Parallel Pipelined VLSI Arrays for Real-Time Image Processing

by

Faridah M. Ali

Dissertation submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Doctor of philosophy

in

Electrical Engineering

APPROVED:

[illegible]
Morton Nadler, Chairman

[illegible]
K. B. YU

[illegible]
J. G. Tront

[illegible]
J. P. Bixler

[illegible]
J. R. Armstrong

May, 1988

Blacksburg, Virginia

Parallel Pipelined VLSI Arrays for Real-Time Image Processing

by

Faridah M. Ali

Morton Nadler, Chairman

Electrical Engineering

(ABSTRACT)

Real-time image processing involves processing a wide spectrum of algorithms on huge data sets. Processing at the pixel data rate demands more powerful parallel machines than those developed for conventional image processing.

This research takes advantage of current VLSI technology to examine a new approach for processing arbitrary algorithms at real-time data rate. It is based on embedding the algorithms, expressed by their dependency graphs, into two dimensional regularly connected processing arrays. Each node in a graph represents an operation which can be processed by an individual processor in the array. The embedding is performed such that data can be processed in a pipeline fashion as they are received. The result is a machine which exploits functional parallelism and data pipelining simultaneously.

The presentation is divided into three parts: the first discusses graphical representation for general image processing algorithms, taking into account the nature of the data flow in real-time systems. The conditions for pipelining the processing of the graph are derived.

Next the logical design of a class of VLSI arrays is considered. These arrays can be configured to embed arbitrary problem graphs. The discussion involves the architecture of the array, the architecture of its processing elements and an efficient programming scheme.

Finally, static embedding of the dependency graphs into the proposed array is considered. Lower and upper bounds on the area needed to embed any graph are found. Three heuristic procedures to embed the graph at minimum cost are developed, implemented and tested.

Acknowledgements

I would like to extend my thanks to Dr. Nadler for his encouragement, excellent suggestions and the long hours of discussion. To my husband, for his support and understanding during my graduate studies, and also for his assistance in producing the final copy of most the figures in this dissertation. Special thanks to Kuwait Institute for Scientific research for supporting me during my graduate studies. And last, but not least thanks to all my committee members for their assistance cooperation and constructive criticism.

Table of Contents

List of Illustrations

List of Tables

Chapter 1: Introduction

1.1 Overview

Image processing applications have inspired many computer architects to design new computing systems beyond the conventional Von Neumann machine. The motivation is the large amount of data and the need for very rapid computation.

A typical image is represented as two-dimensional array of six to eight bit non-negative integer values. These values, called the grey level, are the scene brightness obtained by a sensing device. Each entry in the array is known as a pixel. The size of the image could vary from 256 by 256 for robot images to 4000 by 4000 for LANDSAT images.

Real-time image processing applications demand high processing rate and fast response time. The processing rate is determined by the data rate, which could be in excess of 20 MHZ for some military applications [1]. Consequently, a computational throughput of several thousand million operations per second is required.

Two basic modes of operation are used in real-time applications: pointwise and window operators. In pointwise operations, the output at a pixel is a function of its value. In window operators, the output at each pixel is a function of the grey level values within a fixed size window centered at that pixel.

Examples of real-time image processing applications are target detection and tracking in defence systems, and vision-based robot systems. Figure 1 shows the basic arrangement of a real-time system which will be considered here. The analog image is digitized to the grey level values in a raster-scan format. The sizes of the input buffer depend on the architecture of the image processing peripheral and on the type of operators to be processed. For instance, point by point operators can be processed without any input buffering if the image processing peripheral can accept the pixels at video rate.

The objective of this research is to study a new architecture which promises to meet these requirements by combining the two basic approaches of concurrency: parallelism and pipelining.

The rest of this chapter introduces local and pointwise operators, the basic modular structure of real-time image processing algorithms, then will survey existing image processing machines and point out their limitations

Chapter 2 presents the proposed architecture which is based on modeling the algorithms by their dependency graphs. These graphs are processed by a network of processors in a pipeline fashion. Formal definition of dependency graphs in iterative processing together with pipelining arbitrary graphs are presented. Chapter 3 is devoted to the architecture of a VLSI (very large scale integration)

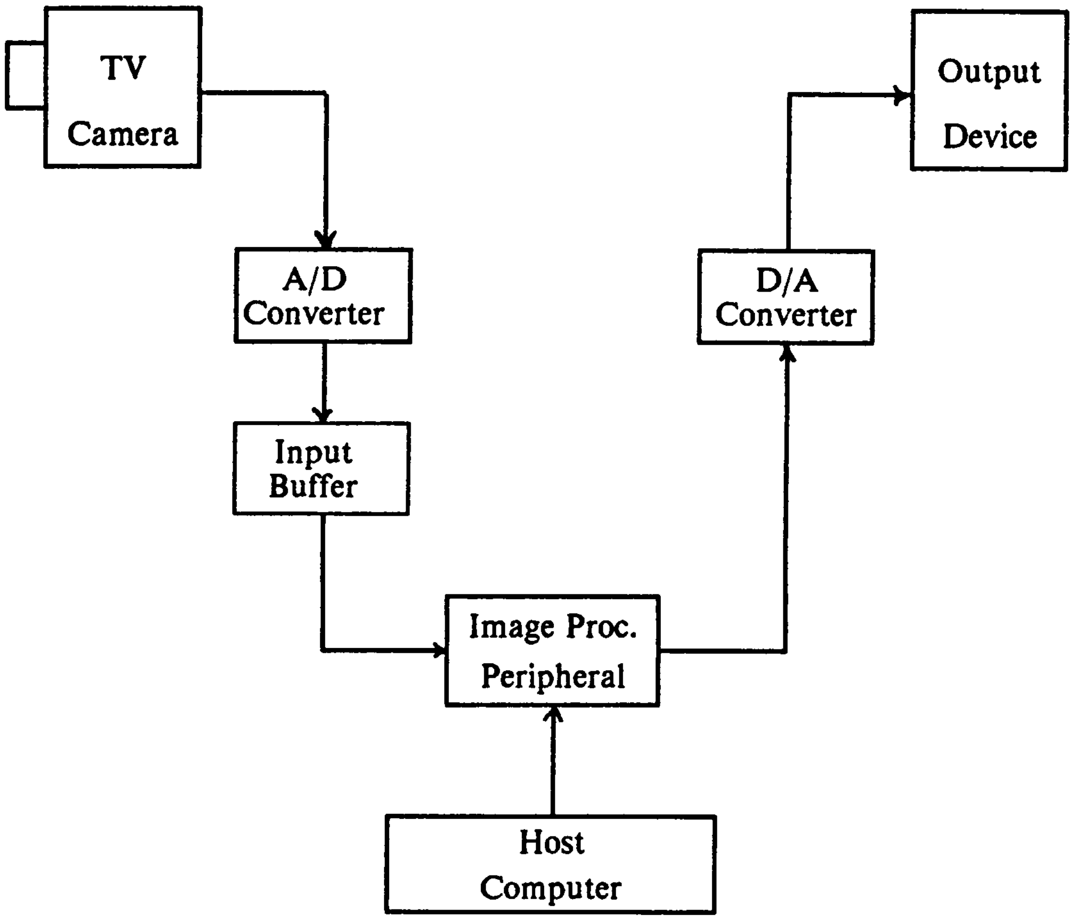


Figure 1. Basic arrangement of a real-time image processing systems.

array which can be configured to process arbitrary problem graph. The design principles of the array, its processing elements and an efficient programming scheme will be proposed.

Chapter 4 integrates the results of Chapter 2 and Chapter 3 by developing and examining systematic methods to map problem graphs into the processing array. This mapping is done according to certain constraints which guarantee proper execution and optimal cost. These mapping techniques have been implemented and tested on pseudo-randomly generated test problems. Finally, Chapter 5 will present the conclusion of this research and possible future work.

1.2 Real-Time Image Processing Operators

The purpose of this section is to illustrate the nature of image processing operators by giving examples of frequently used operators in this area. It is not intended to survey existing operators, nor study their role in image processing; these topics have been studied extensively in [3]-[6].

Pointwise Operators

Pointwise operators process each pixel individually and independently from its location in the image. These operators can be classified according to the size of the output they produce. Some pointwise operators produce one output per

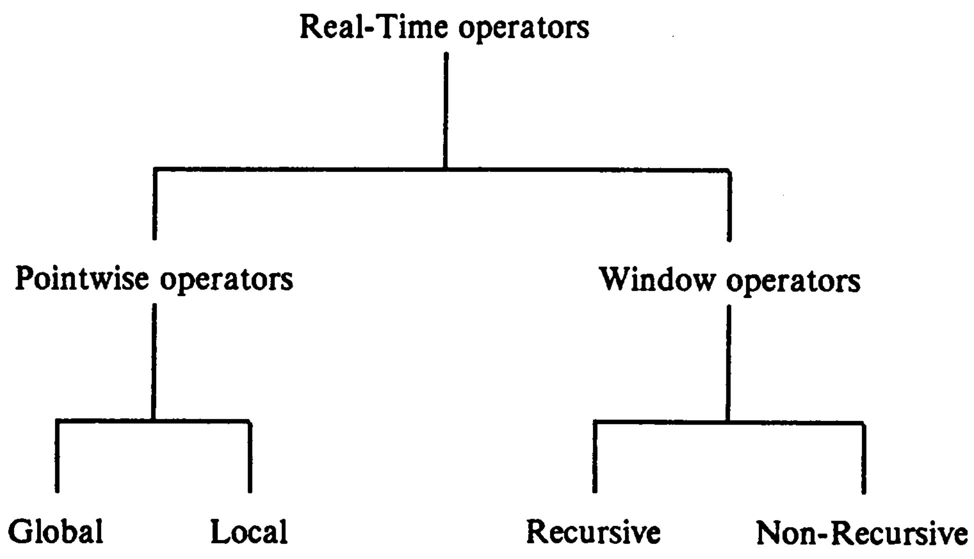


Figure 2. Classification of real-time image processing operators.

pixel by performing logical and/or arithmetic operations or by a table-lookup. Examples of this class are thresholding and histogram modification.

