is as follows. Section 8.1 discusses the tools developed for the CGM in order to apply the template algorithms for the triangulation problems, to this model of computation. This is followed by Section 8.2, where the application of the template algorithms to provide computationally optimal algorithms on the CGM is discussed.

8.1 TOOLS

In addition to the tools developed in Chapter 5, the following tools are essential to port the template algorithms designed for the ACM(n, p, M) to the CGM(n, p).

• *ANLV:* The ANLV problem is solved on the CGM(n, p) as discussed in Subsection 8.1.1.

• *Convex Hull* : The convex hull of a set of points in the plane is computed on the CGM(n,p) as described in Subsection 8.1.3.

Also, Subsection 8.1.2 discusses the problem of merging two convex hulls which is an essential ingredient of the convex hull algorithm discussed in Subsection 8.1.3.

8.1.1 ALL NEAREST LARGER VALUES

The purpose of this section is to exhibit an efficient solution for the ANLV problem on a CGM(n,p). It can be solved by viewing the ANLV as special instance of EV problem in $O(\frac{n \log n}{p})+ O(\log p T_{Alltoall}(n,p))$ time. However, the ANLV problem can be solved in $O(\frac{n}{p})+ O(T_{Alltoall}(n,p))$ time using the dynamic load balancing scheme discussed in Chapter 5. Since the sequential lower bound for this problem is $\Omega(n)$, this algorithm is computationally time-optimal.

Consider a sequence of n real numbers $< a_1, a_2, \ldots, a_n >$, $\frac{n}{p}$ per processor in a CGM(n,p), such that any processor P_i stores the items $A_i = a_{i*\frac{n}{p}+1}, \ldots, a_{(i+1)*\frac{n}{p}}$. Let us discuss only the computation of the nearest larger value to the left of every a_i, the computation of the ones to the right can be done symmetrically.

Given the input sequence of real numbers $< a_1, \ldots, a_n >$, a sequence of vertical segments is obtained by associating the element a_j with a segment s_j with its top endpoint specified by the coordinated (j, a_j) and the bottom endpoint represented by $(j, -\infty)$. Now, every P_i stores the subsequence $S_i =< s_{i*\frac{n}{p}+1}, \ldots, s_{(i+1)*\frac{n}{p}} >$. Note that the sequence of segments $< s_1, \ldots, s_n >$ are sorted by their x-coordinates.

Step 1. Let every processor P_i solve a local instance of ANLV problem for the items in $A_i =< a_{i*\frac{n}{p}+1}, \ldots, a_{(i+1)*\frac{n}{p}} >$, where every item determines the nearest larger value to its left and right. This is equivalent to determining the nearest line segment that is blocking a horizontal ray emanating from each of the top endpoints

of $s_j \in S_i$, in positive and negative x directions. This can be accomplished using the sequential algorithm to compute the ANLV in $O(\frac{n}{p})$ time [78]. Consider the subset of segments in S_i, whose top endpoints did not find their solution, in negative x direction,within the set S_i. This subset of S_i is said to be the *left contour* and is referred to as $LC(S_i)$. Similarly, the subset of S_i, whose top endpoints that did not find their solutions in the *right* direction are said to belong to the *right contour* and are referred to as $RC(S_i)$.

After determining the left and the right contours of S_i, every P_i needs to determine if any of the segments in $RC(S_k)$, $k < i$, block the horizontal ray emanating from the top endpoint of each $s_j \in LC(S_i)$. This can be accomplished using a successive refinement technique, where as a first step, every P_i determines for every $s_j \in LC(S_i)$, the *pocket* to which its solution belongs to. Note that, the pocket of $s_j \in LC(S_i)$ is k if the $RC(S_k)$ contains the solution to s_j. Once this information is available, the dynamic load balancing scheme detailed in the Chapter 5 could be applied to obtain the actual solutions to every s_j. The details are as follows.

Step 2. Every processor P_i determines the *tallest* segment it holds, and that is considered the *sample* item t_i. Once $LC(S_i)$ and $RC(S_i)$ are determined, t_i can be obtained in $O(1)$ time. Now, perform an all-gather operation so that every processor has a copy of the sequence of sample items from every processor, $\mathcal{T} =< T_0, t_1, \ldots, t_{p-1} >$. This can be accomplished in $T_{Allgather}(p, p)$ time.

In every P_i perform the following computation in parallel. Determine the right contour of the sample $\mathcal{T}$, given by $RC(\mathcal{T})$. Now, for every $s_j \in LC(S_i)$, determine if any of the segments $t_k \in RC(\mathcal{T})$ block the horizontal ray emanating from its top endpoint. This can be accomplished in $O(\frac{n}{p})$ time. For each endpoint in $LC(S_i)$, determine the pocket to be k, if it is blocked by the segment $t_k \in RC(T)$.

Observation 8.1. The actual segment that would block $s_j \in \mathrm{LC}(S_i)$ is contained in $\mathrm{RC}(S_k)$, where k is its pocket of s_j. □

Step 3. The dynamic load balancing scheme discussed in Subsection 5.1.1 can be applied to determine the final solutions for every $s_j \in \mathrm{LC}(S_i)$. This can be accomplished in $O(\frac{n}{p})$ computational time, and $O(T_{Alltoall}(n,p))$ communication time, by virtue of Lemma 5.2. Thus, the following result is obtained.

Theorem 8.2. The All Nearest Larger Values problem for a sequence of n items, stored $\frac{n}{p}$ per processor on a CGM(n,p), can be solved in $O(\frac{n}{p})$ computational time, and $O(T_{Alltoall}(n,p))$ communication time. □

8.1.2 HULL MERGE

This subsection discusses the problem of merging two upper hulls of size $\frac{n}{2}$ vertices, stored in $\frac{p}{2}$ processors each, on a CGM(n,p). This is accomplished by computing the supporting line of the two upper hulls and updating the ranks of the vertices on the resulting hull. The running time of the algorithm is $O(\frac{n}{p})$ computational time and $O(T_{Broadcast}(\frac{n}{p},p))$ time. Since the sequential lower bound for this problem is $\Omega(n)$, this algorithm is computationally time-optimal.

Let us discuss a few terms that are used in the following discussion. Consider the upper hull $U = u_1, u_2, \ldots, u_k$ of a set S of points in the plane. A *sample* of U is a subset of vertices in U enumerated in the same order as in U. Consider an arbitrary sample $A = (u_1 = a_0, a_1, \ldots, a_s = u_k)$ of U. The sample A partitions U into s *pockets* A_1, A_2, ..., A_s, such that A_i involves the vertices in U lying between a_{i-1} and a_i.

Now, let us discuss the problem of computing the supporting line of two separable upper hulls $U = u_1, u_2, \ldots, u_{\frac{n}{2}}$ and $V = v_1, v_2, \ldots, v_{\frac{n}{2}}$, having $\frac{n}{2}$ vertices each. The $\frac{n}{2}$ vertices of U are stored in the processors $P_0, P_1, \ldots, P_{\frac{p}{2}-1}$, $\frac{n}{p}$ per

processor, in the CGM(n,p). Again, V vertices is stored in processors $P_{\frac{p}{2}},\ldots,P_{p-1}$ of the CGM(n,p), $\frac{n}{p}$ per processor. Consider the sample A of U consisting of every $\frac{n}{p}^{th}$ vertex in U (including the last vertex) and is enumerated as $a_0 = u_1, a_1 = u_{\frac{n}{p}+1},\ldots,a_{\frac{p}{2}-1} = u_{(\frac{p}{2}-1)\frac{n}{p}+1}, a_{\frac{p}{2}} = u_{\frac{p}{2}}$. Similarly, let B be the sample of V given by $b_0 = v_1, b_1 = v_{\frac{n}{p}+1},\ldots,b_{\frac{p}{2}-1} = v_{(\frac{p}{2}-1)\frac{n}{p}+1}, b_{\frac{p}{2}} = v_{\frac{p}{2}}$. The two samples determine pockets A_1, A_2, ..., $A_{\frac{p}{2}}$ and B_1, B_2, ..., $B_{\frac{p}{2}}$ in U and V, respectively. Let the supporting line of A and B be achieved by a_i and b_j, and let the supporting line of U and V be achieved by u_p and v_q. The following technical result has been established in [5].

