National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

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

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

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

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

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

AVIS

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

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

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

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, tests publiés, etc.) ne sont pas microfilmés.

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

**VISIBILITY FOR BINARY IMAGES ON A
MESH-CONNECTED COMPUTER.**

by

Anne-Lise Hassenklover

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

April 29, 1988

© copyright

1988 Anne-Lise Hassenklover

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced, without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-40626-7

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

Visibility for Binary Images on a Mesh-Connected Computer

submitted by Anne-Lise Hassenklover, B.C.S.

in partial fulfilment of the requirements for
the degree of Master of Computer Science

Thesis Supervisor

Chairman, School of Computer Science

Carleton University

April 29, 1988

ABSTRACT

In this thesis, the Mesh-Connected Computer (MCC) systolic architecture is described, for which new algorithms solving visibility problems on digitized images are designed. MCC algorithms frequently make use of standard operations; these are presented in the thesis with an outline of their implementation. A survey of MCC algorithms, classified under computational geometry, graph theory, and image processing is given that emphasizes computational geometry problems. The survey illustrates the use of the standard MCC operations, and provides examples in systolic algorithm design. New MCC algorithms for visibility related problems on digitized objects are presented to compute parallel, perspective, and rectangular visibility; to solve separability problems, and to determine the configuration space of a set of obstacles. Extensions to the parallel and perspective visibility algorithms that solve related problems are presented as well. The implementation of many proposed MCC algorithms is difficult, due to the lack of support in existing systolic languages for standard operations and techniques such as recursion. Consequently, a high-level language to be employed in the implementation of MCC algorithms is defined.

ACKNOWLEDGEMENTS

I would like to acknowledge Dr. Jörg-Rüdiger Sack, Dr. Frank Dehne, and Dr. Nicola Santoro for their ideas on many algorithms of this thesis. I especially thank Dr. Sack and Dr. Dehne for their effort and advice on the style of the thesis, and for suggesting parallel computing as a research area, a topic in which I am now very interested.

TABLE OF CONTENTS

CHAPTER I: Introduction	1
1.1. Systolic Architectures	2
1.2. The Mesh-Connected Computer	2
1.3. The Systolic Screen	4
1.4. Other Systolic Architectures	5
1.5. Thesis Overview	7
CHAPTER II: MCC Standard Operations and Surveyed Algorithms	9
2.1. Standard Operations and Techniques on an MCC	9
2.1.1. Basic MCC Operations	10
2.1.2. Data Compression	12
2.1.3. Recursion	12
2.1.4. Sorting as a Fundamental Operation	14
2.1.5. The RAR and RAW Operations	18
2.2. A Survey of Parallel Algorithms	21
2.2.1. Computational Geometry Definitions	21
2.2.2. Computational Geometry in Object Space	22
2.2.3. Computational Geometry in Image Space	28
2.2.4. Image Processing Problems	33
2.2.5. Graph Problems	34
CHAPTER III: Visibility Problems	36
3.1. Object Space Visibility	37
3.2. Digitized Parallel Visibility	40
3.2.2. Systolic Parallel Visibility	41
3.2.3. Mesh Implementation of Parallel Visibility	44
3.2.4. Three-dimensional Extensions to Parallel Visibility	47
3.3. Unidirectional Separability	48
3.3.1. Unidirectional Separability Using a Movability Graph	49
3.3.2. Separability Using the Systolic Screen	51
3.3.3. Solutions to the Separability Problems	55

3.4. Digitized Perspective Visibility	56
3.4.1. Mesh Implementation of Perspective Visibility	63
3.4.2. Variations on Digitized Perspective Visibility	65
3.5. Rectangular Visibility	66
3.5.1. Systolic Digitized Rectangular Visibility	67
3.6. Configuration Space	69
3.6.1. An Encoding for the Robot	70
3.6.2. Pipe-lining the Encoded Robot	73
3.6.3. Digitized Configuration Space	74
3.6.4. Shortest Paths and the Digitized Configuration Space	77
3.7. Conclusions on Visibility	78
CHAPTER IV: Mesh-Connected Computer High-Level Language	80
4.1. Previous Systolic Languages	81
4.2. Overview of Mesh Control	82
4.3. PE Architecture and ML Requirements	83
4.4. The MCC Languages	85
4.4.1. Notation	85
4.4.2. Mesh Instruction Language	86
4.4.3. Mesh Controller Language	87
4.4.4. Mesh Language	91
4.5. An Example Using ML	101
4.6. Conclusions and Improvements	106
REFERENCES	109

LIST OF FIGURES

FIGURE	DESCRIPTION	PAGE
1.1.	An $n \times n$ Mesh-Connected Computer	3
1.2.	Image on a MCC	4
1.3.	A systolic screen	4
1.4.	A 3-cube	5
1.5.	A 32 PE CCC	5
1.6.	A 4x4 Mesh of Trees	6
1.7.	A 4x4 Pyramid	6
2.1.	Details of the row broadcast operation	11
2.2.	Recursively merging subsquares on an 8x8 MCC	13
2.3.	Divide-and-conquer splitting on an 8x8 MCC	14
2.4.	Sorted orderings on a Mesh-Connected Computer	15
2.5.	Odd-even transposition sort of eight elements	16
2.6.	The merge step to sort a $k \times k$ mesh into snake-like order	17
2.7.	A numerical sorting example	18
2.8.	Quadrant locations	24
2.9.	Tangent points and rays	24
2.10.	The smallest enclosing rectangle for edge e	26
2.11.	The border elements of a $k \times k$ square	28
3.1.	The visibility polygon of a simple polygon with holes	37
3.2.	Computation of parallel visibility and the visibility hull	39
3.3.	A strip S_k , and its associated interval I_k of width w	42
3.4.	Update of interval I_k when scanning strip S_k	42
3.5.	Definition of width w of a strip	43
3.6.	Subdivision of an interval	44
3.7.	S_k intersects the top left-hand corner of a pixel	46
3.8.	S_k touches the bottom right-hand corner of a pixel	46
3.9.	Three-dimensional parallel visibility strips	47
3.10.	A set of polygons	49

3.11.	The movability graph for Figure 3.10 in direction d	50
3.12.	The cyclic movability graph of Figure 3.10 in direction d ...	51
3.13.	The black PE sees $n-1$ different objects in the strip	52
3.14.	The interval used in computing separability	53
3.15.	The update of I_x of the PE labeled 'a'	54
3.16.	Perspective visibility from a point p to a simple polygon '...	56
3.17.	The layers from PE(3,4)	57
3.18.	Perspective visibility wedge PVW(q)	57
3.19.	Perspective visibility on the systolic screen	58
3.20.	The projections of PVWs onto a line l	59
3.21.	Intersection of rays from p to pixels of the MCC	60
3.22.	The number of PVWs of a layer that intersect a pixel	61
3.23.	The growth and exchange steps of PVWs for a layer	62
3.24.	Rectangular visibility, and p 's visibility interval	66
3.25.	Digitized rectangular visibility	67
3.26.	The update of a pixel p 's record as it is shifted right	67
3.27.	The configuration space of polygonal figures	69
3.28.	Encoding the left side of a rectilinear robot	71
3.29.	Encoding special types of robots	72
3.30.	Pipe-lining and waves of an 8x8 MCC	73
4.1.	Overview of mesh control	83
4.2.	PE registers and processor	85
4.3.	MIL basic instructions	87
4.4.	Syntax of MCL	88
4.5.	ML configuration and program format	92
4.6.	ML program syntax	93
4.7.	Recursively finding maximal points	102

LIST OF EXAMPLES

EXAMPLE	DESCRIPTION	PAGE
2.1.	A RAW example on 8 processors	19

1. INTRODUCTION.

Many of the sequential algorithms that exist in computer science today are optimal in the big-oh sense ([1] p. 17). Consequently, any improvement on the execution time of these algorithms can only be made by using parallel processing. This thesis concentrates on parallel solutions to computational geometry problems. These algorithms are written for a particular systolic architecture referred to as the Mesh-Connected Computer (MCC). Optimal sequential solutions to many computational geometry problems exist [31]. A survey of parallel solutions to selected computational geometry problems is presented in this thesis to illustrate the improved execution time.

Sequential solutions exist to compute hidden-line elimination in graphics [65], or to plan a path for a robot through a set of obstacles [34]. These solutions often compute visibility from a point (perspective visibility) or from a given direction (parallel visibility) to a set of obstacles. New parallel solutions to the computation of perspective and parallel visibility, and other visibility related problems for a set of digitized obstacles are presented for the MCC. Further details of the problems studied in this thesis may be found in the overview of Section 1.5.

Many high-level languages exist, such as Pascal, Fortran or C, to allow easy implementation of an algorithm on a sequential machine. Many of the languages designed for parallel architectures [28,30,58,59] are still quite low-level in that they only provide low-level parallel 'send' operations, or operations specific to the given hardware. This makes it difficult to program in a high-level manner, and makes these algorithms hardware dependent. The last chapter of this thesis addresses the implementation of MCC algorithms by designing a high-level language for that systolic architecture.

Chapter one introduces systolic architectures, which are a class of parallel architectures in which the processors are highly synchronized. Section 1.1 defines a systolic architecture more formally, and Section 1.2 describes the Mesh-Connected Computer type, for which algorithms of this thesis are presented or designed. The new algorithms designed in this thesis operate on digitized images, hence, a model in which these digitized objects may be represented on the MCC, referred to as a systolic screen, is introduced in Section 1.3. Other systolic architectures exist, and those most frequently used are introduced in Section 1.4. Some problems solved on these architectures are mentioned along with those surveyed for the MCC in Chapter 2. Finally, Section 1.5 presents an overview discussing the contents of the remaining chapters of this thesis.

1.1. SYSTOLIC ARCHITECTURES.

Systolic architectures have been studied by Kung [26] who provides a general methodology for mapping high-level computations into parallel hardware structures. The idea behind a systolic architecture is to allow data to be processed in parallel by either having each processor perform a step on the data before passing it to another processor (pipelining) or by having the processors perform a sequence of steps on the data (parallel execution) or a combination of both. The systolic architecture is used to speed up throughput in the pipelining case, or to speed up execution in the parallel case. A formal definition of a systolic architecture is presented next, examples of which are found in subsequent sections.

A *systolic architecture* is a set of processing elements, arranged in a (regular) bounded degree interconnection network [26]. The *interconnection network* is defined by a graph $G(V,E)$, where V represents the set of PEs, and E represents a set of communication links (usually bi-directional) between the PEs in V . This interconnection network G is *regular* if the degree of each node V of G is the same. Each *processing element* (PE) of V is a simple processor containing an arithmetic/logic unit, a set of registers, and a set of communication links that link it to other processing elements. A processing element can perform certain arithmetic and boolean operations in unit time, contains a fixed amount of local memory (as registers), and a unique identifier (ID). A *neighboring* PE of a processor x is one that is connected by a communication link to x . In a single *routing step* a PE can transmit a fixed-size data element to a neighboring PE in unit time [10]. The execution of arithmetic/boolean or send instructions by these processors is highly synchronized. Some systolic architectures, such as the cube-connected cycles architecture presented in Section 1.4, allow the degree of certain nodes to be different. The overall interconnection network of such architectures, though, are usually defined in a systematic fashion that allows them to be represented by regular, or equivalent but smaller, parts.

1.2. THE MESH-CONNECTED COMPUTER.

One type of interconnection network is a two-dimensional array of processors, referred to as the *mesh* or *Mesh-Connected Computer* (MCC). In this configuration, the PEs form an $n \times m$ matrix of n rows and m columns, with each PE connected to its four up, down, left and right neighbors if they exist. The PEs are referred to by their row and column index. For example PE (1,2) is the second processor from the left in the first row, and in general we have PE (i,j) for $i=1..n, j=1..m$. An *sxt submesh* on an $n \times m$ MCC is a subarray of neighboring processors $P(i,j)$, of k rows by l columns ($1 \leq s, t \leq n$), such that $a < i \leq a+s$ and $b < j \leq b+t$ where $0 \leq a \leq n-s$ and $0 \leq b \leq m-t$. An example of an $n \times n$ MCC is shown in Figure

1.1 where the boxes represent PEs and the lines represent communication links.

All PEs execute the arithmetic, boolean or send instructions in a synchronous fashion, i.e. in lock step. This is usually accomplished by a host computer that controls the mesh by broadcasting the next instruction to be executed to each PE. The communication links on the boundary PEs, not shown in subsequent MCC figures, are used by the host to load or remove data from the MCC.

Figure 1.1. An $n \times n$ Mesh-Connected Computer

A mesh operates under one of two different control strategies, one of which is the SIMD (Single Instruction stream, Multiple Data stream) machine discussed in Thompson and Kung [64]. In the SIMD model, a single instruction is broadcast to all PEs, and executed simultaneously by those PEs specified in the instruction. A SIMD instruction could be: all PEs (i,j) with i even send the data of register 20 to the right. Another type of control strategy is the multiple-instruction, multiple-data (MIMD) model in which the PE's may execute different instructions at each step. In this model, a program is typically loaded into each PE, and the PEs execute independently, synchronizing only when data must be transmitted between them. The control strategy used in this thesis is a SIMD $n \times n$ MCC for $n=2^t$, where t is a positive integer. Unless mentioned otherwise, all algorithms and time complexities are given for their execution on an $n \times n$ MCC.

1.3. THE SYSTOLIC SCREEN.

One of the first applications for which the MCC has actually been built is (low-level) image processing. Obviously, the mesh architecture is well suited to store and manipulate digital images because of its one-to-one correspondence between the pixels of an image and the PEs of the mesh. An example of such a machine is the Massively Parallel Processor (MPP) of Batcher [4], which is a 128×128 mesh of PEs used to process images produced by satellites. The use of an MCC to store and manipulate digital images gives rise to the concept of a systolic screen. In the following we will formally define digital images and the systolic screen as used in the remainder of this thesis.

An *image* is defined as a subset of a digitized plane Π of size n^2 , where Π is a rectangular array of lattice points with integer coordinates (i,j) , where $i,j \in \{1, \dots, n\}$. A *picture element* (pixel) is a unit area square, and pixel (i,j) covers the square $[i-0.5, i+0.5]$ by $[j-0.5, j+0.5]$. A *systolic screen* of size n^2 is an $n \times n$ MCC where each PE P_{ij} represents a pixel centered about the lattice point (i,j) , with pixel $p_{1,1}$ occurring at the top left-hand corner of the MCC [14]. Figure 1.2 shows an image in an MCC, where the dark pixels are occupied by a digitized object. For an $n \times n$ systolic screen, an equivalent diagram that does not explicitly indicate the communication links will be employed. The corresponding diagram for the image of Figure 1.2 is shown in Figure 1.3.

Figure 1.2. Image on an MCC.

Figure 1.3. A systolic screen.

In the remainder of the thesis, a digitized image is denoted by black or shaded pixels, and unoccupied space is represented by white pixels. An *occupied* pixel therefore refers to a black or shaded pixel that is part of some object, and an *unoccupied* pixel refers to a white pixel. In algorithms that deal with images, the term pixel will most often be used when referring to that element of an image, while the term PE will be used when referring to (the contents of) the processor that represents that pixel. The difference between a systolic screen and the regular MCC is that each PE now represents a pixel, and a register in each

PE is used to denote the shade of its pixel. If only two shades are allowed, a binary black and white image results.

1.4. OTHER SYSTOLIC ARCHITECTURES.

The MCC is just one of the many systolic architectures that exist. Kung [26] illustrates several designs for processors arranged in a linear array, in which the processors are linked to form a list, to solve two-dimensional convolution. He presents variations on the mesh like 8-neighbor connected and triangular meshes to list a few, and demonstrates how they may be combined to solve matrix problems such as matrix multiplication, triangulation, or solving systems of equations. The remainder of this section presents different systolic architectures such as the linear array, mesh of trees, cube-connected cycles, hypercube, and pyramid. These architectures (and the MCC) are used as a model for which parallel solutions to problems are found. Variations of these architectures are often proposed to speed up algorithm execution, or for special purpose areas like image processing. The survey of Section 2.2 mentions several problems that are solved on these architectures.

A hypercube or (binary) k -cube computer is a systolic computer with $n=2^k$ processors interconnected as a k -dimensional binary cube. Each processor has its own CPU and local main memory, and is connected to k other processors by connections that correspond to the edges of a k -dimensional cube. That is, the processors are addressed between 0 and $n-1$, and two processors are connected by a communication link if the binary representation of their addresses differs by only one bit. Figure 1.4 illustrates the 3-cube hypercube interconnection network. Colley et. al. [6] mention that meshes of all dimensions, and trees may be embedded on a hypercube. Additionally, they note that the communication structures used in the Fast Fourier Transform and the Bitonic Sort algorithm can also be embedded on a hypercube.

Figure 1.4. A 3-cube.

Figure 1.5. A 32 PE CCC.

Preparata and Vuilleman [53], use the Cube-Connected Cycles (CCC) systolic architecture to perform sorting, Fast Fourier Transforms, matrix multiplication, permutations, and other algorithms. They also show that the CCC may emulate the k -cube with no significant degradation in performance. The CCC consists of $n = 2^k$ processors for $k = r + 2^r$ (for arbitrary k , r is chosen as the smallest integer for which $r + 2^r \geq k$), where each processor has an address m expressed as a pair (l, p) of integers with $(k-r)$ and r bits respectively. These processors are connected to three others: Forward, Backward and Lateral PEs (F, B, L) as:

$F(l, p)$ is connected to $B(l, (p+1) \bmod 2^r)$;

$B(l, p)$ is connected to $F(l, (p-1) \bmod 2^r)$;

$L(l, p)$ is connected to $L(l + e \cdot 2^p, p)$, where $e = 1 - 2 \cdot \text{bit}_p(l)$.

This interconnection scheme is displayed in Figure 1.5 for $k=5$ and $r=2$ giving $n=32$ processors. Different layouts give time/space requirements for the algorithms listed above. A time of $O(\log n)$ with $O(n^2 / \log^2 n)$ area, or $\log n \leq T \leq \sqrt{n}$ time and $A = O(n^2 / T^2)$ area may be achieved, or a time of $T = O(\sqrt{n})$ and area $A = O(n)$ layout may be used to emulate a $\sqrt{n} \times \sqrt{n}$ MCC.

The mesh of trees [25,33] is an $n \times n$ mesh whose rows and columns are the leaf nodes of trees. The nodes of the trees are processors that are used to communicate between nodes of the $n \times n$ mesh, and can perform simple operations such as summing, and finding the minimum. A 4×4 MOT is illustrated in Figure 1.6.

Figure 1.6. A 4×4 Mesh of Trees.

Figure 1.7. A 4×4 Pyramid.

An architecture similar to the MOT architecture is the Pyramid computer [15] which is a pyramid of successively smaller meshes in which each PE in the mesh is joined (if they exist) to one parent PE in the next smaller mesh above it, and four children PEs in the next larger mesh beneath it, in addition to its four mesh neighbors. In specifying a Pyramid's

dimensions, the size of the largest mesh is given, as for example the 4x4 Pyramid of Figure 1.7. The Pyramid has primarily been used in image processing [49,54], and multi-resolution images may be stored on it by averaging the pixels of the children PEs.

Further details on the architectures and algorithms mentioned may be found in [5,6,15,25,26,33,49,53,54,56,63] or references therein. The next section presents an overview of the problems solved on the MCC architecture defined in Section 1.2.

1.5 THESIS OVERVIEW.

Sequential solutions for representative problems in the field of computational geometry have been surveyed by Lee and Preparata [31]. Preparata and Shamos [52] present detailed solutions for many of these problems. Recently, parallel versions of several computational geometry problems have been designed for the mesh [40], and other parallel architectures as discussed in Section 1.4. Although the mesh was originally designed for low-level image processing [4], Miller and Stout [41] present parallel solutions for computational geometry problems on digitized pictures, thereby solving many high-level image processing problems. This thesis studies systolic algorithms of selected computational geometry problems, particularly those involved in solving visibility and visibility related problems on digitized pictures. Issues concerning the implementation of MCC algorithms are addressed by proposing a high-level language for the mesh. Chapters 2 and 3 address algorithmic issues of MCC computation, whereas Chapter 4 deals with a support environment for their implementation.

In Chapter 2, standard operations and techniques for algorithm design on an MCC are presented, and many algorithms in selected fields are surveyed. Implementations for send, broadcast and rotation operations are outlined. Sorting, random-access-read and random-access-write operations are used in complex data movement on the MCC. They are considered as algorithms in their own right, but they are significant as steps within other MCC algorithms. Consequently, a more detailed description of their implementation is presented. This is followed by a survey of algorithms in the areas of computational geometry, graph theory and image processing, which amply illustrates the use of the standard operations, as well as MCC programming technique.

In Chapter 3 we study parallel MCC solutions to visibility problems and introduce new solutions to a variety of visibility, separability and robotics problems for digitized images. In particular, new algorithms are designed to solve problems in parallel, perspective and rectangular visibility, as well as separability and the computation of the configuration space for a set of objects as discussed below.

In parallel visibility, the portions of a figure that are illuminated by parallel rays of light must be found. For perspective visibility, the portions that are illuminated by rays emanating from a point must be determined. These problems are solved for digitized objects; an extension to three-dimensions is presented for parallel visibility, and extensions that solve related problems are presented for the perspective visibility algorithm. A problem related to parallel visibility is the separability problem [57], in which a set of objects are sequentially separated in a given direction by moving them one by one to an arbitrary distance in that direction without colliding with the remaining obstacles. This problem is solved using ideas from the parallel visibility algorithm. Variations on the meaning of visibility exist [44], such as the concept of rectangular visibility in which two points are visible if the rectangle defined by using the points as opposite corners contains no other point. An algorithm for computing the digitized rectangular visibility between all black PEs is presented.

Another problem related to visibility is planning the path of a robot through a set of obstacles. This problem is referred to as path planning [34] and it consists of navigating a robot from one point to another while avoiding obstacles in its path. The computation of the configuration space, in which the obstacles are "grown" by the dimensions of the robot, is important in solving this problem since the robot may then be represented as a point. A parallel solution to compute the configuration space of a set of digitized objects is presented for certain types of robot.

Stout [62] has noted that the actual implementation of many of the MCC algorithms such as those discussed Chapters 2 and 3 is difficult due to the lack of language and system support. He indicates that standard operations and frequently used techniques such as recursion should be provided as part of the language, just as the recursion stack, for instance, is handled automatically in sequential machines. As a case in point, some algorithms such as Fast Fourier transforms, matrix multiplication and other arithmetic problems, have been implemented and executed on Batcher's [4] MPP, but MPP users have found its language and environmental support to be insufficient [51]. For these reasons, the design of a high-level language for the MCC is presented in Chapter 4.

2. MCC STANDARD OPERATIONS AND SURVEYED ALGORITHMS.

Systolic algorithms are presented by listing a sequence of parallel operations. This chapter defines the standard MCC operations, and the technique used for recursion and data compression. In particular, an algorithm for the fundamental operation of sorting a set of data elements stored on the MCC is presented. Recursion and data compression are techniques that are frequently used on the mesh. The meaning of recursion and data compression for a mesh layout is addressed in this chapter. Parallel data movement is a common feature of MCC algorithms; consequently, an example for a Random Access Read operation on the mesh is presented that illustrates the parallel control of data movement and sketches its implementation. This chapter surveys selected systolic algorithms that illustrate the design of systolic algorithms, and the use of the operations and techniques that were presented. Since the algorithms developed in this thesis solve computational geometry problems, the survey concentrates on existing MCC algorithms in this field. Problems solved on the architectures presented in Section 1.4 are mentioned as well. Finally, algorithms in graph theory and image processing are briefly surveyed, affording a more general overview of the types of problems solved on the mesh.

2.1. STANDARD OPERATIONS AND TECHNIQUES ON AN MCC.

This section describes the operations and techniques most frequently used in MCC algorithms. The operations allow a high-level description of MCC algorithms since they appear as steps in the algorithm, but are actually composed of many parallel steps. The basic operations, discussed in [10, 27, 41, 45, 46, 64] for an $n \times n$ MCC, are listed below. An outline of the implementation of these operations is presented in the sections that follow.

Send: A PE (i, j) sends a register value to another PE (s, t).

Broadcast within a Row [Column]: A PE (i, j) sends the value of a register to all PEs in its row [column].

Broadcast: A PE (i, j) transmits the value of a register to all PEs in the MCC.

Row [Column] Rotation: The value of register R of each PE in a row [column] is copied. These copies are then rotated in the row [column] in such a way that each PE in the row [column] sees them all.

Row [Column] Broadcast: A specific register of all PEs in a row [column] is seen by all PEs in the MCC.

Sort: A set of *key* and *data* registers, denoted (K,D) are specified. The key registers K are arranged in the order they are to be used in the sort. The data registers are not used for sorting, but are simply passed along with the key registers. The records (K,D) , are sorted into some specific order on the MCC. Sorted orderings are presented in Section 2.1.4.

Random Access Read (RAR): The value of register R of each PE (i,j) denotes the ID of the PE (s,t) from which data is to be read. Upon completion of the operation, PE (i,j) contains the data of the PE with the ID of register R . Register R contains a special illegal ID if a PE does not read.

Random Access Write (RAW): The value of register W of each PE (i,j) denotes the ID of the PE (s,t) to which data is to be written. Register W contains a special illegal ID if the PE does not write. If d values of register W are the same (excluding the illegal ID) then $O(dn)$ time is required.

All of the operations discussed above require worst case $O(n)$ execution time. The send operation may require less time, depending on the separation between the source and destination PEs.

Techniques used in some MCC algorithms are *data compression* and *recursion*. In data compression [10], relevant data is clustered into a submesh of the mesh. Many sophisticated MCC algorithms such as sorting and labeling the components of an image use recursion. In recursion, the same algorithm is executed on successively smaller submeshes of the MCC. Similarly to the standard operations, recursion allows a high-level description of an algorithm. One reason is that only the operations used on the data after split or a merge step are specified, and the steps involved in performing the mechanics of subdividing or merging submeshes are not explicitly mentioned. Two common types of recursion, and a possible implementation of one, are discussed after the basic operations in Sections 2.2.2 and 2.2.3.

2.1.1. Basic MCC Operations.

First, the basic operations for sending, broadcasting and rotating data through the mesh are discussed. The more sophisticated data movement operations of sorting, RAR and RAW are presented later.

The *send* operation is accomplished using the routing links of the interconnection network such that the number of links used is minimal. Data is routed from a source PE to a destination PE. Since the topology of the MCC is known, the routing of data presents no problem. The send operation is usually employed to send data to a neighboring PE. A set of selected PEs may simultaneously send data in a fixed direction. Routing conflicts do not arise in this case since at most one record arrives at any PE. If instead, these PEs were to simultaneously perform a send to an *arbitrary* PE, routing conflicts may occur when many data elements must pass through the same PE. To avoid this, the RAR or RAW operations

should be used when the destination PEs of parallel sends are in arbitrary positions on the mesh. The time complexity depends on the number of routing steps in the mesh between the source and destination PEs.

The *broadcast within a row [column]* for $PE(i,j)$ can be implemented as having $PE(i,j)$ send copies of the data to the left [up] and to the right [down]. The copies are shifted onwards in both directions until the sides of the row [column] have been reached. This operation is mainly used as a step in the broadcast operation.

The *row [column] rotation* is implemented similarly to the broadcast within a row [column], except that all the PEs in the row [column] make two copies of the data of the registers to be rotated. These copies form two lists, whose elements are shifted simultaneously through the row [column]. One list is shifted right [up] and the other left [down] until all of the elements of the list have been shifted off of a side of the row [column]. This operation is useful in order to perform some function on the data of each PE in a row or column by comparing it to all other data elements in that row or column.

The *broadcasting* of the value of a register R in $PE(i,j)$, is accomplished by performing a row broadcast of R in row j , followed by column broadcasts of the data stored in register R of $PE(i,k)$ $1 \leq k \leq n$. This operation communicates the results computed by one PE to the remainder of the mesh.

After execution of a *row broadcast* operation, all PEs of the mesh must have seen the values of the registers of a specified row. The row broadcast operation is accomplished by performing broadcasts within a column of the data in a specified row, followed by row rotations. A row broadcast for row two is illustrated for a 4×4 MCC in Figure 2.1. Notice that after the broadcast within the columns is executed, all rows are exactly alike in that their data is a copy of row two. Row rotations of the data in Figure 2.1(b) will then ensure that all PEs have seen the data of row two. The implementation of the *column broadcast* operation is similar to that of a row broadcast. This operation may be used to communicate a set of at most n values to all other PEs.