Other pointwise operators produce some global statistical measurements on the whole image. Histogram evaluation and bit-counting are examples of this class.

Example 1.2.1 Histogram Modification: The grey level of the individual pixels is modified according to a given table, the result is an image of given grey distribution.

Example 1.2.2 Thresholding:The grey level at each pixel is compared to a threshold value g_{th} the result $f(x,y)$ will be

$$f(x,y) = \begin{cases} 1 & \text{if } g(x,y) > g_{th} \\ 0 & \text{otherwise} \end{cases}$$

Example 1.2.3 Histogram Evaluation: The histogram H(g) of an image function g(x,y) is defined as

$$H(g) = n_g , \qquad 0 \le g \le g_{\max}$$

where n_g is the number of pixels having grey level g .

Window Operators

In window operators, sometimes refered to as local operators, the output at each location is a function of the pixel values within a neighborhood centered at that location. Typical windows used in image processing are described as $(2n + 1)$ by $(2m + 1)$ rectangular areas.

Window operators are classified into recursive and non-recursive operators. Recursive operators replace the original value of the pixel by the value they compute. In other words, they use the same memory for their input and output. Therefore, the value computed at a certain pixel will influence the output of its neighbors which have not been processed yet. On the other hand, non-recursive operators use different memory for their output from the one used for their input.

Another classification of the window operators would be according to the primitive operation they use. In this case they can be classified into [1]:

1. Arithmetic: e.g. convolution, correlation, local mean.

2. Compare & Sort: e.g. local Min/Max, median filter.

3. Logical: e.g. bit counting, region growing, shrinking.

The following are examples of some frequently used window operators. Throughout these examples $g(x,y)$ will denote the value of the pixel at location (x,y).

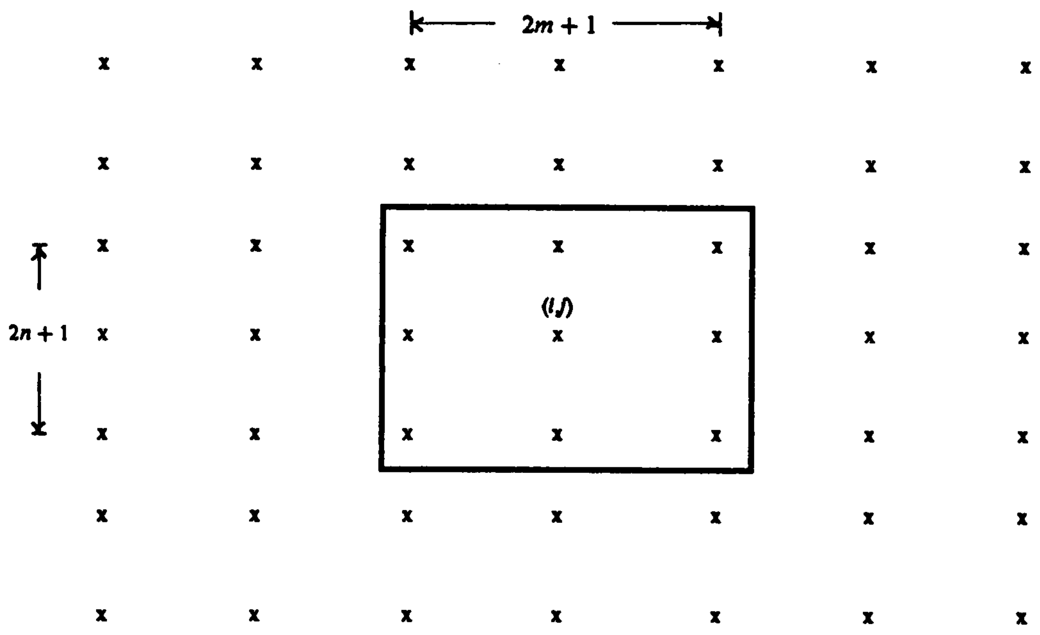


Figure 3. A window centered at (i,j)

Example 1.2.4 Local average:: used for smoothing the image. The output $f(x, y)$ can be defined as:

$$f(x,y) = \frac{1}{(2n+1)(2m+1)} \sum_{i=-n}^{n} \sum_{j=-m}^{m} g(x+i, y+j)$$

Example 1.2.5 Median filter: defined as

$$M(x,y) = \text{median}\,\{\, g(x-n, y-m),\ \dots\ , g(x,y),\ \dots\ g(x+n, y+m) \,\}$$

Example 1.6 Convolution: used in filtering and template matching. If $w(i,j)$ is the $n \, x \, n$ weight matrix, then the convolution is defined as:

$$f(x,y) = \sum_{i=-n}^{n} \sum_{j=-n}^{n} g(x-i, y-j) \cdot w(i,j)$$

An example of image processing filters is the Laplacian filter, a typical weight matrix w for this filter is defined as:

$$w = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

1.3 Classes of Computer Architecture

The literature of computer architecture indicates that two basic approaches have been adopted to achieve high speed computing: parallelism and pipelining.

A purely parallel system consists of identical subsystems. Each subsystem functions on its own set of inputs independently from the other subsystems. The throughput[1] of a pure parallel system is a function of the number of subsystems it contains.

On the other hand, a pure pipelined system is composed of several cascaded stages. Each stage is dedicated to perform a certain subfunction from the overall function performed by the pipeline. The throughput of a purely pipelined system is a function of the delay within each stage and the time interval between successive sets of input data and is independent of the number of stages in the pipeline.

The most widely used classification of computer architecture is the one defined by Flynn [16]. The classification is according to the number of different instructions executed simultaneously, and the multiplicity of the data streams. Hence, four classes can be defined: the Single Instruction Single Data Stream (SISD), Single Instruction Multiple Data Stream (SIMD), Multiple Instruction Single Data Stream (MISD), and Multiple Instructions Multiple Data Stream (MIMD).

[1] defined as the number of output sets produced by the system per unit time.

SISD machines correspond to the conventional computers where each instruction triggers only one set of operands, while SIMD machines correspond to machines where each instruction triggers more than one operand set. The operations on all data sets can be done in parallel as in parallel arrays (e.g. MPP [2,10], CLIP [10,12]), or in a pipeline fashion as in vector processors (e.g. Cray-1[8]).

MIMD machines execute different programs on different data sets simultaneously. Finally, MISD systems execute different instructions on the same data stream.

The problem with this classification is that it does not describe the structure of the machines in each class. For instance, vector processors and parallel arrays both belong to SIMD class, although vector processors achieve their concurrency by purely pipelining the instruction, while parallel arrays achieve their concurrency by employing an array of processors which execute the same instruction in a pure parallel fashion.

1.4 Survey of Image Processing Architectures.

Several image processing architectures have been proposed and implemented in the past few year [10, 12, 13] The focus will be on those that can efficiently process a wide range of local and pointwise image processing operators. Therefore, although there are many dedicated devices which operate efficiently at pixel rate, they will not be considered here since their functional capability is limited.

1.4.1 SIMD Parallel Array:

The typical structure of SIMD parallel arrays is shown in Figure 4 [10]. It is composed of relatively simple processing elements. Each element is connected to its immediate neighbors and has its own memory. A global control unit broadcasts the same instruction to all processing elements.

Ideally, the image is mapped into the array such that there is one to one correspondance between the pixels of the image and the processing elements. In practice, no machine has been built yet which can process the whole image in parallel; instead the image is partitioned into subimages of the size of the array, and one subimage is processed at a time. Another advantage is the direct access of each pixel to its neighbors without requiring any indexing or address generation.

The following are examples SIMD arrays designed for image processing applications:

The Massively Parallel Processor (MPP)

The MPP machine was designed for NASA by Goodyear and came into operation in 1983 [2, 12, 23]. It contains 16 384 processing elements (PE's) arranged in a 128 by 128 square. Each PE is a bit-serial processor with 1024 bits of local memory and is connected to its north, south, east and west neighbor processors. The cycle time for the array is 100 ns.

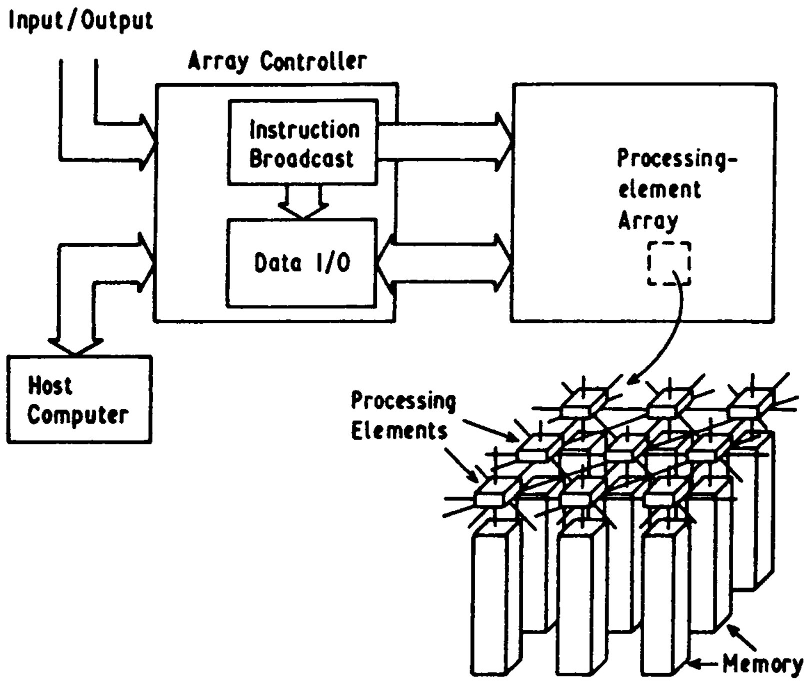


Figure 4. A schematic block diagram of a typical SIMD parallel array [10].