Proposition 8.3. At least one of the following statements is true:

(a) $u_p \in A_i$;

(b) $u_p \in A_{i+1}$;

(c) $v_q \in B_j$;

(d) $v_q \in B_{j+1}$. □

Proposition 8.3 suggests the following procedure to determine the supporting line of the two hulls. In an all-gather operation, the samples A and B are replicated in every processor P_i $(0 \leq i \leq p-1)$ of the CGM(n,p). This is accomplished in $T_{Allgather}(p,p)$ time. In O(log p) time, let every P_i compute the supporting line for A and B, using the sequential algorithm [78], and let a_i and b_j achieve the supporting line of A and B. The next task is to check which of the four conditions in Proposition 8.3 holds. For example, condition (b) is equivalent to saying that u_p lies to the right of a_i and left of a_{i+1}. To check (b), the supporting lines s and s' from a_i and a_{i+1} to V are computed, as follows. Every processor P_i $(\frac{p}{2} \leq i \leq p-1)$, determines if any of the vertices v_k of V it holds is such that $v_k a_i$ is the supporting line s to V. Exactly one processor determines s, and broadcasts the value of v_k and similarly s' is also

computed. This takes $O(\frac{n}{p})+O(T_{Broadcast}(1,p))$ time. Next, the processor storing a_{i+1} checks if the right neighbor of a_i in U lies above s'. Similarly, the processor storing a_i checks if the right neighbor of a_i in U lies above s. It is easy to see that u_p belongs to A_{i+1} if, and only if, both these conditions hold. The other conditions are checked similarly.

Assume without loss of generality that condition (b) holds. The next target is to compute the supporting line of A_{i+1} and B, which is accomplished by the processor holding pocket A_{i+1} in $O(\log \frac{n}{p})$ time, using the sequential algorithm. It is important to note that convexity guarantees that if the supporting line of A_{i+1} and B is not a supporting line to U and V, then the pocket B_t that contains v_q can be determined. Therefore, the supporting line of U and V can be determined by identifying the pocket B_t and determining the supporting line of A_{i+1} and B_t, which is nothing but the supporting line of U and V. Note that, this step would require $O(\log \frac{n}{p})$ computational time and also $O(T_{Broadcast}(\frac{n}{p},p))$ communication time to move B_t to processor storing A_{i+1}.

Once the supporting line of U and V is determined, in $T_{Broadcast}(1,p)$ time all the processors can be informed of the supporting line, and in $O(\frac{n}{p})$ computational time, the ranks of the various vertices on the upper hull can be updated. Thus, the following result is obtained.

Lemma 8.4. Given two separable upperhulls U and V of $\frac{n}{2}$ vertices each, stored $\frac{n}{p}$ vertices per processor in the p processors of a CGM(n,p), the two hulls can be merged in $O(\frac{n}{p})$ computational time and $O(T_{Broadcast}(\frac{n}{p},p))$ time. □

8.1.3 CONVEX HULL

This subsection discusses the convex hull algorithm and as stated earlier uses the algorithm to merge convex hulls, described in Subsection 8.1.2. The running time

of the algorithm is $O(\frac{n \log n}{p})$ computational time and $O(\log p T_{Alltoall}(n,p))$ communication time. Since the sequential lower bound for this problem is $\Omega(n \log n)$, this algorithm is computationally time-optimal.

Consider a set $S = \{s_1, s_2, \ldots, s_n\}$ of n points in the plane, stored $\frac{n}{p}$ per processor, in an CGM(n,p). To avoid tedious details, assume without loss of generality, that the points in S are in general position, with no three points collinear and no two having the same x and y coordinates. The algorithm proceeds by determining the upper and lower hulls of S separately and then merges them. The details of the computation of the upper hull is as follows. Note that, the lower hull can be computed similarly.

Step 1. Sort the points in S in increasing order of their x coordinates, and this can be done in $O(\frac{n \log n}{p})$ computational time, and $O(\log p T_{Alltoall}(n,p))$ communication time, as stated in Lemma 5.5. Next, in each processor P_i, the convex hull of the $\frac{n}{p}$ points it holds is determined in $O(\frac{n}{p} \log \frac{n}{p})$ time, using the sequential algorithm to compute the convex hull of a set of points [78].

Step 2. This step involves $\log p$ iterations. In the first iteration, the CGM(n,p) can be viewed as $\frac{p}{2}$ independent CGM's, given by CGM$(\frac{2n}{p}, 2)$ and the upper hulls held in the two processors of each CGM can be merged using the algorithm discussed in previous subsection. In general, in any iteration t, the CGM(n,p) can be viewed as consisting of $\frac{p}{2^t}$ independent CGM's, given by CGM$(\frac{2^t n}{p}, 2^t)$ and in each such CGM, the pair of hulls obtained in iteration $t-1$ are merged. At the end of $\log p$ steps, the convex hull of S is obtained. The running time of each of the steps is bounded by $O(\frac{n}{p})$ computational time and $O(T_{Alltoall}(n,p))$ communication time. Thus, the following result is obtained.

Lemma 8.5. The convex hull of a set S of n points in the plane, stored $\frac{n}{p}$ per processor, on a CGM(n,p), can be determined in $O(\frac{n \log n}{p})$ computational time and $O(\log p T_{Alltoall}(n,p))$ communication time. □

8.2 TRIANGULATION ALGORITHMS

With the various tools in hand, the porting of the template algorithms for the triangulation problems to the CGM(n,p) is accomplished as described in the following subsections.

8.2.1 TRIANGULATING A SPECIAL MONOTONE POLYGON

Let $\mathcal{M} = v_1, v_2, \ldots, v_n$ be an n-vertex special monotone polygon with its vertices specified in clockwise order and with v_1v_n denoting the base edge, stored $\frac{n}{p}$ vertices per processor in a CGM(n,p).

As in the template algorithm 6.1, each vertex $v_i = (x_i, y_i)$ of $\mathcal{M}$ is associated with an element y_i and solve the resulting instance of the ANLV problem. Every vertex v_i that has identified (at least) a match v_j adds the diagonal v_iv_j to the triangulation. Mark the vertices as specified in Step 4 of the template algorithm. Let $v_1 = v_{i_1}, v_{i_2}, \ldots, v_{i_r} = v_n$ be the sequence of marked vertices enumerated by increasing x-coordinate and let $\mathcal{M}'$ be the monotone polygon determined by these marked vertices. Rotate $\mathcal{M}'$ so that v_1v_n becomes parallel to the x-axis and repeat the above process. Thus the following result is obtained.

Theorem 8.6. The problem of triangulating an n-vertex *special* monotone polygon can be solved in $O(\frac{n}{p})$ computational time and $O(T_{Alltoall}(n,p))$ communication time, on a CGM(n,p). □

8.2.2 TRIANGULATING A SET OF POINTS

This subsection discusses the problem of triangulating a given set S of n points in the plane, on a CGM(n,p), obtained by applying template algorithm 6.2. The running time of the algorithm is $O(\frac{n \log n}{p})$ computational time and $O(\log p T_{Alltoall}(n,p))$ communication time. Since the sequential lower bound for this problem is $\Omega(n \log n)$, this algorithm is computationally time-optimal.