	1	2	3	4
1				
2	a	b	c	d
3				
4				

(a) Data in row two.

	1	2	3	4
1	a	b	c	d
2	a	b	c	d
3	a	b	c	d
4	a	b	c	d

(b) Broadcasts within columns.

Figure 2.1. Details of the row broadcast operation.

The operations presented in this section are abstractions of common sequences of steps performed in many MCC algorithms. Their format is mainly due to the physical interconnection scheme used for the mesh. Many of the algorithms surveyed make use of more sophisticated data movement, to achieve an ordering on a set of data elements, or to exchange data between arbitrary PEs. In view of this, Sections 2.1.4 and 2.1.5 present sorting and random access read and write operations.

2.1.2. Data Compression.

Frequently the routing of data during the execution of a standard operation dominates the execution time of an algorithm. Thus, if important data can be compressed into a smaller submesh, the performance of the algorithm might be improved. The operation of relocating the data relevant to an algorithm into a submesh is called *data compression*.

As an example, say an algorithm A has $N=n \times n$ data elements and performs $\log n$ steps on the data elements, and after each step the relevant data is reduced by a certain fraction k $0 < k < 1$. Assume further that, the work done at each step depends linearly on the maximum routing distance between the elements. If data compression is used each step will cost $O(k^i \cdot n)$ for $i=2,3, \dots, \log_k n$ which will result in an $O(n)$ algorithm. If data compression were not used the communication costs would imply a worst case $O(n)$ time complexity for each step of A resulting in a time complexity of $O(n \log n)$.

Data compression is mainly used if a linear fraction of the data becomes irrelevant after a step of an algorithm on a mesh. An example, discussed in Section 2.2.2, is a binary search technique used by Miller and Stout [40] in their convex hull algorithm. In implementing data compression, a key of 0 or 1 may be used, where relevant data is marked by a 0, and irrelevant data by a 1. If the data is sorted by this key using the *Sort* operation, all of the irrelevant data elements will occur at the end of the sorted order. The amount of relevant data can therefore easily be established using row and column rotations, from which the size of the minimum $k \times k$ submesh that contains it may be determined. The relevant data can then be rearranged to fit into the submesh by computing the ID of the PE of the submesh into which it must fall (based on its place in the sorted order) and performing a RAW to this ID.

2.1.3. Recursion.

The divide-and-conquer paradigm, popular in sequential programming, is commonly used to solve problems on the mesh and other architectures. Some examples may be found in many of the algorithms surveyed in Section 2.2, as well as those found in [10,12,22,35,36,37,40,41,42,45,46,62,64]. Stout [62] discusses recursion, and gives an overview of systolic algorithms that use recursion to solve problems such as finding the connected

components (objects) in an image, or finding the extreme points for digitized objects.

The main method of recursion repeatedly subdivides an $n \times n$ MCC, with $n=2^t$, into four equal-size square submeshes until the number of PEs in the smallest submesh satisfies the algorithm's requirements for the first merge step to occur. The subdivision process is accomplished by storing the value $k=\log n$ in each PE, and decrementing it by one at each step until 2^k is small enough. The four submeshes generated by the previous subdivision step must now be merged. All PEs can compute the submesh in which they lie based on k and their PE's ID (i,j). During the merge step, k is incremented by one thereby merging the four neighboring submeshes generated in the split step. The merging process continues with the new value of k , and stops when $k=\log n$. Figure 2.2 demonstrates the recursive merge steps for an 8×8 MCC, where the minimum value of k is zero and the maximum value is three. The dashed lines separate the subsquares to be merged in a step. Figure 2.2(c) shows the merge step before the entire 8×8 MCC becomes undivided. The final undivided mesh has not been illustrated, but it is the mesh after the dashed lines of Figure 2.2(c) have been removed.

Figure 2.2. Recursively merging subsquares on an 8×8 MCC.

Many algorithms (see [12,22,35,42]) use the divide-and-conquer paradigm to divide the data in half at each step instead of into four. For this strategy, the mesh is alternately divided in half along the middle column or middle row of the resulting submeshes, and the data is split and placed in each half of these submeshes. Figure 2.3 illustrates the splitting of the mesh for this type of recursion.

(a) The first split step. (b) The second split step. (c) The third split step.

Figure 2.3. Divide-and-conquer splitting on an 8x8 MCC.

The type of recursion where the MCC is divided into four submeshes, at each step, will be referred to as *square recursion*, while the type of recursion that divides the data in half, at each step, will be called *divide-and-conquer recursion*. Notice that divide-and-conquer recursion may be transformed to the more natural (for the mesh) square recursion by performing two consecutive splits of the data, and two consecutive merges to merge the submeshes. If the complexity of the merge (or split) step of an algorithm is $c2^i$, for c a constant, then the total complexity is $\sum c2^i, (i=0,1, \dots, \log n) = 2cn = O(n)$ for square recursion, and $4cn = O(n)$ in divide-and-conquer recursion.

2.1.4. Sorting as a Fundamental Operation.

In sorting data, two sets of registers are specified: the key K , and data D registers. The key registers are used in a specific order to compare two elements, and the data registers are transmitted with the keys but are not used in the sort operation. In the sort operation the contents of registers (K,D) are placed in some specific ordering among the PEs using the keys specified in K for the comparison operation.

There are four main types of sorted orderings: row-major, shuffled row-major, snake-like row-major and proximity order, as indicated in Figures 2.4(a), (b), (c) and (d) respectively. The first three are the most common, while the fourth is a variant of the shuffled row-major order used by Miller and Stout [40] in an algorithm to be surveyed later. The algorithms in [27,45,64] (as presented or with few modifications) can also sort a $k \times k$ submesh for $k = 2^i, 1 \leq i \leq t, t$ an integer. Instead of re-sorting, one ordering may be transformed into another in faster time (by a multiplicative constant) by rearranging the sorted data. For example row-major is transformable into snake-like row-major in $2(n-1)$ steps on a $n \times n$ MCC by simply reversing the keys in the even rows.

The snake-like row-major ordering has the advantage that data elements adjacent in the sorted list are stored in neighboring processors. Data may be rotated (within intervals) of the snake-like order, just as data is rotated in the row rotation operation. The shuffled row-major ordering distributes the sorted data in such a way that the first quarter of it resides in one quadrant, the second, third and fourth quarters in the remaining three quadrants. This holds recursively in each quadrant. Divide-and-conquer type solutions of problems on the MCC frequently make use of this ordering, since it follows the same layout presented for square recursion in Section 2.1.3. After a sort in shuffled row-major order, a recursive algorithm would only have to merge submeshes in a bottom-up fashion. In the proximity ordering of Figure 2.4(d) elements adjacent in their sorted order are also found in neighboring PEs, and the layout follows the recursive subdivision type of the shuffled row-major ordering. Consequently, this ordering has the advantages of the snake-like row-major and shuffled row-major orderings.

(a) Row-Major.

(b) Shuffled Row-Major.

(c) Snake-like Row Major.

(d) Proximity.

Figure 2.4. Sorted orderings on a Mesh-Connected Computer.

Processors are usually assigned ID's i , in the range $1 \leq i \leq N$, where $N = n^q$ for a q -dimensional $n \times n \times \dots \times n$ MCC, based on one of the sorted orderings. For example, if the rows and columns of Figure 2.4 are numbered from one to four, then PE(4,2) has row-major, shuffled row-major, snake-like row-major and proximity ID's of 14, 12, 15, and 7 respectively.

Many different algorithms have been written to sort data on a MCC. Data is usually sorted on an $n \times n$ MCC where n is a positive integer power of two. Two papers among those written are Thompson and Kung [64] who have sorted data using ideas from Batcher's bitonic sort, and Nassimi and Sahni [45] who present a different adaptation of the bitonic sort technique to produce the row-major ordering. The above papers also extend their algorithms to q -dimensionally connected $n \times n \times \dots \times n$ MCCs with a time complexity of $O(n)$. Additionally, Thompson and Kung describe an algorithm for an $n \times m$ MCC, where n and m are integer powers of two. An algorithm by Lang, Schimmler, Schmeck, and Schröder [27] that sorts data into snake-like row-major order is now discussed in detail.

In the discussion that follows the term snake-like will be used to refer to the snake-like row-major ordering. The algorithm by Lang et. al. uses the square recursion technique discussed in Section 2.1.3, although they refer to their technique as iteratively merging subsquares of the MCC. The merge step merges four neighboring subsquares of size $k/2 \times k/2$ whose elements are sorted into snake-like order, resulting in a $k \times k$ square which is sorted in snake-like order.

The algorithm uses two operations: the *shuffle* operation and the *odd-even transposition sort*. A *shuffle* operation takes a sequence $(a_1, a_2, a_3, \dots, a_n)$ and produces the shuffled sequence $(a_1, a_{n/2}, a_2, a_{n/2+1}, \dots, a_{n/2-1}, a_n)$. This operation is often referred to as a perfect shuffle since the sequence is divided in half, and starting from the first element of each half-sequence, the elements are alternately taken from the first and second half-sequence to produce the final sequence. The *odd-even transposition sort* sorts a sequence of numbers $(a_1, a_2, a_3, \dots, a_n)$ by alternating between odd and even comparison steps. In an odd comparison step, the odd indexed data elements i and their neighbors $i+1$, are compared and exchanged if the smaller keyed data element is at index $i+1$. The even step is the same as the odd step except that the even indexed data elements are compared with their neighbors and exchanged if they are unsorted. A proof that at most n steps of the odd-even transposition sort are required to sort a sequence of n numbers can be found in [32]. Figure 2.5 illustrates the steps in sorting eight numbers where dashes are used to indicate a comparison. The sequence is sorted after the seventh step.

1.	7-6	8-3	2-5	4-1	5.	3-2	6-1	7-4	8-5		
2.	6	7-3	8-2	5-1	4	6.	2	3-1	6-4	7-5	8
3.	6-3	7-2	8-1	5-4	7.	2-1	3-4	6-5	7-8		
4.	3	6-2	7-1	8-4	5	8.	1	2-3	4-5	6-7	8

Figure 2.5. Odd-even transposition sort of eight elements.

The algorithm's merge step first performs a shuffle operation, in which the elements within each column remain in fixed order, while the columns of the $k \times k$ array are perfectly shuffled among themselves. The odd numbered columns i , and column $i+1$ are then formed into $k \times 2$ double columns, which are sorted into snake-like order using $2k$ steps of odd-even transposition sort. Lastly $2k$ steps of odd-even transposition sort are applied to the entire $k \times k$ array, where the elements of the array are assumed to be indexed in snake-like order forming a sequence of length k^2 . A proof of the correctness of the algorithm is presented in [27], which uses the 0-1 Principle [24] to prove that $2k$ steps of odd-even transposition sort are sufficient to sort the $k \times k$ array after the double columns have been sorted. The merge steps are illustrated in Figure 2.4 where Figure 2.6(a) shows the four sorted submeshes with the higher $k/2$ columns of the $k \times k$ mesh shaded. Figure 2.6(b) shows the columns after the shuffle operation, and Figure 2.6(c) shows the mesh after the $k \times 2$ double columns have been sorted. A final $2k$ steps of odd-even transposition sort produces the sorted mesh of Figure 2.6(d).—

(a) The 4 sorted submeshes.

(b) The shuffled columns.

(c) Sorted $k \times 2$ columns. (d) The final sorted $k \times k$ mesh.

Figure 2.6. The merge step to sort a $k \times k$ mesh into snake-like order.

A numerical example shown in Figure 2.7 outlines the steps once more. Figure 2.7(a) shows the 4x4 MCC where the 2x2 submeshes have been sorted into snake-like order. Figure 2.7(b) illustrates the shuffle step, where columns two and three are interchanged. Figure 2.7(c) gives the ordering after the $2k$, for $k=4$, odd-even transposition steps have been performed on the $k \times 2$ double columns. Figures 2.7(d) to 2.7(g) illustrate the odd-even transposition sort if a snake-like ordering of the 4x4 array is assumed. The bold lines indicate a comparison. Although at most 4x2 odd-even transposition steps are performed in general, the mesh is sorted after 4 steps.

Figure 2.7. A numerical sorting example.

2.1.5. The RAR and RAW Operations.

As previously mentioned in this chapter, if many PEs perform read or write data to arbitrary PEs in parallel, routing conflicts may result due to the simple nature of the data routing strategy used in the send operation. To allow the orderly routing of data through the mesh, two operations to read and write arbitrary PEs are discussed. The Random Access Read and Random Access Write algorithms are described by Nassimi and Sahni [46] for a q -dimensional $n \times n \times \dots \times n$ MCC, and execute with a time complexity of $O(q^2n)$ for the RAR, and $O(q^2n + dqn)$ for the RAW operation provided that at most d processors write to any one PE. The main steps of the algorithm for RAR are outlined in Example 2.1. In the example, the PEs are assumed to have been assigned an ID based on either the shuffled row-major or row-major order as discussed in the previous section.

Example 2.1: A RAW example on 8 processors.

Let: - $S(i)$ be the register of $PE(i)$ indicating the PE from which data is to be read.
 - $D(i)$ be the data register of $PE(i)$.

At the end of the RAR operation, a register of $PE(i)$ has the contents of register $D(S(i))$. A particular value, denoted by ∞ , indicates that the PE does not request to read and dashes, "-", will be used to indicate that the value of the register is not important in the remainder of the operation. Procedures used in the RAW are first presented, followed by the example.

rank: The relevant data records are first *flagged* by setting a boolean register to true. All flagged records are to be ranked, and the remaining records are ignored but not destroyed. The rank of a PE is the number of flagged PEs below it where the PE's are listed in row-major or shuffled row-major order.

concentrate: Ranked data is concentrated by placing the data of a $PE(i)$ of rank r into $PE(r)$.

distribute: A distribute operation sends data that has been concentrated back to $PE(i)$.

generalize: Generalize makes multiple copies of data.

The initial contents of register $S(i)$ for $1 \leq i \leq 8$ are indicated below, from which it can be seen that $PE(1)$ wishes to read from $PE(3)$, $PE(2)$ from $PE(7)$, etc....

ID(i)	1	2	3	4	5	6	7	8
S(i)	3	7	4	3	∞	3	1	7

Each PE first forms the record $\langle S(i), T(i) \rangle$, where $T(i)$ is assigned ID(i). These records are then sorted on $S(i)$ using $T(i)$ to break ties. After sorting these records, all distinct non-infinite keys $S(i)$ with the highest value for $T(i)$ are flagged. The flagged PE's are indicated by *'s in the example below. The *rank* procedure is then used to rank all flagged PE's, placing the resulting rank in register $R(i)$ as illustrated.

ID(i)	1	2	3	4	5	6	7	8
S(i)	1*	3	3	3*	4*	7	7*	∞
T(i)	7	1	4	6	3	2	8	5
R(i)	1	-	-	2	3	-	4	-

New records $\langle R(i), U(i), S(i) \rangle$, where $U(i) = ID(i)$, are defined for the flagged PE's, and these records are *concentrated* as illustrated below. The contents of register $T(i)$ were not lost but have been omitted for brevity. This concentration step will ensure that the data of the PEs on whom a read request was made is read once only.

ID(i)	1	2	3	4	5	6	7	8
R(i)	1	2	3	4	-	-	-	-
S(i)	1	3	4	7	-	-	-	-
U(i)	1	4	5	7	-	-	-	-

The record $\langle S(i), V(i) \rangle$, where $V(i) = ID(i)$, is defined for the ranked PEs, and these records are *distributed* in the format shown below using the value of $S(i)$ as a PE index. The data D of the PEs on whom a read request was made is now read into $S(i)$.

ID(i)	1	2	3	4	5	6	7	8
S(i)	D(1)	-	D(3)	D(4)	-	-	D(7)	-
V(i)	1	-	2	3	-	-	4	-

The values $D(i)$ read must now be duplicated the number of times a read request was made for them. To accomplish this records $\langle V(i), S(i) \rangle$ are first *concentrated* into the configuration shown below.

ID(i)	1	2	3	4	5	6	7	8
S(i)	D(1)	D(3)	D(4)	D(7)	-	-	-	-
V(i)	1	2	3	4	-	-	-	-
U(i)	1	4	5	7	-	-	-	-

The number of copies of $D(i)$ required is the difference $U(i) - U(i-1)$ above, with $U(0)$ defined as zero. The $D(i)$'s are duplicated using the *generalize* operation, resulting in the data configuration shown below. The value of $T(i)$ is indicated again.

ID(i)	1	2	3	4	5	6	7	8
S(i)	D(1)	D(3)	D(3)	D(3)	D(4)	D(7)	D(7)	-
T(i)	7	1	4	6	3	2	8	5

A sort on the record $\langle T(i), S(i) \rangle$ will result in the register $S(i)$ of $PE(i)$ containing the data requested for the read operation. The final contents of $S(i)$ are illustrated below along with the original read requests stored in $S(i)$.

ID(i)	1	2	3	4	5	6	7	8
S(i)	D(3)	D(7)	D(4)	D(3)	-	D(3)	D(1)	D(7)
S(i)	3	7	4	3	∞	3	1	7

(final)
(original)

This concludes Example 2.1. The operations to *flag*, *rank*, *concentrate*, *distribute*, and *generalize* data have been defined formally, and are discussed in detail in [46]. The rank operation may be performed by sorting all records, which also contain the current PE's ID (as a return address), into shuffled row-major order, by the specified key, such that flagged records occur before all unflagged records. The shuffled row-major index of the PEs in which the records now reside is assigned as the rank of the flagged records. A sort on the all records (both flagged and unflagged) by the return address will restore the records to their previous positions. Nassimi and Sahni use the order already present in the records due to the previous sort operation to present a ranking algorithm with $2q(n-1)$ unit-routing steps. A similar technique is applied to the concentrate and generalize operations. Since

the sort operation of Thompson and Kung [64] or Nassimi and Sahni [45] requires $O(q^2n)$ time, it dominates the execution time of the rank, concentrate, distribute and generalize operations resulting in the overall $O(q^2n)$ time given for the RAR operation.

The RAW operation is detailed in [46] and is similar in execution to the RAR, except that duplicate writes are handled differently. Its execution time requires an additional $O(qdn)$ steps since the d records sent to a destination PE are treated as one record in the rank, concentrate and generalize operations. This results in the execution time of $O(q^2n + dqn)$ mentioned for the RAW operation thereby concluding this section.

In the next section, the use of operations and techniques presented are illustrated using MCC algorithms. Variations of these operations are introduced for some problems, while Chapter 4 presents them in the context of an MCC language.

2.2. A SURVEY OF PARALLEL ALGORITHMS.

This section surveys MCC algorithms for certain computational geometry problems in detail, first presenting some introductory computational geometry terms. Additional terms are defined when they appear. Representative problems in computational geometry or image processing are mentioned for the architectures presented in Section 1.4.

Computational geometry problems are presented in Section 2.2.2 for objects whose boundary (usually represented by a sequence of line segments) is stored on the MCC. The problems solved in this section improve the time bounds of their sequential counterparts. The corresponding versions of these problems, that now operate on objects that are digitized on a systolic screen (Section 1.3), are presented in Section 2.2.3. This section illustrates that the MCC need not be solely concerned with low-level image processing operations, some of which are presented in Section 2.2.4. Many of the computational geometry problems that are discussed in the survey make use of algorithms from graph theory that have been implemented on the MCC. A cross-section of these algorithms is presented in Section 2.2.5. All problems in this chapter, unless otherwise mentioned, are solved on an $n \times n$ MCC where n is a positive integer power of 2.

2.2.1. Computational Geometry Definitions.

Computational geometry [31,52] is a field within computer science concerned with geometric problems, their complexity and solutions. Computational geometry problems may be solved in *image space* which uses a digitized representation of a set of objects, or in *object space* in which a description of the boundary of an object is used.

In object space, the boundary of an object is usually represented by a *polygon* P , which is a set of points $P = (p_0, p_1, \dots, p_{m-1})$, where edge e_i is defined as the line segment (p_i, p_{i+1}) with subscripts taken modulo m . Most algorithms operate on *simple polygons*, which are

polygons with the additional condition that any pair of edges e_i and e_j ($0 \leq i < j \leq m$) do not intersect unless $j = (i+1)$, in which case they intersect at point p_{i+1} only. A *convex polygon* is a simple polygon whose interior angles are convex.

In image space the digitized image of an object, such as a simple polygon, is stored one pixel per PE on an $n \times n$ MCC. In object space the attributes of the object, such as edges or points, represented using Cartesian coordinates, are usually stored one attribute per PE. For example, a set of line segments of a simple polygon, a set of points, or a set of circles represented by their center and radius are all stored in separate PEs. In both object and image space, a unique label is usually associated with the data of an object thereby permitting more than one object to be stored in the MCC without ambiguity.

The *4-neighborhood* [14] of a pixel (i, j) is the set of its four horizontal and vertical neighbours $(i \pm 1, j)$ and $(i, j \pm 1)$. A *path* from pixel p to pixel q is a sequence of pixels $p = p_0, \dots, p_r = q$ such that p_i is a 4-neighbor of p_{i-1} , $1 \leq i \leq r$, and both contain the same label. The *length* of a path is one less than the number of PEs in the path, and corresponds to the Manhattan metric distance between the pixels. A pixel p is at the *border* or *boundary* of a digitized object P if any of its neighboring pixels are not part of P . In fact, the number of borders at p is the number of 4-neighboring PEs whose label (or color) is different from that of p . Some algorithms, such as finding a nearest neighbor, require a *monotone* distance metric, which is a non-decreasing function of the separation between the points. Examples of monotone metrics are the Euclidean and the Manhattan metric.

Pixels p and q are said to be *connected* if there exists a path from p to q . A *connected component* of an image is a set of pixels that are all connected. A component P is *rectilinearly convex* if any horizontal or vertical line cuts borders of P at most twice. Miller and Stout [41] define a set of PEs on a systolic screen as *convex* if and only if the corresponding set of integer lattice points can be contained in a convex polygon that contains no other lattice points. The convexity of a digitized image depends on the digitization process used. The process used by Miller and Stout is to color a PE black if all of the pixel area is covered by the object being digitized. The *convex hull* of a set S of PEs, denoted $CH(S)$, is the smallest convex set of PEs containing S . A PE p in S is an *extreme point* of S if p is not inside $CH(S-p)$.

2.2.2. Computational Geometry in Object Space.

This section surveys computational geometry algorithms for figures represented in object space, i.e., by points and/or line segments on an $n \times n$ MCC. Problems for convex hulls, rectangles, computing the minimum spanning tree, and Voronoi diagrams are among those considered. Detailed descriptions are first presented for convex hull and the smallest

enclosing rectangle problems, followed by a brief problem description, with the execution time, of the remaining problems.

Additional operations, used by Miller and Stout [40], based on the proximity order discussed in Section 2.1.4, to broadcast or rotate data within that ordering and to perform a variation on the random access read and write operation, are introduced. Details on their implementation may be found in [39]. As stated in [40], the proximity order index of a PE may be computed in $O(\log n)$ time by each processor using a binary search technique. The first operations are *Broadcasting* or *Rotating Data within Intervals*. The idea used is to broadcast a value to all PEs in an interval of the proximity order, or to rotate the records in this interval among themselves. In both cases, each PE contains a record $\langle \text{label}, \text{data}, \text{flag} \rangle$, sorted by label to form an (ordered) interval in the proximity ordering. The data of all records with $\text{flag} = \text{true}$ can be seen by all PEs holding the same label in $O(\max\{m(r) + i(r)^{1/2} \mid r \text{ a label}\})$ time, where $m(r)$ is the number of marked records with label r , and $i(r)$ is the total number of records with label r . Essentially, a breadth first spanning tree is built by connecting PEs of an interval to the 4-neighbor of the highest ID that contains the same label and/or interval. Since the proximity order places elements of the same label in quadrants, (with leftovers in the next quadrant), the height of this tree is $O(i(r)^{1/2})$. The marked data is passed up the tree to the root, after which it is passed back down the tree where each parent passes the data to all of its children. This operation is a broadcast within an (ordered) interval if one data record is passed back, and a rotation within an (ordered) interval if multiple records are circulated within their interval. The same "tree-technique" scheme may be used to solve operations such as summing, and finding the minimum of values over an interval in $O(\max\{i(r)^{1/2}, \mid r \text{ a label}\})$ time, since the records are combined as they move up the tree.

The remaining operations to be discussed are *concurrent read* (CR) and *concurrent write* (CW). In the CR, the source PEs generate a record $\langle \text{key}, \text{return_address}, \text{data}, \text{SEND} \rangle$ (all keys are distinct). Each destination PE specifies a constant c number of records $\langle \text{key}, \text{return_address}, \text{data}, \text{RECEIVE} \rangle$ where the data is as yet undefined. A PE may be both a source and a destination. The CR then transfers the records to the destination PEs that request its key. In CW, the destination PEs simply indicate their willingness to receive records. The source PEs generate the records as before except the keys may not be distinct. At the end of the CW, a destination PE has between zero, and the fixed number of requested records. Each key is received by exactly one destination PE, and if two or more keys are the same, then a destination PE will receive the minimum such record (any other, associative and commutative binary operation may be used).

These, and the standard operations presented in this chapter, are used for solving problems such as finding the extreme points of a point set to be discussed next. Serial

algorithms that computing the convex hull of a point set [52,67] have been found to have applications in image processing, feature extraction, shape decomposition in pattern recognition, and obtaining triangulations of point sets to name a few. The parallel $O(n)$ algorithms of Miller and Stout [40] for marking a convex hull for one or more figures, and detecting their intersection improve the $O(n^2 \log n)$ time bounds of the serial versions for n^2 planar points.

Mark Convex Hull: Mark the extreme points of a set of at most n^2 planar points stored one point per PE.

The points are first sorted into non-decreasing proximity order by x -coordinate, using the y -coordinate to break ties. Duplicate points are then eliminated by having each PE P_i , $2 \leq i \leq n^2$ examine the point stored at PE P_{i-1} and mark itself the points are equal. As the algorithm progresses, points that are not a part of the convex hull become marked, and only the unmarked points are considered in further steps. The convex hull and the position of each extreme point in a counter-clockwise (cc-wise) ordering of the convex hull, are found recursively for quadrants A_1, A_2, A_3 , and A_4 illustrated in Figure 2.8. The convex hulls $B_1 = CH(A_1 \cup A_2)$ and $B_2 = CH(A_3 \cup A_4)$ are then computed simultaneously, and finally $CH(B_1 \cup B_2)$ is found.

Figure 2.8. Quadrant locations. Figure 2.9. Tangent points and rays.

This merge step is now outlined for $A \cup B$, with the points of $A = A_1$ taken to the left of those in $B = A_2$. The convex hulls of A and B are non-intersecting since sorting was performed on the x -coordinates, which are assumed to be distinct after the removal of duplicates. In computing $CH(A \cup B)$, the parallel lines of support p, p' , and q, q' for the

tangent lines (illustrated in bold in Figure 2.9) to $CH(A)$ and $CH(B)$ must be identified. A binary search to identify p and p' is given, and the search for q and q' is similar. The points t and u , as the leftmost and rightmost points respectively of $CH(A)$, and t' and u' , as the leftmost and rightmost points of $CH(B)$ are first found. This is accomplished using the tree-technique (discussed above) to find the minimum and maximum x -coordinate points of $CH(A)$, after which all (unmarked) PEs of A know the values of t and u as well as their positions in the cc-wise ordering of $CH(A)$. Each PE in A that contains an extreme point then decides if it is above the line tu . The candidate for p chosen as the middle vertex of the chain of points found to be above tu using the broadcast within intervals by the points' cc-wise position. The left and right neighbors x and y of p in the cc-wise order are then determined from which the rays xp and py , as illustrated in Figure 2.9, are formed. Similarly, t' , u' and the candidate p' and rays $p'x'$ and $y'p'$, as well as all extreme points that are above the line $t'u'$ are identified for B .