Table 1. Examples of SIMD systems for image processing.

System	Connection	Size	Processor	Memory/PE	Technology
MPP	square	128*128	bit-serial	1 k bits	LSI
DAP	square	64*64	bit-serial	4 k bits	SSI/MSI
CLIP4	8 neighbors	96*96	bit-serial	32 bits	LSI
CLIP7	8 neighbors	4*512	16-bit	?	LSI
GRID	4 direct + NNS	8*8	bit-serial	64-bits	CMOS VLSI

To process an image, the whole image has to be read into an input buffer first, then one 128 by 128 bit-plane would be loaded at a time into the array. This is done through a one bit shift register associated with each PE. Although loading a bit plane can be done simultaneously while another plane is in processing stage, the I/O time dominates the computation time for image processing algorithms that do not require extensive computations. Another disadvantage of MPP is its lack of interface with video data which makes it inadequate for real-time processing.

The Cellular Logic Image Processor (CLIP) machine

The CLIP series is another example of SIMD parallel arrays [12,13]. CLIP4 is one of this series which is in commercial operation. It is a $96x96$ bit-serial processor array designed to process video input from a TV camera. Each PE can communicate with six nearest neighbors or eight nearest neighbors depending on the selected communication mode, and is supported by 32 bits of local memory. The video input is stored in 9216 by 6 bit memory first, then one bit plane is transferred to the array at a time in a similar fashion to MPP.

CLIP7 [12] is an extension of CLIP4 intended to process 512 by 512 images at the same rate as CLIP4. The major difference between the two machines is that CLIP7 uses 16-bit instead of bit-serial processors.

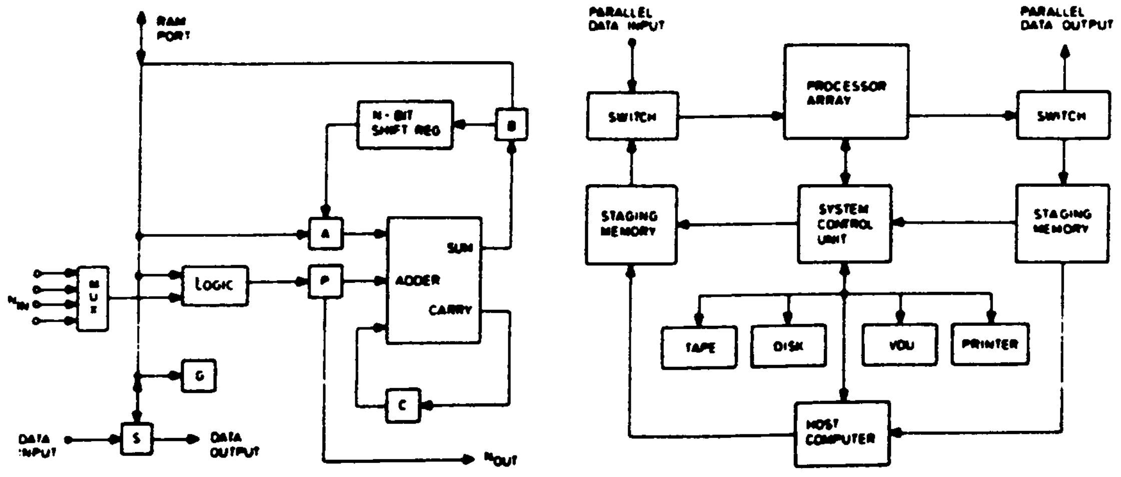


Figure 5. The massively parallel processor (MPP) system [12].

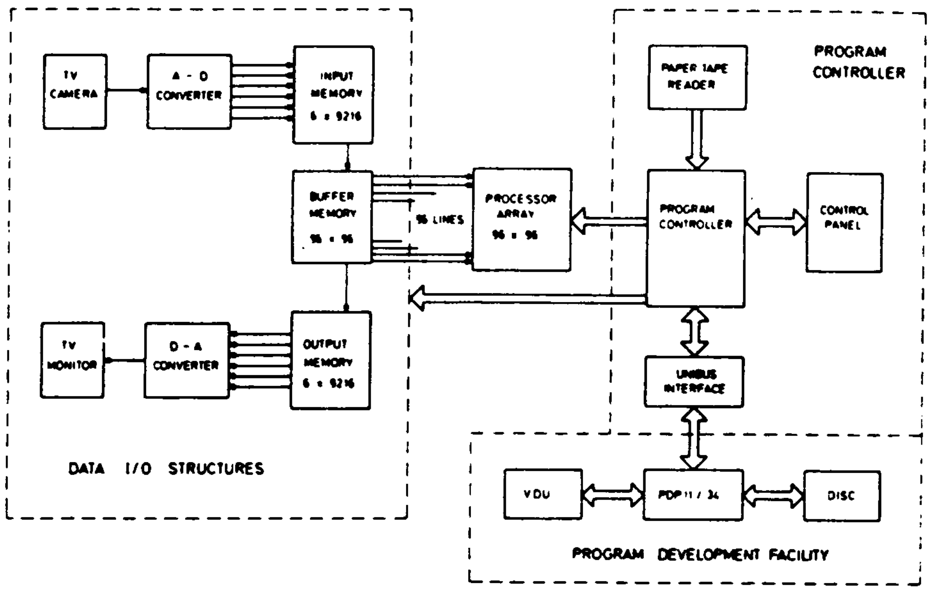


Figure 6. CLIP4 system [12].

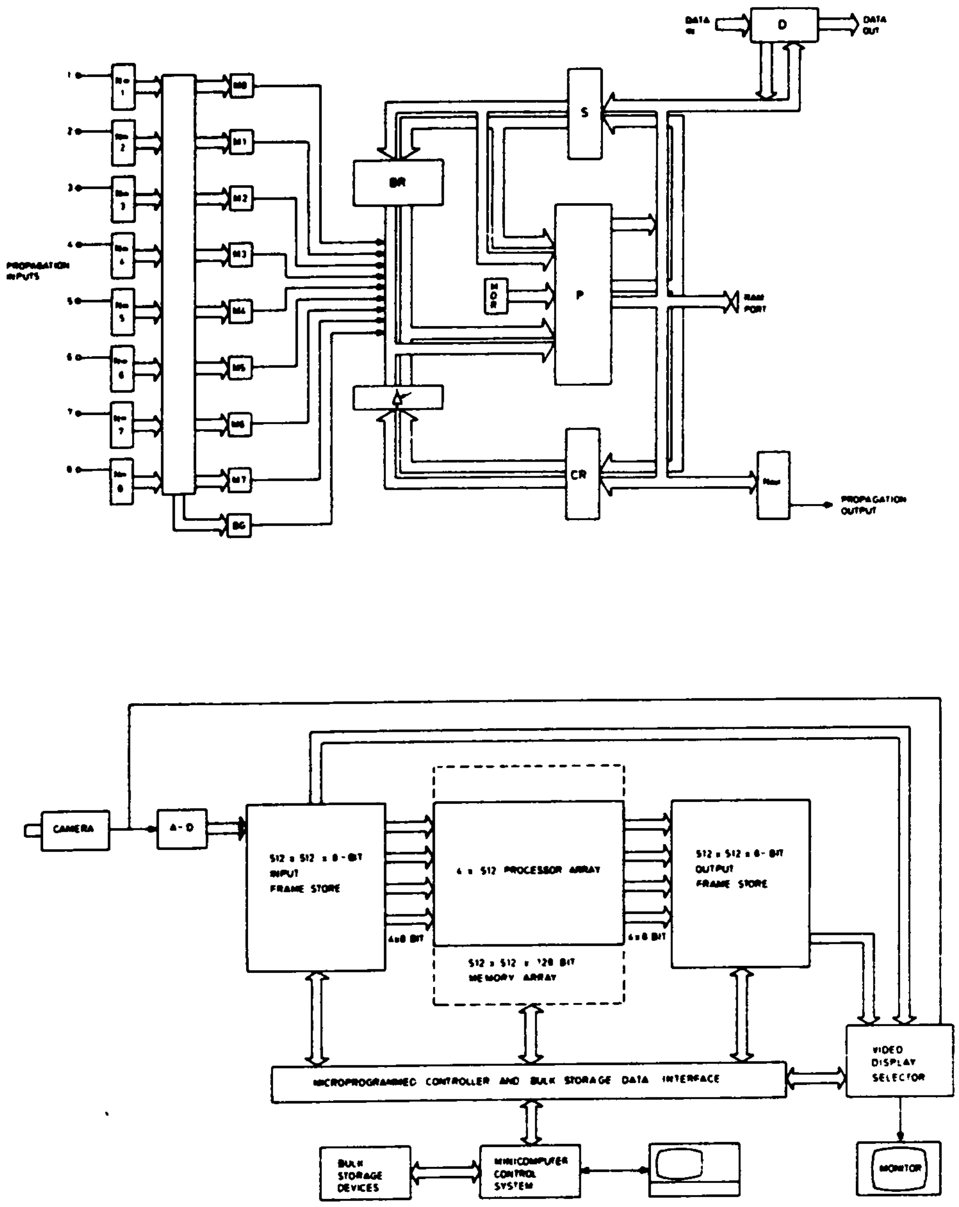


Figure 7. CLIP7 processor and system [12].

The Distributed Array Processor (DAP) system

The DAP system [12] is 64 by 64 bit-serial processor array. Each processor is directly connected to its four neighbors and has 4k-bits of random access memory. The DAP memory is a portion of the host memory. All the input and output of the array are done through the host memory.

The GEC Rectangular Image and Data (GRID) machine

The GRID system has been developed by the GEC (General Electric Company) in England for image signal and numerical processing [10],[12].