Begin by computing the convex hull of S, and by Lemma 8.5, this task can be performed in $O(\frac{n \log n}{p})$ computational time and $O(\log p T_{Alltoall}(n,p))$ communication time. Next, sort all the points in S by their x coordinates. By virtue of Lemma 5.5, this task can be performed in $O(\frac{n \log n}{p})$ computational time and $O(\log p T_{Alltoall}(n,p))$ communication time. Further, join every point with its immediate neighbor in the sequence sorted by x. All the convex hull edges and the edges drawn between two adjacent points are included in the triangulation. The chain determined by joining adjacent points in the sorted sequence divides the entire region within the hull into special monotone polygons. Each of these polygons with a base edge on the lower hull can be triangulated independently in parallel using the algorithm described above. The same can be repeated for the polygons with a base edge belonging to the upper hull. Now, Theorem 8.6 guarantees that each of the above steps can be performed in $O(\frac{n}{p})$ computational time and $O(T_{Alltoall}(n,p))$ communication time. Thus, the following result is obtained.

Theorem 8.7. The problem of triangulating a set S of n points in the plane can be solved in $O(\frac{n \log n}{p})$ computational time and $O(\log p T_{Alltoall}(n,p))$ communication time, on a CGM(n,p). □

8.2.3 TRIANGULATING A CONVEX HULL WITH A CONVEX HOLE

In this subsection let us discuss the algorithm to triangulate a convex hull with a convex hole, which is based on template algorithm 6.3.

Let C be stored $\frac{2n}{p}$ vertices per processor among the first $\frac{p}{2}$ processors of the CGM(n,p) and H be stored $\frac{2n}{p}$ vertices per processor in the next $\frac{m}{\frac{2n}{p}}$ processors of the CGM(n,p). The triangulation algorithm proceeds as in the template algorithm 6.3, where an arbitrary point ω interior to H is chosen and the vertices of C and H are converted to polar coordinates having ω as pole and the positive x-direction as polar axis. This can be accomplished in $O(\frac{n}{p})$ time. Next, the two sequences of vertices of C and H are merged in $O(T_{Merge}(n,p))$ time. Let $B = b_1, b_2, \ldots, b_{n+m}$ be the resulting sequence sorted by polar angle. Case 1 and Case 2 subsequences are identified and solved in parallel as specified in the template algorithm. Thus the following result is obtained.

Theorem 8.8. Given a convex hull C be stored $\frac{2n}{p}$ vertices per processor among the first $\frac{p}{2}$ processors of the CGM(n,p) and convex hole H stored $\frac{2n}{p}$ vertices per processor in the next $\frac{m}{\frac{2n}{p}}$ processors of the CGM(n,p), the planar region $C \setminus H$ can be triangulated in $O(\frac{n}{p})$ computational time and $O(T_{Alltoall}(n,p))$ communication time. □

8.2.4 TRIANGULATING A CONVEX REGION WITH RECTANGULAR HOLES

This subsection discusses the algorithm to triangulate a convex region with rectangular holes on a CGM(n,p), based on template algorithm 6.4.

Let $C = c_1, c_2, \ldots, c_n$ be a convex region containing n rectangular holes specified by a set $\mathcal{R}$= $\{R_1, R_2, \ldots, R_n\}$ of rectangles with their sides parallel to the

axes. The task at hand is to triangulate $C \setminus \mathcal{R}$. Let C' be the convex hull of the set $\mathcal{R}$ of rectangles. Triangulate $C \setminus C'$ using the algorithm discussed in Subsection 8.2.3. This is accomplished in $O(\frac{n}{p})$ computation time and $O(T_{Alltoall}(n,p))$ communication time,as stated in Theorem 8.8.

As in the template algorithm, add two rectangles R_0 and R_{n+1} to the given set of rectangles and solve the RV problem. Associate with each corner point of rectangle R_i an information packet containing its coordinates and two numbers u and v, as specified in the template algorithm. Sort the information packets, first on the u value and then on the y-coordinate. Determine the closest contours, and identify the special trapezoids and special monotone polygons which are then triangulated in parallel. By virtue of Lemma 5.5, Theorem 5.10 and Theorem 8.6, the following result is obtained.

Theorem 8.9. Triangulation of a convex region, of size n, containing a given set of n iso-oriented rectangular holes can be solved in $O(\frac{n \log n}{p})$ computational time and $O(\log p T_{Alltoall}(n,p))$ communication time, on a CGM(n,p). □

8.2.4 TRIANGULATING A CONVEX REGION WITH ORDERED SEGMENTS

This subsection briefly presents the result of porting template algorithm 6.5 to triangulate a convex region containing a set of ordered segments to a CGM(n,p). Consider a set of n well ordered segments $S = s_1, s_2, \ldots, s_n$ in the plane, stored $\frac{n}{p}$ per processor in the CGM(n,p). The vertices of C are also stored $\frac{n}{p}$ per processor in the CGM(n,p). The approach to this problem is similar to the triangulation in the presence of rectangular holes and the details are omitted. Thus, the following result is obtained.

Theorem 8.10. The problem of triangulating a convex region, of size n, containing a given set of n ordered segments $S = s_1, s_2, \ldots, s_n$ stored $\frac{n}{p}$ per processor on a CGM(n,p) is solved in $O(\frac{n \log n}{p})$ computational time and $O(\log p T_{Alltoall}(n,p))$ communication time. □

CHAPTER 9

IMPLEMENTATION NOTES AND CONCLUSIONS

9.1 EXPERIMENTAL RESULTS

To demonstrate the practical relevance of the several algorithms presented in this thesis, two fundamental algorithms discussed in this work were implemented. The problems chosen to be implemented are two of the basic algorithms used by the various visibility-related problems as very useful tools, namely the endpoint visibility algorithm (EV), and the algorithm for triangulating a special monotone polygon. These algorithms were implemented using MPI and timed on IBM-SP2. Note that, the code can be ported to several commercially available parallel computers, including shared memory computers, by just recompiling the code.

Before going into the implementation details, let us briefly discuss the IBM-SP2 architecture. It consists of RISC System/6000 processors connected via the SP2 *communication subsystem.* This subsystem is based upon a low latency, high bandwidth switching network called the *High-Performance Switch.* The primary goal of the SP2 communication subsystem is to be scalable, modular, and easily integrated. The communication network consists of bidirectional multistage interconnection net-

works [87]. Clearly, the SP2 can be classified as a Coarse-Grain Multicomputer (CGM), the coarse-grain computational model discussed in this thesis.

9.1.1 ENDPOINT VISIBILITY

Given a set of n ordered segments in the plane, template algorithm 3.1 can be applied to solve the EV problem. The implementation of the algorithm was straightforward and the program was timed on IBM-SP2 using 16 processors. A sequential algorithm for solving the EV problem was also implemented and run on a single processor of the SP2 and the speed up was determined.

The code was tested for several input sets assuming that the viewpoint is at $(\infty, 0)$. The input sets were assumed to be vertical segments and were sorted by their x-values to ensure that they are *well ordered* (see Chapter 3). The code was timed for segment sets where the y-values of the endpoints were generated using a random number generator. The size of the input sets varied from 2^{15} to 2^{20} segments. Since the timing of the program is dependent on certain geometric patterns in the set of input segments, several special cases were also timed.