All extreme points in B above line $t'u'$ perform a CR to obtain rays xp and py . The PEs then determine if the points they contain are all below xp , and some are above py . If this is true, the candidate p selected is a tangent point. If not, and if xp is not above all extreme points of B , u is set to x , otherwise xp and py are both above all the extreme points of B and t is set to y . Extreme points in the chain of A that are above line tu perform the same operation to check p' . If p and p' are not tangent points the data in A and B is compressed by $1/2$, eliminating the chains below the lines tu and $t'u'$ respectively, and the binary search is iterated. If p is a point on a parallel line of support but p' is not (or vice versa), then p and the chain above $t'u'$ are compressed and the operation is iterated. In the data compression, the points of A and B do not remain in quadrants A_1 and A_2 , rather the remaining points are compressed into one subsquare. This compression ensures that the $O(\log n)$ iterations of the binary search for p, p', q, q' are completed in $O(n)$ time. Once p, p', q , and q' are found, the positions of the extreme points in the cc-wise ordering of $CH(A \cup B)$ are computed using the positions of p, p', q , and q' and the extreme points' previous positions. The chains that are not a part of the convex hull are marked, thereby completing the merge step.

Since data is compressed by $1/2$ at each iteration of the binary search, the communication costs are bounded by the furthest distance between PEs resulting in an $O(n)$ algorithm.

This algorithm is easily extended for a set of figures labeled S if the points are sorted with the label of their component as the primary key, and x - and y -coordinates as the secondary and tertiary keys. The Mark Convex Hull algorithm can then be executed simultaneously for each figure. Once the convex hulls have been determined, the area,

perimeter, and centroid of each can be determined simultaneously in $O(n)$ time using the broadcasting or rotating data within intervals operations. A further application of the convex hull algorithm presented is to solve the smallest enclosing rectangle problem.

Smallest Enclosing Rectangle: For each labeled point set S find a minimum area rectangle that encloses it.

The $CH(S)$ for each figure is first found as discussed previously. Since the positions of the points in the cc-wise order are known, the edges of the convex hulls are known. For each edge e of a convex figure, a smallest enclosing box is found by using a binary search [42] to find the points r, s and t illustrated in Figure 2.10, and the minimum area rectangle over all rectangles generated for the edges e is selected. Notice that the smallest enclosing rectangle from each edge of S is computed as a by product of the algorithm's execution.

Figure 2.10. The smallest enclosing rectangle for edge e .

Intersection of Convex Hulls: Given two figures A and B , determine whether their convex hulls intersect.

Let there be r extreme points in A and s in B in convex hulls of A and B identified using the Mark Convex Hull algorithm for multiple figures. A binary search (from [42]) is then used, where line segments between the extreme points numbered $1, r/4, r/2, 3r/4$ points are constructed as $(1, r/4)$, $(r/4, r/2)$, $(r/2, 3r/4)$ and $(3r/4, 1)$ for A . Similar line segments are constructed for B . These line segments are tested for intersection, and the minimum distance of each of the selected extreme points to the other figure is calculated. If these line segments intersect, then the convex hulls of A and B must intersect. Otherwise, the distances computed to the special points are used to discard half of the extreme points of each figure that are farthest away from the other figure. The remaining data is compressed and the process is continued recursively for at most $O(\log n)$ iterations resulting in an $O(n)$

overall execution time for the algorithm. The case of A being contained in B , or vice-versa, is handled by checking if the endpoints of the 4 line segments of one figure are contained in the other figure.

Additional algorithms solved by Miller and Stout [40] are the all points nearest neighbor problem, and detection of line segment intersection among a set of line segments that may intersect at their endpoints. Extensions of these problems for sets of labeled points and line segments are outlined as well. Solutions to all nearest neighbor problems for a set or sets of points, between sets of line segments, for a set of rectangles or circles are presented where the label and distance between the vertices or a vertex and line segment may be used. Algorithms that determine whether sets of differing labeled line segments intersect, whether many simple labeled polygons intersect, and whether a set of line segments forms a simple polygon (for each labeled set) and many other related algorithms are given. Among the problems solved for n^2 rectangles (or circles), stored one per PE, are: the total area covered by all rectangles, the area of each rectangle not covered by another, the number of rectangles containing a point (for n^2 points simultaneously), and the nearest neighbor rectangle (circle). An $O(n \log n)$ algorithm to find a minimum spanning tree that is superior to Lu and Varman's [37] is also presented. Extensions to higher dimensions of some of the above algorithms are mentioned.

-) Lu and Varman [36] solve rectangle problems for determining the total area covered by all rectangles, and the area of each rectangle not covered by another. Another solution to the all-nearest neighbor problem [37] solved by Miller and Stout [40] is presented, and using an $n^{2/3} \times n^{2/3}$ MCC, Lu and Varman present an $O(n^{2/3} \log n)$ algorithm to compute the Euclidean Minimum Spanning Tree for n planar points. The Voronoi diagram has been computed in $O(n \log n)$ time by Lu [35] using a transformation to 3-space, and a three-dimensional convex hull algorithm. Recently Jeong and Lee [22] have solved computation of the Voronoi diagram in $O(n)$ time by parallelizing the merge step of the sequential divide-and-conquer Voronoi diagram algorithm. Planar point location, trapezoidal decomposition, intersection detection, smallest enclosing circle problems are also solved within that time bound.

Chazelle [5] uses a one-dimensional array of n PEs, which is a single row of the MCC without the column links, to solve in $O(n)$ time dynamic versions of convex hull, point inclusion, range and intersection search, planar point location, intersection, triangulation, and closest point problems. In the dynamic cases, such as adding a new point and recomputing the convex hull, the time complexity is also $O(n)$ but the points may be pipelined, that is the points are sent one after the other with a constant delay time between them.

The next section presents image-space solutions to many of the algorithms solved in this

section.

2.2.3. Computational Geometry in Image Space.

The MCC has been used for low-level image processing [18,29] as discussed in Section 2.2.4. Miller and Stout [41] describe some high level algorithms that may be used to classify digitized objects. One algorithm finds the *internal distance* of all PEs to a marked PE m in the same component, and the other finds the *extreme points* of each connected component. Both problems, although interesting in themselves, may be used to solve many related problems as discussed later.

All algorithms presented in this section work with images on an $n \times n$ MCC. Unless otherwise stated, capital letters will be used to refer to the set of all the PEs in a connected component, while lowercase letters will refer to a PE of the component. For PEs a and b of the same connected component, a *minimal a - b path* is an a - b path of minimum length over all such paths. The *internal distance* from a to b , denoted $\text{DIST}(a,b)$, is the length of a minimal a - b path.

Internal Distance Problem: Determine $\text{DIST}(s,m)$ for each PE s of S to m a fixed PE of S .

The problem is broken into two subproblems that are solved using recursion and by only considering distances between certain PEs of the MCC. The PEs used in the subproblems are first defined.

The *outer border elements* of a $k \times k$ square are those PEs in columns or rows 1 and k . The *inner border elements* of a $k \times k$ square are those PEs in columns or rows $k/2$ and $(k/2)+1$. The term *border elements* is used to refer to both inner and outer border elements of a $k \times k$ square. Figure 2.11 illustrates these border elements for a $k \times k$ square.

Figure 2.11. The border elements of a $k \times k$ square.

Problem I determines $\text{DIST}(s,m)$ between the border elements of the $n \times n$ MCC, and between the border elements and m . Problem II determines $\text{DIST}(s,m)$ between all elements of S and m using the algorithm and results obtained for Problem I.

The solution to Problem I uses an $(8k-15) \times (8k-15)$ matrix $A_k(i,j)$ for a $k \times k$ square of the MCC to store the *restricted* internal distance from PE i to PE j , where paths are restricted to the PEs in the square. The restricted internal distance between the border elements, and from the border elements to m are kept in the matrix. If no path exists between two PEs, or m is not present in the square, the distance infinity is used. The PE IDs responsible for the minimum distance are also stored for each element of the matrix. Since the matrix is larger than the $k \times k$ square in which it must reside, each PE stores approximately 64 elements of the matrix.

A divide-and-conquer technique is used to solve Problem I, where the $n \times n$ MCC is recursively divided into subsquares until 4×4 subsquares result. The internal distance matrices for these squares are computed, and the neighboring matrices of a recursive step are merged until the $n \times n$ MCC remains.

Four neighboring squares of the MCC are merged by merging their matrices. This is accomplished by first eliminating the rows and columns that represent the inner border elements in each of the four neighboring matrices. These matrices are then merged, and the PEs that become inner border elements set the distance to their neighboring PE as one. The merge process therefore gives the same configuration as Figure 2.11, except for the value of k , which is now twice as large. The restricted internal distances between the new outer border elements, and m are now updated using a *transitive closure* function $A_k(i,j) = \min\{A_k(i,j), A_{k-1}(i,k) + A_{k-1}(k,j)\}$. This operation can be executed in time $O(k)$ by a modification to a transitive closure algorithm for graphs presented in [68].

Problem II is now solved by recursively subdividing the MCC. The correct distances of the border elements found in Problem I are used to compute the distance to m . Notice that in solving Problem I, the distance to m was not defined in the squares where m was not present. Now that the distance to m is known, the correct distances to m can be determined for each PE. The same algorithm is used as for Problem I, except that squares are divided instead of merged. After $O(\log n)$ steps the border elements of the subsquares cover all PEs of the MCC, and hence the distance from all PEs to m is known.

Since each step of the $O(\log n)$ steps of recursion requires $O(k)$ time, the entire algorithm requires $O(n)$ time. Also, since two PEs of a figure may be at opposite corners of the MCC, implying a communication cost of $O(n)$, this algorithm is worst case optimal.

Internal Distance Related Problems:

Given two marked PEs a and b , if $\text{DIST}(a,b)$ is finite, then the following have been solved in $O(n)$ time:

- (1) Mark all minimal a - b paths.
- (2) Mark a single minimal a - b path.
- (3) Determine the number of all minimal a - b paths.

Problem (1) is solved by executing the Internal Distance Problem's algorithm once for $m=a$, and again for $m=b$. The value of $\text{DIST}(a,b)$ is broadcast to each PE c , which also knows $\text{DIST}(c,a)$ and $\text{DIST}(c,b)$ from the previous two steps. Every PE c for which $\text{DIST}(a,c) + \text{DIST}(c,b) = \text{DIST}(a,b)$ creates an edge (c,d) to each neighbor d such that $\text{DIST}(d,b) = \text{DIST}(c,b) - 1$. Problem (2) is solved using (1) and Problem (3) is solved using (1) and an algorithm similar to the Internal Distance Problem, except that the matrices now store the number of minimal a - b paths.

The above problems can also be solved between sets of PEs A and B , where the distance between the two sets is defined as the minimum distance between a PE of A and a PE of B . Variations, such as using positive weights between neighboring PEs, may also be solved using the techniques in the above algorithms.

Mark Extreme Points Problem: Identify, simultaneously for each connected component S , the extreme points (defined in Section 2.2.1) of S .

Notice that a digitized figure labeled S has at most two extreme points in any row of the MCC. Therefore there are at most $2n$ extreme points for any figure stored on the MCC. Each PE first determines whether it is the leftmost or rightmost PE of its label in its row. This is done by performing row rotations of the record $\langle \text{PE ID}, \text{PE label} \rangle$. The right and leftmost PEs then form the record $R = \langle \text{PE label}, \text{PE ID} \rangle$, while all others form the record $\langle \infty, \infty \rangle$. These records are sorted into snake-like row-major order by label. The labels of the at most $2n$ candidate extreme points for a figure are now consecutive in the snake-like order. The PE ID is used to determine the lattice coordinates of the PEs, and the coordinates for non-infinite labels are rotated in snake-like fashion among the PEs whose record R contains the same label.

Let X be a candidate extreme point sorted into snake-like order. Let the first two of the rotating points Y and U be arranged such that the angle α from XY to XU is convex. For all other points W that arrive, if the angle XW falls between XY and XU they are ignored, otherwise XY is replaced by XW if it lies between XW and XU , or XU is replaced by XW if it lies between XW and XY . If angle α from XY to XU is now reflex, then X is not an extreme point. If, at the end of the snake-like rotations of the candidate extreme points, the

angle α is convex, then X is an extreme point. A RAW is then made to the PE ID of record R returning the decision on whether that PE is an extreme point in its component. Since a constant amount of work is required between the at most $2n$ steps in the snake-like rotation of PE coordinates, and row rotations, sorting and RAW require $O(n)$ time, the algorithm has an execution time of $O(n)$.

This algorithm also works if a figure digitizes into disconnected components, but these components must then be labeled as an input since Nassimi and Sahni's algorithm [47] will label them as separate components.

Problems Solved Using Extreme Points:

- (4) Each connected component S , simultaneously in $O(n)$ time, can decide if it is convex.
- (5) In $O(n)$ time, all components can decide whether their convex hull contains PEs of another component. Further, each component can decide if any of its PEs are contained in the convex hull of another.
- (6) In $O(n)$ time, it can be decided if two sets of pixels labeled A and B are linearly separable. That is, if there exists a straight line separating the two sets of pixels.
- (7) In $O(n)$ time, all PEs labeled A , for all labels A simultaneously, can determine a smallest enclosing rectangle for their component. Further, in $O(n)$ time every PE labeled A can determine the same rectangle as every other processor labeled A .
- (8) In $O(n)$ time, simultaneously for all components, each labeled processor s can determine the external diameter of its component, where the distance metric used may be any non-decreasing (i.e. monotone) distance metric.
- (9) In $O(n)$ time, for any monotone metric, the all-points farthest point problem can be solved, in which for each labeled PE, the greatest distance to a PE with the same label is found.
- (10) In $O(n)$ time, each processor can determine the distance to the nearest differing labeled PE, if such a PE exists. (All points nearest-neighbor problem).

Additionally, problems such finding the largest empty circle, the smallest enclosing circle, the restricted planar center, the restricted planar radius, and the all points fixed-radius nearest neighbor problem have been solved in [41]. Details on some of the above problems are now presented.

Problem (4) is solved by first performing row rotations. This finds inconvexities, where PEs of the same label are separated by empty PEs, or PEs of a differing label; the decision found for the labels in each row is remembered. This step detects 'O' shaped inconvexities. Then, the Mark Extreme Points problem is run, except that the left neighbor

of the leftmost PE, and the right neighbor of the rightmost PE are now tested to see whether all of them would be extreme points if they were included in the component. This will detect 'Z' shaped inconvexities, where PEs that are candidate extreme points in a row, are actually contained by the convex hull of the figure. The decision for the 'O'-shaped inconvexities is passed with the left and right neighbors, and combined with the decision on the 'Z' shaped ones. The results are returned by a RAW operation to the neighbors, where they are rotated in their rows so that all PEs of the component know the answer.

Problem (5) is solved using ideas from Problem (4), and Problem (6) is solved by noting that two sets are linearly separable if and only if their convex hulls are disjoint. Smallest enclosing rectangles (7) are found by determining the edges of the convex hull of the object using a variation of the extreme points algorithm. A smallest rectangle is then found for each edge e of the convex hull, where one side of the box must pass through that edge. The points r, s, t of the other three bounding sides, see Figure 2.10, are found in the step where data is rotated in snake-like order. Problems (8) and (9) are also solved by using the extreme points.

The nearest neighbor problem (10) is solved by performing row rotations to find the nearest differing labeled PE on both sides of all PEs in the row. This data is then rotated in the columns. Any monotone distance metric that requires constant time to compute may be used.

Other Image Space Problems:

Lee and Preparata [31] discuss the problem of computing the *depth* and *convex layers* of a set S of n points in the plane. The computation involves finding the sequence $(CH(S_i) \mid i = 1, \dots, d)$ where $S_i = S_{i-1} - V(CH(S_{i-1}))$, $i = 1, \dots, d$ where $V(CH(T))$ are the set of vertices of T used in $CH(T)$. This sequence defines the convex layers of S , and the depth of S is the number d of these layers. This problem is needed in statistical estimation, since outlying points (or noise) should be eliminated before the estimation is computed. Dehne et. al. [14] present an $O(n)$ algorithm to compute the rectilinearly convex layers and depth of each pixel for an object digitized on an $n \times n$ systolic screen. That is, for the digitized object X , denote by X^0 all the border elements of image X . Then, the k^{th} rectilinear convex hull $RCH^k(S)$ of an image S is defined as $RCH^k(S) = RCH(RCH^{k-1}(S) \cap S) \cup RCH^{k-1}(S)^0$. The algorithm presented computes all k^{th} rectilinear convex hulls of an image, for which the depth k of each pixel of the object is stored in its PE. Furthermore, if we say a pixel $p=(i,j)$ *dominates* another pixel $q=(s,t)$ if $i \geq s$ and $j \geq t$, then p is called *maximal* in a set S of occupied pixels if there is no other pixel $r \in S$ that dominates it. We

may then define the 1st m-contour of S , denoted by $\text{MAX}(S)$, as all maximal pixels of S . The k^{th} m-contour of S , denoted by $\text{MAX}(S, k)$, may be recursively defined as: $\text{MAX}(S, k) = \text{MAX}(S - \text{MAX}(S, k-1))$, where $\text{MAX}(S, 1) = \text{MAX}(S)$. An $O(n)$ algorithm for finding all k^{th} m-contours of a set S may also be found in [14].

Kumar and Eshaghian [25] and Lodi and Pagli [33] use the Mesh of Trees (MOT) network to solve computational geometry problems. Some problems solved in [25] are component labeling, drawing digitized straight lines, computing convexity, determining distances, etc..., for pictures digitized on the mesh. Lodi and Pagli [33] find the visibility between n vertical line segments in a plane. The above problems require $O(\log n)$ time on the MOT.

Stout [63] presents lower bounds for non-local image processing problems solved in the Pyramid computer (Section 1.4), i.e. for problems in which there is no fixed bound on the distance that information must travel. $O(\log n)$ algorithms for detecting concavities in an image, and for determining the minimum distance between any pair of black pixels (closest pair) have been constructed. An algorithm for finding the extreme points of an image is presented that executes in $O(\log^3 n)$ time. Stout notes that strongly global operations such as sorting, where $\Omega(n^2)$ data elements cross a vertical cut through the center of the Pyramid, require $\Omega(n)$ time on the Pyramid since any cut has at most $O(n)$ connections. He presents standard Pyramid operations, that operate for global problems that use bottom-up divide-and-conquer solutions. Low-level image processing problems are presented in the next section.

2.2.4. Image Processing Problems.

Due to the easy mapping of a screen to the MCC, as described in Section 1.4, the MCC has been frequently used for image processing. In addition, many image processing problems have been solved on the Pyramid architecture presented in Section 1.4. Applications in biomedical image processing and pattern recognition are given by Sternberg [61] and Toussaint [67]. This section presents many MCC solutions for low-level image processing and line-detection problems, followed by a list of problems solved on the Pyramid.

Low level image processing problems such as point and local operations on a grey-scale image stored on an $n \times n$ systolic screen have been addressed by Kushner et. al. [29]. Algorithms for the 128x128 Massively Parallel Processor MPP [4] to compute a histogram of the number of pixels at a certain grey-scale value, a 'co-occurrence matrix' which is essentially a histogram of all pairs of grey levels, and to compute the two-dimensional discrete Fourier transform have been implemented. The histograms are stored by keeping

the buckets for each grey level in a column. That is row i of each column will count grey scale i . The grey scales in each column are added to their bucket by column rotations, the sums for each bucket are then added by performing row rotations. The Fourier transform is solved by computing the discrete transform row-wise then column-wise. These operations therefore require $O(n)$ time.

Kushner also compares the actual execution time for running these algorithms on the MPP, as opposed to a sequential VAX and notes that the problem size cannot be large due to the memory limitations of the MPP (1024 bits per PE). He determines that the histogram operation time on the MPP is only a marginal improvement over the time for the same operation on the VAX, although local operations are usually more than 10 times faster on the MPP as opposed to the VAX. These finding may be due to the fact that the amount of data processed is small, due to the small amount of storage present in a PE of the MPP, and the size of the MPP. This implies that a speed up by a factor of $128 \times 128 = 16384$ PEs will be encountered for local operations, but this factor is reduced to 128 for the histogram and transform operations since the 128 parallel sums for the rows and columns are performed sequentially. If more computation is present at each step in an algorithm, then the improvement will be more dramatic because the computations are done in parallel.

Guerra and Hambruch [18] present a more global image processing algorithm that detects lines in an image stored on an $n \times n$ systolic screen. They use the Hough Transform to transform each occupied pixel with coordinates (x, y) to a line with parameters (θ, ρ) , where ρ is the perpendicular distance to the line passing through (x, y) with angle θ from the positive x -axis. Points lying on a line are transformed into lines that intersect at a point in the parameter space. The number of values for θ and ρ is discretized giving k and m values respectively, and two algorithms, one that runs in time $O(k\sqrt{m} + n)$ and the other in time $O(n + k)$, are presented. If $k = m = n$ the first algorithm requires $O(n\sqrt{n})$ time, while the second requires $O(n)$ time.

Rosenfeld [56] describes pyramid techniques for extracting global structures such as features, regions and patterns from an image in $O(\log d)$ time where d is the image diameter. Some features recognized are spots, streaks, curves, edges of all sizes, and closed contours (by detecting curves or edges).

2.2.5. Graph Problems.

The most frequently used algorithm in graph theory is the determination of the connected components of a figure. Nassimi and Sahni [47] solve for $O(n^2)$ vertices: the connected components for degree d graphs (in $O(n \log n)$), the connected components for degree 2

graphs in $O(n)$, and the connected ones (used to label digitized objects) in $O(n)$ time. The algorithms presented generalize to higher dimension meshes.

Atallah and Kosaraju [3] solve a variety of graph problems such as, determining the length of a shortest cycle, finding a minimum spanning tree, and finding bridges and articulation points for $O(n)$ vertices on an $n \times n$ MCC in $O(n)$ time. They store the adjacency matrix to solve these problems, which gives the $O(n)$ bound on the number of vertices for the graphs used. Atallah and Hambrusch [2] store arcs instead, implying that $O(n^2)$ arcs of a graph may be stored. Their algorithms, which may be generalized to higher dimensional meshes, determine the depth, height, number of descendants, and preorder (inorder and postorder) number of every node of a tree T in $O(n)$ time. They also compute the Minimax value of a game tree. Hambrusch [20] presents graph algorithms to find bridge- and bi-connected components of undirected graphs of n^2 vertices and e edges. For bi-connectivity, $O(n^3)$ time is used if input is stored in adjacency matrix form, and $O(n^3 + e)$ time for input in the form of edges. In these algorithms, the $n \times n$ MCC receives n^4 (respectively e) inputs in the form of input waves, since the algorithms cannot explicitly store the entire input on the mesh.

This concludes the presentation of existing MCC algorithms. The next chapter presents new algorithms to solve visibility problems in computational geometry. These algorithms use a sweep approach, as opposed to the divide-and-conquer approach used in many of the algorithms discussed in this survey.

3. VISIBILITY PROBLEMS.

In this chapter we use the systolic screen defined in Section 1.3 to solve visibility problems in image space. Two-dimensional arrays of processors [64] have long been proposed for image processing [55] since images can be naturally mapped to an MCC. In image processing, arrays of processors are mostly used for low level local operations such as image restoration, noise removal, computation of connected components and edge detection (see [7,18,23,47,49,54,55]). The algorithms described in Chapter 2 illustrate that the MCC need not be used exclusively for low-level image processing, and can indeed solve more complex problems. This chapter examines problems in the area of visibility from the object and image space perspective.

In the *parallel visibility* problem, a light source located at infinity emits rays parallel to some direction d . In the *perspective visibility* problem, the light source is a point source p which emits rays in every direction and is located at a finite distance from some objects. In both cases we desire the portions of the objects that are visible from the light source. Parallel visibility has been used to solve the *sequential separability* of a set of objects in a direction d (see [13,57]). That is, to find an ordering, if it exists, in which the objects may be translated one by one to infinity in direction d so that no two objects intersect at any time in the translation. A parallel algorithm that finds the movability graph defined in [13] to find an ordering of the objects is given. An algorithm is presented for finding all rectangularly visible [44] pairs of black PEs, in which two black PEs are considered visible if the rectangle defined by using the PEs as opposite corners contains no other black PE. Due to the ordered nature of images digitized on a systolic screen, algorithms that sweep data across the screen were designed for these problems.

In planning the path of a robot through a set of obstacles [34], the configuration space of the obstacles is constructed in order that the robot may be reduced to a point. The last section of this chapter presents an algorithm to compute the configuration space of a set of digitized obstacles from which a shortest path that directs the robot (now a point) through the obstacles may be found. A data encoding technique is used to encode the robot. This data is then pipelined through the MCC.

Systolic versions of visibility problems solved in object space by Dehne [12] are discussed in the next section. He solves the perspective visibility problem from a point p inside a simple polygon with holes, and the parallel visibility problem in direction d to the exterior of a simple polygon. He also constructs the visibility hulls for two simple polygons using the parallel visibility algorithm to determine whether they are separable in direction d .

3.1. OBJECT SPACE VISIBILITY.

The problems solved by Dehne [12] are:

- I Compute the perspective visibility from a point p interior to a n^2 -vertex polygon with holes. (The holes themselves are simple polygons and no polygon intersects another.)
- II Compute parallel visibility in direction d to the exterior of an n^2 -vertex simple polygon.
- III Determine whether two n^2 -vertex polygons are separable by a single translation in direction d .

Problems I, II, and III have been solved in $O(n)$ time on an nxn MCC. The algorithm for Problem I will be outlined first since the algorithms for the remaining problems make use of the results. Some definitions for Problem I are presented.

A point q is visible from p if the line segment pq does not intersect an edge of the simple polygon P . All edges of P are directed so that the interior of P (exterior of the holes) lies to the left of the edge. For a directed edge $e=(s,f)$, s is the *start vertex* and f is the *end vertex*. An edge of P is a *luff edge* if p lies on or to the left of an infinite line through e directed from s to f , and is a *lee edge* otherwise. The *visible interval* of an edge, will be the segment of the edge that is visible from p .

Figure 3.1. The visibility polygon of a simple polygon with holes.

Initially, all PEs will store an arbitrary edge of P , and the point p from which visibility is being computed. Upon termination of the algorithm, a set of all visible intervals for the edges will be stored one per PE in arbitrary PEs. An ordered list of vertices to construct the *visibility polygon*, which is the polygon that encloses all points q that are visible from

p , can be constructed by sorting the vertices of the visibility intervals by their polar coordinates with respect to p (hereto referred to as *polar p -coordinates*). An example of a visibility polygon, as well as some examples of luff and lee edges, is given in Figure 3.1.

Dehne uses a divide-and-conquer strategy to solve the perspective visibility problem. Initially, two horizontal rays, R_l and R_r , directed to the left and right of p respectively, are constructed. The perspective visibility problem can be solved in parallel for those edges that lie above, and those that lie below these rays. The solution is presented for the edges lying below the rays. The edges below R_l and R_r will be recursively divided into two equal size subsets by selecting a ray R from p to the vertex of an edge that divides the edges, until a small constant number of edges remain in a subset. The visibility interval for these edges is then found. Since edges will be split if they cross R , the visibility intervals found are re-linked in the merge steps.

During execution of the algorithm, edges will be labeled *active* if visible intervals must be computed for them, and *passive* otherwise. Initially, all edges are flagged as active. By definition, the lee edges are not visible from p . In fact, the luff edges themselves may only have been used to decide the visibility intervals, except that some luff edges are shortened or eliminated during the algorithm's execution and this would result in an incorrect visibility computation at the last split step.

A split of an edge set is done by sorting the vertices of the edges by their polar p -coordinates and choosing the vertex v that divides the set of vertices in half. A ray R directed from p through v is constructed which divides the sector defined by R_l and R_r into two left and right sectors called $Sect_l$ and $Sect_r$ respectively. The mesh is recursively split, as described in Section 2.1.3, and those edges whose start and end vertices lie in the same sector are sent to the corresponding submeshes that store the edges in its sector. The active lee edges that cross R are split, and the resulting two edges are stored as passive edges in their respective sector. Notice that passive edges are not split in successive recursive steps, therefore, the number of passive edges is at most double the number of lee edges. The active luff edges whose vertices lie in each sector must now be split. Since active luff edges may be split at each recursive step, care must be taken to ensure that the problem size in each subdivision decreases by a certain fraction at each recursive step. Dehne addresses this problem by using the passive edges to remove the active edges in a sector that are not visible, and determines the visible intervals on the remaining luff edges by shortening or eliminating those that are blocked by other luff edges in the sector. These operations result in fewer and shorter edges of which only one will intersect a ray in the sector. If the number of edges in each sector are not approximately equal, a single rebalancing step is

performed in which R is recomputed based on the remaining edges.