Each processor is a bit-serial processor directly connected to its north, east, west and south neighbors, and can access the four nearest neighbors through a local Nearest Neighbor Switch (NNS). The GRID processing element is shown in Figure 8 .

The new feature in the GRID system which does not exist in the previously discussed SIMD arrays, is that its controller includes special hardware which supports mapping larger images into small arrays. The controller provides two mapping techniques: the window and the pyramidal mapping.

In window mapping the data array is partitioned into regions equal to the size of the processor array, and are stacked on top of each other in the array. In pyramidal mapping the data array is also partitioned into regions, but each region is mapped to a single processor in the array.

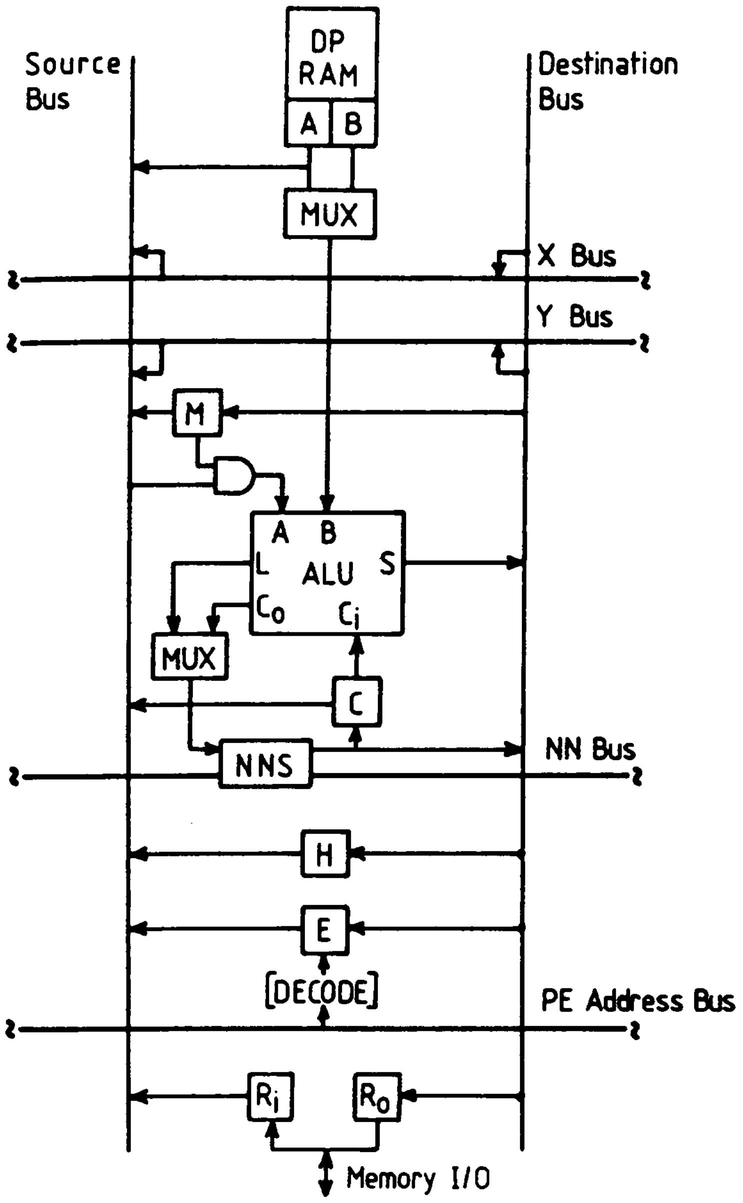


Figure 8. **GRID processing element [12].**

Another advantage of the GRID system is that it employs two-dimensional video line buffers for input and output. These buffers allow the video signals to be input, processed and output in a pipeline fashion, hence overcoming the need of extensive buffering to store the whole image as in other SIMD machines.

Although the GRID system overcame the problem of direct interface with I/O devices, the use of memory within each PE degraded the cycle time to 100 ns.

1.4.2 Hardwired Pipeline Processors

Another candidate architecture for image processing applications is the serial pipeline. It consists of several cascaded (pipelined) programmable processor stages with hardwired data paths between adjacent stages. Each stage will perform a specific operation on the input data and send the result to the next stage. Examples of pipelined image processing systems are the Cytocomputer and the NEC μPD7281 image pipelined processor.

The CytoComputer

The Cyto-Computer [13, 21, 22] was designed by ERIM (Environmental Research Institute of Michigan). It is a serial pipeline machine composed of identical stages (presently 80). Each stage, as shown in Figure 9, is table-driven and is capable of performing either a point-by-point logic function or a 3 by 3 neighborhood logic function. The latter would require two (M-3) buffers at each stage, where M is the image width. The output for each stage is determined by lookup table.

The advantage of this machine over parallel array processor is that it can be directly interfaced with the sensor without extensive buffering; this diminishes the overhead involved with image I/O. Also, the interconnections between processor stages is simple and fault tolerance is easy to incorporate by bypassing the faulty stage.

However it has several disadvantages: the complexity of the logic function at each processor stage is limited by the pixel transfer rate; using a lookup table to evaluate the logic function at each stage limits the processing speed by the memory access time. Another disadvantage, is the limited window size which is a confining constraint for image processing systems.

NEC μPD7281 Image Pipelined Processor

The NEC μPD7281 Image pipelined processor is the first commercial VLSI device to exploit pipelined data flow architecture [10]. It is supported by a powerful instruction set designed specifically for digital image processing applications. The processor employs a novel circular pipeline architecture as shown in Figure 10.

The pipeline is formed by five functional blocks which allow the processor unit to operate at a continuous 5 MHz rate.

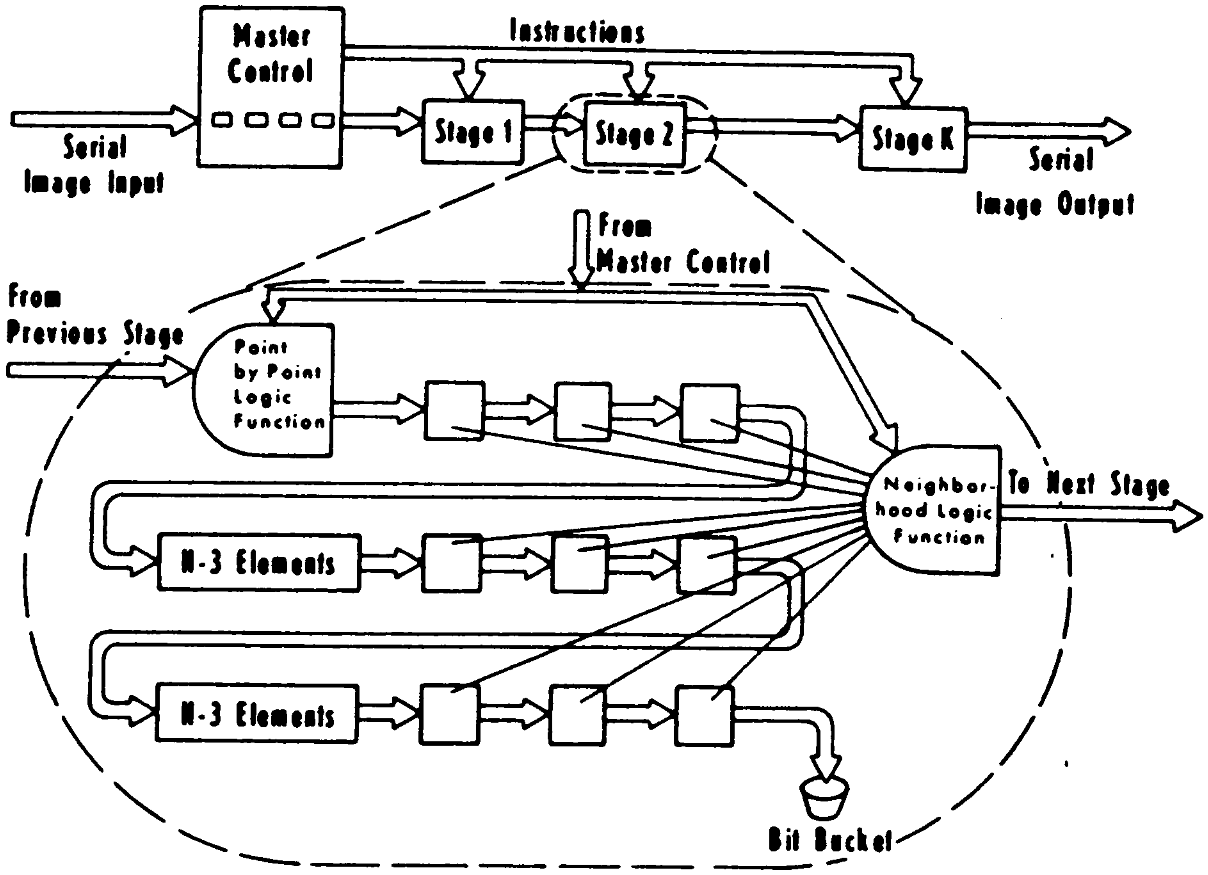


Figure 9. The Cytocomputer [22].