Figure 9.1 shows the running times of the parallel EV algorithm on 16 processors of the SP2. The curve labeled Case 1 corresponds to input sets where the endpoints are generated using a random number generator. The randomness in the coordinates of the endpoints diminishes the possibility of having dense pockets during the last $\log p$ merge steps corresponding to the top $\log p$ levels of the tree $\mathcal{T}$. The curve labeled Case 3 corresponds to the input sets where the endpoints are in a geometric pattern guaranteeing that all the endpoints belong to dense pockets during each of the $\log p$ merge operations, forcing the algorithm to use dynamic load-balancing at every step. This results in an increase in the running time by a small quantity over Case 1 because of the extra overhead in processing dense pock-

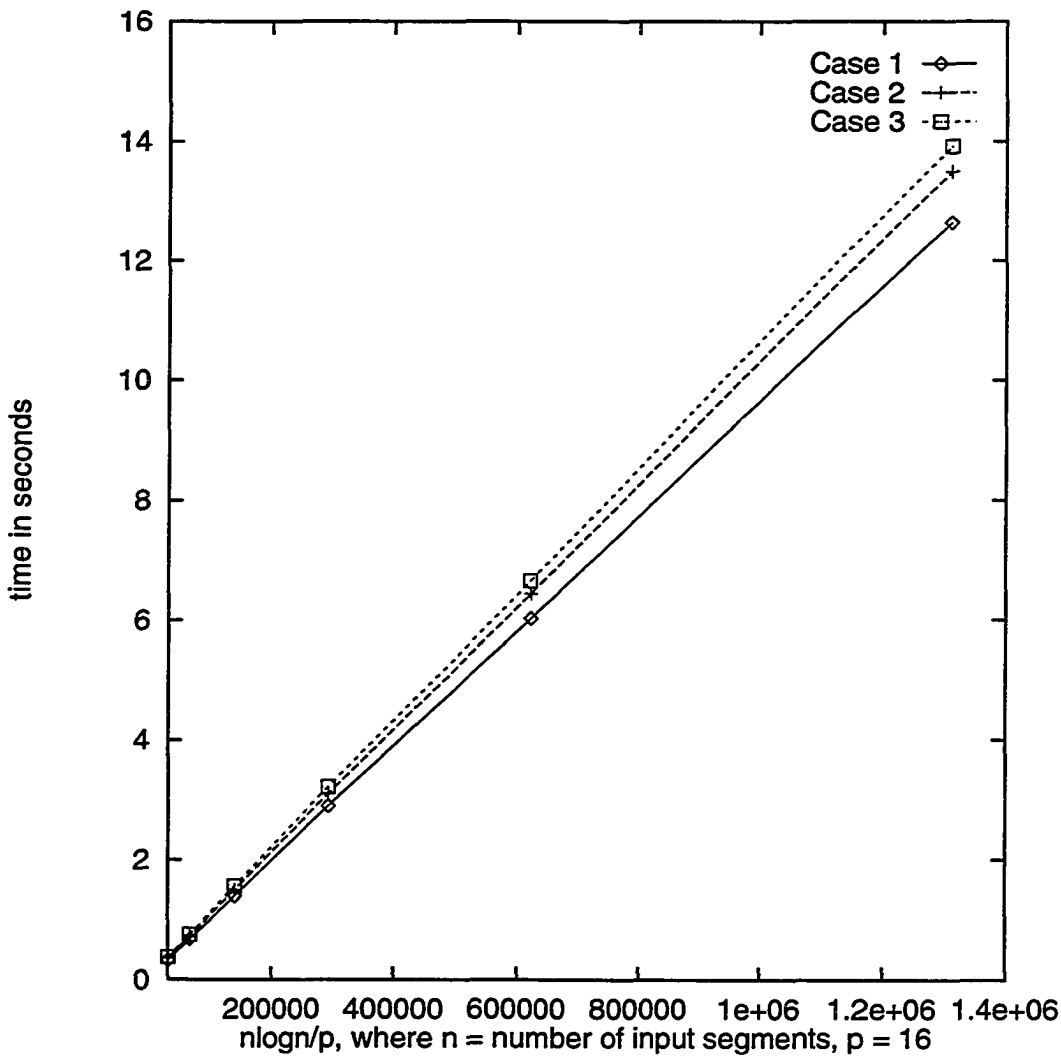

Figure 9.1: Running time of Stage 1 of EV

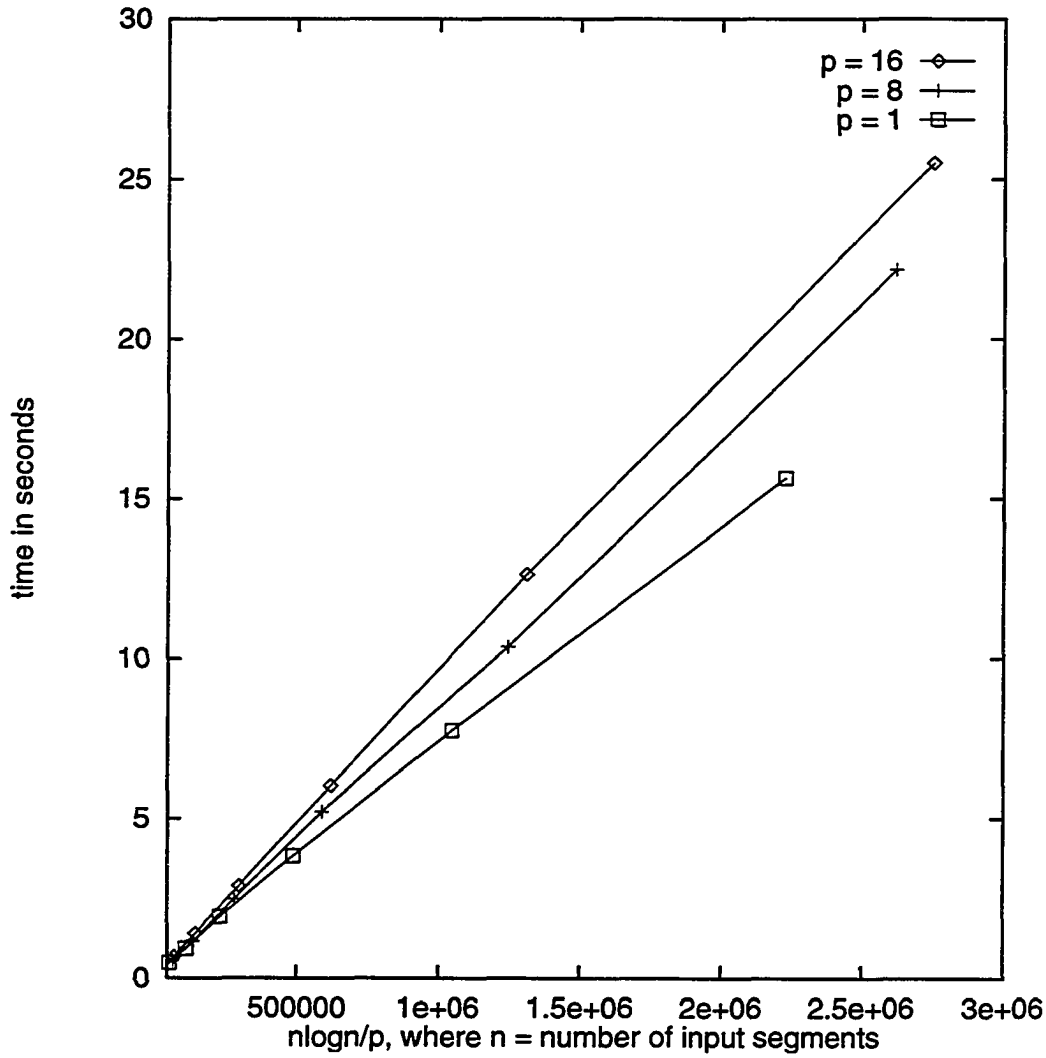

Figure 9.2: Comparison of sequential and parallel algorithms for EV

ets. The curve labeled Case 2 corresponds to the input sets where the endpoints are in a geometric pattern such that during each of the $\log p$ merge steps about half the endpoints belong to sparse pockets and rest of them belong to dense pockets. As expected, for Case 2 the running time is slightly less than Case 3 and slightly more than Case 1. Figure 9.2 compares the average running times of the sequential and parallel algorithms for randomly generated input sets. The speedup of the parallel algorithm over the sequential algorithm was found be about 6.2 for 8 processors and about 10.74 for 16 processors. It has also been observed that a single processor cannot handle input sizes of the order of 2^{20} as it runs out of memory for that large a input size.