This completes the discussion of the splitting steps. In merging the resulting visibility intervals, all that needs to be done is to sort the vertices of the intervals by their distance to p . Those intervals that were split by a ray will now be consecutive in the ordering, and may be merged. Thus, Problem I is solved in $O(n)$ time since the (constant number of) operations at each step require $O(k)$ time, where k is the maximum width of the submeshes. The number of split steps is $O(\log n)$ since care is taken to reduce the problem size at each split step.

Figure 3.2. Computation of parallel visibility and the visibility hull.

The solution to the parallel visibility problem in direction d is accomplished by rotating P until d is parallel to the x -axis, and then translating the result such that all vertices of P have positive x -coordinates. One may then select a point p on the negative x -axis, transform P using a parallel to perspective transform (the inverse of the perspective to parallel transform presented in [17]), thereby solving the perspective visibility problem, and then perform the inverse transform of the visible segments to arrive at the parallel visibility intervals. Notice that the definition of the luff and lee edges is the reverse of the one in the perspective visibility algorithm since the visibility is computed to the exterior of the polygon. The visibility intervals for the simple polygon in Figure 3.2 are illustrated by heavy bold lines. If parallel visibility is computed in directions d and $-d$, and the resulting visibility intervals are connected into a simple polygon as illustrated in Figure 3.2, then the resulting polygon P' is the *visibility hull* of P in direction d . In Problem III, the determination of the separability of two simple polygons in direction d is found by computing the visibility hulls for the two polygons. If they do not intersect, then the two

polygons are separable in direction d . The detection of intersection for $O(n^2)$ line segments has been solved in $O(n)$ time by Miller and Stout [40].

The sections that follow present versions of the parallel and perspective visibility problems for a set of digitized images. Solutions to some separability problems are also presented. These problems are different than those solved by Dehne [12] in object space, as the objects are represented by pixels rather than line segments. The parts of each pixel that are visible must be determined, and the images may be of any shape. In addition, for perspective visibility, the viewpoint p may be taken from any point and the algorithm is the same. Another factor is that images on a systolic screen are stored in a more orderly fashion than a set of line segments, allowing simpler algorithms to be developed.

3.2. DIGITIZED PARALLEL VISIBILITY.

Two-dimensional parallel (or perspective) visibility has been computed (sequentially) to the exterior or interior of a simple polygon by Dean and Sack [9]. These algorithms may be applied to the hidden-line elimination problem [65] where two-dimensional projections of three-dimensional objects are to be displayed. Srihari [60] presents a representation of three-dimensional digital images to allow cross-sections and shaded surface reconstruction on a graphics terminal. These images may be generated using computed tomography or a scanning electron microscope, for example, producing a 3D array of numbers. The parallel visibility algorithm presented in this section is extended for images digitized on a three-dimensional systolic screen in the last subsection. A cross-section may be displayed on a screen from a three-dimensional image by ignoring the visibility of the PEs before the cut-plane, and mapping the results of the parallel visibility for the PEs on this plane onto the screen. The definition of parallel visibility for digitized pictures is presented first. Notice that if the actual boundary of an object is stored at each PE, instead of a flag indicating whether the pixel is on or off, the actual visibility may also be computed.

The *parallel visibility problem* is stated as follows: Given a light source located at infinity emitting rays of light parallel to a specified direction d , compute the portion of each pixel of an object that is illuminated by the light source.

In the following section we discuss a sweep method which will be used to solve this parallel visibility problem. In Section 3.2.3, the implementation of the algorithm on a systolic screen is described in some detail, and its time complexity is analyzed.

3.2.2. Systolic Parallel Visibility.

Given a direction d , the *parallel visibility* problem on the systolic screen is the problem of determining for each object in the screen which portion of each pixel is *visible* in the given direction; that is, to determine the region of each pixel illuminated by a light source located at infinity, which emits rays parallel to the given direction d . Pixels that are part of a digitized object are called *occupied* pixels and they block the light rays.

The first approach that may suggest itself is to follow the path, in direction $-d$, of the Bresenham digitization [17] of a line through the lattice point of a pixel, and return to that pixel once the boundary of the systolic screen is encountered. That is, each PE p creates a record with its ID and a flag, and sends it up along this path. The record returns by the same path to p with its flag set to true if any occupied pixel was encountered on the Bresenham line that is parallel to direction d and passes through p 's lattice point. This approach doesn't find the exact portion of each pixel that is blocked; since the pixels on a Bresenham line must only intersect a certain fraction of the line. Due to the fact that Bresenham's algorithm does not digitize all pixels that intersect the line through PE p , not all PEs that block p are traversed. Even if this were so (we could use a thick line in the Bresenham algorithm), a systolic algorithm requires that data be sent in a regular fashion. Although each record initially follows the same sequence of routing steps for each digitized line (since the lines are parallel), some records reach the boundary of the systolic screen before others. These records cannot wait until all records are ready to return, because each PE has constant memory (at worst $1/2n$ records must wait). As a result, the computation of the next pixel to which a record must be sent is difficult since the formula to compute it has many conditions. A method that divides the systolic screen into strips was devised since it resulted in a fixed number of motions among those possible to send all records, and in an exact solution as to the portion of a pixel that is visible.

A *strip* S is a region of the systolic screen bounded by two lines parallel to direction d . In computing the visible portions of a pixel, the systolic screen will be divided into fixed-size consecutive strips $S_1, S_2, \dots, S_m$. For a strip S_k , denote by s_k and s_{k+1} the directed lines parallel to d form the *boundaries* of S_k . Since the visibility of a strip is affected only by the pixels intersecting that strip, we will consider only the problem of computing visibility in a strip. The *visibility information* for a strip S_k is an interval I_k perpendicular to the direction d , whose length is initially set to the width w of the entire strip (see Figure 3.3). The strip will be scanned; during this process, the visibility information will be updated so that, at any point in time, the interval indicates which parts of the strip are still visible from the light source. The actual value of the width w of a strip depends on the direction d . The method for choosing w will be discussed later.

Figure 3.3. A strip S_k , and its associated interval I_k of width w .

Initially, the entire strip is considered visible, and the length of the interval is w . The interval is passed along the strip in direction d , and is updated whenever an occupied pixel is encountered. For example, in Figure 3.3, the interval I_k initially covers the entire strip S_k . In Figure 3.4, the same strip S_k is shown with only the occupied pixels being indicated. After the first occupied pixel is encountered, I_k is reduced to I'_k . It is further reduced when the next occupied pixel is encountered, resulting in the final interval I''_k .

Figure 3.4. Update of interval I_k when scanning strip S_k .

The updated visibility information stored in I_k must be sent to all pixels intersecting strip S_k . Let $P_k = \{p_1, p_2, \dots, p_m\}$ be the set of all pixels intersecting S_k . If pixel p_i blocks pixel p_j , to correctly compute the visibility in S_k it must be ensured that interval I_k reaches p_i (and is updated there) *before* it reaches p_j . A linear ordering $\mu(P_k)$ of the elements of P_k is said to be *correct* if $p_i <_\mu p_j$ whenever p_i blocks p_j . If interval I_k travels in strip S_k according to a correct linear ordering, a correct computation of the visibility is ensured. In particular, a correct ordering of the pixels in P_k is the ordering $\pi(P_k)$ induced by the

projections of the centers of the pixels onto s_k . This particular order of the pixels in P_k can easily be proved correct for any two PEs p and q of P_k since only the pixel area of any PE p whose projected center occurs before another's, say q , may block q 's pixel area by the physical geometry of the mesh and the fact that parallel rays are used. The case where p and q 's projections are equal is discussed in Section 3.2.3. With respect to $\pi(P_k)$, we may now define the *successor* [*predecessor*], denoted by $\text{succ}(\text{PE})$ [$\text{pred}(\text{PE})$], of a PE in a strip S_k as the pixel that follows [precedes] PE in $\pi(P_k)$.

We shall now consider the requirements on the width of the strip. The width w must be selected so as to satisfy Requirements 3.1 and 3.2 below:

Requirement 3.1: The size of the messages sent by each PE is bounded by a small constant.

Note that this requirement is easily met by avoiding fragmentation of I_k ; that is, by ensuring that interval I_k remains continuous.

Requirement 3.2: The number of messages a PE sends or receives at each time step is bounded by a constant.

Proposition 3.1: Let the width w equal $\sqrt{2} \cos(45^\circ - \beta)$ for angles $\beta \in [0, 90]$ measured from the positive x -axis to the direction d . Requirements 1 and 2 are then met.

Proof: Choosing $w = \sqrt{2} \cos(45^\circ - \beta)$ ensures that any pixel of the mesh intersects at most two strips (see Figure 3.5). This implies that a PE sends and receives at most two messages; thus, Requirement 3.2 is met. Fragmentation of interval I_k can only occur if a pixel is totally contained inside a strip, possibly leaving gaps at either end as shown in Figure 3.6. However, this situation can never occur, for any direction d , since the boundaries of a strip form lines of support to a pixel region, which ensures that in general, any pixel of the MCC either blocks an entire strip, or one side of a strip; thus, Requirement 3.1 is met. $\square$

Figure 3.5. Definition of width w of a strip.

Directions of visibility other than those described by $\beta=[0, 90]$ are analogous and are thus omitted here.

Figure 3.6. Subdivision of an interval.

We shall now describe the general strategy for solving the parallel visibility problem on a systolic screen. In the presentation, the general case of a PE intersecting two strips is considered; the case of a PE intersecting only one strip is implied in the discussion.

General Strategy

1. Given the direction d of the light rays, each PE computes the strips in which it lies, and determines for each strip its successor and predecessor pixels in the linear ordering $\pi(P_k)$. (This computation involves the projection of a constant number of pixels onto s_k , as discussed later.) Each PE then computes interval I_k (set initially to be the entire strip width) for each of the strips it intersects. If the PE is occupied, it transmits to its successor in $\pi(P_k)$ the portion of I_k which it does not block.
2. Upon receiving an interval from his predecessor in $\pi(P_k)$, a PE uses it to update its own interval, and sends the resulting interval I'_k to its successor in $\pi(P_k)$.
3. Processing terminates when no further messages are sent in the mesh. As shown later, this occurs after at most $O(n)$ time steps. Upon termination, the final interval (one for each strip) available at a PE represents the portion of the pixel which is illuminated within that strip.

3.2.3. Mesh Implementation Of Parallel Visibility.

The algorithm assumes that the image has been digitized on the systolic array. The direction d of the incoming light is sent to one PE, and then broadcast to the other PEs in the MCC.

If PE x finds that there are two or more PE's whose projected centers coincide when computing its successor in a strip, then the PE (i, j) with minimum j is selected as x 's successor. Should the projected center of a PE p coincide with the projected center of x , and its j index is less than that of x , then p is not the successor of x . A similar approach is used to select the predecessor PE. This ensures that infinite loops do not result in the

selection of a successor.

The PEs at the edges of the screen may not have successors or predecessors. To ensure simple and symmetric computation, the two outer border layers of pixels on the systolic screen should contain only unoccupied pixels. These PE's perform the algorithm, but they are not part of the final result.

The pseudo-code below gives the operation of a PE x . Note that any message contains identification of both source and destination PE. Subscript i in $succ_i(x)$ [$pred_i(x)$] will indicate the successor [predecessor] of x in strip S_i provided that x intersects strip S_i .

PARALLEL VISIBILITY ALGORITHM (For PE x)

1. if direction d is received then
 - * calculate the width w and compute strips S_k and S_{k+1} that are intersected.
 - * compute $succ_k(x)$, $pred_k(x)$, $succ_{k+1}(x)$, and $pred_{k+1}(x)$ for strips S_k and S_{k+1} .
 - * set I_k and I_{k+1} to be of width w .
 - * if x is occupied then
 - * remove from I_k and I_{k+1} the portion blocked by x .
 - * send updated intervals I_k and I_{k+1} to $succ_k(x)$ and $succ_{k+1}(x)$ respectively.
- endif
2. if interval I is received from $pred_h(x)$ (where $h=k$ or $h=k+1$) then
 - * update I_h by intersecting it with I .
 - * send the updated I_h to $succ_h(x)$.
- endif
3. if received interval I is not for x then
 - * route I to its destination
- endif
4. if the number of elapsed time steps is equal to the number of steps for completion then
 - * I_k and I_{k+1} are the regions of x visible from the light source.
- endif

END PARALLEL VISIBILITY.

Lemma 3.1 will ensure that the number of routing steps required to route an interval from a PE to its successor in a strip is constant. Another result of Lemma 3.1 is that $succ$ (PE) and $pred$ (PE) may be computed in constant time.

Lemma 3.1. *The distance (in the L_1 , or Manhattan metric) between a PE x and its successor in $\pi(P_k)$ of strip S_k is at most 3.*

Proof: Three distinct cases may occur.

Case 1: the boundary line s_k touches PE x at the top left-hand corner. This implies, by definition of w , that s_{k+1} touches x at the bottom right-hand corner. If the direction is changed continuously from 0° to 90° , the successor of x is either the left or bottom neighboring PE (see Figure 3.7).

Case 2: the boundary line s_k touches x almost at the bottom right-hand corner. This implies, by definition of w , that if the direction is changed continuously from 0° to 90° , the possible successor of x is at distance $d \leq 3$ from x (see Figure 3.8).

Case 3: the boundary line s_k lies between the top left-hand corner and the bottom right-hand corner of x . This case is easily reduced to the above two cases. $\square$

Figure 3.7. S_k intersects the top left-hand corner of a pixel.

Figure 3.8. S_k touches the bottom right-hand corner of a pixel.

We can now prove the overall time complexity of the proposed algorithm.

Lemma 3.2. *The Parallel Visibility Algorithm requires $O(n)$ time on a $n \times n$ MCC.*

Proof: Once the value of direction d is received (after a broadcast), each step (1-3) of the algorithm requires a constant number of time steps. In fact, the successors and predecessors of a PE can be computed in constant time, and the time to route an interval to a successor requires no more than 3 routing steps (by Lemma 3.1). Furthermore, since a PE intersects at most two strips, during each step of the algorithm a constant number of fixed size messages will be sent (by Requirements 3.1 and 3.2). Finally, there are $O(n)$ PE's intersecting each strip and, in the worst case, an interval travels down the entire strip. $\square$

3.2.4. Three-dimensional Extensions to Parallel Visibility.

The digitized parallel visibility algorithm may easily be extended to three dimensional $n \times n \times n$ systolic screens by defining three dimensional strips, now referred to as strip volumes. Only directions d of parallel visibility that lie in a principal plane may be used in order to ensure that the size of the visibility information sent in each strip volume remains a constant. A strip volume is constructed by creating an initial strip volume with the same dimensions and orientation of a vertical column of the systolic screen, then rotating this column about a pixel so that the strip volume remains in the principal plane to which d is parallel. During the rotation, the strip volume is expanded to ensure that the pixel is contained in it, and that at least two pairs of opposite edges of the pixel touch one of its planes. Figure 3.9(a) illustrates this definition of a strip volume using a unit-cube pixel, where the indicated edges of the cube touch a plane of the strip volume. A strip volume now has two widths, w_a and w_b which is the perpendicular distance between the two parallel sides of a strip volume as illustrated in Figure 3.9(a). Width w_a is 1 and width w_b is between 1 and the square root of 2, since the projection of the unit pixel that defines a strip volume onto the side of the strip volume whose width is w_b will result in the picture shown in Figure 3.5.

(a) A strip volume.

(b) Strip volumes that intersect a pixel.

Figure 3.9. Three-dimensional parallel visibility strips.

If the three-dimensional systolic screen is projected onto the principal plane in which direction d lies, then the outermost projected edges of a strip volume are the same as those illustrated in Figures 3.7 and 3.8. This implies that Lemma 3.1 is extendible to the three-dimensional case to give 3 as the maximum L_1 distance to a successor in a strip volume. Any pixel will intersect at most two strip volumes, since the strip volumes lie in a principal plane of the systolic screen. These observations about the widths and number of strip volumes intersected by a pixel imply that Requirements 3.1 and 3.2 are met. The succes-

sors and predecessors of a PE in a strip volume are defined as the projections of the centers of the pixels on a face of the strip volume, followed by a projection onto the face perpendicular to the one previously chosen. Many PEs may now have the same projection, and any method to order these may be used.

The visibility information transmitted from one PE to the next is defined as an interval perpendicular to direction d and parallel to the principal plane in which d lies, and it is initially set to have a length of w_b . Only one interval is required per strip since the strip volumes' sides touch the two opposite sides of the PE that are parallel to d .

The fact that a successor is a constant L_1 routing distance away, and the fact that the requirements hold, imply an $O(n)$ three-dimensional parallel visibility algorithm for an $n \times n \times n$ systolic screen. This completes the digitized parallel visibility section. An application of this algorithm occurs in the unidirectional separability section below. Other examples of sweep techniques may be found in the rectangular visibility and digitized perspective visibility sections.

3.3. UNIDIRECTIONAL SEPARABILITY.

Motion or separability problems occur in areas such as robotics where, for example, an object such as a ball bearing is to be removed from a box of ball bearings and moved without colliding with other obstacles to a specific place; or in areas of computer graphics, such as animation, where solid objects must not pass through one another.

For unidirectional separability, objects are translated in a given direction one at a time in such an order that no object touches another. Nurmi [48] presents sequential algorithms (for 2D and 3D objects) to determine whether the translation is possible, and if so to find the order in which the objects may be translated. These problems are extensions of the unidirectional separability problem now defined. The *unidirectional separability problem* of a set Q of simple polygons is: Given a direction d and a polygon q of Q , q is *separable* from Q if q may be moved by a single translation to an arbitrary distance in direction d without the interior of q intersecting the interior of any of the polygons in $Q - q$. Variations of this problem, such as finding *all* directions in which p is separable from Q , have been investigated by [13,57,66] for polygons in object space. Reference [13] finds, for a direction d , all orderings of the polygons in Q so that the entire set Q is sequentially separable.

A set Q of objects on a systolic screen may be any digitized object such as simple polygons, circles, polygons with holes, etc., where the interior of each object is filled. A restriction on the set Q is that no two objects intersect (share the same pixel). An object q of Q is said to *collide* with another object t of Q if a pixel of q intersects a part of any pixel of t during a translational motion of q . The object q is *unidirectionally sequentially*

separable from Q in direction d , if q may be translated in a single motion in direction d to an arbitrary distance without colliding with an object in $Q-q$. The problems that are solved (for all q in Q) either on the MCC or by using a graph generated by the MCC are listed below.

Problem I: Does there exist an ordering for which the elements of Q are unidirectionally separable?

Problem I: Is q unidirectional separable from Q in direction d ?

Problem II: Find an ordering $(q_1, q_2, \dots, q_n)$ of the objects in Q such that q_i is unidirectional sequentially separable, in direction d , from $\{Q - q_j \mid 1 \leq j \leq i\}$.

The systolic algorithm presented later may easily be extended to three dimensions since it is a variation of the parallel visibility algorithm. Thus, the problems solved in 2D in this section are extendable for 3D images. Nurmi uses a dominance relation, where one object A dominates another object B if A must be translated before B . Dehne and Sack [13] use dominance relations between all objects to find a total ordering in which the objects may be separated. These dominances may be expressed in a movability graph as discussed in the next section.

3.3.1. Unidirectional Separability Using A Movability Graph.

The idea used to solve problems I and II is to generate a directed graph MG called a *movability graph* [13]. The vertices of MG consist of the objects $p_1, p_2, \dots, p_m$ of Q . A directed edge (or arc) $p_i \rightarrow p_j$ exists if, and only if, p_j must be translated in direction d before p_i . If no edge exists between p_i and p_j then they may be translated in any order in direction d . An example of a movability graph for a set of polygons in Figure 3.10 is shown in Figure 3.11, where the direction d of separability is towards positive infinity on the x -axis.

Figure 3.10. A set of polygons.

In [13] it is shown that if there exists at least one translational ordering of Q with respect to a given direction d , then the set of all translational orderings of Q with respect to d is given by the set of all topological sortings of graph MG . A *topological sorting* of a directed graph $G(V,E)$ is defined as a linear ordering of all vertices, such that, if there is an arc from vertex v_i to vertex v_j , then v_i appears before v_j in the linear ordering. Topological sortings apply to connected graphs, that is, graphs with one component, since there can be any ordering on a set of unconnected elements. Therefore the topological sort must be applied to each of the components of a disconnected graph G in order that a linear ordering for the entire graph be obtained.

Figure 3.11. The movability graph for Figure 3.10 in direction d .

Since the elements of differing components have no order-restrictions with respect to the other components, the topological order of two components may be interleaved in any way, as long as the order of the elements in each component is maintained. The listing of any one of the topological sortings of all components of MG will give an order in which the set Q may be unidirectionally sequentially separated, thereby answering Problem II. Problem I may be answered by checking to see if q has any edges directed to another object in Q . A possible ordering from the movability graph in Figure 3.11 is obtained by topologically sorting the two components, and appending the two sorted lists giving, for example, the ordering (4 2 3 5 6 1 7). The polygons that answer yes to Problem I for the graph in Figure 3.11 are polygons 2, 4, 6 and 7. Any of these polygons may be chosen as the first one to be moved.

Before the algorithm to topologically sort graph MG is applied, we must determine whether there are any cycles in MG . If there is a cycle, then there exists no ordering of MG for which the set is unidirectionally sequentially separable in direction d . A cycle involving two objects q and t implies q must be translated before t , and vice versa, which is impossible. Figure 3.12 shows the cyclic movability graph resulting if the polygons of Figure 3.10 are to be separated in a direction d that is directed towards negative infinity

along the y -axis. The cycle is between vertices 1 and 5 and means that there is no solution for Problem II. The graph of Figure 3.12 may still be used to solve Problem I for the objects in Q . Polygon 7 is the only polygon that will answer yes to Problem I since all other vertices have a directed edge leaving them.

Figure 3.12. The cyclic movability graph of Figure 3.10 in direction d .

Now that the graph used in solving the separability problems has been defined, we shall explain how to construct a related graph on the MCC. This graph will have some of the transitive dominance relations removed.

3.3.2. Separability Using The Systolic Screen.

This section discusses the computation of a movability graph for the separability of a set of objects digitized on an $n \times n$ systolic screen. A modified version of the systolic parallel visibility algorithm is used to solve problems I and II. If the direction of separability is d , then the systolic screen is divided into strips S_i , $i=1, 2, \dots, m$ whose boundaries are parallel and directed in direction $-d$. Each digitized object in Q now has a distinct label which is stored in all of its PEs. If the objects are unlabeled, the $O(n)$ algorithm for finding connected components on a $n \times n$ MCC [47] may be used to label the objects, provided no two different objects have neighboring PEs otherwise they are considered to be the same object.

If a modified version of the parallel visibility algorithm were used in which a label is sent with each interval I_k indicating the PE that blocks that region, then, as shown in Figure 3.13, a PE may have to store $O(n)$ arcs for the movability graph MG . Also, the movability graph MG requires worst-case $O(n^4)$ storage if $O(n^2)$ objects are present on the systolic screen. To ensure that a constant number of arcs are stored in each PE, and that only $O(n^2)$ arcs are generated in the worst case, additional information will be sent with I_k and most of the transitive arcs of MG will be eliminated. In fact, a graph MG' , which is the graph MG with the transitive arcs of relations within strips removed, is generated. MG' will have at most $4n^2$ edges, as we will demonstrate later. The way in which the movability graph MG'

may be used to solve problems II and III is shown in Section 3.3.3,

Figure 3.13. The black PE sees $n-1$ different objects in the strip.

The information passed in the parallel visibility algorithm has to be modified in order to be able to decide if an arc is transitive within a strip. The interval I_k for strip S_k , with boundaries s_k and s_{k+1} directed in direction $-d$, is now considered to have three sections. Interval I_k has values (i_l, i_r) denoting the left and right boundaries of I_k in S_k , and the interval is now used to indicate the region of the strip that is blocked. Additionally, three labels (l_1, l_2, l_3) are sent with I_k to indicate which object blocks the regions 1: $[s_k, i_l]$, 2: $[i_l, i_r]$, and 3: $(i_r, s_{k+1}]$ respectively (l_2 is the label of the PE from which the interval originates). Labels l_1 and l_3 correspond to the last occupied PEs to have blocked a part of (i_l, i_r) . If such PEs do not exist, a special *clear label* denoted by Ψ is used. The symbol I_k now denotes the tuple $((i_l, i_r), (l_1, l_2, l_3))$, the region (i_l, i_r) is referred to as the *interval of I_k* or simply the *interval*, and l_1, l_2 and l_3 are the *labels of I_k* . The occupied PE that created I_k is called the *originator of I_k* . See Figure 3.14 for an example of an interval.

The labels l_1 and l_3 are used to eliminate the transitive arcs in strip S_k while the label l_2 is used to generate arcs in MG' . If an occupied PE labeled x intersects region i , for $i=1,2,3$ of the interval of I_k , then the arc $x \rightarrow l_i$ is said to be *known* at x . The arcs $l_1 \rightarrow l_2$ and $l_3 \rightarrow l_2$ of I_k , unless a label is Ψ , are *known* in all cases. Arc $x \rightarrow l_2$ is *transitive* if arcs $x \rightarrow a$ and $a \rightarrow l_2$ exist among the known arcs of I_k for some label a not equal to Ψ or x ; the arc $x \rightarrow l_2$ is referred to as *nontransitive* otherwise, and will be stored at x .

Figure 3.14. The interval used in computing separability.

Initially, each occupied PE computes the interval region (i_l, i_r) and labels l_1, l_2, l_3 in the one or two strips it intersects, and transmits I_k to its successor PE in S_k . When an occupied PE with label x receives I_k from its predecessor in S_k , a check is made to see which regions of I_k it intersects. If it only intersects region 1 or 3 then I_k is simply sent on to the successor, otherwise, arc $x \rightarrow l_2$ is stored if it is nontransitive, in which case the interval (i_l, i_r) is updated by removing the parts that intersect x . If i_l is modified l_1 is set to label x , or if i_r is modified l_3 is set to label x . Note that if both i_l and i_r are modified, then the interval must be empty. If the interval (i_l, i_r) is non-empty, the updated I_k is sent to the successor of x in S_k . All unoccupied PEs that receive intervals I_k simply send them on to their successors. After all intervals I_k have passed through their strip, the edges of the movability graph MG' for direction d are stored in the PEs of the systolic screen.

Proposition 3.2. *The known arcs of I_k that were received by an occupied PE p labeled x in strip S_k are sufficient to determine whether the arc $x \rightarrow l_2$ exists and whether it is transitive in S_k .*

Proof: Arc $x \rightarrow l_2$ exists if region (i_l, i_r) intersects p 's pixel region. It is transitive if region 1 intersects p 's pixel region and l_1 is not equal to Y or x , since the arc $x \rightarrow l_1$ and the known arc $l_1 \rightarrow l_2$ imply that $x \rightarrow l_2$. The same argument holds for region 3 intersecting p 's pixel region. $\square$

Proposition 3.3: *At most four arcs need to be stored at any occupied PE of the systolic screen.*

Proof: The width w of a strip implies that either a PE intersects the entire strip or one side of the strip. If PE x intersects the entire strip or the left side of strip S_k , then the occupied PE y that occurs first after x in S_k and intersects the left side of S_k must store the arc $y \rightarrow x$. Furthermore, since PE y blocks the "left" region of the strip, arcs involving PE x and PE's z that occur after PE y in S_k and intersect the left side of S_k are not stored since they are transitive. The same argument applies to PEs that intersect the right side of S_k . An occupied PE may be blocked by PEs that intersect the left or right side of S_k , depending on its position in S_k . Therefore at most two different nontransitive arcs can be stored at any occupied PE for strip S_k . Since a PE intersects at most two strips, at most four nontransitive arcs are stored at an occupied PE. $\square$

From Proposition 3.3 the systolic screen stores at most $4n^2$ arcs. Proposition 3.2 proves that the nontransitive arcs of a strip are correctly determined by using the arcs already stored in the PE. Since each strip is independent of the others, the graph MG' is correctly computed for the entire systolic screen.