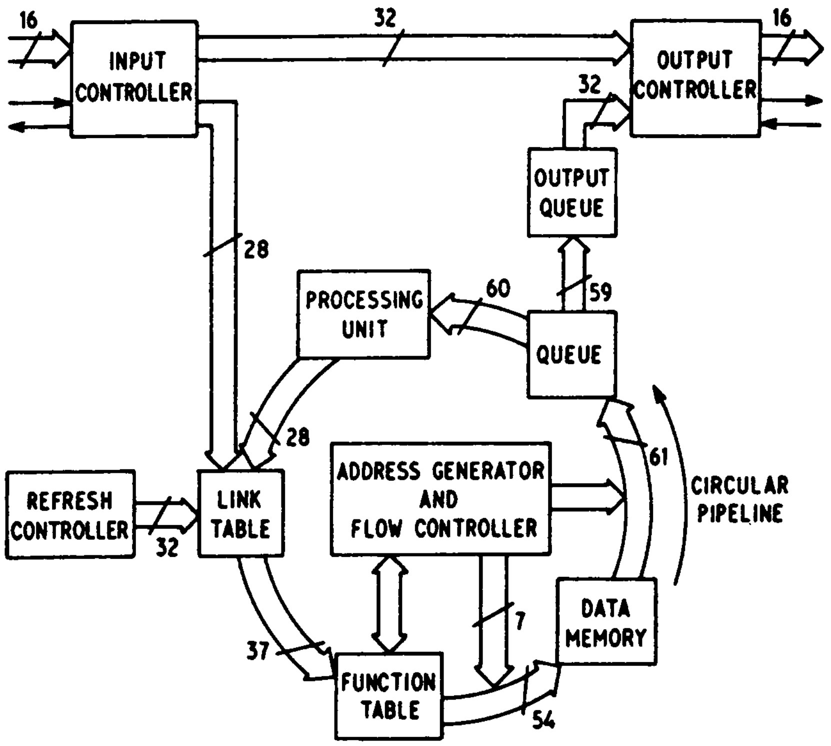


Figure 10. NEC μ PD7281 Image Pipelined Processor [10].

More than one processor can be cascaded to increase the throughput rate. In its current implementation, the μPD7281 is designed as a peripheral to a mini-computer or a microcomputer without interfacing the I/O devices.

1.4.3 Systolic Arrays

A systolic array [9, 14, 20] consists of one or a few types of cells arranged in linear or two-dimensional arrays with regular cell interconnections. Once the data enter the array, they will be used extensively in order to minimize the I/O time. At each clock cycle, each cell will operate on its input data and send the result(s) to the next cell. Therefore, these arrays derive their computational efficiency from using multiprocessing and pipelining simultaneously.

Systolic implementation have been defined for a few image processing algorithms such as 2-D convolution [14, 24] and median filtering [10].

Although these arrays have high performance, are fault tolerant, and can directly interfaced with the sensor without image I/O overhead, they lack the universality we are seeking in image processing machines.

1.5 Conclusions

Several SIMD and pipelined systems designed for image processing applications have been discussed. From their performance for real-time applications, the following can be concluded:

1. Pipelining the processing and the I/O is needed to overcome the use of extensive buffers and reduce the response time.

2. The use of memory devices in the processing stage degrades and limits the processing speed in SIMD arrays and serial pipelined machines.

3. Parallelism is essential to achieve the computational throughput needed for real-time applications.

Chapter 2: Parallel Pipelined Processing

2.1 Introduction

For most pointwise and window operators the number of program execution (equivalent to the number of pixels in the image) exceeds the program length (proportional to the number of instructions). Based on this, one can say that spatial program mapping would require less area than pixel mapping.

Previous attempts have been made to assign one pixel per SISD processing element, hence utilizing the maximum possible data parallelism without exploiting any functional parallelism. This approach generated the whole class of SIMD parallel arrays, which have been discussed in Chapter 1.

In this research, another type of mapping is considered, where both functional parallelism and data pipelining are utilized simultaneously. It is based on representing the algorithm by its dependency graph and then embedding the graph in a processing array. The embedding procedure involves mapping each node in the graph to a processing element and each arc to a directed path.

By inserting proper delay nodes the data can be "pumped" into the network continuously in order to keep the processors as busy as possible and to reduce the waiting time for the new coming data. By this approach, functional parallelism is fully exploited since parallel nodes are processed simultaneously. Also by pipelining the data, different sets of parallel nodes will be operating on different problem instances (in image processing, each problem instances will process a pixel or a window of pixels).

In this chapter, the focus will be on the dependency graph as the computational model. The goal is to test the pipelineability of the graph knowing its connectivity and the nodal type. Since ultimately these graphs are embedded in a processor array we would like to know the condition which should be imposed on the embedding so that the embedded graph is pipelined. For graph related terms, the reader is referred to Appendix A.

2.2 Dependency Graphs.

The dependency graph (DG) is a directed graph with nodes representing operations and edges representing their data dependency.

Processing the dependency graphs by a network of PEs brings up the following issues:

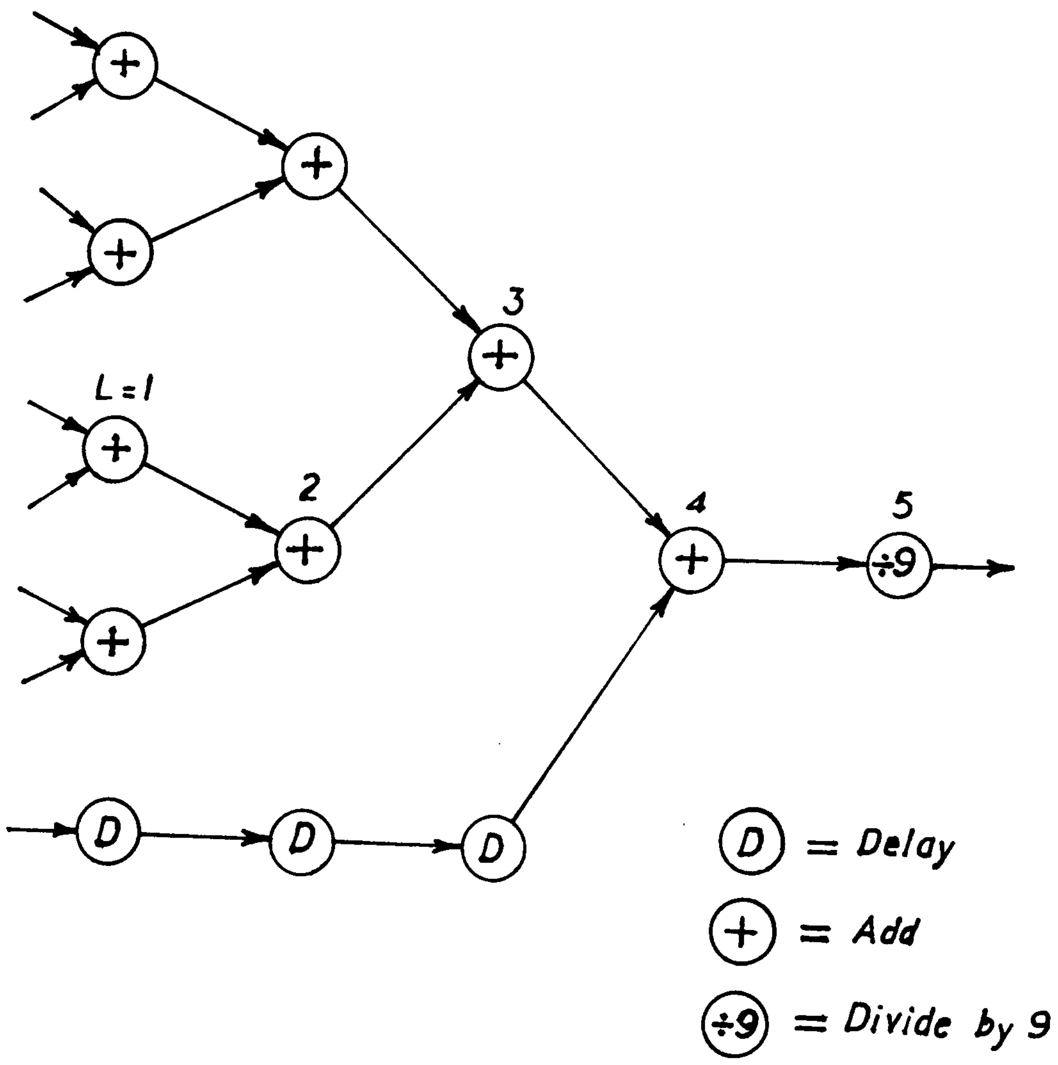


Figure 11. Pipelined network for 3x3 average operator.

1. The indegree of nodes in DG are limited by the number of operands the target processor can accept. Throughout this research only unary and binary operators are assumed.

2. The outdegree of the nodes is also limited by the processor fanout. This limitation is overcome by introducing *splitting* nodes, where each node generates two or a few copies of the same value. Arbitrary fanout can be achieved by using a tree of these splitting nodes.

3. The operation of each node in the graph can be processed by the target processor individually.

4. The processing time for each node should be less than the data rate. Therefore for high speed computations, the node assignment will be at instruction level.

The dependency graph is synchronized by inserting delay nodes. The location and the number of the delay nodes can be determined by finding the level of each node v in the graph.