9.1.2 TRIANGULATION OF A SPECIAL MONOTONE POLYGON

The problem of triangulating a special monotone polygon, where the base edge is assumed to be parallel to the x-axis, has been implemented based on the template algorithm 6.1. As in the case of the EV algorithm, the performance of the parallel algorithm, running on 16 processors of IBM-SP2, was compared against a $O(n)$ time sequential implementation for the triangulation problem running on a single processor of the SP2. The program was timed for special monotone polygons whose vertices generated using a random number generator. The number of vertices in the input polygons varied from 2^{16} to 2^{21}. Again, since the timing of the algorithm is dependent on the geometrical patterns within the set of input vertices, several special cases were timed.

In Figure 9.3, the curve labeled Case 1 corresponds to the randomly generated vertex sets, and the low run time can be explained because of the fact the randomness increases the likelihood of a vertex finding its match (refer to template

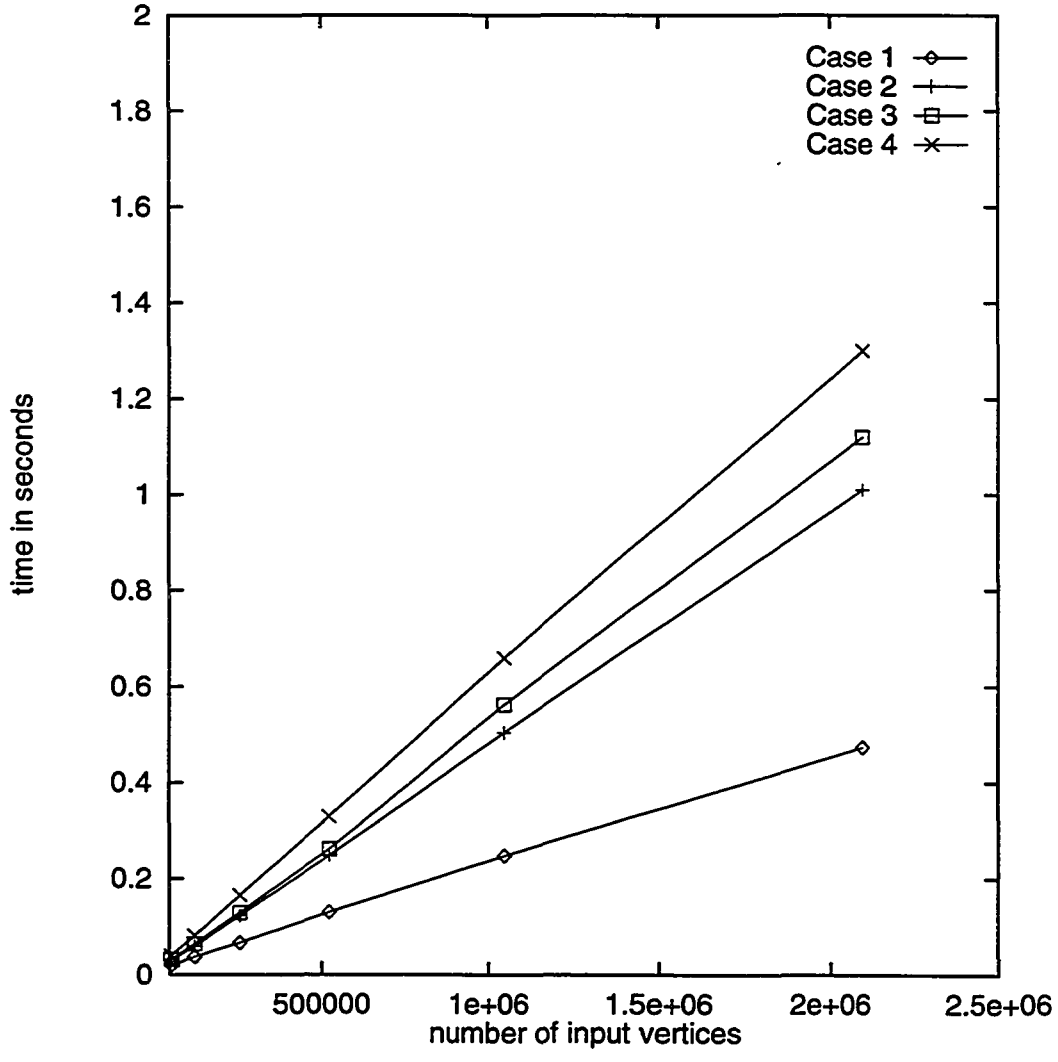

Figure 9.3: Running times of triangulation of special monotone polygon

algorithm 6.1) within the same processor and this corresponds to the situation where there are no vertices that belong to dense pockets and there are O(N) (where $N<<n$) vertices belonging to sparse pockets. The curve labeled Case 2 corresponds to the arrangement of the vertices of the special monotone polygon, where the resulting instance of ANLV in Step 2 of the template algorithm is such that O(n) vertices belong to sparse pockets. As expected the running time for Case 2 is slightly higher than that of Case 1 because of the fact that $O(n)$ vertices move across the 16 processors to determine their solutions. The curve labeled Case 4 corresponds to the case where O(n) vertices belong to dense pockets, thus increasing the running time because of the extra overhead involved in processing dense pockets. The curve labeled Case 3 corresponds to the case where O($\frac{n}{2}$) vertices belong to sparse pockets and O($\frac{n}{2}$) vertices belong to dense pockets. The comparison of the average running times of the parallel algorithm and the sequential algorithm is given in Figure 9.4 and the speed up is found to be about 14.2.

9.2 CONCLUSIONS

As stated in the introduction, the design of optimal parallel algorithms poses two major challenges to an algorithm designer. For a given problem, the first challenge is to design optimal algorithm for the particular model of computation under consideration. The second and the more difficult challenge to meet is to develop a template solution that can be ported to diverse computational platforms to give an optimal solution on that platform.