Figure 3.15. The update of I_k of the PE labeled 'a'.

An example for the interval I_k of the occupied PE labeled a at the top of S_k is shown in Figure 3.15. The labels of each PE are shown (in each PE) using normal-sized letters, while the labels of the objects with which arcs are formed are shown in smaller-sized letters. The different updates of I_k are labeled consecutively, and the values for the three regions are shown beside each interval. Notice that at the PE labeled d , the arc $d \rightarrow a$ is transitive since $d \rightarrow e$ and $e \rightarrow a$ are known. The steps in the proof of Proposition 3.2 may be used as an algorithm to determine whether an arc is transitive. The interval for I_k is empty at this PE and is therefore no longer transmitted. The intervals for occupied PEs in S_k are sent simultaneously, that is, intervals are routed in parallel to their successors, calculations are performed, and the process repeats.

This completes the presentation of a method to construct the movability graph MG' , which may be used to solve the problems listed at the beginning of the separability section. These solutions are presented in the next section.

3.3.3. Solutions to the Separability Problems.

Problem I: In deciding whether an ordering of the objects in Q exists, it must be determined whether a set of the arcs forms a cycle. This can be detected by using the arcs in a strip since a cycle among the objects in Q exists if and only if a cycle occurs within a strip. To illustrate this, imagine a line parallel to direction d that cuts the objects where the cycle is formed, this line must be part of some strip. To detect a cycle, the PEs labeled x in a strip send the arcs $x \rightarrow y$ that it stores from the algorithm that determined MG' to their successor in each strip. Each PE labeled z with an arc of the form $z \rightarrow x$ that receives an arc of the form $x \rightarrow y$ from its predecessor performs a *transitive operation* on the arcs to generate the transitive arc $z \rightarrow y$. If ever the arc $z \rightarrow z$ is achieved by this transitive operation, then the set is not unidirectionally separable. Notice by the physical ordering of the objects in a strip, and the way transitive arcs are eliminated in the algorithm, the arcs for the transitive operation will arrive in the proper order.

Problem II: All objects q on the systolic screen may determine in $O(n)$ time whether they are separable from the set $Q - q$. If any pixel of q stores an arc not of the form $q \rightarrow \Psi$, then another object blocks q in direction d . The label of an object q is actually the PE ID of the smallest PE in the connected component [47]. This label, call it id , may be used to as the destination of a RAW in which each labeled PE sends a 1 if it contains no arc except $q \rightarrow \Psi$, and a 0 otherwise. The multiple RAWs are combined using the *and* operation, meaning that if a 1 remains, the object represented by this label may be separated in direction d . A RAR by each labeled PE is made to id to obtain the decision, allowing each pixel of an object to

know whether its component is separable from the set by a single translation in direction d .

Problem III: Since MG' is planar by construction, it contains $O(|Q|)$ edges where $|Q|$ is the number of digitized objects. If this number is $O(n)$ then the adjacency matrix for MG' may be generated on the MCC, and a modified version of the transitive closure operation of Van Scoy [68] may be used to compute the topological sort based on the longest path from a node to the $\forall$ node. Otherwise, the graph MG' must be read off of the mesh and used to solve this problem sequentially by topologically sorting MG' in $O(e)$ time [1], where e is the number of edges in the graph.

3.4. DIGITIZED PERSPECTIVE VISIBILITY.

This section presents an algorithm to compute perspective visibility from a point p of a digitized image on an $n \times n$ systolic screen. Perspective visibility involves finding all parts of a set of objects that are illuminated from a point light source located at p , as illustrated in Figure 3.16. Computing the perspective (or parallel) visibility of an obstacle set provides a solution to the hidden-line elimination problem [50,65]. This section presents a solution for the perspective visibility to an image in two dimensions, and presents extensions to the algorithm for related problems.

Figure 3.16. Perspective visibility from a point p to a simple polygon.

In the remainder of the section, p will be a point with *integer* coordinates from which perspective visibility is being computed. For the object space perspective visibility problem, the regions of the objects illuminated by rays from p are found. The object space perspective visibility solution for a simple polygon whose edges are nontransparent is indicated in bold in Figure 3.16. In *image space perspective visibility*, also referred to as digitized perspective visibility, we shall determine the portion of each pixel that is illu-

minated by rays emanating from p . Additionally, the number of PE's that block a certain fraction of each PE's pixel area will be determined. The case where p is a pixel on the systolic screen is discussed first, and the general case of p being any point (of integer coordinates) at a finite distance from the pixels in the systolic screen is addressed later. If p is at an infinite distance from the systolic screen then the parallel visibility algorithm of Section 3.2 should be used.

In the following discussion we let p be the origin of a cartesian coordinate system whose principal axes are parallel to the rows and columns of the systolic screen. The PEs (excluding p) on a principal axis are defined as belonging to the two quadrants they separate. The algorithm for perspective visibility uses a sweep technique where the sweep proceeds in layers outwards from p . These layers form a square of pixels centered about p , that is, a layer d from PE $p(x,y)$ is the square of PEs $q(i,j)$ at distance $d = d(p,q) = \max(|i-x|, |j-y|)$. The rows and columns of a layer refer respectively to the top and bottom edge, and the left and right sides respectively of the layer. The layers of an 8x8 systolic screen for $p(3,4)$ are shown in Figure 3.17.

	1	2	3	4	5	6	7	8
1	3	2	2	2	2	2	3	4
2	3	2	1	1	1	2	3	4
3	3	2	1	0	1	2	3	4
4	3	2	1	1	1	2	3	4
5	3	2	2	2	2	2	3	4
6	3	3	3	3	3	3	3	4
7	4	4	4	4	4	4	4	4
8	5	5	5	5	5	5	5	5

Figure 3.17.
The layers from PE (3,4).

Figure 3.18.
Perspective visibility wedge PVW(q).

The units of information used in computing the portions of a pixel that are visible from p are called *perspective visibility wedges* (PVWs). PVWs indicate the parts of the plane that are not visible from p . The PVW for a PE q of the systolic screen, denoted as $PVW(q)$, is the region between the two rays of support to the pixel region of q , as illustrated in Figure 3.18. There are two sections in a PVW, one between p and q , and the other after q . The section that occurs after q is not visible from p and is referred to as the

blocked region of the PVW since it blocks the parts of the PEs that it intersects. Similarly, the section that occurs before q is referred to as the *blocking region*, since all occupied PEs that intersect this region hide parts of q from p .

Figure 3.19. Perspective visibility from a point p .

In the perspective visibility algorithm, the PVWs of the PEs in a layer are transmitted outward layer by layer to all the PEs that intersect their blocked region. This process occurs in lockstep for all layers of the systolic screen. All PEs q have a count register C , that stores the number of PEs that have blocked a certain fraction f ($0 < f \leq 1$) of q 's pixel area. They store the union of all PVWs that intersect q 's pixel region in the registers U . At the end of the algorithm, U will give the portions of q that are not visible from p , and C will give the number of PEs whose PVWs block at least a fraction f of the total area of q 's pixel region. The shaded areas of Figure 3.19 indicate the parts of each pixel that are not visible from p , thereby determining the perspective visibility for the images on the systolic screen.

In order for the algorithm to require $O(n)$ time on an $n \times n$ MCC, the amount of information transmitted by each PE to a PE in the next layer must be a constant. The amount of storage for the union U must also be constant. These two requirements are addressed below.

Figure 3.20. The projections of PVWs onto a line l .

Lemma 3.3: *At most two PVWs are required to represent the union U of all PVWs whose blocked regions intersect a PE q 's pixel area.*

Proof: The proof is presented for PEs in the first quadrant defined by p , and extends by symmetry to the remaining quadrants. The PEs in the same row or column as p are special cases which are treated at the end of the proof. For the remaining PEs, define the supporting diagonal of a PE r as the diagonal whose corners t_r and s_r define the rays of support of $PVW(r)$, as illustrated in Figure 3.20. All PEs in the first quadrant (excluding the exceptions) have the same supporting diagonal. Define a line l parallel to these supporting diagonals, and place it so that the blocked region of $PVW(q)$ intersects it. Call the endpoints of the projected supporting diagonal for PE r onto l , t'_r and s'_r . Notice that triangle p, t_r, s_r is similar to triangle p, t'_r, s'_r .

Call the set of PEs intersected by the blocked region of $PVW(q)$ I_q ; only these PEs may block parts of q from p . The distance from the center of q to p is greater than the distance from the center of any PE in I_q to p . Since the supporting diagonals of these PEs are of the same length, this implies, by the similar triangles defined for a PVW and its projection, that the length of the line segment (t'_r, s'_r) for a PE r of I_q is greater than the length of line segment (t'_q, s'_q) for $PVW(q)$. Therefore a PE of I_q must block at least one corner of the supporting diagonal of q , implying that at most two disjoint PVWs, call them a and b , may block parts of q . If this is the case, any other PVW of I_q either joins a and b together, meaning q is completely blocked from p , or one of a or b grows. In the first case, one wedge is required to represent the union U of the PVWs that block q , and in the second

case two wedges are still required.

If q is in the same row (or column) as p , then the blocking region of $PVW(q)$ intersects PEs between p and q in that row. These PEs (and q) have PVWs whose rays of support are formed by the side of the PE whose two endpoints are closest to p . If l is defined to be parallel to these faces, then the arguments given above for the general case also hold. If instead q is intersected by a PVW of a PE x in the same row or column as p , but q is not itself in that row or column, then the projection line l is defined by the supporting diagonal of PE q as before. The projection of $PVW(x)$ onto l must be greater in length than the projection of its diagonal that is parallel to l , or the diagonal would have been chosen to define $PVW(x)$. This implies that this special case may be generalized as discussed above.

□

Figure 3.21. Intersection of rays from p to pixels of the MCC.

By Lemma 3.3, U may be represented by at most two PVWs. The case where two wedges are required is illustrated in Figure 3.19 for PE q . The amount of information that a PE receives from the PEs of a layer must also be constant. Lemma 3.4 and Theorem 3.1 will be used to demonstrate this.

Lemma 3.4: *Only the PVWs of neighboring PEs in a layer may intersect.*

Proof: The proof requires that the support lines of the PVW intersect at most two neighboring PEs in a layer. To illustrate this, using Figure 3.21, notice that the angle of incidence b to the shaded PE in the bottom row of the layer must be 45 degrees. This is a consequence of the fact that the shaded PE p and the PE of the layer on the y -axis, form a right-angled isosceles triangle. This implies that angles of incidence between the x -axis and a ray from p to the part of the bottom row in the fourth quadrant of a layer must range from 90° down to 45° . Since the shaded PE is a corner PE of the layer, 45° is the smallest angle of incidence achieved along the row of a layer. That is, the case where a ray from p may intersect more than two PEs in a row, as illustrated by the ray forming angle c in Figure

3.21, cannot occur. The angle of incidence α of a ray through the PEs on the positive x -axis of a layer is between 45 and 90 degrees, and therefore, the ray can intersect PEs within a unit distance on the x -axis. Since all PEs are unit square, only neighboring PEs of a layer may be intersected by a ray from p . Notice that the above discussion applies to all quadrants and sides of a layer by symmetry.

The PVW of a pixel may be considered to be a continuous sweep of a ray from one supporting ray of the PE to the other. From this fact and the above argument, it can be seen that only the PVWs of neighboring pixels of a layer may intersect. $\square$

Theorem 3.1. *A pixel q of the systolic screen is intersected by at most three different blocked regions of PVWs of the pixels in a given layer.*

Proof: From Lemma 3.4, the PVWs of a layer partition the plane into regions which either zero, one or two pixels block. This is illustrated in Figure 3.21 for the PEs of layer 3, where the single shaded pixels are occupied, and the number of pixels blocking a region is illustrated on the side of the systolic screen. If a PVW r intersects all of q 's pixel region, then at most two other neighboring PVWs of a layer, call them s and t , may intersect q by Lemma 3.4. By Lemma 3.3, s and t may intersect one side of q , assuming as a worst case that both s and t intersect q . This implies that there are two regions $r \cap s$, and $r \cap t$ where two PEs block q . Since r blocks all of q , these two regions block either side of q . If a PVW other than r, s , or t (but in the same layer) were to block q , then it must cut through one of the two regions, implying that a region where three PVWs intersect a pixel exists. This is, however, a contradiction of Lemma 3.4. Other cases where a PVW intersects q are solved similarly by rearranging or removing some of r, s , and t . Therefore, at most three PVWs may intersect q . $\square$

The PE of Figure 3.22 with three shaded regions is an example where three different PEs of a layer block a PE.

Figure 3.22. The number of PVWs of a layer that intersect a pixel.

Theorem 3.1 implies that a PE may receive at most three PVWs from the PEs of a layer. In fact, more will be received but the total number is bounded by six. The steps in sending PVWs from one layer to the PEs in the next layer are now described. They are referred to as growth and exchange steps. In the discussion that follows, the term *data* will refer to the list of PVWs that must be transmitted.

(a) Layer 2 performs the growth step. (b) The exchange step.

Figure 3.23. The growth and exchange steps of PVWs for a layer.

In the *growth* step, the PEs of layer d send data to their neighboring PE(s) of layer $d+1$ as indicated in Figure 3.23(a) where the black PE is p . This is followed by an *exchange* step where data is sent one step left or right for PEs in the row, and one step up or down for PEs in the column of layer $d+1$. In the exchange step, the PEs of a row of the layer transmit data one step away from the y -axis in the row, and the columns transmit data one step away from the x -axis in the column. The corners do not transmit data, and the PEs on a principle axis transmit in both directions away from the principle axis they lie on. The exchange step is illustrated in Figure 3.23(b) where the arrows indicate the one step of data movement.

The PEs of layer $d+1$ have at most six PVWs after the exchange step is completed. Some of these PVWs are copies, and they are discarded. Additionally, some of the PVWs do not intersect the pixel region of the PE, these are also discarded. The PE then updates the union U of the PVW regions that intersect its pixel region. It also counts those that intersect at least a fraction f of the pixel region, and sums this to C . By Theorem 3.1, at most three PVWs will remain, and this is the data transmitted in the next growth step. The next section uses the results of this section to presents an algorithm for perspective visibility on the systolic screen.

3.4.1. Mesh Implementation of Perspective Visibility.

This algorithm assumes that an image has been digitized on the systolic array. The algorithm presented is more general than is required to compute the perspective visibility. In order to more easily extend the algorithm (as discussed in Section 3.4.2), a count C is kept of the number of pixels that intersect a fraction f of the pixel. If this count is to be correctly updated, the simple algorithm of sweeping out in layers from p , and sending the union of the PVWs that block a pixel is not used since the union operation on PVWs destroys information about the number, and size of the PVWs that it represents.

The point p from which visibility is taken, as well as the fraction f of the pixel region to be covered when a pixel is considered blocked, is sent to one PE, say PE(1,1), as an input. The pseudo-code below describes the perspective visibility algorithm.

DIGITIZED PERSPECTIVE VISIBILITY (p, f); { For PE x }

1. **Broadcast** p and f from PE(1,1);
2. Compute the layer in which x falls, and whether it is at the corner of its layer.
3. $PVWList := \emptyset$; { Empty PVW list, implemented as an array. }
4. $UnionPVW := \emptyset$; { Initialize union of PVWs to be empty. }
5. $Count := 0$; { No PEs block x yet. }
6. **if** x is occupied **and** $x \neq p$ **then** { Add PVW(x) to the list so that it is exchanged. }
 $InsertPVW(PVWList, PVW(x))$;
endif
7. $ExchangeStep(PVWList)$; { Exchange PVWs within the home layer. }
8. $DeletePVW(PVWList, PVW(x))$; { Remove the PVW for x if it exists. }
9. $Update(PVWList, UnionPVW, Count)$; { Update visibility. }
10. **if** x is occupied **and** $x \neq p$ **then** { Add PVW(x) to the list for the growth step. }
 $InsertPVW(PVWList, PVW(x))$;
endif
11. **For** $i := 1$ **to** n **Do** { Alternate between growth and exchange steps n times. }
 $GrowthStep(PVWList)$;
 $Update(PVWList, UnionPVW, Count)$; { Update visibility. }
 $ExchangeStep(PVWList)$;
 $Update(PVWList, UnionPVW, Count)$; { Update visibility and PVWList. }
Endfor

END DIGITIZED PERSPECTIVE VISIBILITY;

Steps 1 to 5 initialize the contents of the registers of each PE. The PVWs that intersect PE x are maintained in a $PVWList$ implemented as a fixed size array. The union of PVW wedges that intersect x will be kept in $UnionPVW$. Line 6 enters $PVW(x)$ into the $PVWList$ if x is occupied. The operation $ExchangeStep$ sends $PVWList$ to the appropriate neighbor in x 's layer, and any received elements are inserted into the local $PVWList$. The

operation GrowthStep sends the PVWList to x 's neighbor in the next layer and sets the local PVWList to empty. Next, the elements of the new PVWList that is received are inserted into the local PVWList using InsertPVW. The InsertPVW operation does not insert duplicate PVWs. DeletePVW is used to remove PVWs from the PVWList if they exist. The Update instruction is used to update the PVWList (by removing PVWs that do not intersect x), the UnionPVW and Count as discussed previously.

An initial exchange step is first performed in line 7 among the PEs in a layer since neighboring PVWs in a layer may intersect. Line 8 removes the PVW for x in order that line 9 updates UnionPVW and Count without counting PVW(x). Line 10 places PVW(x) back into PVWList so that it is sent to the next layer in the growth step. All layers now perform n growth and exchange steps in line 11.

Notice that the data of the PEs of a layer is swept across the systolic screen layer by layer, and that all layers act in lock step. Since, after sending the PVWList in a growth step the PVWList is emptied, eventually a layer will receive only empty PVWLists. In fact, after at most n growth and exchange steps all layers must have received the PVW's of occupied PEs in lower numbered layers. Upon termination of the algorithm, the value of UnionPVW for PE x will indicate the portions of x 's pixel that are blocked from p , and the value of Count will indicate the number of PVWs that intersect a fraction f of x 's pixel region.

For the case where p is not a pixel on the systolic screen, the layers are computed as if the systolic screen extended to p . This implies that the layers are correctly defined meaning Lemmas 3.3 and 3.4 hold. The numbering of the layers may have to be reduced if the distance between p and the first layer PE's is larger than a register can hold.

The correctness of the perspective visibility algorithm is now proven using Theorem 3.2. Variations on the perspective visibility problem are given afterwards.

Theorem 3.2: *The PVW of an occupied pixel q is transmitted to all pixels that intersect its blocked region.*

Proof: A proof by induction will be given for PVW(q) of PE q of layer j .

We shall consider the parts of the layers in the fourth quadrant only, as illustrated in Figure 3.21, since the proof applies by symmetry to the remaining quadrants. The PEs (s, t) in the row of a layer block the PEs $(s, t+1)$ in the same layer, while the PEs (k, l) in the column of a layer block PEs $(k+1, l)$. The first exchange step, of PVWs of a layer, therefore ensures that PVW(q) is transmitted to the neighbor in its layer that it intersects.

Assume that the PEs P_i of layer $i \geq j$ that are intersected by the blocked region of PVW(q) have received PVW(q). The PEs P_i are neighbors of all (except perhaps one) of PEs P_{i+1} in layer $i+1$ that PVW(q) intersects since PVW(q) is even wider at layer $i+1$ than

it is at layer i . These PEs P_{i+1} receive $PVW(q)$ after the growth step from layer i . The PE of P_{i+1} that did not receive $PVW(q)$ from that growth step must be a neighbor of those that have received $PVW(q)$ from it since $PVW(q)$ may intersect only one extra neighboring PE by Lemma 3.4. In fact, it must be the PE that is reached after the exchange step in layer $i+1$ is performed since this case is similar to the first exchange step of the algorithm which has been discussed above. The exchange step in layer $i+1$ is performed, and any PEs of layer $i+1$ that do not intersect $PVW(q)$ discard it. Now all PEs of layer $i+1$ that are intersected by the blocked region of $PVW(q)$ have received that wedge. $\square$

The algorithm presented in this section solves the spy problem [8], where a spy has equipment that can hear or see through a fixed number of walls. At the end of the algorithm, the counter C contains the number of pixels that block it. The PEs which have a count of zero or one give the regions can be seen or heard by the spy since they have more than $(1-f)$ percent visible area. This problem takes into account the thickness of a wall. The variations in the next section may be used to ignore the thickness by counting boundaries passed.

3.4.2. Variations on Digitized Perspective Visibility.

In the stabbing problem [16], a count of the number of edges of a figure P crossed by a ray originating from point p is found. In the digitized version, referred to as the digitized stabbing problem, the number of crossed boundaries of a figure is determined. Label the pixels of the figure 1, and the remaining pixels 0. The label of a pixel is now associated with the PVW of that pixel. Each PE contains register C as before, which stores a count, and a new register L to store the label of the last PVW received that blocked at least a fraction f of the pixel. Initially $L=1$ and $C=0$. All pixels are considered to be occupied, i.e. they all send PVWs to the PEs of the next layer. Any PE q that receives PVWs first checks whether the label associated with it is equal to q 's label L . If it is, this PVW is ignored, otherwise if the PVW intersects at least a fraction f of q , C is incremented by one if q 's label L is 1. The local L is then set to $1-L$. The perspective visibility algorithm is executed, and upon termination C is set to $2C$ if p has label 0 and to $2C-1$ otherwise. The value of C in a PE is now the count of the number of boundaries of the figure that were crossed by a ray from p to that PE.

A simple extension of this algorithm can be made to determine the number of times a figure spirals around p [9]. The digitized stabbing algorithm is used except for the following modification; upon termination the value of C is set to $2(C-1)$ if p has label 1, and remains the same otherwise. By performing row and column rotations, the minimum

count may be determined, thereby giving the number of times the figure spirals around p . Additionally, the digitized stabbing problem may be extended to determine the number of boundaries crossed for more than one figure by labeling each figure distinctly and applying the above algorithm, except that L is updated to the label of the PVW received.

3.5. RECTANGULAR VISIBILITY.

Munro, Overmars and Wood [44] define two planar points to be *rectangularly visible* if the orthogonal rectangle with the two points as opposite vertices contains no other point. They give a sequential algorithm to find all pairs of n points that are rectangularly visible in time $O(n \log n + k)$, where k is the number of reported pairs, using $O(n)$ space. They use a planar sweep technique and keep *visibility intervals*. These are intervals on a vertical line that indicate the region to the right of the interval that is rectangularly visible to a point p . The points are sorted by x -coordinate, and a tree of visibility intervals for the points through which the sweepline has passed is kept, and intervals are added, deleted or updated as a line is swept to the right across the point set. That is, a visibility interval for each point to the left of the sweepline is kept. In Figure 3.24, a sweepline is shown and the visibility interval for point p is indicated. The points p that are rectangularly visible to p are circled, those that will be rectangularly visible to p are enclosed by rectangles. This sweep technique is appropriate for solving digitized rectangular visibility.

Figure 3.24. Rectangular visibility, and p 's visibility interval.

This problem may be solved for two- or three- dimensional point sets Q using a dominance relation; a point p *dominates* a point q if each coordinate of p is greater than the corresponding coordinate in q . A point p *directly dominates* a set of points P in Q if p dominates P and no point in P is dominated by a point in $\{Q-p\}$. The dominance relation stated finds all direct dominances for the points in the first quadrant defined by p . By altering the dominance relation, the directly dominated points P in the remaining three quadrants defined by p may be determined. The key is that a point p is rectangularly visible to all points that are directly dominated by it. Güting, Nurmi and Ottman [29] compute direct dominance relations for each point in a 2- or 3-dimensional point set. Consequently, all rectangularly visible points are obtainable using their relations.

The sweep technique of [44] will be adapted to solve the problem for digitized points as discussed below.

3.5.1. Systolic Digitized Rectangular Visibility.

Digitized rectangular visibility for a set of lattice points P that represent occupied PEs, is defined as all those pairs (p,q) for p and q in P , where the rectangle formed by (p,q) contains no lattice point from $P - \{p,q\}$ where the lattice points that are on the boundary of the rectangle are considered as contained. The term *visible* will also be used to mean digitized rectangular visibility for the remainder of this section.

Figure 3.25. Digitized rectangular visibility.

Figure 3.26. The update of pixel p 's record as it is shifted right.

The algorithm outlined will find all pairs of such points for a set of digitized PEs on an $n \times n$ MCC. Figure 3.25 displays the pairs of lattice points that are visible by connecting the pairs by rectangles. Degenerate rectangles (that should be a line) are indicated by very narrow rectangles. Notice that the rectangles intersect and contain only the two occupied

PEs they join. An algorithm to find the visible pairs is first presented. It will be assumed that a constant number of visible pairs will be stored per PE. This assumption will be justified after the presentation of the algorithm.

Digitized Rectangular Visibility:

- (1) Column rotations are performed, whereby each $PE(i,j)$ determines the row value h of the first occupied PE above and the row value l of the first below itself. If no occupied PE above $PE(i,j)$ exists, then h is set to infinity. The same applies to l .
- (2) Each occupied $PE(i,j)$ forms a record R of the interval $[H=h, L=l]$ and its ID (i,j) , and stores pair $\{(i,j), (h,j)\}$ if h is non-infinite, and pair $\{(i,j), (l,j)\}$ if l is non-infinite.
- (3) All PEs (s,t) with records R shift them one step to the right. If $PE(s,t)$ is unoccupied and receives a record, then it checks if h is greater than H and stores pair $\{(i,j), (h,j)\}$ if it is. It also checks if l is less than L , and stores pair $\{(i,j), (l,j)\}$ if it is. The interval is updated to have $H = \max\{H, h\} + 1$, and $L = \min\{L, l\} - 1$. If $PE(s,t)$ is occupied and receives a record, the pair $\{(i,j), (s,t)\}$ is generated and the record is destroyed. This step is repeated n times.
- (4) Perform column rotations to remove any duplicate pairs in a column, by choosing the pair stored in the PE with the greatest row value.

In the above algorithm, intervals for each occupied PE are swept to the right across the MCC and the pairs seen are stored in the PE that detects them. As it happens, this will always be a corner PE of the visibility rectangle for that pair. Figure 3.26 illustrates this process for the leftmost PE p . Remember that the interval only moves in the row of that PE, and the vertical lines are only used to illustrate the updating of the interval. The x 'd PEs are stored along with p as visible pairs in p 's row. It remains to show that a constant number of pairs is stored per PE. This implies that at most $O(n^2)$ pairs will result.

Lemma 3.5. *There are at most 3 pairs stored per PE throughout the execution of the Digitized Rectangular Visibility algorithm.*

Proof: Since records, used to generate pairs, are shifted to the right in the rows, each row is independent of the other. If a record arrives at $PE(i,j)$, then at most two pairs may be generated for an unoccupied PE, and at most one for an occupied PE as can be seen in step 3. No other record can arrive at $PE(i,j)$ since these records are absorbed by any occupied PE to the left of $PE(i,j)$ in the row, and if no such PE exists, $PE(i,j)$ will receive no record. $\square$

Theorem 3.3. *All pairs of rectangularly visible black PEs are determined in $O(n)$ time by the Digitized Rectangular Visibility algorithm.*

Proof: The time bound holds since a sweep across the rows is performed. The fact that all points that are rectangularly visible are found follows from the fact that at most two PEs in a column may be visible to any PE in a row. The column rotations of step (1) of the algorithm ensure that these PEs are seen. The interval created in step (2) and updated in step (3) ensures that only the correct pairs are stored. $\square$

Notice that if lattice points that intersected the edge of a rectangle (i.e. those in the same row or column as the rectangle) are allowed, then there may be $O(n^3)$ pairs since each lattice point may see $O(n)$ other lattice points in its row and/or column.

3.6. CONFIGURATION SPACE.

In path planning [34] a convex robot R must navigate through a set of convex polygonal obstacles O . Instead of moving the robot through the obstacles, the robot is shrunk to a specific reference point r , and the obstacles are *grown* by the same "dimensions" that R was shrunk. The growing operation can be visualized by placing a pen on the reference vertex of the robot, and sliding the robot around an obstacle keeping the same orientation. The lines traced out by the pen will give the grown obstacle. This process is performed for each obstacle, and is referred to as constructing the *configuration space*, denoted $CO_R(O)$, of a set of obstacles. The point r may now be used, instead of R , to plan a path from a source point s to a destination point d in the grown obstacles. This is done by constructing a visibility graph on the grown obstacles in which a shortest path may be found.