Definition 2.1: The level $L(v)$of node v is defined as:

$$L(v) = \begin{cases} 1 & \text{if } v \text{ is a source node} \\ 1 + Max\,\{\, L(pred_1(v)),\ L(pred_2(v))\,\} & \text{otherwise} \end{cases}$$

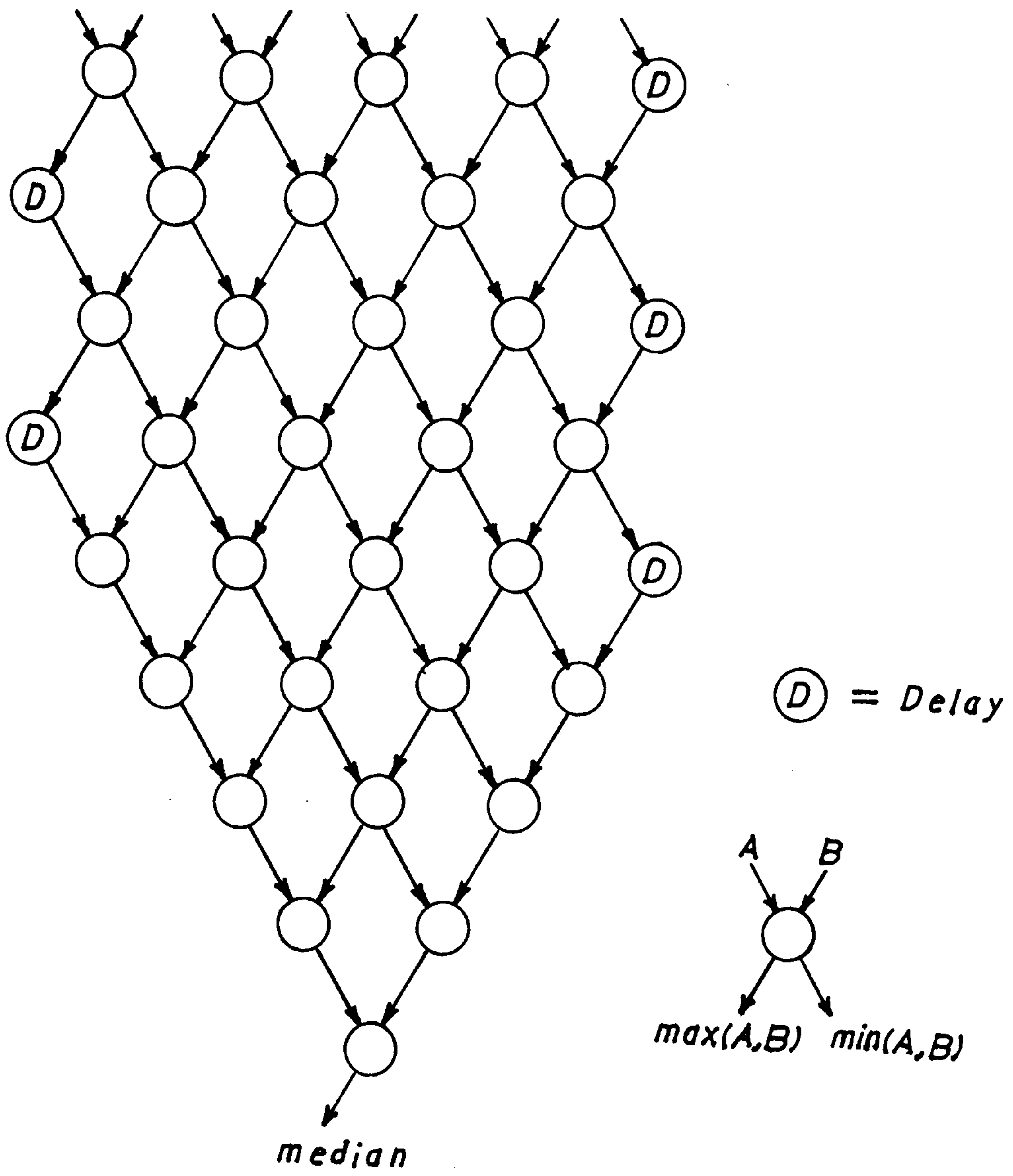


Figure 12. Pipelined network for median filter operator.

Delay nodes should be inserted on each arc connecting two nodes having discontinuity in their levels, i.e. the level of the successor is not one more than the level of the predecessor. The latency [2] in the delay node is the difference in their levels subtracted by one.

Figure 11 shows the pipelined network for the 3*3 average operator assuming that all the nodes have the same computation time. Figure 12 shows the dependency graph for the median filter.

2.3 Iterative Dependency Graphs (IDG)

Iterative computations are defined as those which use values computed at previous iterations. Graphs representing such types of algorithms will be named Iterative Dependency Graphs. Let T_i represent the iteration period.

Definition 2.2: The *Iterative Dependency Graph* $IDG = < V,E, \tau(v) >$ is a directed graph where:

1. $V = C \cup P$, where C is a set of computational nodes and P is a set of separator nodes.

2. E is a set of directed edges.

[2] defined by the number of clock cycles

3. $\tau(v)$ is a function on V. Where $\tau(c)$ denotes the nodal computation time. $\tau(p) = T_i$ for all separation nodes.

The separator nodes are storage nodes which hold information from one iteration to another; they perform the function of the Z^{-1} operator. Although the separator nodes are functionally equivalent to the delay nodes, their presence is essential, as opposed to the delay nodes which can be removed from the graph without affecting the logical definition of the graph. Moreover, the latency within the separator node is always T_i while the delay latency depends on the structure of the graph.

Example 2.1 Separable median filter [4]: Although the separable median filter gives different results than the original filter, it can be used in real-time systems to reduce the computational intensity. a 3x3 separable median filter can be defined as:

$$Y_i = Median\,\{x_1, x_2, x_3\}$$
$$Med = Median\,\{Y_i, Y_{i-1}, Y_{i-2}\}$$

The pipelined network for the separable median filter is shown on Figure 13, assuming the latency in the delay nodes and the splitting node are equal to that of the sorting node.

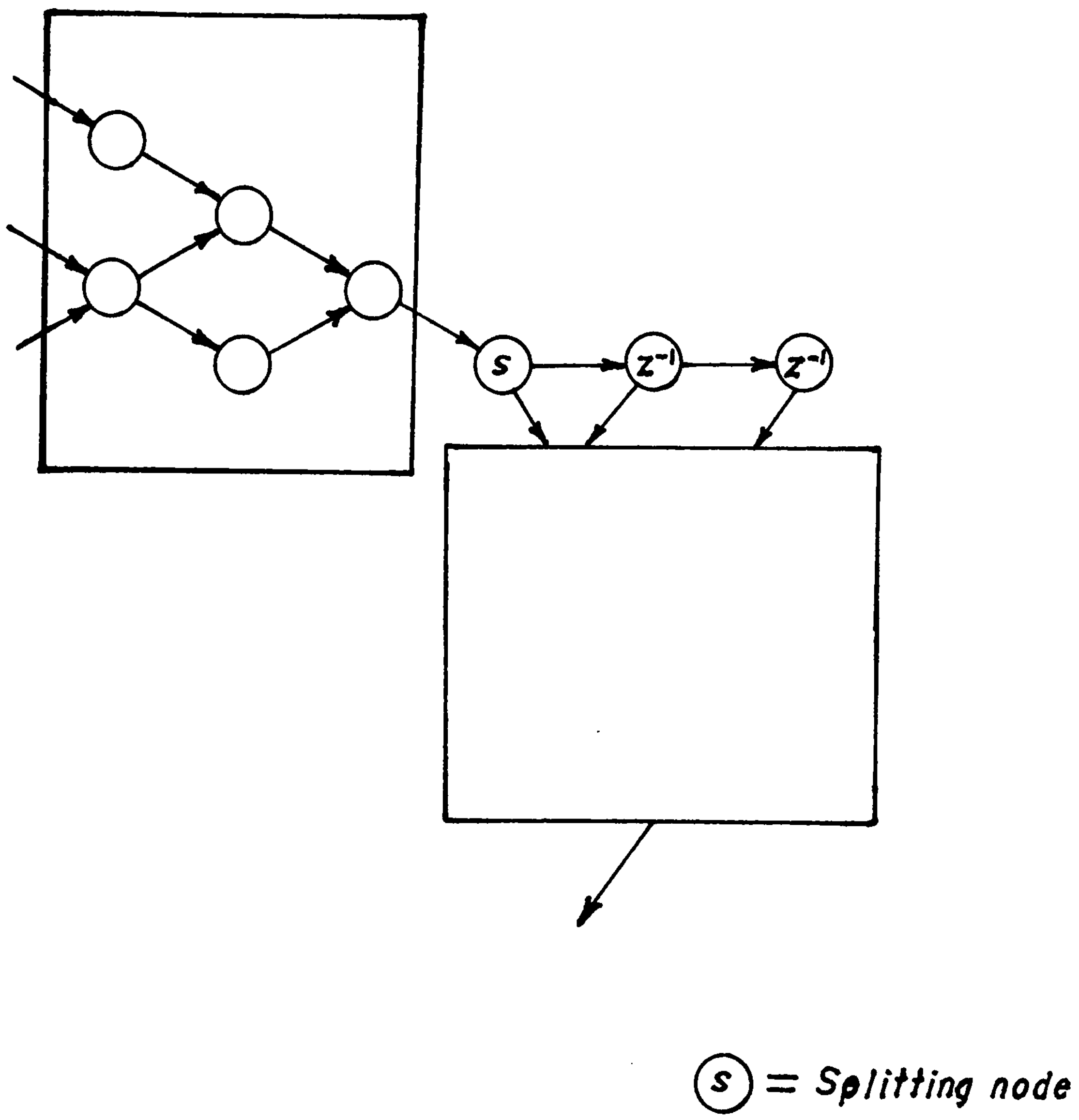


Figure 13. Pipelined network for separable median filter.

2.3.1 Pipelining Acyclic IDGs

To generalize the synchronization procedure used in the two previous examples, Definition 2.1 has to be modified to the following:

Definition 2.2[3]: The level $L(v)$ of node v is defined as:

$$L(v) = \begin{cases} 1 & \text{if } v \text{ is a source node} \\ L(pred(v)) & \text{if } v \text{ is a separator node} \\ 1 + Max\,\{L(pred_1(v)),\ L(pred_2(v))\} & \text{otherwise} \end{cases}$$

The synchronization is completed by inserting delay nodes between every pair of nodes which have discontinuity in their levels.

To test the pipelineability of the synchronized graph ,the following theorem, which was introduced in [53], will be presented. It provides a simple test for pipelineability of n-dimensional iterative arrays (An iterative array is a network of identical processing elements, where each element is directly connected to its neighbors).

Theorem 2.1: [53] An n-dimensional iterative array is pipelined if we can define a set of wavefront cuts on it such that:

[3] not to be confused with the level definition in graph theory

1. All signals that cross the cuts in one sense are buffered, i.e., they have separator nodes whenever they cross the cuts.