In this thesis, the class of visibility-related problems was studied with the intent of investigating the process of developing architecture independent techniques that serve as template algorithms across various parallel computational models. As

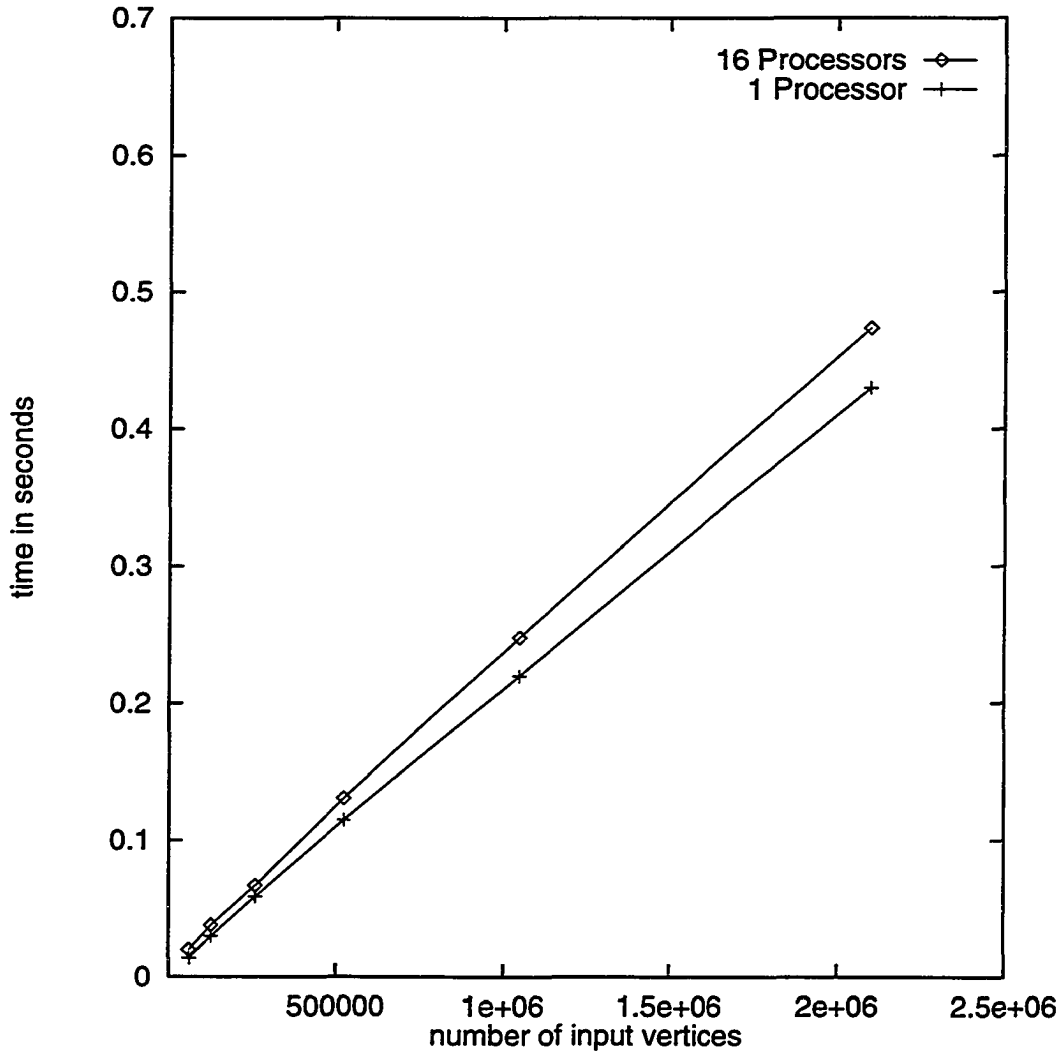

Figure 9.4: Comparison of sequential and parallel algorithms for monotone polygon triangulation

stated in the introduction, these problems find applications in seemingly unrelated and diverse fields such as computer graphics, scene analysis, robotics and VLSI design. Considering the fact that the existing solutions to various members of this class of problems do not exploit the common threads that run between them, this thesis provided an unified approach to these problems by identifying the commonality between them.

The problems investigated in this work can be broadly classified into object visibility and closely related triangulation problems. This thesis has studied these problems in great detail and to a significant extent met the challenges of developing optimal solutions to the problems at hand on various computational models, which in fact are the instantiations of template algorithms designed for an abstract computational model.

First, a detailed discussion on the class of object visibility problems including segment/endpoint visibility, disk visibility, rectangle visibility, dominance graph problems, was presented. Template algorithms for each of these problems were discussed on the abstract computational model and it was observed that the solutions to the problems are inter-dependent and revealed a number of aspects that are common to visibility relations among general objects in the plane. The segment/endpoint visibility problem for a set of ordered segments has been discovered as a powerful tool which makes the solutions to the rest of the problems very simple. In addition to various object visibility problems discussed here, others like determining the visibility pairs among a given set of segments, ANLV, and several constrained triangulations use this solution to obtain optimal solutions.

Next, various tools required to port the template algorithms for various object visibility problems to the fine-grain enhanced mesh connected computers,

namely the meshes with multiple broadcasting and reconfigurable meshes were designed. The template algorithms when ported to the meshes with multiple broadcasting resulted in time-optimal solutions to the object visibility problems as shown by the various lower-bound arguments presented. Not surprisingly, the same algorithms when applied to the reconfigurable meshes resulted in O(1) time solutions to the various problems. Following this, a detailed discussion on the various tools developed on the coarse-grain multicomputers and their application to the template algorithms for the object visibility problems to provide computationally optimal algorithms was presented.

The class of triangulation problems, which is closely related to object visibility, is the other class of interesting problems that received focus in this thesis. Again, the segment/endpoint visibility problem for ordered segments is a very important important tool for the various template algorithms developed. The concept of special monotone polygons and their triangulation emerged as another fundamental result which can be used in the template algorithms to various constrained triangulation problems.

Next, the development of required tools to apply the template algorithms to enchanced mesh connected computers was discussed, followed by the discussion on porting the template algorithms to these platforms. Once again this resulted in optimal algorithms on meshes with multiple broadcasting and O(1) time algorithms on reconfigurable meshes. Next, a detailed discussion on the additions to the rich collection of tools developed for the coarse-grain multicomputers was presented. The tools developed were than applied to the template algorithms to give computationally optimal solutions to various triangulations on the CGM.

As already mentioned a byproduct of the exercise of porting the template

algorithms to these diverse computational models is a rich collection of tools that can be reused in other contexts. The powerful tools that were developed for the enhanced meshes include the compaction algorithm, the EV algorithm, and the triangulation of special monotone polygons. For the coarse-grain multicomputers, a very vast collection of tools has been designed. These include the algorithms to merge two sorted sequences, to sort a collection of items from a totally ordered universe, to determine the all nearest larger values for a given sequence of items, to solve the segment visibility problem for a set of well ordered segments, to merge two convex hulls and to determine the convex hull for a given set of points in the plane.

To demonstrate the practical relevance of the various algorithms discussed in this work, the two most fundamental algorithms for segment visibility and triangulation of special monotone polygons were implemented using MPI, and their running times analyzed on an IBM-SP2. It has been observed that the parallel algorithms provide significant speedup over their sequential counterparts. The code developed can be readily ported to various commercially available parallel machines including shared memory machines.

This work opens avenue to several open problems. It would be of interest to see what other visibility related problems can be solved using the various concepts and template algorithms designed in this thesis. In particular, the segment visibility problem, involving a collection of ordered segments, has been discovered as the stepping stone for almost all the other algorithms discussed in this work. It seems to have a lot of potential that could be exploited in the context of several other geometric problems.

BIBLIOGRAPHY

[1] A. Aggarwal, Optimal bounds for finding maximum on array of processors with k global buses, *IEEE Transactions on Computers*, C-35, 1986, 62–64.

[2] S.G.Akl, and K.S. Lyons, *Parallel Computational Geometry*, Prentice Hall, Englewood Cliffs, NJ 07632.

[3] S. G. Akl, *The design and analysis of parallel algorithms*, Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[4] T. Asano, and H. Umeo, Systolic algorithms for computing the visibility polygon for a polygonal region with holes,*Transactions of the IECE Japan E-68*, V ol 9, 1985, 557-559.

[5] M.J. Atallah, and M.T. Goodrich, Efficient plane sweeping in parallel, *Proceedings of the Second Annual ACM Symposium on Computational Geometry,* Yorktown Heights, New York, June 1986, 216-225.

[6] M. J. Atallah, R. Cole and M. T. Goodrich, Cascading divide-and-conquer: A technique for designing parallel algorithms, *SIAM Journal on Computing*, 18, 1989, 499–532.