Figure 3.27. The configuration space of polygonal figures.

The motions of point r correspond to straight line translations along edges in the visibility graph. An example of this procedure is shown in Figure 3.27 where the robot R 's reference vertex is the bottom right hand corner of the square. Obstacles O_1 and O_2 are shown shaded, and their grown polygons are shown in bold. If r is placed on any point on a grown boundary $CO_R(O_i)$, then R will not intersect O_i , $i=1,2$. The shortest path from s to d using 3 translations of R is also shown.

This section presents an algorithm to construct the *digitized configuration space* for a set of digitized obstacles. Section 3.6.4 gives a solution to the shortest path between two pixels in a digitized configuration space. This path may be used to navigate the robot through the obstacles. The conditions on the robot, and a method of *encoding* the robot will be discussed in Section 3.6.1. The encoded data will be pipelined across the systolic screen, and each obstacle will use the data to generate its configuration space. The method of sending this encoded data is presented in Section 3.6.2, and the final algorithm, and additional problems are presented in Section 3.6.3.

3.6.1. An Encoding for the Robot.

The encoding of the robot to facilitate construction of the digitized configuration space $DCO_R(O)$ is now addressed. In object space configuration space construction, the vertices of the polygonal obstacles and the robot are used. In digitized configuration space, the pixels of the obstacles and robot will be used. Definitions expressed in [34] for object space are now translated to image space.

For two digitized figures A and B , whose pixels are denoted by lowercase letters, we have:

$$A \oplus B = \{a + b \mid a \in A, b \in B\}$$

$$A - B = \{a - b \mid a \in A, b \in B\}$$

$$-A = \{-a \mid a \in A\},$$

where $a + b$ means the lattice coordinates $a(i,j)$ and $b(s,t)$ are summed as $(i+s, j+t)$, and similarly for set subtraction or negation. The term $(R)_0$ is used to denote $\{a - r, \text{ for } a \in R \text{ and } r \text{ as the reference vertex of } R\}$. For two planar figures A and R $DCO_R(A) = A - (R)_0$. This fact is proven in [34] for polygonal figures, and also applies to digitized figures since their lattice points are used.

The computation of $DCO_R(O)$ can be simplified if R is a rectilinearly convex figure (Section 2.2.1) as follows: Let r be chosen to be the $PE(i,j)$ of R with the greatest j value and, if ties exist, the smallest i valued PE . All PEs of R compute $-(R)_0$, using their lattice points. Since the new values of $-(R)_0$ may not be on the systolic screen, they must be adjusted. This is done by determining the coordinates of the leftmost, rightmost, topmost and bottommost PEs of $-(R)_0$. Using these four lattice points, the values of $-(R)_0$ may be translated back on the systolic screen by adding a constant value to the i and j coordinates. A RAW can now be used to write the values of the translated figure $-(R)_0$ to the new lattice coordinates (i,j) computed. These PEs mark themselves as R' , and the old data for the robot is discarded.

Define the *left boundary* of the robot, as all PEs L of R' whose left neighbor is not in R' , and those PEs required to 4-connect PEs in L . Refer to Figure 3.28, which illustrates the left boundary of R' , where the left diagonally shaded PEs have no left neighbor, the cross shaded PEs connect L PEs in neighboring rows, and the remaining diagonally shaded PEs define the rest of R' . The connecting PEs may be found by having each PE ask if its up or down neighbor is a left boundary PE . Row rotations can then be performed to determine the connecting PEs .

Figure 3.28. Encoding the left side of a rectilinear robot.

The reference point r' of R' must now be the bottommost and leftmost PE , in case of ties, as indicated in Figure 3.28. The left boundary PEs of R' encode a sequence of motions, where r' will start the sequence. These will be stored in records $\langle instr\#, count, direction \rangle$, where $instr\#$ gives the number of the motion, $count$ is the number of PEs of R' to the right of the left boundary PE the instruction is stored in, and $direction$ is the motion

required to get to the next left boundary PE (which is the next instruction). The instructions for the robot in Figure 3.28 are illustrated beside the MCC. The inst# can be determined using row and column rotations, and the locations of PEs with no left neighbor. Since the robot is rectilinear, R' will be rectilinear, and since the number of PEs in its left boundary is $O(n)$ on an $n \times n$ MCC, at most two records need to be stored per PE on the left boundary.

The robot need not be entirely rectilinearly convex. *Right*, *top*, and *bottom* boundaries may be defined similarly as for the left boundary, with the right boundary defined as being opposite to the left, and the top as opposite to the bottom boundary. The left [right] boundary is rectilinearly convex if, when we color all PEs to the right [left] of the left [right] boundary pixels black, the resulting figure is rectilinearly convex. If the right [left] boundary may be encoded so that only up, left and right directions are used then that boundary is *monotone* with respect to the columns of the MCC, and the robot can be encoded along the left [right] boundary.

Figure 3.29. Encoding special types of robots.

An example of a robot which is not rectilinearly convex, but whose right boundary is monotone and whose left boundary is rectilinearly convex, is shown shaded in Figure 3.29. Similar arguments apply to encoding the top or bottom sides of the robot if one is rectilinearly convex and its opposite is monotone with respect to the rows of the MCC.

The decision of whether one side of a robot is rectilinearly convex, and the opposite is monotone can be done in $\tilde{O}(n)$ time on an $n \times n$ MCC. The detection of whether the left side is rectilinear is done by first considering all PEs to the right of the left boundary of R as being on, i.e. mark them and the left boundary of R as R'' , and rotating the pixels in the columns. If all pixels of R'' form connected components within each column, then the left boundary is rectilinearly convex. The detection of whether the right boundary is monotone

can be done by rotating pixels of R in each row to determine if each row of R by itself is a connected component. Other boundaries may be treated in a similar manner.

This encoding scheme is used to construct $DCO_R(O)$, by sending the instruction sequence to each pixel of the obstacles as outlined in the next section.

3.6.2. Pipelining the Encoded Robot.

Pipelining is the synchronous motion of a set of records through a set of processors. For instance, in the row rotation operation, copies of a register are simultaneously shifted left in the row. In pipelining, a set of data is sent through a set of processors in some specified order in which the first data element is in one set of processors, and the second in a neighboring set, etc..., forming waves of processors through which the data is shifted. We can define a *wave* w on a MCC as a set of processors on the MCC where the PEs of wave w should have at least one neighbor at a constant routing distance away from it in wave $w+1$, and every PE is contained in exactly one wave. The number of waves, denoted by $\#w$, depends on the sets of processors chosen for each wave. For example, we can define the rows of the MCC as waves, with the PEs of the i th row being in wave i , then $\#w=n$. Another method would be to define diagonal waves where $PE(i,j)$ belongs to the wave $w = (n+j-i)$, $1 \leq i, j \leq n$. The wave number of these diagonal PE waves is indicated in each PE of the MCC in Figure 3.30, where the total number of waves is: $\#w=2n-1$. If data $(d_1, d_2, \dots, d_m)$ is to be pipelined, then at step k ($2 \leq k \leq \#w+m$) of the pipelining, record d_i ($1 \leq i \leq m$) should be in the PEs of wave $k-i$ if such a wave exists. The data for $m=5$ and $k=12$ is shown on the right side of the MCC in Figure 3.30, and the waves in which the data resides are illustrated by diagonal lines.

Figure 3.30. Pipelining and waves of an 8x8 MCC.

Notice that $O(\#w+m)$ constant size steps are required to pipeline a data sequence across an MCC since the neighbors of wave w in wave $w+1$ are a constant routing distance away. If m is $O(n)$, then this process will require $O(n)$ time for row, column or diagonal waves since $\#w$ will be $O(n)$.

The encoding technique, and the method of pipelining data may now be used to find an algorithm that computes the configuration space of a set of obstacles.

3.6.3. Digitized Configuration Space.

The encoded boundary of R' is to be pipelined across the MCC using diagonal waves. If a black (occupied) PE were to trace out the left boundary of R' and then fill-out the rows of R' using the count field of the encoded instructions for R' (Section 3.6.1), the configuration space of the robot with respect to that pixel will result. The algorithm performs tracing action in parallel for each pixel, and the counts are then used to fill in the configuration space for the pixels. Since the paths pixels follow in the tracing may overlap, care must be taken to ensure that the counts used to fill in the remainder of configuration space are not lost. Each black PE initially waits for the first instruction of the encoded robot. Once this instruction is received, the PE will form a record which must wait for the second instruction, and send it to the PE that is at the same offset from itself as the next encoded boundary PE of the robot. The PE will then store the number of pixels to its right that must be colored black. The record that is waiting for instruction 2 of the encoded robot repeats this process. Since many black PEs act in parallel, PEs that receive these records will store the *maximum* count field value received. After this process is finished, each pixel of O will have traced out the left boundary of R' . The count values at each PE may then be used to fill in the remainder of the configuration space. Issues, such as records for black pixels being shifted off of the MCC, will be addressed later. The records that PEs must generate, and the directions in which to pipeline them, are discussed next.

Each PE will contain two records of the format $\langle \text{used}, \text{wait_instr\#} \rangle$, where *used* is a flag indicating whether the record is used, and *wait_instr#* gives the next instruction of the encoded robot to execute. A register *max_count*, initially set to zero, will store the maximum count value received by the PE. If the PE is from a digitized obstacle, one of these records is initialized to $\langle \text{true}, 1 \rangle$ while the other remains as $\langle \text{false}, 0 \rangle$. The instructions are stored in records $\langle \text{instr\#}, \text{count}, \text{move} \rangle$, where *instr#* is the instruction number, *count* is the number of pixels to color to the right (if the left boundary was encoded), and *move* is the direction an occupied pixel should move before awaiting the next instruction. The waves used in transmitting the encoded instructions, and the actions

taken by the PEs are described in the *pipeline_instructions* algorithm below.

The waves for the pipelining of the encoded data must be chosen in such a way that the records generated by the black PEs remain ahead of the instructions they await. Notice that, in general, the encoding of R' is a sequence L_1 of up and left motions, followed by a sequence L_2 of right and up motions. This implies that L_1 should be pipelined in diagonal waves that start at $PE(n,n)$ and end at $PE(1,1)$, while L_2 should be pipelined in diagonal waves that start at $PE(n,1)$ and end at $PE(1,n)$. The algorithm for pipelining the instructions for one wave is presented below. The instructions of L_1 are pipelined first, and when it is through the instructions for L_2 are pipelined. The instructions are assumed to be entered at first corner PE of wave 1.

PIPELINE_INSTRUCTIONS:
repeat

- * Send(<instr#, count, move>) to PEs of the next wave.
- * if (an instruction <instr#, count, move> is received) then
 - * if (instr# = wait_instr#) and (used is true) in a record of the PE then
 - * max_count := max{max_count, count};
 - * Send(<true, wait_instr# + 1>) in direction move, unless move = done.
 - * used := false;
 - * endif
- * endif
- * if (a record <true, wait_instr#> is received) then
 - * Store it in a record where used is false.
- * endif

until All data has been pipelined.
END PIPELINE_INSTRUCTIONS.

If one pixel of an obstacle is observed, the left boundary of R' will have been traced out after the encoded instructions have been transmitted, and counts of the pixels to color to the right would be stored in these PEs. It now remains to fill in the remainder of the configuration space for that pixel by filling in all pixels with non-zero count values, and filling in the value of count number of pixels to their right. Therefore, for a single pixel p , the configuration space will be $\{p\} - (R_0)$, and an obstacle A having many pixels will have the $A - (R_0)$ as required.

The encoded instructions for R' may be read off of the mesh, and entered into the first wave by a corner PE, or the instructions may be kept on the mesh, and shifted one by one into the corner PE of the first wave. This may be accomplished by sorting the instructions in L_1 and the instructions in L_2 into two separate sequences, lining these sequences up

along the boundary of the MCC so that they can easily be shifted to the PE of their first wave when they are required. This means an additional step in the pipelining is required, to shift encoded instructions toward the first wave from which they are pipelined onwards.

There remains the problem of the record of an occupied PE being shifted off of the left side of the MCC. If this occurs, the instructions for the remainder of the left boundary of R' will never reach this record, and an incorrect configuration space will result. To prevent this, each column of the MCC will simulate two columns, and all obstacles O are shifted as far right as possible in this new configuration. Since the robot is no greater than n rows in width, no record will fall off of the left side, and the image can be re-expanded once computation is complete. Any pixels of obstacles that fall off of the top of the MCC are also lost, but the configuration space displayed will be correct. If the entire configuration space for each obstacle is desired, each column of the MCC may also simulate two rows (each PE now simulates 4 pixels) and the MCC is now $2n \times 2n$. The robot may also be restricted in width to $1/2n$ columns in the first case, and $1/2n$ rows and $1/2n$ columns in the second case.

Two questions remain:

1. Are a constant number of records per (simulated) PE enough to store the records of pixels tracing out a boundary of the robot?
2. If a record in a PE is waiting for an instruction, does it arrive in a constant number of steps?

If we can answer the first question in the affirmative, then constant storage is required and each PE must only examine and store a constant number of records at each step in the pipeline algorithm. The pipelined instructions take up at most two records in any PE, depending on whether they are kept in the mesh or read off of it and pipelined into a corner. The answer to the second question depends on the pipeline algorithm, and the direction of motion of the encoded instructions. If the direction of motion is before the wave of the instruction, the instruction that any record in a PE awaits must be two waves behind. This means that if there are $O(n)$ instructions and the answer to Question 1 is affirmative, then the algorithm requires $O(n)$ constant-time steps to execute.

Proposition 3.4: *Two records are sufficient to store data representing shifting pixels.*

Prpof: Initially, each PE of an obstacle stores one record which is waiting for instruction 1. When instruction 1 arrives at these PEs, a record $\langle \text{true}, 2 \rangle$ is sent to the PEs of the wave w in front of the wave containing instruction 1, by the definition of the sequences L_1 and L_2 for R' . Wave w is therefore the wave instruction 1 passes through next. Since all records of a wave are shifted in the same direction (i.e. left, right or up), a PE in wave w

must now have at most one record waiting for instruction 2 from a PE in wave $w-1$, and it may have its own record for itself waiting for instruction 1.

In general, if a PE receives instruction i that it is waiting for, it will transmit its data to a PE in the next wave w instruction i must arrive at. Since all records of a wave are shifted in the same direction, a PE in wave w can have at most one record waiting for instruction i and one record waiting for instruction $i-1$. Instruction $i-1$ is currently in wave w , meaning that the records in wave w waiting for it are shifted forward to the wave in front leaving room for the records sent forward by instruction i . This process occurs simultaneously for all PEs containing instructions, that is, a set of records is shifted one wave forward, and the case for the wave containing instruction 1 is addressed in the first paragraph. $\square$

Notice that, since pixels are treated individually, the obstacles may be of any shape. Note that holes in obstacles should be filled in before the start of the algorithm since the robot can never reach those sections.

Theorem 3.4: *The digitized configuration space of a set of obstacles may be created for the types of robot specified in Section 3.6.1 in $O(n)$ time.*

Proof: The encoding of the robot requires $O(n)$ time as discussed in Section 3.6.1. The pipelining of the encoded instructions requires $O(n)$ time since there are $O(n)$ instructions, and the filling in of the configuration space requires $O(n)$ steps since a robot is at most n columns wide. $\square$

3.6.4. Shortest Paths and the Digitized Configuration Space.

Existing algorithms for finding minimum paths in digitized objects [41] may be used to find the minimum path to be traveled by the robot (now a pixel) between a source and destination PE in the configuration space. For instance, Problem (2) of the Internal Distance Related Problems of Section 2.2.3 finds a minimal $a-b$ path between two connected pixels a and b in $O(n)$ time. If the unoccupied space is considered to be a connected component, then this algorithm may be used to find the minimum path between any two specified unoccupied PEs. As specified in the algorithm, any monotone distance metric may be used. In fact, the Internal Distance Problem (Section 2.2.3) may be used to determine whether a path exists between two PEs a and b , since if the distance between these PEs is infinite, then no path on the systolic screen exists.

3.7. CONCLUSIONS ON VISIBILITY.

Existing systolic solutions for parallel, perspective and unidirectional separability problems for polygons in object space were presented in Section 3.1. Solutions to these visibility problems in image space were then presented in Sections 3.2, 3.3 and 3.4. Rectangular visibility has been solved in image space in Section 3.5, and the computation of the digitized configuration space was addressed for certain types of monotone robots in Section 3.6.

The image space visibility problems solved are optimal in that a pixel at one corner of the systolic screen may block another at the opposite side or corner. At most $2n$ routing steps are required to send this information, implying a lower bound of $\Omega(n)$ time. In fact, for the digitized configuration space problem, even if the robot may be encoded in a constant number of records, this data must still be transmitted to all obstacles, which may be separated by $2n$ routing steps in the worst case, implying that this algorithm is also optimal.

The image space visibility algorithms use sweeps to compute their visibility function (parallel, perspective and rectangularly visible). The digitized configuration space uses the pipeline technique, which may be considered as a diagonal sweep. This illustrates the usefulness of the sweep paradigm for visibility on images, whereas a divide-and-conquer approach was used in object space MCC algorithms since the data is not stored in an ordered fashion as an image on the systolic screen.

Further improvements can be made on the digitized configuration space problem to allow more general types of robot. If the pipelining method is to be used though, the amount of encoded data must be $O(n)$ and the number of moves between each step must be a constant. Also, although the movability graph for the image space unidirectional separability problem can be computed, it has a worst case of $O(n^2)$ edges. If an algorithm to topologically sort a graph of $O(n^2)$ edges on an $n \times n$ MCC is found, the separability problem number 3 listed in Section 3.4 may be solved directly on the MCC.

The digitized parallel visibility problem was solved for particular directions d of visibility on a three-dimensional systolic screen. This was accomplished by extending the strips to rectangular volumes followed by simple modifications on the parallel visibility algorithm for two dimensions. Since the separability algorithm uses the strip idea, the extension for digitized unidirectional separability into three space for these directions of visibility is also straightforward. We are currently investigating the extension of the perspective visibility algorithm for a three-dimensional systolic screen, and have some results in that area. A problem with extending the rectangular visibility algorithm is that there can be $O(n)$ pairs of PEs stored at a PE. To illustrate this, imagine a staircase of black PEs going down row by row on the systolic screen. A black PE on the bottom of the

screen, under the highest row, can see $n-1$ pixels above it in the vertical plane that cuts the staircase. Computing the digitized configuration space for three dimensional obstacles and robots on a three-dimensional systolic screen presents another problem. Extending the two-dimensional solution is not a good approach, since, the encoding of the (rectilinearly convex) robot in $O(n)$ elements in order to pipeline it, is in general not possible. Extensions to general two-dimensional monotone robots are being investigated.

Further research could be made into solving three-dimensional visibility problems in object space for $n \times n$ MCCs. If the perspective visibility algorithm presented in Section 3.1 is solved in three-space, the parallel and separability problems presented should also be easily extendible.

4. MESH-CONNECTED COMPUTER HIGH-LEVEL LANGUAGE.

This chapter introduces a design of a high-level parallel language called the Mesh Language (ML) to be used in programming the mesh-connected computer. Most mesh algorithms, like those presented in Section 2.2, are described in a top-down manner by using high-level parallel mesh operations, recursion and, submeshes. These operations (see Section 2.1 for details) are used as if they have already been programmed or were implemented by the hardware. In current mesh languages though, the user must code these operations first. The Mesh Language is designed to present a top-down programming view of the mesh, since the user thinks in terms of parallel operations that operate on the mesh instead of the steps performed in each processor. This programming view follows that found in most research papers, where algorithm descriptions are presented using high-level parallel operations specific to the interconnection network. A second view would be to program the steps performed by each PE to accomplish a high-level operation, which results in bottom-up programming.

Stout [62] has recently noted that further language and programming support is necessary to allow the implementation of many proposed systolic algorithms, especially those that make extensive use of high-level parallel operations. In addition to expediting an algorithm's implementation, designing a language that includes these operations will ensure that ML code is portable. ML code is machine independent since the details of the implementation of the standard operations for a particular mesh are hidden from the user.

Existing languages and environments for parallel architectures usually take one of two views. If they take the second view, programming is performed from the point of view of a processor and the user must code all operations. Otherwise programming is done from the point of view of a controller that broadcasts instructions to the mesh, except that high-level parallel operations are not supplied or they are machine dependent. Examples of these cases are presented in Section 4.1. In our system, programming is done from the point of view of a controller that broadcasts instructions to the PEs of the mesh. This makes it easier to design high-level instructions, and incorporate portability of code. The structure of our system is outlined in Section 4.2 where a hierarchy of languages is presented.

The concept of limiting the scope of an operation to one or several submeshes is useful in recursion, data compression, or cases where the same algorithm is executed in many submeshes in parallel. Section 4.3 addresses this, and other requirements of ML, and presents a suitable architecture for a PE of the mesh. The overview presented in Section 4.2 is expanded in Section 4.4 which presents the low-level language that the PEs execute,

the language used in programming the parallel operations, and the syntax of the ML operations. An example of an ML program is presented in Section 4.5 to illustrate the structure of an ML program and the use of some ML operations. Finally, Section 4.6 summarizes this chapter and mentions some extensions that may be made to ML.

4.1. PREVIOUS SYSTOLIC LANGUAGES.

Many existing parallel languages such as Glypnir [30] give support for the machine's parallelism by providing instructions that are specific to the architecture. For instance, the Glypnir programming language, which was designed for the Illiac IV mesh architecture, provides instructions to access the same register of all PEs in parallel, or the register of a specific PE. Control statements that resemble those of Fortran and Algol are provided to select the code to be executed (IF statements) or to perform loops. An example of a control statement is found in IVTRAN [43], an extension of Fortran for the Illiac IV, which uses a DO loop to access elements arranged in the Illiac IV's registers. Glypnir and IVTRAN programmers make use of the instructions that access a set of registers in parallel. This implies that the data must be arranged in the PEs in a way that allows the most parallelism using these instructions. These languages are therefore designed to exploit the parallelism in the machine, instead of the parallelism of an algorithm, where parallel operations are used. In fact, Perrot and Stevenson [51] have noted in their survey of Illiac IV users that future languages should provide abstraction mechanisms to represent a *program's* parallelism rather than the *machine's*.

Another approach is found in Poker [58,59] where programming is done from the processors point of view. The Poker programming environment allows the user to specify an interconnection network by embedding it on an $n \times n$ lattice which Poker then simulates. That is, the user defines types of PEs for which program segments are written using a high-level language such as C, and links them to other PEs using the lattice. A general notation for sending data to a neighboring PE is provided by a "send" operation. The program segments for the PEs are analyzed and converted by Poker into synchronously communicating programs. In Poker, programming is performed from the PE's point of view, since the programmer describes what each type of PE does at each step instead of writing instructions that are executed by a group of enabled PEs. This results in bottom-up programming since the operations required by the algorithm must first be implemented before the algorithm can be coded. Since the user codes the operations for his problem, they may be coded in a problem-dependent manner that makes them difficult to reuse. The embedding of general interconnection networks on Poker's mesh structure means that it is very difficult to write high-level parallel operations, since these operations are usually

dependent on the interconnection pattern. Therefore, although Poker provides high-level language control structures (in C), no general parallel operations, aside from a low-level send operation, are defined due to the dependence of operations on the interconnection scheme.

Some systolic languages are designed for specific areas such as image processing [28]. Examples are Uhr's Pascal PL, and Balston's Instruction Forming Language (IFL). These languages provide image processing operations, but are not general enough to be used as a language in other problem areas. This is often due to the fact that the architecture on which they run has been built specifically for image processing.

CMLisp is an extension of Lisp that runs on the Connection Machine [21], an n-cube architecture of 65536 processors with 4096 bits of memory each. CMLisp provides a good interface to the Connection Machine by defining high-level operations for a data type called Xectors, which are arrays (vectors) whose elements are mapped to separate processors. The user must use Xector operations to build higher level functions since the elements of Xectors are mapped to arbitrary PEs. Unfortunately, the user is provided with a fixed data type, and no means to control the placement of data. This makes it difficult to write algorithms that are specific to certain interconnection networks such as the Hypercube, Pyramid or MCC discussed in Chapter 1.

4.2. OVERVIEW OF MESH CONTROL.

In contrast to previous approaches, the system presented in this section takes a top-down programming view of the mesh by providing high-level Mesh Language operations. A program's parallelism is easily expressed using these operations, and the usual low-level send and arithmetic or boolean operations are provided to cover local operations on the data in a PEs registers and data exchanges between neighboring processors. The programmer writes programs in ML which are executed by a mesh controller that controls the entire mesh by sending instructions that perform the high-level ML operations, such as a broadcast, sort or recursion operation, to the PEs of the mesh. This achieves a top-down view since ML operations are used as if the mesh hardware provided them. This type of control, where a controller sends the instruction to execute to all PEs, is the SIMD model of Section 1.2. Note that ML programs that use the SIMD model can easily be converted to run on MIMD machines by loading the same program into each PE. The SIMD model also provides the simplest hardware, since programs need not be kept within each PE.

In our approach, we use a hierarchy of languages: the Mesh Language (ML), which is designed for the mesh programmer; the Mesh Controller Language (MCL), which controls the mesh since it implements the ML operations; and the Mesh Instruction Language (MIL), which describes the format of the primitive instructions finally sent to all PEs. The mesh

programmer writes programs in ML using high-level parallel operations that are executed on the mesh. The mesh controller executes the ML program by compiling the operations into sequences of MIL instructions that implement the operation. These instructions of these sequences are broadcast to the PEs of the mesh, which then execute them. The sequences may be composed of enable or disable and simple arithmetic/boolean or "send" instructions. For example, if a ML programmer asks all PEs in even rows to send the contents of a register to the right, the mesh controller will send the MIL instructions to enable all PEs in even rows, followed by the send instruction. That is, the PEs that execute the instruction must first be enabled, and then the instruction to execute is sent. The components of the MCC system are shown in Figure 4.1.

Figure 4.1. Overview of Mesh Control.

The ML operations are converted by the mesh controller into a sequence of low-level instructions executable by the PEs of the mesh. Therefore, if the hardware changes, only the controller implementation of the ML instructions have to be modified. This makes the Mesh Language a *machine independent* high-level language since the mesh controller implements the standard ML operations used by the ML programmer. In addition, if the user selects a mesh size that is larger than the physical mesh, the MCL control programs could attempt to execute the program by having each PE simulate a submesh such as a 2x2 region. Section 4.4 presents more detail on the three languages outlined in this section, but first, the next section describes the architecture of a PE based on some requirements of ML operations.

4.3. PE ARCHITECTURE AND ML REQUIREMENTS.

In order to provide an abstraction mechanism for a program's parallelism, ML contains the standard operations of Section 2.1, as well as automatic recursion, data compression

the capability to define multiple submeshes. A standard operation may act on the entire mesh, such as a broadcast of a register value to all PEs, or on a submesh generated by data compression for instance. Some MCC algorithms, such as Miller and Stout's convex hull algorithm [40] (presented in Section 2.2.2), are generalizable for more than one figure by creating a submesh for each figure, and performing the algorithm in parallel on each submesh. If, for example, a row-rotation is performed as part of a recursive step, the user would like the data to be rotated within all recursive subdivisions in parallel, not on the entire mesh. Additionally, the user may employ nested recursion in solving a problem as in the solution of the largest empty rectangle algorithm presented in [10]. In this case, the algorithm recursively subdivides the mesh, and at each merge step, a second recursive operation is performed.

The above examples illustrate the need for a bounded region (or regions) to be associated with an ML operation. This issue is handled by defining a boundary within which a high-level ML operation executes. A *boundary* is a set of (possibly one) non-overlapping submeshes of the mesh. To implement this, a *Bounds Stack* (see Figure 4.2) is used. The *Bounds Stack* is a constant size stack of boundaries for operations. The initial boundary on this stack is the entire $n \times n$ MCC, i.e. the PEs (i,j) with $1 \leq i,j \leq n$. All ML operations execute within this boundary, unless it is changed by performing data compression, recursion, or by adding user-defined boundaries. Since only a constant number of boundaries may be kept, there may only be a constant number calls to operations that add new boundaries to the stack. The *diameter* of a boundary (for a submesh) is the maximum number of rows or columns in the submesh. Since ML operations may operate on different size submeshes in parallel, the diameter of the largest of these submeshes must be known by the controller in order that the correct number of steps are performed to complete the operation in each submesh. Boundaries and diameters of submeshes are discussed further in Section 4.4.4.