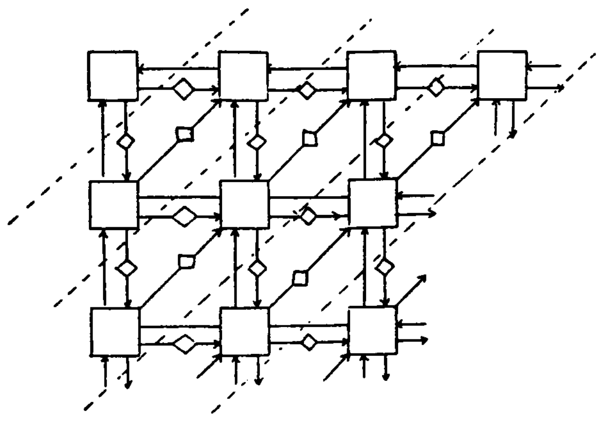
2. All signal lines between processors that do not cross a wavefront cut are buffered.

A *wavefront cut* is a set of parallel cuts through the graph of the array. Each cut is a line (not necessary a straight line) which divides the graph into two parts. Each cut line is uniquely defined by its cut-set. A *cut-set* is a set of edges the cut line intersects. Two cuts are parallel if the intersection of their cut sets is the empty set.

Theorem 2.2: Every acyclic synchronous IDG Network is pipelineable.

Proof: let $CS_i = \{(a,b) \in E \mid L(a) = i \text{ and } L(b) = i + 1\}$ be a cut-set which defines the cut C_i. Since each node has a unique value L, then every edge can belong to one cut-set only. Hence, $\bigcap_{i=1}^{k} C_i = \Phi$, where k is is the total number of cuts. Since the graph is loop-free then all the edges will intersect the lines in one sense (left to right or right to left) and we can assume that the edges which will intersect the cut in the opposite direction are the ones which will be buffered. Therefore the first condition is satisfied.

From Definition 2.2, we know that the only edges which will not belong to any cut-set are those which connect either two separator nodes or a separator node and its source since in both cases L(successor) = L(predecessor). Since these nodes are already buffered, then the second condition is satisfied. •


(a)

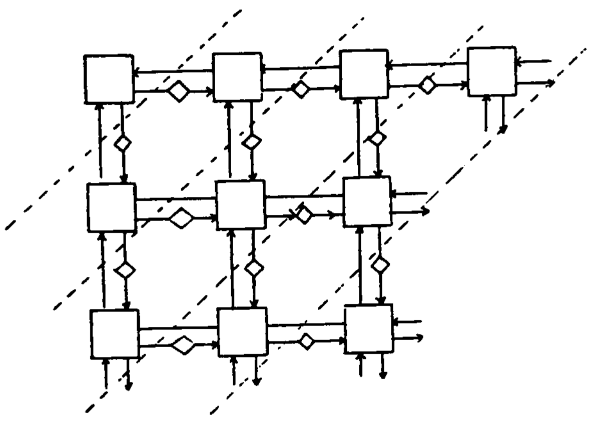

(b)

Figure 14. Examples of 2-D arrays with wavefront cuts drawn [53].

Example 2.2 Local Average: Consider the $3x3$ average operator discussed previously, and suppose that a column of three values is received every T_i then the operator can be defined as:

$$Y = x_1 + x_2 + x_3$$
$$AVG = (Y + Z^{-1}[Y] + Z^{-2}[Y])/9$$

The cut lines as defined by the theorem are shown on Figure 15

From the above definition of the cut lines, the nodes between consecutive lines can be viewed as a single stage in a one dimensional pipeline. All the stages will have the same input to output latency of T_i, and in steady state all the PEs within the network will be active if their computation time is equal to the iteration period. Moreover, if at all edges cut by the same cut line, the same number of delay nodes is inserted then the network can still be pipelined.

Since the arcs connecting separation nodes are never cut by any cut line, this implies that when embedding such a graph in processor array these nodes should be mapped to directly connected processors.

2.3.2 Pipelining Cyclic IDGs

Cyclic IDG represent recursive operators. Formally, they can be defined as:

$$y = f(a, g(Z^{-k}[y]))$$

where

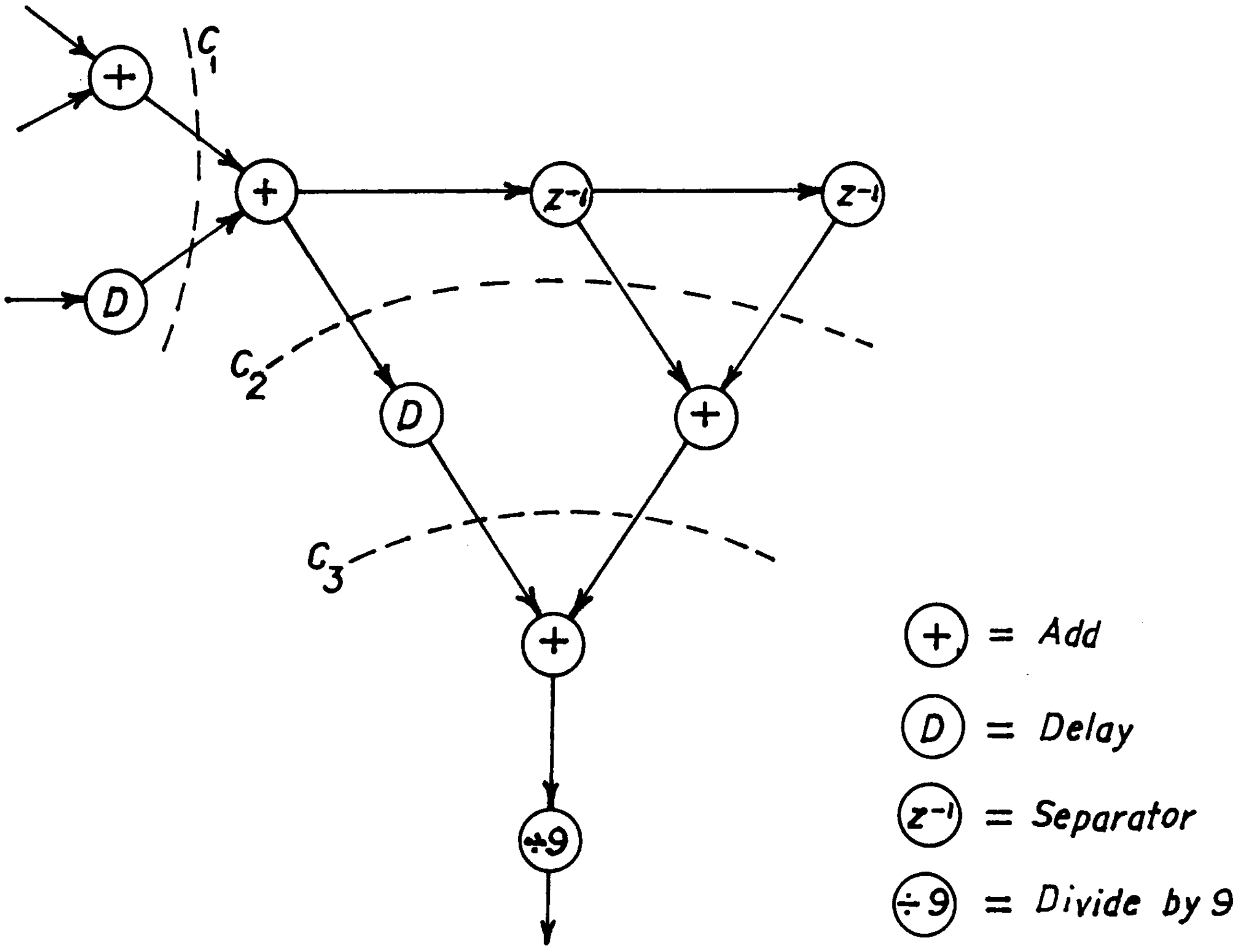


Figure 15. Pipelined network for 3x3 average operator using separators.

f : the recursive operator function.

y : the value computed by the recursive operator.

a : is a fresh input.

Z : the delay operator. Therefore, $Z^{-k}[y]$ is the k^{th} previous y value.

g : is a primitive or complex function which the output of the operator has to go through before reentering the operator node.

In general, g can be presented as:

$$g(y) = g_n(g_{n-1}(\ldots\ldots(g_1(y)\ldots\ldots)))$$

each g_i can be a single primitive node or a set of parallel nodes.

Theorem 2.2: A recursive operator $y = f(a, g(Z^{-k}[y]))$ where $g(y) = g_n(g_{n-1}(\ldots g_1(y)..))$ and each g_i is a sequential operator, can be processed with the highest throughput of $\frac{1}{T_i}$ if $n \leq (k-1)$.

Proof: There are n primitive operators g_i which y has to go through before reentering the recursive operator. Insertion $(T_ik - 1 - n)$ separator nodes within the loop, will make the total length of the loop T_i k. Therefore, at every T_i clock cycles a new $g(y)$, where y has been computed in the previous k^{th} iteration, will enter the recursive operator. •

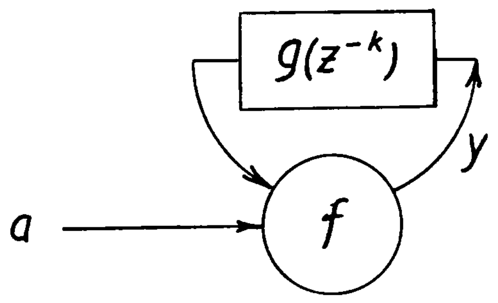


Figure 16. Recursive IDG

2.4 Conclusions

The rules for embedding any DG such that the processing is pipelined are as follows:

1. Cycles have to be embedded in fixed path loops.

2. The delay between a node and its separator nodes should be equal to the iteration period. Therefore these nodes should be mapped to directly connected processors.

3. The predecessors of each binary node have the same level. Therefore if the graph does not include any separator nodes, then the two incident paths to a binary operator will have the same length.

Chapter 3: Architecture of Parallel Pipelined VLSI Arrays.