[7] M. J. Atallah, and D. Z. Chen, An optimal parallel algorithm for the visibility of a simple polygon from a point, *Proceedings of the 5th Annual ACM Symposium*

on Computational Geometry, Saarbruchen, Germany, June 1989, 114–123.

[8] M.J. Atallah, J.J. Tsay, On the parallel-decomposability of geometric problems, *Proceedings of the Fifth Annual Symposium on Computational Geometry,* Saarbruchen, Germany, June 1989, 104-113.

[9] M.J. Atallah, F. Dehne and S.E. Hambrusch. *A coarse-grained, architecture-independent approach for connected component labeling,* Manuscript, Purdue University, 1993.

[10] A. Bar-Noy and D. Peleg, Square meshes are not always optimal, *IEEE Transactions on Computers,* 40, 1991, 196–204.

[11] G.H.Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick, and R.A. Stokes, The ILLIAC IV computer, *IEEE Transactions on Computers,* C-17, 1968, 746-757.

[12] K. E. Batcher, Design of Massively Parallel Processor, *IEEE Transactions on Computers,* C-29, 1980, 836–840.

[13] Y. Ben-Asher, D. Peleg, R. Ramaswani, and A. Schuster, The power of reconfiguration, *Journal of Parallel and Distributed Computing,* 13, 1991, 139–153.

[14] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin, Highly parallelizable problems, *Proceedings of Annual Symposium of Theory of Computing,* 1989, 770–780.

[15] D. Bhagavathi, S. Olariu, J. L. Schwing, and J. Zhang, Convex polygon problems on meshes with multiple broadcasting, *Parallel Processing Letters,* 2, 1992, 249–256.

[16] D. Bhagavathi, H. Gurla, R. Lin, S. Olariu, J. L. Schwing and J. Zhang, Time- and VLSI-optimal sorting on meshes with multiple broadcasting, *Proceedings of International Conference on Parallel Processing*, St-Charles, Illinois, August 1993, III, 196–201.

[17] D. Bhagavathi, S. Olariu, W. Shen, and L. Wilson, A unifying look at semigroup computations on meshes with multiple broadcasting, *Proceedings of Parallel Architectures and Languages Europe*, München, Germany, June 1993, LNCS 694, 561–569.

[18] D. Bhagavathi, V. Bokka, H. Gurla, S. Olariu, J. L. Schwing, and Zhang, Square meshes are not optimal for convex hull computation, *Proceedings of International Conference on Parallel Processing*, St-Charles, Illinois, August 1993, III, 307–311.

[19] D. Bhagavathi, P. J. Looges, S. Olariu, J. L. Schwing, and J. Zhang, A fast selection algorithm on meshes with multiple broadcasting, *IEEE Transactions on Parallel and Distributed Systems*, 5, 1994, 772–778.

[20] D. Bhagavathi, H. Gurla, S. Olariu, J. L. Schwing, and Zhang, Time-optimal parallel algorithms for Dominance and Visibility graphs, *Journal of VLSI design*, Vol 4, 1996, 33–40.

[21] D. Bhagavathi, S. Olariu, W. Shen, and L. Wilson, A time-optimal multiple search algorithm on enhanced meshes, with applications, *Journal of Parallel and Distributed Computing*, to appear.

[22] S. H. Bokhari, Finding maximum on an array processor with a global bus, *IEEE Transactions on Computers*, 33, 1984, 133–139.

[23] W. Briggs, *Multigrid tutorial*, SIAM Press, Philadelphia, 1987.

[24] I.-W. Chan, and D. K. Friesen, An optimal parallel algorithm for the vertical segment visibility reporting problem, *Proceedings of ICCI'91*, Lecture Notes in Computer Science Vol. 497, Springer-Verlag, 1991, 323–334.

[25] B. Chazelle, Computational Geometry on a systolic chip, *IEEE Transactions on Computers*, Vol. C-33, No. 9, September 1984, 774-785.

[26] Y. C. Chen, W. T. Chen, G.-H. Chen and J. P. Sheu, Designing efficient parallel algorithms on mesh connected computers with multiple broadcasting, *IEEE Transactions on Parallel and Distributed Systems*, 1, 1990.

[27] A.L.Chow, Parallel algorithms for geometric problems, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1980.

[28] C.F. Codd, *Cellular Automata*, Academic Press, Newyork, 1968.

[29] S. A. Cook, C. Dwork, and R. Reischuk, Upper and lower time bounds for parallel random access machines without simultaneous writes, *SIAM Journal on Computing*, 15, 1986, 87–97.

[30] David L. Waltz, "Application of the Connection Machine", *Computer*, January 1987, 85–97.

[31] F. Dehne, Solving visibility and separability problems on a mesh-of-processors, *The Visual Computer*, Vol 3., 1988, 356-370.

[32] F. Dehne, A. Fabri and A. Rau-Chaplin, Scalable Parallel Geometric Algorithms for Coarse Grained Multicomputers, *Proceedings of the ACM Symposium on Computational Geometry*, 1993.

[33] Oliver Devillers, and Andreas Fabri, Scalable algorithms for bichromatic line segment intersection problems on Coarse Grained Multicomputers, 1993.

[34] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*, Wiley and Sons, New York, 1973.

[35] C.R. Dyer and A. Rosenfeld, Parallel image processing by memory augumented cellular automata, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol PAMI-3, 1981, 29–41.

[36] H. ElGindy, An optimal speed-up parallel algorithm for triangulating simplicial point sets in space, *International Journal of Parallel Programming*, Vol 15, No. 5, 1986, 389-398.

[37] J. Encarnacao and E. G. Schlechtendahl, *Computer Aided Design*, Springer-Verlag, Berlin, 1990.

[38] D. G. Feitelson, *Optical Computing*, MIT Press, 1988.

[39] L. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer graphics, principles and practice*, Second Edition, Addison-Wesley, Reading, MA, 1990.

[40] H. Freeman and G. Pieroni, Eds., *Computer architecture for spatially distributed data*, Springer-Verlag, Heidelberg, Berlin, 1985.

[41] Gabriela Hristescu, Parallel Triangulation of a Set of Points for Coarse Grained Multicomputers, Rutgers University, Technical Report DCS-TR-313, September 1994.

[42] *Grand Challenges: High Performance Computing and Communication*, a report by the Committee on Physical, Mathematical, and Engineering Sciences, to supplement the U.S. President's Fiscal Year 1992 Budget.

[43] D. Hillis, *The Connection Machine*, MIT press. Cambridge, Mass., 1985.

[44] J. JáJá, *An introduction to parallel algorithms,* Addison-Wesley, Reading, MA, 1992.

[45] J. Jang and V. Prasanna, Parallel geometric problems on the reconfigurable mesh, *Proceedings of the International Conference of Parallel Processing,* St. Charles, Illinois, III, 1992, 127–130.

[46] J. Jang and V. K. Prasanna, An optimal sorting algorithm on reconfigurable meshes, *Proceedings of International Parallel Processing Symposium*, 1992, 130–137.

[47] V. Prasanna Kumar and C. S. Raghavendra, Array processor with multiple broadcasting, *Journal of Parallel and Distributed Computing,* 2, 1987, 173–190.

[48] V. Prasanna Kumar and D. I. Reisis, Image computations on meshes with multiple broadcast, *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 11, 1989, 1194–1201.

[49] J.-P. Laumond, Obstacle growing in a non-polygonal world, *Information Processing Letters*, 25, 1987, 41–50.

[50] H. Li and M. Maresca, Polymorphic-torus network, *IEEE Transactions on Computers,* 38, 1989, 1345–1351.