Some additional requirements for ML are: allowing PEs to be accessed by a row-major ID, allowing a PE's registers to be referred to by arbitrary identifiers which are later mapped to physical registers, providing arithmetic/boolean operations, and sending data to neighboring PEs. In communicating with the outside world, three forms of input and output (I/O) are defined: (1) The mesh controller broadcasts an ML instruction to the mesh. (2) A constant number of I/O ports are defined, to be used by the controller or ML user in gaining the status of the computation, i.e. to determine whether an algorithm has terminated. (3) The sides of the mesh are used to load data as I/O streams into the registers. Forms (2) and (3) are captured by defining I/O ports and streams, while form (1) is reserved for the mesh controller alone.

Figure 4.2. PE registers and processor.

These requirements imply that a certain amount of information must be maintained in each PE of the mesh. Figure 4.2 illustrates the types of registers that each PE contains, and the Arithmetic/Logic Unit (ALU) of a PE which may be disabled if it is not to execute an instruction. The *standard registers*, are reserved for special purposes and are read-only to ML users. The standard registers consist of the PE ID, and a *Bounds Stack* which is used to handle the boundaries of an operation. Registers reserved for ML users' programs are referred to as *general registers*, and are read/write. There are also *communication registers* that are used by the PEs in send operations, and cannot be accessed by ML users. The special PEs that are I/O ports have additional registers (not illustrated) that are linked to the mesh controller.

4.4. THE MCC LANGUAGES.

This section presents further details on the hierarchy of languages defined in Section 4.2. The notation used to present these languages is first presented followed by the definition of the Mesh Instruction Language. Details on the Mesh Controller Language that is used to implement the ML operations is presented in Section 4.4.3. Finally, the syntax of the ML operations is presented, along with their function and any restrictions on their use.

4.4.1. Notation.

In the sections that follow, bold identifiers are used to denote reserved words, while italic identifiers denote Pascal reserved words. Statements in angle brackets, < and >, must be substituted by the identifier or identifiers named. Angle brackets are also used to show where code is to be substituted. Plain face identifiers are pre-defined identifiers

which the user may access. Comments are delimited by curly braces, { and }, and anything included in square brackets, [and], is optional or context sensitive. The symbol | is used to mean *or*, that is, one of the listed items is chosen. The symbol | will be used if the syntax for an instruction is similar to another. The identifier to the right of | replaces the identifier to the left throughout the instruction, giving a new instruction. Three dots "..." follow clauses that may be repeated as required.

4.4.2. Mesh Instruction Language.

The Mesh Instruction Language consists of instructions to enable or disable a PE, execute boolean or arithmetic expressions, or send and load registers. The arithmetic or boolean expressions contained in the *execute* or *enable* instructions consist of simple one or two register assembly-like instructions generated by the MIL compiler. The instructions may use parameters: register-register, register-constant or register. The register-constant allows operations such as $x = x * \pi$, where x is a register and π is a constant. The constant π is stored in the controller, not in each PE, and is broadcast to each PE as part of the instruction. A list of the instructions is given in Figure 4.3.

The *Enable* and *disable* instructions are used for if-then-else type of control. Sending one register to a neighbor is implemented using the *send* instruction which loads the specified register into a communication register, the enabled PEs then simultaneously send the contents of the communication register to the same register of the PE in the direction specified. The *load* operation is then used to load the value in the communication register, into the specified PE register.

Some PEs are connected directly to the mesh controller by I/O ports to allow direct access to them alone. The registers of these PEs may be read from or written to by the mesh controller. The read instruction informs the PE with the given port ID that the controller wants to read one of its registers. That PE then sends the value of the register to the controller. The write instruction informs the PE with the given port ID that the controller wants to write to one of its registers. The PE then receives the data sent by the controller and places it into the specified register.

The side connections of a mesh (Figure 1.1) are used as I/O streams to load or read register data. The left side of the mesh is called the left I/O stream. Similar definitions apply to the three remaining sides. In loading data for a specific register of all PEs, the *stream_in* instruction is used to load the first column (or row) of register values. The *send* instruction is then used to shift the data in the communication registers to the next row (or column). This repeats until all data is on the mesh and in the communication register of the correct PE. A final load instruction copies the value of the communication register into the

2 of/de 2

by an enhanced Pascal compiler. The resulting code is executed on the controller, which sends the required MIL instruction sequences that execute the ML program.

Many of the currently existing systolic languages, such as Glypnir (one version of which is a superset of Fortran) operate at the controller level and provide machine dependent instructions. This means Glypnir users must code any standard operations that they employ. The purpose of providing MCL is to allow easy programming of ML operations by system (MCL) programmers, and to make ML code portable.

The overall syntax of MCL is illustrated in Figure 4.4 below. The MCL language is composed of a configuration and program part. The configuration section defines the physical parameters of the mesh, followed by functions or procedures that implement ML operations, and a main program which consists of the main program of the compiled ML program.

Program Mesh_Controller;

Configuration

StandardMeshConst

N = <actual size of NxN mesh>;
 ConnectionType = 4 | 8; { Four or eight nearest neighbor-connection scheme. }
 NumberIntegerRegisters = <number of integer registers in each PE>;
 NumberRealRegisters = <number of real registers in each PE>;
 NumberBooleanRegisters = <number of boolean registers in each PE>;
 MaxBoundsStack = <physical size of the stack for boundaries>;

{ names of connections between PEs }

Left, Right, Up, Down [other neighbor connections if ConnectionType is 8];

StandardPEType

- the Pascal *integer*, *real*, and *boolean* types.

PE_ID = <record of row, column for a Row Major ID>

Bounds = <record of top-left and bottom-right PE IDs for a boundary>

StandardPE_Registers { Read/write for MCL, read only for ML programmers. }

PE: PE_ID; { A record of registers used to store the fixed ID of a PE. }

BoundsStack : Array[1..MaxBoundsStack] of Bounds;

StackTop: Integer; { A pointer to the top of Bounds Stack, for use by MCL programmers only. }

PE_Registers { Users' registers are mapped to these physical general registers. }

IntegerReg: Array[1..NumberIntegerRegisters] of Integer;

RealReg: Array[1..NumberRealRegisters] of Real;

BooleanReg: Array[1..NumberBooleanRegisters] of Boolean;

{ The communication registers used in the send or load MIL operation. }

IntegerCommReg: Integer;

RealCommReg: Integer;

BooleanCommReg: Integer;

```

I/O Ports
{ Define the constant number of I/O ports that physically exist. }
  PE 1,1; { PE 1,1 is an existing I/O port }
I/O Streams
{ Set the I/O streams listed below, which are boolean constants, to be true if the I/O
  stream physically exists and false otherwise. }
  LeftInputStream, RightInputStream, TopInputStream, BottomInputStream,
  LeftOutputStream, RightOutputStream, TopOutputStream, BottomOutputStream
EndConfiguration

ContrConst
{ MCL programmer defined constants }

ContrType
- all Pascal types -
StandardPETypes

ContrVar
{ MCL programmer variables of type ContrType. }

ContrProcedures
{ MCL programmer defined procedures and functions of ContrType. }

ContrBegin
{ The translated ML program sequence to be executed. }
ContrEnd.

```

Figure 4.4. Syntax of MCL.

The standard mesh constants define the actual physical parameters of the mesh, such as its size, the number of its registers etc.... Standard PE types are register types that are used in MCL, and in MIL instructions. These registers are present in each PE, and are read/write to MCL programmers. The register *PE*, which gives the ID of that PE, is read only to ML programmers. All PE general and communication registers are then declared and any I/O ports and streams are declared lastly.

The constants, types and variables used by the MCL programmer (allocated in the controller) are defined after the configuration section. Procedures and functions used in the implementation of standard ML operations are given afterwards. The main program consists of code to initialize the mesh, and the ML program which has been compiled into a sequence of MCL statements and control code that decides which instructions are sent, or loops through a set of instructions. The five instructions below are used to send their corresponding MIL instruction to the mesh. They represent form (3) of mesh I/O; forms (1) and (2) are supported by the instructions presented later.

```

MCL_ENABLE(<compiled boolean condition using PE registers>);
MCL_DISABLE(<compiled boolean condition using PE registers>);

```

```

MCL_EXECUTE(<an expression using PE registers>);
MCL_SEND(a PE general register, a PE communication register, direction);
MCL_LOAD(a PE communication register, a PE general register);

```

The identifiers used by the ML programmer are converted into physical register names. Any code for arithmetic statements or boolean conditions is compiled into simple assembly-like format executable by the PEs, and any temporary registers required in a computation are selected from those not mapped to the ML user's program.

I/O ports are provided to the MCC to allow communication between the controller and the PEs of the MCC. This communications is accomplished by the MCL instructions:

```

MCLRead( <I/O_port>, <PE Register>, <Controller variable> );
MCLWrite( <I/O_port>, <Controller variable>, <PE Register> );

```

Where the type of the PE register and the controller variable must correspond. The *MCLRead* instruction will read a register from an I/O port PE into a controller variable, and the *MCLWrite* instruction performs the opposite. This is form (2) of I/O discussed in Section 4.3.

I/O streams are used to load or save data of the general registers of the MCC. A general register is specified for all PEs of the mesh, and the values for these registers are loaded into the PEs at one side of the mesh using the *MCLStreamInput* instruction. The *MCLStreamOutput* instruction is used to read the values of a register from the PEs on a side of the mesh. Data may be initially loaded into the registers of an $n \times n$ MCC by calling the *MCLStreamInput* instruction and then a send to shift the data, repeating this procedure until all values are loaded as discussed in Section 4.4.2. Unloading the results of computation is accomplished in the reverse fashion. The proper I/O bandwidth must be provided to the edges of the MCC to ensure that it can be loaded and unloaded in $O(dn)$ time, where d is the number of registers to load. The <I/O stream> is one of the I/O Streams listed in the configuration section. The two instructions are:

```

MCLStreamInput(<I/O stream>, <PE General Register>);
MCLStreamOutput(<I/O stream>, <PE General Register>);

```

The MCL programmer may access any register of a PE, in particular, the *StackTop* register is used to access the top of the *BoundsStack*, which initially contains the physical mesh bounds of (1,1) to (N,N). The mesh controller programs are responsible for ensuring that the ML operations execute within the boundaries presently at the top of the *BoundsStack*. A maximum diameter is associated with all boundaries at the top of the *BoundsStack*, this enables the MCL programmer to write control loops that send the correct number of steps to execute an ML operation in all submeshes. The controller is informed of

the maximum diameter of the current set of boundaries by the ML operations that add boundaries. It remembers these diameters using a stack in the controller.

4.4.4. Mesh Language.

Most systems stop at the controller and MCL level, but as Stout [62] and Perrot and Stevenson [51] indicate, more support is needed for the high-level parallel operations used in many systolic algorithms. The MCL programmer implements the standard operations for the ML language, and provides them as a ML library. One may have instead allowed the user to program in MCL, but this implies that he would have to write hardware specific code. In addition, he must provide a support package for recursion and the standard operations, and update this package should the mesh hardware change. The user would like to write high-level code for some "ideal" mesh that supports these operations automatically so that the parallelism in his *algorithm* is captured by that language instead of the parallelism of the *machine*.

The global structure of ML is a sequential list of ML parallel operations, and possible Pascal control statements that determine which ML operations are executed, or loop through a sequence of ML statements many times. This is similar to the format found in research papers, where a sequential description of parallel operations or arithmetic statements is given for the PEs of the mesh. The scope, or boundary, on which an instruction acts, introduced in Section 4.3, is used to conveniently express recursive subdivision, data compression and submeshes.

The overall format of a ML program follows Figure 4.5 below. The configuration section allows the user to set constants that define the mesh size and number of neighbors of a PE (4 or 8) required for his program. The constants for *ConnectionType* and *StandardPETypes* are automatically included in the configuration section in order that the MCL programmer may declare variables of those types. The ML programmer declares any PE registers used by his program in the PE variables section. A variable *current_bound* is included in the PE variables section to allow the user to refer to the top of the *BoundsStack*. The boolean constants set up by the MCL programmer are included in the I/O streams section of the ML program to allow the ML user to check which streams are active. Finally, the the ML programmer gives logical names to any I/O ports used in his program by referring to the MCL ports IDs that are included. In porting code from one mesh to another, the I/O ports section may have to be updated if they are different on the new mesh.

CONFIGURATION

Constants to define the mesh size and link strategy.

User variables that represent physical PE registers.

User named I/O ports and streams to be used in the algorithm.

PROGRAM

Variables for use by the algorithm to provide control of the operations called.

A set of user-defined procedures that perform certain tasks.

A user program, of standard MCC operations and procedure calls.

Figure 4.5. ML configuration and program format.

In the program section, the user defines any constants, types and variables that are to be used in Pascal control statements that are executed in the mesh controller. The `StandardMeshConst` and all `StandardPETypes` of Section 4.4.3 are also included in this section, in order that the ML user may know the physical dimensions of the mesh if required and define controller variables of `StandardPE` types. The declaration section is followed by user-defined procedures or functions, which contain sequences of instructions that are either Pascal or ML parallel operations. The type of the function can be any Pascal type, the *Bounds* type, or *PE_ID* types. Call by name is permitted as a parameter passing mechanism for PE variables only. This allows the user to build procedures where the PE variables may be passed into the procedure and substituted into ML operations inside the

procedure. Finally, the main ML program that is to be executed is given. As discussed previously, this section contains a sequential order of either Pascal code or standard ML operations. No expression that uses PE registers may appear outside of the standard ML operations. The Pascal code, that is used to control the sequence of ML instructions executed on the mesh, may only use the control variables, which are allocated in the controller.

MCC_Program <identifier>;

MCC_Configuration

MCC_N = <mesh size>; { The definition for an NxN mesh. }
 MCC_Connection = 8 | 4; { User selects connection scheme. }

{ StandardMeshConst for ConnectionType and connection names of Section 4.4.3. }
 { The StandardPETypes Section 4.4.3 are included here. }

PE_Variables { User-declared variables or one-dimensional constant-size arrays. }
 { The standard register named *PE* of Section 4.4.3 is included here. }
 current_bound: Bounds; { a variable that represents the top of the bounds stack. }
 [<identifier>[(<constant>)] .. : StandardPEType;] ... { See Section 4.4.3. }

I/O_Ports { A *constant* number *d* of those PEs specified by the mesh controller which
 can communicate directly to the mesh. }
 <Port_identifier> : <RowNo>, <ColumnNo>; ... { A logical name for the port. }
 <Port_identifierd> : <RowNo>, <ColumnNo>;

I/O_Streams

{ The boolean constants for the I/O streams of Section 4.4.2 are included here. }

EndConfiguration

Control_Types

- PE_ID and Bounds -
 { Standard and user-defined Pascal types. }

Control_Variables { Variables used in I/O. }

[<Control_identifier>, ... : <Control_Type>] ...

MCC_Procedures

{ User defined procedures or functions that may use control or PE code and ML
 standard operations. }

MCC_Begin

{ Control and/or PE code and ML standard operations. }

MCC_End.

Figure 4.6. ML program syntax.

The details of the syntax for a user's MCC program are provided in Figure 4.6 above. A PE's registers are referred to as variables in ML, and in the remainder of this section. The statement *<PE execute>* will refer to the *PE_Execute* operation, which contains instructions for modifying a PE's variables. That is, assignment statements and arithmetic and boolean expressions, that use PE variables only. Control code, denoted by *<control code>*, is executed in the mesh controller and determines which set of ML operations or statements in *PE_Execute* are to be executed by the mesh. Control code may contain any valid Pascal statements that use control variables only since this code is executed in the mesh controller.

A list of the standard operations are provided with a brief explanation of their function. The low-level execute, send and I/O operations are first presented, followed by the high-level standard ML operations. The notation of Section 4.4.1 is particularly important here.

LOW_LEVEL ML OPERATIONS:

The *PE_Execute* operation allows Pascal arithmetic and/or boolean instructions and assignments to be carried out in each enabled PE. Only PE variables may be specified in the expressions used, and any other ML standard operations are not permitted. If *All* is specified in the enable clause then all PEs execute the instructions sent by the controller. Control code may also appear in the enable clause, but the resulting executed statement must be a boolean condition using PE variables only. The enable condition enables the PEs that execute the enclosed code, and it must take constant time to execute. Typically a condition on a PE variable is specified. If no enable clause is provided, the default is *All*.

PE_Execute

```
<PE execute> [<control code>]; ...
[Enable All | <boolean condition using PE variables> [<control code>] ];
End;
```

EXAMPLE:

```
PE_Execute
  if grey_scale8 then
    color := shade mod 8
  else if grey_scale16 then
    color := shade mod 16
  else
    color := shade mod 32;
Enable
  shade > 0;
End;
```

In the example for *PE_Execute*, all underlined identifiers are control variables or constants, and have been allocated in the controller. The remaining are PE variables which are allocated as registers in each PE. The *if* statement controls which section of PE code is to be executed. In the enable clause, all PEs whose *shade* variable is greater than zero are

enabled.

Send

```
<List of readable PE variables>;
To Left | Right | Up | Down [ | UpLeft | UpRight | DownLeft | DownRight ];
Into <List of writable PE variables>;
[Enable All | <boolean condition using PE variables> [<control code>] ];
End;
```

The Send operation allows the contents of a set of source variables to be sent in a specified direction into a set of destination variables (in the *Into* clause) whose type match the corresponding variable in the *To* clause. The directions in square brackets are only permissible if the *ConnectionType* is 8. The enable clause is the same as for *PE_Execute*.

The operations to read and write from and I/O port are:

```
Read( <I/O_port>, <List of PE variables>, <List of Control variables> );
Write( <I/O_port>, <List of Control variables>, <List of PE variables> );
```

The types of the PE registers and the controller variables in the lists of each instruction must correspond. The read instruction will read a set of registers from a pre-defined port into control variables, and the write instruction performs the opposite.

The operations used to access an input or output stream are:

```
StreamInput(<InputStream>, <List of PE variables>);
StreamOutput(<OutputStream>, <List of PE variables>);
```

PE variables specified are loaded from, or put to, the specified stream. Files may be attached to streams, or if this is not done the controller will ask the user from which file the data is to be read. The values for the PE variables must appear in the file in the same order as specified in the list of PE variables.

HIGH-LEVEL ML OPERATIONS:

These are the standard ML operations that operate within boundaries. The operations that are used to set, change or remove boundaries are: *AddCurrent*, *RemoveCurrent*, and *CompressData*. These operations and their effect on other ML operations are discussed first.

The *BoundsStack* is a constant size stack of *Bounds* type variables. The top of the *BoundsStack* in each PE is referred to as the *current boundary*. That is, this is the boundary (submesh) in which the PE is enclosed. Any boundaries beneath the top of the *BoundsStack*, are from previously defined bounds that were added by previous ML operations which modify mesh boundaries. For instance, a call to the recursion operation automatically copies the top of the *BoundsStack* and places the new copy on top. This boundary is used in the split step of the recursion, which divides the mesh by dividing the

boundary found at the top of the *BoundsStack*. On each merge step the inverse operation is performed and, at the end of the recursive operation, the top of the *BoundsStack* is removed. Notice that the *BoundsStack* size does not grow with recursive subdivisions, but rather by calls to operations that place a new boundary on the stack. For example, if a data compression were performed at a recursive subdivision, then the new boundary of the compressed data would be placed on top of the stack until it is removed by the user. Care must be taken by the user to ensure that this boundary of the compressed data is removed before further subdivision or merging is performed by the recursion operation. The ML programmer may refer to the contents of the top of the stack by using the reserved variable *current_bound* which has two records, *top_left* and *bottom_right* of type *PE_ID*, that denote the top leftmost and bottom rightmost PEs in the boundary.

A general instruction to add a boundary to the top of the *BoundsStack* is *AddCurrent*, presented below.

AddCurrent

```
[<PE Bounds variable> MaxDiameter | <control variable> | <constant>];  
| Mesh;
```

End;

The *AddCurrent* operation is used to place the boundary defined in the PE variable of type *Bounds*, which is a record of the top-left and bottom-right PE IDs of the submesh, onto the top of the stack. The *MaxDiameter* clause must include a control variable or constant that gives the maximum number of rows or columns over all submeshes defined by the current boundary. This diameter may be obtained by performing a maximum over all diameters of the submeshes, and reading the result into a control variable using the *Read* operation. If the maximum diameter of the set of submeshes is known, or can be computed, it may be supplied as a constant or in a controller variable. If the clause *Mesh* is specified instead of a PE variable of type *Bounds* then the boundary (1,1) to (MCC_N, MCC_N) is placed on the *BoundsStack* and the maximum diameter is automatically set to MCC_N. This operation can be used by the ML programmer to define a set of submeshes on the mesh. ML instructions will then be executed in parallel in each submesh. It is the user's responsibility to ensure that each PE within a defined submesh has the correct bounds of the submesh.

RemoveCurrent;

The *RemoveCurrent* operation is used to remove to top of the *BoundsStack*. The user may only remove boundaries that were added using *AddCurrent*, or *CompressData* where the *Giving* clause specified *NewCurrent*. In other words, only boundaries specifically added by the user may be removed.

CopyCurrent <PE Bounds variable> End;

The *CopyCurrent* operation is used to copy the top of the *BoundsStack* into a PE variable of type *Bounds*.

CompressData

<List of read/write PE variables>;
 To TopLeft | TopRight | BottomLeft | BottomRight;
 Using <PE boolean variable>;
 Giving [NewCurrent | Current];
 MaxDiameter ReturnTo(<control variable>) | <control variable> | <constant>;
 End;

The *CompressData* operation compresses data into a submesh at a specified corner within the current boundary. Before using the *CompressData* operation, the user must first flag the PEs that contain relevant data by setting the boolean variable specified in the *Using* clause to be true. The listed variables of all flagged PEs are moved as a unit to another PE in a submesh that will contain the compressed data. Since this operation executes within the current boundary, only the flagged PEs within each boundary are compressed into a submesh, and this is placed in the specified corner of that boundary. The *Giving* clause determines whether the new boundary (or boundaries) of the compressed data are added to the top of the stack, or whether they replace the top of the *BoundsStack*. If *NewCurrent* is specified in the *Giving* clause, the new boundary of the compressed data is added to the top of the *BoundsStack* in each PE within that boundary. If *Current* is specified in the *Giving* clause, the top of the *BoundsStack* is replaced by the new compressed boundary. For PEs outside the compressed boundary, a special non-active boundary is used, implying that these PEs are not part of the computation as long as the compressed boundary remains. *Current* should only be used if the current boundary was added by the user, or by a previous call to *CompressData*.

The ML operations, for example a row rotation, must know the maximum number of steps to execute in order that each submesh completes the operation. The maximum number of rows or columns over all of the compressed boundaries must either be supplied using a control variable or constant, or it can be returned from the mesh into a control variable using the *ReturnTo* operation. Since *ReturnTo* performs a maximum operation over all boundaries, it requires $O(MCC_N)$ time to execute. The *CompressData* operation generates the smallest $k \times k$ submesh that contains the data of the selected set of PEs. Since only integer values may be used for k , some of the PEs in each submesh may not contain valid data, or if too much data is relevant the new square submesh may be larger than the current boundary (an error). The user must be aware of this when writing programs that use data compression.

BoundedSend

```

<List of readable PE variables>;
To Left | Right | Up | Down [ | UpLeft | UpRight | DownLeft | DownRight ];
Into <List of wrtable PE variables>;
[Enable All | <boolean condition using PE variables> [<control code>] ];
End;

```

The *BoundedSend* operation performs the same function as the *Send* operation except that it is boundary sensitive, i.e. it allows no data to be sent across the current boundary. If the user issues a *BoundedSend* to the *Left* say, then all PEs on the left side of the current boundary do not send.

Broadcast

```

<List of read/write PE variables>;
From <boolean condition using PE variables> > [<control code>];
End;

```

The *Broadcast* operation broadcasts a list of variables from an enabled PE into the same variables of the PEs within the current boundary. Since the broadcast operation, and the operations listed below, may operate in recursive subdivisions or on submeshes, the boolean function or condition provided must enable *one* (or none, if a broadcast is not desired) PE only in each subdivision or submesh.

BroadcastInRows | BroadcastInColumns

```

<List of PE variables>;
Rows | Columns
<boolean condition using PE variables> [<control code>];
End;

```

The *BroadcastInRows* or *BroadcastInColumns* instruction must have one PE per submesh row or column enabled (unless no broadcast is to be performed) using the *Rows* or *Columns* clause. The listed variables of the enabled PE are then broadcast in its row or column within the current boundary.

RowRotation | ColumnRotation

```

<List of readable PE variables>;
Rows | Columns All | <boolean condition using PE variables> [<control code>];
Temporary <List of read/write PE variables>;
Perform
<PE execute> [<control code>]; { executed after each step in the rotation }
End;

```

In the *RowRotation* or *ColumnRotation* operation, all PEs of the row or column to be rotated must be enabled using the *Rows* or *Columns* clause. The first list of variables is then rotated through the PEs of the row or column within the current boundary, using the list of temporary variables provided in the *Temporary* clause to store a copy of the shifting data. The *Temporary* clause variable set must correspond in type to the list of variables being rotated. At each step in the rotation, i.e. after the contents of the variables of another

PE of the MCC are loaded, the code in the *Perform* clause is executed.

RowBroadcast | ColumnBroadcast

```

<List readable PE variables>;
Row | Column <boolean condition using PE variables> [<control code>];
Temporary <List of read/write PE variables>;
Perform
  <PE execute> [<control code>]; {after each step in the rotation}
End;
```

In the *RowBroadcast* or *ColumnBroadcast* operation, the PEs of the row or column to be broadcast must all be enabled using the *Row* or *Column* clause's boolean condition. The contents of the list of variables in the row or column enabled are then broadcast within the current boundary. The *Temporary* and *Perform* clauses are similar in function to the *RowRotation* operation.

RAR | RAW

```

<List of readable PE variables>;
From | To <PE_ID variable>;
Into <List of read/write PE variables>;
End;
```

In the *RAR* or *RAW* operation, a list of variables to read or write, and a variable of type *PE_ID* in which the row-major ID of the PE to read or write to, are provided. For *RAR*, the variables specified in the PE addressed in the *From* clause are read into the *Into* clause's variables of the current PE. If the ID contained in the *From* clause is infinite that PE does not read. In *RAW*, the contents of the first list of variables of the PE are written into the *Into* clause's variables of the *To* clause PE whose ID is contained in the *PE_ID* variable specified. If that ID is infinite then this PE does not write.

Sort

```

Key <List of readable PE variables>;
[Data <List of readable PE variables>;]
Into increasing | decreasing;
Order Row-Major | Column-Major | Shuffled-Row-Major | Snake-Like-Row-Major |
  Snake-Like-Column-Major | Proximity;
Temporary
  Key <List of read/write PE variables>;
  [Data <List of read/write PE variables>;]
End;
```

In *Sort* the *Key* variable set is used in the order listed to perform key comparisons. The data within each current boundary is sorted into one of increasing or decreasing order, specified in the *Order* clause. Some of the sorted orderings listed above are described in Section 2.1.4. The *Temporary* variables are used in comparing two records; for this reason they must correspond in type and be distinct from the variables listed in the first *Key* and *Data* clauses. The *Data* variables should also be distinct from the *Key* variable list. At the

end of the sort, the contents of all variables specified in the *Key* and *Data* section will have been sorted into their appropriate PE. If no data is specified, only the key records are moved.

The next instruction, *EnableSubmesh*, is used to enable submeshes in which the ML programmer wants ML operations to be performed. For instance, some submeshes may be too small for certain operations, such as a recursive subdivision, or a call to a sort. The ML programmer would therefore like a sequence of ML operations to apply only to those submeshes which are not degenerate cases (small submeshes) or on submeshes in which the data is relevant.