[51] R. Lin, S. Olariu, J. L. Schwing, and J. Zhang, Sorting in O(1) time on a reconfigurable mesh of size $N \times N$, *Parallel Computing: From Theory to Sound Practice, Proceedings of EWPC'92*, Plenary Address, IOS Press, Amsterdam, 1992, 16–27.

[52] R. Lin, S. Olariu, J. L. Schwing, and J. Zhang, Simulating enhanced meshes, with applications, *Parallel Processing Letters*, 3, 1993, 59–70.

[53] E. Lodi and L. Pagli, A VLSI solution to the vertical segment visibility problem, *IEEE Transactions on Computers*, 35, 1986, 923–928.

[54] T. Lozano-Perez, Spatial planning: a configurational space approach, *IEEE Transactions on Computers*, 32, 1983, 108–119.

[55] F. Luccio, S. Mazzone, and C. K. Wong, A note on visibility graphs, *Discrete Mathematics*, 64, 1987, 209–219.

[56] M. Lu and P. Varman, Solving geometric proximity problems on mesh-connected computers, *Proceedings of 1985 Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, 248–255.

[57] P.D. MacKenzie, and Q.F. Stout, Asymptotically efficient hypercube algorithms for computational geometry, *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation*, College Park, Maryland, October 1990, 8–11.

[58] A. A. Malik, An efficient algorithm for generation of constraint graph for compaction, *Proceedings of International Conference on CAD*, 1987, 130–133.

[59] M. Maresca and H. Li, Connection autonomy and SIMD computers: a VLSI implementation, *Journal of Parallel and Distributed Computing*, 7, 1989, 302–320.

[60] M. Maresca, H. Li, and P. Baglietto, Hardware support for fast reconfigurability in processor arrays, *Proceedings of International Conference on Parallel Processing*, St. Charles, Illinois, I, 1993, 282–289.

[61] C. A. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA, 1979.

[62] E. Merks, An optimal parallel algorithm for triangulating a set of points in the plane, *International Journal of Parallel Programming,* Vol. 15, No. 5 , 1986, 399-411.

[63] R. Miller, and Q.F. Stout, Geometric algorithms for digitized pictures on a mesh-connected computer, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7, 1985, 216–228.

[64] R. Miller, and Q.F. Stout, Mesh computer algorithms for computational geometry, Technical Report No. 86-18, State University of New York at Buffalo, Department of Computer Science, July 1986.

[65] R. Miller, and Q.F. Stout, Mesh Computer Algorithms for Line Segments and Simple Polygons.

[66] R. Miller, V. K. P. Kumar, D. Reisis, and Q. F. Stout, Parallel Computations on Reconfigurable Meshes, *IEEE Transactions on Computers*, 42, 1993, 678–692.

[67] Message Passing Interface Forum. Document for a standard message-passing interface standard. Technical Report No. CS-93-214 (revised), University of Tennesse, April 1994.

[68] M. Nigam and S. Sahni, Sorting n numbers on $n \times n$ reconfigurable mesh with buses, CIS, University of Florida, Technical Report TR-92-04, 1992.

[69] M. Nigam and S. Sahni, Constant time computational geometry on reconfigurable meshes with buses, CIS, University of Florida, Technical Report TR-92-35, 1992.

[70] S. Olariu, J. L. Schwing, and J. Zhang, Fundamental Data Movement for Reconfigurable Meshes, *Proceedings of the International Phoenix Conference on Computers and Communications*, Scottsdale, Arizona, April 1992, 472–480.

[71] S. Olariu, J. L. Schwing, and J. Zhang, Time-Optimal Convex Hull Algorithms on Enhanced Meshes, *BIT*, 33, 1993, 396–410.

[72] S. Olariu, J. L. Schwing, and J. Zhang, Optimal convex hull algorithms on enhanced meshes, *BIT*, 33, 1993, 396–410.

[73] S. Olariu and I. Stojmenović, Time-optimal proximity algorithms on meshes with multiple broadcasting, *Proceedings of 8th International Parallel Processing Symposium*, Cancun, Mexico, April 1994, 94–101.

[74] S. Olariu and I. Stojmenović, Time-optimal nearest-neighbor computations on enhanced meshes, *Proceedings of PARLE*, Patras, Greece, July 1994.

[75] D. Parkinson, D. J. Hunt, and K. S. MacQueen, The AMT DAP 500, 33^{rd} *IEEE Computer Society International Conference*, 1988, 196–199.

[76] T. Pavlidis, *Computer Graphics*, Computer Science Press, Potomac, MD, 1978.

[77] B. T. Preas and M. J. Lorenzetti (Eds.), *Physical Design Automation of VLSI Systems*, Benjamin/Cummings, Menlo Park, 1988.

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

[79] J.H. Reif and S. Sen, Randomized algorithms for binary search and load balancing on fixed interconnection networks with geometric applications, *Proceedings*

of the Second ACM Symposium on Parallel Algorithms and Architectures, Crete, July 1990, 327–337.

[80] F. Rosenblatt, *Principles of Neurodynamics*, Spartan Books, New York, 1962.

[81] J. Rothstein, Bus automata, brains, and mental models, *IEEE Transactions on Systems Man Cybernetics* 18 (1988) 522–531.

[82] M. Schlag, F. Luccio, P. Maestrini, D. T. Lee, and C. K. Wong, A visibility problem in VLSI layout compaction, in F. P. Preparata (Ed.), *Advances in Computing Research*, Vol. 2, 1985, 259–282.

[83] A. Schuster and Y. Ben-Asher, Algorithms and optic implementation for reconfigurable networks, *Proceedings of the 5th Jerusalem Conference on Information Technology*, October 1990.

[84] D. B. Shu, L. W. Chow, and J. G. Nash, A content addressable, bit serial associate processor, *Proceedings of the IEEE Workshop on VLSI Signal Processing*, Monterey CA, November 1988.

[85] D. B. Shu and J. G. Nash, The gated interconnection network for dynamic programming, S. K. Tewsburg *et al.* (Eds.), Concurrent Computations, Plenum Publishing, 1988.

[86] H. S. Stone, *High-Performance Computer Architecture*, Second Edition, Addison-Wesley, Reading, MA, 1990.

[87] C. B. Stunkel, G. D. Shea, B. Abali, M. Atkins, C. A. Bender, Grice D. G., P. H. Hochschild, D. J. Joseph, R. A. Swetz, R. F. Stucke, M. Tsao, P. R. Varker, The SP2 Communication Subsystem, *URL:http://ibm.tc.cornell.edu/ibm/pps/doc/css/css.ps.*

[88] G. T. Toussaint, *Computational Geometry*, Elsevier Science Publishers, North-Holland, Amsterdam, 1985.

[89] D. Vernon, *Machine vision, automated visual inspection and robot vision,* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[90] B. F. Wang and G. H. Chen, Sorting and computing convex hulls on processor arrays with reconfigurable bus systems, *Information Sciences*, to appear.

[91] C. A. Wang and Y. H. Tsin, An $O(\log n)$ time parallel algorithm for triangulating a set of points in the plane, *Information Processing Letters*, 25, 1988, 55–60.

VITA

Himabindu Gurla was born in Hyderabad, India on June 19, 1970. She received her Bachelor of Engineering in Computer Science from Osmania University, India, in July 1991. She worked as a Systems Engineer for the Research and Development Division of Computer Maintenance Corporation, India, from August 1991 to August 1992. In September 1992, she started working on her Ph.D Degree in Computer Science at Old Dominion University, Virginia. Since August 1995, she has been working for the Business Computing Services Division of AT&T Bell Laboratories, NJ.