EnableSubmesh

```
<boolean condition on PE variable current_bound>;
Perform
  [ML operations] [<control code>];
End;
```

In the *EnableSubmesh* clause, a boolean condition on the *current_bound* of each PE is used to determine whether to enable the PEs of that submesh. The code in the *Perform* clause is executed in each enabled submesh. *EnableSubmesh* is discussed further in the *Recurse* operation, where it is particularly useful in disabling a split (or merge) in a subdivision generated by recursion.

The *Recurse* operation provides support for recursion on the mesh. The type of recursion could be one of the predefined types of *Square* for square recursion, and *Divide* for divide-and-conquer recursion (see Section 2.1.3). If the *Divide* type is specified, the first split is vertical (i.e. along the middle column), and the splitting then alternates horizontally and vertically. Code executed during the split step is provided (if any) in the *Split* clause, in which the keyword *SplitMesh* must occur once. The point at which the keyword *SplitMesh* occurs is when the mesh is split according to the recursion type specified. A condition to stop execution of the statements in the *Split* clause is supplied in the clause *StopSplit*. *StopSplit*'s condition is used in determining whether all submeshes have split to the final level. The recursion operation's syntax is:

Recurse

```
RecursionType Square | Divide | <user provided split and merge procedures>;
Split
  [<PE execute>] [ML operations] [<control code>];
StopSplit <control code: boolean condition>;
InitialSolve [<PE execute>] [ML operations] [<control code>];
Merge
  [<PE execute>] [ML operations] [<control code>];
StopMerge <control code: boolean condition>;
End;
```

After splitting to the desired depth, the code for the initial merge step in the *InitialSolve* clause is performed. The general merge steps in the *Merge* clause are then applied to all subdivisions until the *StopMerge* condition is satisfied. The instruction *MergeMesh* must occur at one point in the *Merge* clause. After this instruction, the neighboring submeshes of a subdivision at a recursive level are merged. The example of Section 4.5 illustrates divide-and-conquer recursion.

EnableSubmesh may be used in the *Split* clause to enable selected subdivisions for a split by enclosing the *SplitMesh* statement (and other statements of the *Split* clause if desired) within the *Perform* clause of *EnableSubmesh*. In the *Merge* clause, care must be taken to ensure that all of the neighbors of the mesh that would be merged in the step are enabled. If this is not done some subdivisions will increase their bounds to include neighbors that have not merged with them because they were not enabled. The same method used in the split step to disable subdivision of the mesh may be employed in the *Merge* clause by enclosing the *MergeMesh* statement in the *EnableSubmesh* operation. This concludes the discussion on the *Recurse* operation.

The next operation is used to flag PEs that are at some specified side of the current boundary; the syntax is:

```
[Top] [Left] [Bottom] [Right]
  <Boolean PE variable>
End;
```

The boolean PE variable is set to true if the PE is on the current boundary of all of the sides listed in the operation. For instance, the top-left PE in a boundary may be selected by calling **Top Left** <boolean PE variable> **End;**. Illegal cases such as specifying **Top** and **Bottom** are checked for at compilation.

The next section presents a recursive algorithm that is programmed using ML operations. The example illustrates the format of an ML program, and the use of some of its operations.

4.5. AN EXAMPLE USING ML.

This section presents an MCC algorithm written in ML to solve the maximal elements problem [11]. Dehne [11] defines a dominance relation among a set S of planar points as: a point q dominates a point p , written $p \leq q$, iff $p.x \leq q.x$ and $p.y \leq q.y$. A point s in S is called *maximal* if there exists no point s' such that $s \leq s'$. The example provided in this section will find the *maximal* elements of a set of n^2 planar points on an $n \times n$ MCC.

The solution presented in [11] uses a divide-and-conquer approach. Initially all points are marked as maximal. The point set S is subdivided by x -coordinate into two sets S_1 and S_2 that are stored in two halves L and R of the mesh. See Section 2.1.3 for details on

subdividing the MCC in divide-and-conquer recursion. The maximal points in S_1 and S_2 are found recursively, and the left, L , and right, R , halves are merged by finding the maximal point p in S_2 that has the minimum x -coordinate (maximum y -coordinate in case of ties). This point is broadcast to S_1 , and all maximal points of S_1 that are dominated by p are no longer maximal.

Figure 4.7. Recursively finding maximal points.

Figure 4.7 illustrates the operation of the algorithm. Figure 4.7(a) shows the divisions of the point set S into two parts. These are stored in the MCC as shown in Figure 4.7(b). The maximal points in each subset are then found recursively. In Figure 4.7(a) the maximal points in each subset have been circled, and the minimum x -coordinate maximal point in S_2 is enclosed in a box. It can be seen that the boxed point in set S_2 may eliminate maximal points in set S_1 .

ML code for finding the maximal points in a set of n^2 planar points stored one point per PE on an $n \times n$ MCC is now presented. The same notation as in Section 4.4.1 applies.

MCC Program Maximal;

MCC Configuration

MCC_N = 16; { a constant (preferably a power of 2) ==> 16x16 points }

MCC_Connection = 4;

PE Variables

px,py: *Real*; { The point being stored at this PE. }

temp_px, temp_py: *Real*; { variables for temporary points. }

temp_x1,temp_y1, temp_x2, temp_y2: *Real*;

maxpoint, temp_max1, temp_max2, temp_maxp: *Boolean*; { Is this maximal? }

BRside: *Boolean*; { Used in testing if a PE is at a certain side of a boundary. }

I/O Streams

LeftInputStream = True; { These streams may be used. }

LeftOutputStream = True; { ... the state of the other streams is not shown. }

EndConfiguration**Control Variables**

cut_vertical: *Boolean*; { Used in determining the split boundary. }

split_depth: *Integer*; { The depth of recursion. }

I: *Integer*; { Loop variable. } —

MCC_Begin

```

    { Input the points. }
1  For I:= 1 to MCC_N Do Begin
    StreamInput(LeftStream,px,py);
    Send { Undefined px,py values are also sent, but will be overwritten, }
      px,py;
    To Right;
    End;
  End;

  { The first recursive split in Divide recursion is always vertical. }
2.  cut_vertical:= True;
    split_depth:= 0; { Depth is currently zero. }

  { Perform the recursive subdivisions. }
3  Recurse

3.1  RecursionType Divide;

3.2  Split { Sort points depending on the split to be made. }

3.3  If cut_vertical Then { Making a vertical split. }
    Sort
    Key px py;
    Into increasing;
    Order Snake-Like-Column-Major;
    Temporary Key temp_x1 temp_y1;
    End
  Else { Making a horizontal split. }
    Sort
    Key px py;
    Into increasing;
    Order Snake-Like-Row-Major;
    Temporary Key temp_x1 temp_y1;
    End;
    split_depth:= split_depth + 1;

3.4  SplitMesh; { Perform a recursive split. }
3.5  cut_vertical:= Not cut_vertical; { Next split line. }

3.6  StopSplit { Stop when the subdivision gives single PE submeshes. }
    split_depth = 8; { 2 * log2(16) splits are done. }

```

3.7 Initial Solve { Initially all points are maximal. }

PE_Execute
 maxpoint := True;
 End;

3.8 Merge

{ Find the maximal point with minimum x-coordinate (handling ties) in each subdivision; and store it in temp_px and temp_py. }

3.9 PE_Execute

temp_x1 := px; { Initial guess for min. x maximal p in the rows. }
 temp_y1 := py;
 temp_max1 := maxpoint; { In case this is a bad guess. }

End;

3.10 RowRotation { Find minimum x-coordinate maximal p in rows. }

px py maxpoint;

Rows All;

Temporary

temp_x2 temp_y2 temp_max2;

Perform

If Not temp_max1 *and* temp_max2 *Then Begin*

temp_x1 := temp_x2;

temp_y1 := temp_y2;

temp_max1 := True;

End;

If temp_max2 *and* (temp_x2 < temp_x1) *Then Begin*

temp_x1 := temp_x2;

temp_y1 := temp_y2;

End

Else If temp_max2 *and* (temp_x1 = temp_x2)

and (temp_y1 < temp_y2) *Then*

temp_y1 := temp_y2;

End;

3.11

PE_Execute

temp_px := temp_x1; { Guess for min. x maximal p in the subdivision. }

temp_py := temp_y1;

temp_maxp := temp_max1; { In case this is a bad guess. }

End;

3.12

ColumnRotation

temp_x1 temp_y1 temp_max1;

Columns All;

Temporary

temp_x2 temp_y2 temp_max2;

Perform

If Not temp_maxp *and* temp_max2 *Then Begin*

temp_px := temp_x2;

temp_py := temp_y2;

temp_maxp := True;

End;

If temp_max2 *and* (temp_x2 < temp_px) *Then Begin*

temp_px := temp_x2;

temp_py := temp_y2;

End

```

        Else If temp_max2 and (temp_px = temp_x2)
            and (temp_px < temp_y2) Then
            temp_py := temp_y2;
        End;

3.13    MergeMesh; { Remove subdivision boundaries. }
        split_depth := split_depth - 1;

3.14    Bottom Right { Set BRside true in all PEs at the bottom right of a
                    subdivision. }
        BRside;
        End;

3.15    Broadcast
        temp_px temp_py; { The min. x coord. maximal point in S2. }
        From side = True; { Select Bottom Right PE }
        End;

3.16    PE_Execute { Eliminate maximal PEs in S1 that are covered. }
        maxpoint := False;
        Enable { Select PEs in S1 using px. Don't eliminate the maximal PE of S2
                that was broadcast. }
            (px <= temp_px) and (py <= temp_py) and (BRside = False);
        End;

3.17    StopMerge split_depth = 0;
        End; {Recurse}
MCC_End.

```

In line 1, the points are input into the mesh. The variable *cut_vertical* of line 2 is used in the split step to determine which is the next subdivision in the Divide type of recursion. Line 3 starts the entire recursive process for this algorithm, and line 3.1 specifies the type as divide-and-conquer. The split step is performed by sorting the data into the appropriate half of the mesh. Notice, in snake-like row major ordering, the mesh can be divided in two horizontally, and the lower half of the elements will be in the top rows, while the upper half of the elements are in the bottom rows of the mesh. Snake-like column major ordering allows a vertical division by the middle column for similar reasons. Line 3.2 starts the split clause, in which line 3.3 determines what the next split line is, and calls the appropriate sort operation. After sorting and incrementing the *split_depth*, the keyword *SplitMesh* of line 3.4 subdivides the mesh using the Divide splitting method. In line 3.5, *cut_vertical* is toggled for the next split, if any. Line 3.6 determines if the subdivision has resulted in single PE submeshes. If so, splitting is terminated.

Since the point in single PE submeshes is maximal, *maxpoint* is set to true as the initial merge step of line 3.7. In the general merge step we are merging two neighboring submeshes. We must determine the maximal PE *p* with the smallest *px* value in submesh storing the points for *S₂*. For simplicity, this point is found in both submeshes. Lines

3.10, 3.11, and 3.12 accomplish this using row and column rotations. The coordinates of p for each submesh is stored in variables *temp_px* and *temp_py*. We now must broadcast p of S_2 to S_1 . This is accomplished by first removing the boundaries between the neighboring submeshes by using *MergeMesh* in line 3.13, where the split depth is also decremented. In the merging of either a vertical or horizontal split, the bottom-rightmost PE is always a member of S_2 . This PE is selected using line 3.14, and line 3.15 broadcasts the value of point p , stored in (*temp_px*, *temp_py*), to the PEs in both S_1 and S_2 for simplicity. The *PE_Execute* of line 3.16 eliminates any maximal PEs in S_1 that are covered by p by enabling the PEs with x -coordinate less than p 's. (Any PEs in S_1 and S_2 with x -coordinate equal to p , but with y -coordinate less than p 's are also eliminated by this statement.) The merging continues until the original boundary (Mesh) is achieved (tested for in line 3.17), after which the PEs whose *maxpoint* variable is true contain maximal points, the coordinates of which are stored in variables *px* and *py*.

4.6. CONCLUSIONS AND IMPROVEMENTS.

Why design a language such as ML? The main reason is to allow the user to easily program a mesh algorithm from a research paper. Algorithms are usually presented in a high-level fashion, using parallel operations, and control statements to decide the order of execution of the algorithm's statements. In addition algorithms are presented in a machine independent manner. That is, low-level hardware details are not discussed in a research paper, unless the algorithm executes on a architecture designed for a specific task, i.e. computing Fast Fourier Transforms by special-purpose hardware. The next paragraphs outline how these features, summarized below, are supported by the design of ML.

- 1) A top-down programming view of the mesh.
- 2) Control of the statements to be executed by the mesh.
- 3) Portability of ML programs.

The top-down programming view of the mesh is accomplished in ML by providing automatic recursion, and the parallel MCC operations discussed in Section 2.1. The ML programmer may therefore code his algorithm as if these operations were present on the MCC, and need not be concerned with the low-level details of their implementation (or of in fact implementing them for a particular mesh architecture). In addition, the recurse operation provides two standard types of subdivision strategies for the MCC, and permits the user to specify his own strategy. The idea of defining a current boundary in which high-level ML operations operate and allowing parameter passing by name for PE variables permits the ML user to write his own operations, or to modularize his program. If ML users write procedures generally, so that they perform operations on the current boundary

instead of the entire mesh, these procedures may be called as steps in recursion, or for multiple submeshes. The ML language allows the user to express the parallelism in his algorithm by using operations designed for the MCC architecture.

Other operations such as pipelining and sweeping data across the mesh, as in the visibility algorithms presented in Chapter 3, are useful and an improvement would be to include standard operations for them. For now, the user may write these as procedures. More powerful or new MCC operations may be defined in research papers; they may be added to ML by an MCL programmer, thereby making them standard operations. In fact, a good addition would be the Concurrent Read and Concurrent Write operations defined by Miller and Stout [40] (see Section 2.2.2).

The PE_Execute, EnableSubmesh and Pascal control statements provide control of the code executed by the mesh. In PE_Execute the user selects the PEs of the mesh which execute arithmetic or boolean statements on their registers. The EnableSubmesh operation performs the corresponding function on submeshes by allowing the user to select the submeshes that will execute a set of ML operations. Finally, the Pascal control statements allow the user to select which ML statements are executed, or to repeatedly execute a set of statements on the mesh.

Portability of code is supported by the hierarchy of languages (MIL, MCL and ML of Section 4.2) defined for the system thereby permitting the physical architecture or instruction set of a PE to change without impacting ML programs. This language could provide a front-end to a MCC simulator as well, by having the simulator execute the MIL instructions sent to it in each PE.

Most of the existing parallel languages, such as Poker, Glypnir and CMLisp discussed in Section 4.1, provide certain control statements. Some, like Glypnir, are not portable since they provide hardware-specific parallel operations. Most do not provide support for top-down programming i.e. support for the parallelism in an algorithm. Additionally, standard operations are difficult to program in the languages that are not designed from the controller's point of view, as in Poker. Cmlisp supports all of the above listed items, but users are restricted to using the Xector data structure which is allocated by the system to PEs of the Connection Machine.

The current ML programming environment consists of a hierarchy of languages, where programming is done from the point of view of a controller which supplies high-level parallel operations. In addition to this, providing a debugger for all levels of the hierarchy, in particular the ML level, would speed up an algorithm's implementation. This may be accomplished by providing high-level ML operations that perform debugging functions, such as displaying a register of all PEs, displaying the top of the *BoundsStack*, or allowing the user to single step through ML operations.

The definition of a high-level language that defines the operations of a parallel architecture is applicable to any of the systolic architectures discussed in Section 1.4. In fact, some authors have defined operations for other architectures, such as the Pyramid computer [63].

REFERENCES.

- [1] Aho A.V., Hopcroft J.E. and Ullman J.D. Data Structures and Algorithms. Addison-Wesley Pub. Co. Reading, Massachusetts U.S.A., Don Mills, Ontario Canada. 1983.
- [2] Atallah M.J. and Hambrusch S.E. *Solving tree problems on a mesh-connected processor array*. Information and Control, vol. 69, Nos. 1-2, April/May/June - 1986, pp. 168-187.
- [3] Atallah M.J. and Kosaraju S.R. *Graph Problems on a Mesh-Connected Processor Array*. J. ACM, vol. 31, no. 3, July 1984, pp. 649-667.
- [4] Batcher K.E. *Design of a massively parallel processor*. IEEE Trans. on Comput. vol. C-29, no. 9, Sept. 1980, pp. 836-840.
- [5] Chazelle B. *Computational geometry on a systolic chip*. IEEE Trans. on Comput., vol. C-33, no. 9, Sept. 1984, pp. 774-785.
- [6] Colley S., Hayes J.P., Mudge T.N., Palmer J., and Stout Q.F. *Architecture of a hypercube supercomputer*. Proc. 1986 Int'l. Conf. on Parallel Processing, pp. 653-660.
- [7] Danielson P.E. and Levialdi S. *Computer architecture for pictorial information systems*. IEEE Computer, Nov. 1981.
- [8] Dean J.A., Lingas A., and Sack J-R. *On recognizing polygons, or how to eavesdrop*. Proc. 24th Annual Allerton Conf. on Communication, Control and Computing, Oct 1-3, 1986, pp. 324-333.
- [9] Dean J.A. and Sack J-R. *Efficient hidden-line elimination by capturing winding information*. Proc. 23rd Annual Allerton Conf. on Communication, Control, and Computing. Allerton House Monticello, Illinois, Oct. 2-4, 1985, pp. 496-505.
- [10] Dehne F. *Parallel computational geometry and clustering methods*. Ph.D. Thesis, Bayrischen Julius-Maximilians Universität Würzburg, Hannover, West Germany 1986.
- [11] Dehne F. *$O(n^{1/2})$ algorithms for the maximal elements and ECDF searching problem on a mesh-connected parallel computer*. Infor. Processing Letters 22 (1986), pp. 303-306.
- [12] Dehne F. *Solving visibility and separability problems on a mesh-of-processors*. School of Comp. Sci., Carleton University, Ottawa Canada, K1S-5B6, Tech. Rep. SCS-TR-182, Nov. 1987.
- [13] Dehne F. and Sack J-R. *Separability of sets of polygons*. School of Comp. Sci., Carleton Univ., Ottawa Canada, K1S-5B6, Tech. Rep. SCS-TR-82, Oct. 1982.

-
- [14] Dehne F., Sack J.-R. and Santoro N. *Computing on a systolic screen: Hulls contours and applications*. Proc. Esprit Conf. on Parallel Processing; Eindhoven Netherlands, June 1987, (Springer Lecture Notes in Computer Science.).
 - [15] Dyer C. R. *A VLSI pyramid machine for hierarchical image processing*. Proc. IEEE 1981 Conf. on Pattern Recognition and Image Processing, Dallas, Texas, Aug. 1-5 1981, pp. 381-386.
 - [16] Edelsbrunner et. al. *Stabbing line segments*. BIT, vol. 22, pp. 274-281, 1982.
 - [17] Foley J.D. and Van Dam A. Fundamentals of Interactive Computer Graphics. Addison-Wesley, Reading Massachusetts, Don Mills Ontario, 1982, pp. 301.
 - [18] Guerra C. and Hambrusch S. *Parallel algorithms for line detection on a mesh*. To appear in Journal of Parallel and Distributed Computing.
 - [19] Güting R.H., Nurmi O., and Ottman T. *The direct dominance problem*. Proc. of the Symposium on Computational Geometry, Baltimore, Maryland, June 5-7 1985, pp. 81-88.
 - [20] Hambrusch S.E. *Parallel algorithms for bridge- and bi-connectivity on minimum area meshes*. Technical Report CSD-TR-506 Jan. 1985, Dept. of Computer Sciences, Purdue University, West Lafayette, IN 47907 U.S.A.
 - [21] Hillis W.D. The Connection Machine. The MIT Press, Cambridge Massachusetts, London England, 1985.
 - [22] Jeong C-S and Lee D.T. *Parallel geometric algorithms on mesh-connected computers*. 1987 Fall Joint Computer Conference.
 - [23] Klette R. *A parallel computer for image processing*. Elektronische Informationsverarbeitung und Kybernetik. Vol. EIK 15 5/6, 1979, pp. 237-263.
 - [24] Knuth D.E. The Art of Computer Programming. Vol. 3: Sorting and Searching. Addison-Wesley, Reading MA, 1973.
 - [25] Kumar V.K.P. and Eshaghian M.M. *Parallel geometric algorithms for digitized pictures on mesh of trees*. Proc. 1986 Int'l. Conf. on Parallel Processing, pp.270-273
 - [26] Kung H.T. *Why systolic architectures?* Computer vol. 15, no. 1, Jan. 1982, pp. 37-42.
 - [27] Lang H.W., Schimpler M., Smeck H. and Schröder H. *Systolic sorting on a mesh-connected network*. IEEE Trans. on Comput., vol. C-34, no. 7, July 1985, pp. 652-658.
 - [28] Languages and Architectures for Image Processing. M.J.B. Duff and S. Levialdi (Eds.), Academic Press, London, New York, Toronto, 1981.
 - [29] Kushner T., Wu A.Y. and Rosenfeld A. *Image processing on MPP*. Pattern Recognition vol. 15, no. 3, 1982, pp. 121-130.

-
- [30] Lawrie D.H., Layman T., Baer D. and Randal J.M. *Glypnir - A programming language for Illiac IV*. Comm. ACM, vol. 18, no. 3, March 1975, pp. 157-164.
 - [31] Lee D.T. and Preparata F.P. *Computational geometry - A survey*. IEEE Trans. on Comput., Vol. C-33, No. 12, December 1984, pp. 1072-1101.
 - [32] van Leeuwen J. *Distributed computing*. Dept. Computer Sci., Univ. Utrecht, Utrecht, The Netherlands, Tech. Report RUU-CS-82-8, 1982.
 - [33] Lodi E. and Pagli L. *A VLSI solution to the vertical segment visibility problem*. IEEE Trans. on Comput., vol. C-35, no. 10, Oct. 1986, pp. 923-928.
 - [34] Lozano-Pérez T. *Spatial planning: A configuration space approach*. IEEE Trans. on Comput., vol. C-32, no. 2, Feb. 1983, pp. 108-120.
 - [35] Lu M. *Constructing the Voronoi diagram on a mesh-connected computer*. Proc. 1986 Int'l. Conf. on Parallel Processing, pp. 806-811.
 - [36] Lu M. and Varman P. *Mesh-connected computer algorithms for rectangle-intersection problems*. Proc. 1986 Int'l. Conf on Parallel Proc., pp. 301-307.
 - [37] Lu M. and Varman P. *Solving geometric proximity problems on mesh-connected computers*. Proc. 1985 Workshop on Computer Architecture for Pattern Analysis and Image Database Management, pp. 248-255.
 - [38] Mao X., Kunii T.L., Fujishiro I., and Noma T. *Hierarchical representations of 2D/3D gray-scale images and their 2D/3D two-way conversion*. IEEE Computer Graphics and Applications, vol. 7, no. 12, Dec. 1987, pp. 37-44.
 - [39] Miller R. and Stout S. Parallel Algorithms for Regular Architectures. MIT Press, Mass., 1987.
 - [40] Miller R. and Stout S. *Mesh computer algorithms for computational geometry*. Research Report 86-18, Dept. of Computer Science State University of New York at Buffalo, 226 Bell Hall Buffalo, New York 14260.
 - [41] Miller R. and Stout S. *Geometric algorithms for digitized pictures on a mesh-connected computer*. IEEE Trans. on Pattern Analysis and Machine Intelligence, vol. PAMI-7, no. 2, March 1985, pp. 216-228.
 - [42] Miller R. and Stout S. *Computational geometry on a mesh-connected computer*. Proc. 1984 Intl. Conf. on Parallel Processing, pp. 66-73.
 - [43] Millstein R.E. *Control structures in Illiac IV Fortran*. CACM, vol. 16, no. 10, Oct. 1973, pp. 621-627.
 - [44] Munro J.I., Overmars M.H., and Wood D. *Variations on visibility*. Proc. 3rd Annual Symp. on Computational Geometry, Waterloo, Ont. Canada, June 8-10 1987, pp. 291-299.
 - [45] Nassimi D. and Sahni S. *Bitonic sort on a mesh-connected parallel computer*. IEEE Trans. on Comput., vol. C-27, no. 1, January 1979, pp. 2-7.

-
- [46] Nassimi D. and Sahni S. *Data broadcasting in SIMD computers*. IEEE Trans. on Comput., vol. C-30, no. 2, February 1981, pp. 101-107.
- [47] Nassimi D. and Sahni S. *Finding connected components and connected ones on a mesh-connected parallel computer*. SIAM J. Comput., vol. 9, no. 4, Nov. 1980, pp. 744-757.
- [48] Nurmi O. *On translating a set of objects in two- and three- dimensional space*. Computer Vision, Graphics and Image Processing, vol. 36, 1986, pp. 42-52.
- [49] Parallel Computer Vision. L. Uhr (Ed.) Academic Press, Inc. Orlando, Florida 1987.
- [50] Ottman T., Widmayer P., and Wood D. *A worst-case efficient algorithm for hidden-line elimination*. Computer Science Tech. Rep. CS-82-33, University of Waterloo, Waterloo Ontario, Oct. 1983.
- [51] Perrott R.H. and Stevenson D.K. *User's experiences with the Illiac IV system and its programming languages*. Sigplan Notices (1981), vol. 16, no. 7, pp. 75-88.
- [52] Preparata F.P. and Shamos M.I. Computational Geometry: An Introduction. Springer-Verlag: New York, Berlin, Heidelberg, Tokyo, 1985.
- [53] Preparata F.P. and Vuilleman J. *The cube-connected cycles: A versatile network for parallel computation*. C. ACM, vol. 24, no. 5, May 1981, pp. 300-309.
- [54] Pyramidal Systems for Computer Vision. V. Cantoni and S. Levialdi (Ed.) Springer-Verlag Berlin, Heidelberg, New York, London, Paris, Tokyo 1986.
- [55] Reeves A.P. *Survey, parallel computer architecture for image processing*. Computer Vision, Graphics and Image Processing, Vol. 25, 1984.
- [56] Rosenfeld A. *Some Pyramid techniques for image segmentation*. Pyramidal Systems for Computer Vision. V. Cantoni and S. Levialdi (Ed.) Springer-Verlag Berlin, Heidelberg, New York, London, Paris, Tokyo 1986, pp. 261-271.
- [57] Sack J-R. and Toussaint G. *Separability of pairs of polygons through single translations*. Proceedings STACS 85, Saarbrücken Germany, Jan. 1985, pp. 310-321. Also to appear in Robotica Jan. 1987.
- [58] Snyder L. *Parallel programming and the Poker programming environment*. Computer, vol. 17, no. 7, July, 1984, pp. 27-36.
- [59] Snyder L. *The Poker (4.0) programmer's reference guide*. Tech. Rep. 86-05-04, University of Washington, 1986.
- [60] Srihari S.N. *Representation of three-dimensional digital images*. Computing Surveys, vol. 13, no. 4, Dec. 1981, pp. 399-423.
- [61] Sternberg S.R. *Biomedical image processing*. IEEE Computer, vol. 16, Jan. 1983, pp. 22-34.

- [62] Stout Q.F. *Supporting divide-and-conquer algorithms for image processing.* Journal of Parallel and Distributed Computing, vol. 4, no. 1, Feb. 1987.
- [63] Stout Q.F. *Pyramid algorithms optimal for the worst case.* Parallel Computer Vision. L. Uhr (Ed.) Academic Press, Inc. Orlando, Florida 1987, pp. 147-168.
- [64] Thompson C.D. and Kung H.T. *Sorting on a mesh-connected parallel computer.* Communications of the ACM vol. 20, no. 4, April 1977, pp. 263-271.
- [65] Sutherland L.E., Sproul R.F., and Schumaker. *A characterization of ten hidden-surface algorithms.* Computing Surveys, vol. 6, no. 1, March 1974, pp. 1-56.
- [66] Toussaint G.T. *Movable separability of sets,* Computational Geometry: Machine Intelligence and Pattern Recognition. G.T. Toussaint (Ed.) North Holland, Amsterdam, New York 1985.
- [67] Toussaint G.T. *Pattern recognition and geometric complexity.* Proc. 5th Int'l. Conf. on Pattern Recognition, 1980, pp. 1324-1347.
- [68] Van Scoy F.L. *The parallel recognition of classes of graphs.* IEEE Trans. Comput., vol. C-29, 1980, pp. 563-570.

END

2008.88

FIN