3.1 Introduction

In order to be able to process arbitrary problem graphs on the same array, the array should support both functional programmability and topological configurability. Functional programmability can easily be obtained by using programmable processors while topological configurability demands the use of a switching mechanism.

In the last few years several configurable architectures have been proposed. The basic difference is in the switching mechanism, which can be classified into independent, local and global switching.

Independent switches have separate hardware modules and are spread between the PEs to form a processor-switch lattice. The CHIP architecture [33] proposed by Snyder is an example of this type of switching. The basic drawback of this switching mechanism is that the number of switches to be inserted between every pair of PEs depends on the implemented algorithm.

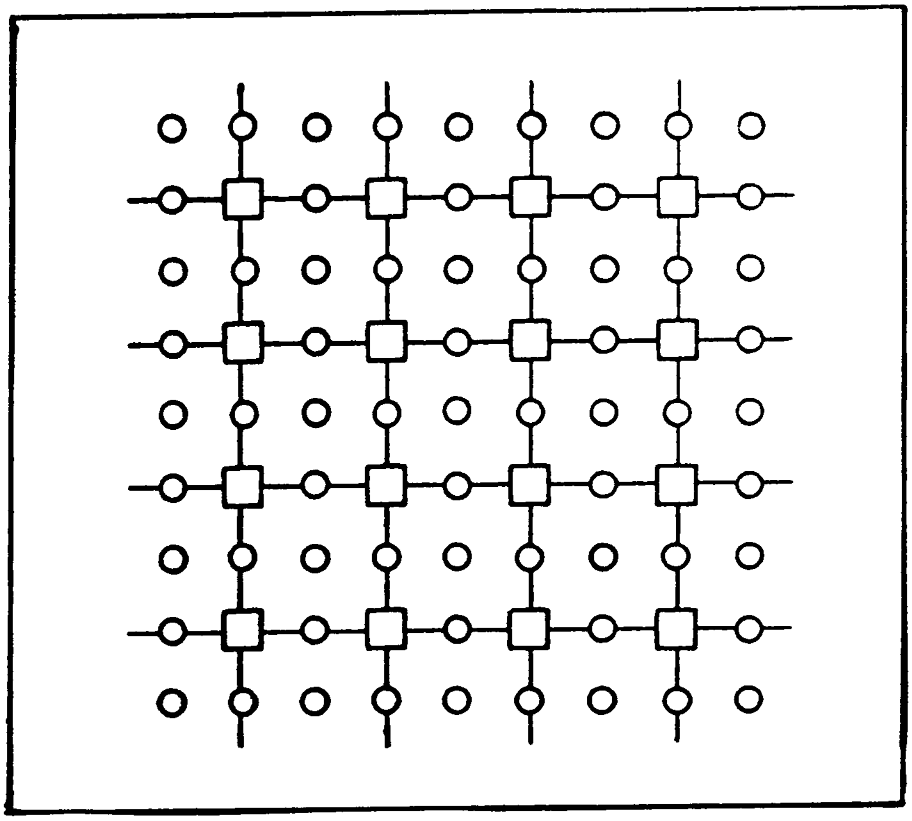


Figure 17. The CHIP architecture.

Local switches are built around the PEs such that the PE and its switch are combined in a single module. The cell can act as a switch or a processor as needed. Koren [25] proposed an array which falls in to this class. The advantage of this type of switching is its modularity and universality as compared to the previous class.

While independent and local switches are simple and can perform connections between their neighbors only, global switches can connect any two processors in the array. This type of switching usually requires the use of a complex switching network. An example of a global switching network is the crossbar switch used by Hwang and Xu [33] in their implementation of multi-pipeline network.

In this chapter the hardware modules of a universal machine capable of processing an arbitrary problem graph will be presented. The discussion will be limited to the logical level and there will still be some decision to be made at design time.

Section 3.2 will present the general structure of a parallel pipelined system. The array and the processing element (PE) architectures will be detailed in sections 3.3 and 3.4 respectively. A scheme to program the proposed array will be presented in Section 3.5. Finally, comparison with other architectures will be made in Section 3.6.

3.2 System Architecture

Figure 18 shows the proposed system. The 2-D array is capable of processing arbitrary problem graphs. It receives, processes and outputs pixels in a complete pipeline fashion. In pointwise processing mode the pixels can directly enter the array, while in window processing mode a few image lines, as many as the window width, have to be buffered.

The host programs the array by specifying the assignment of each node in the array. During programming stage the host communicates with the cells on the boundary only.

Mapping Algorithms to the Array

The first mapping stage involves translating each algorithm to a Fine-grained Dependency Graph (FDG). The node definition and the indegree and the outdegree for each node should be consistent with the target PE. At this stage the nodes of the dependency graph are either functional or splitting nodes.

The next stage would be to embed the graph in the processor array. During the embedding procedure necessary delay nodes are inserted for pipelining the processing of the graph.

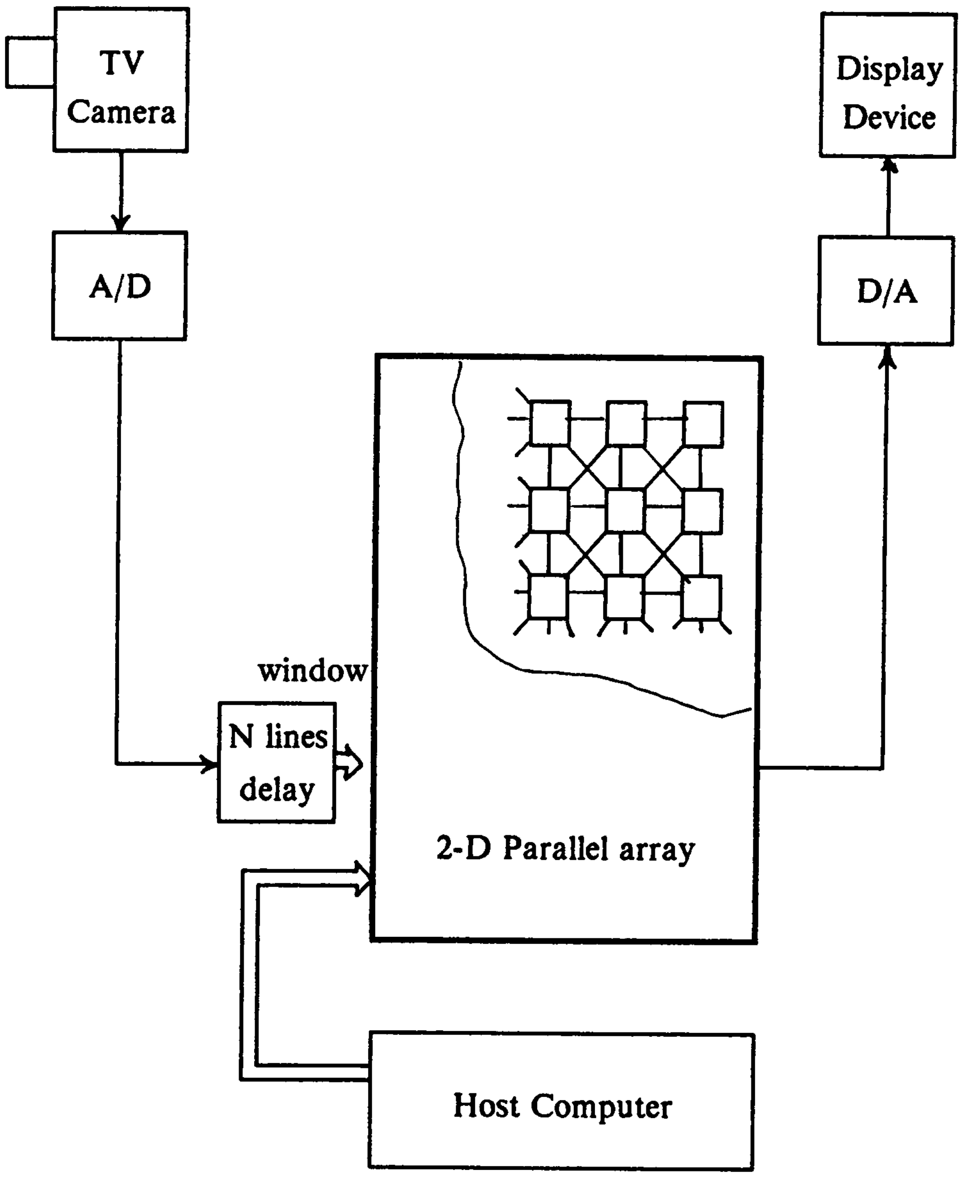


Figure 18. The proposed parallel pipelined image processing system.

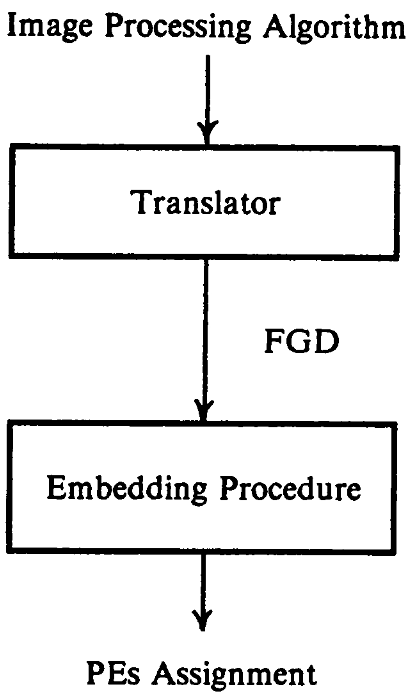


Figure 19. The stages of mapping an algorithm to the processing array.

3.3 The Architecture of the Parallel Pipelined Array

The array is a 2-D arrangement of N-bit processors (N = 8 seems to be a reasonable choice for image processing). Each processor is directly connected to its immediate neighbors through an N-bit connection line. Although the figure shows eight-nearest-neighbor-connected array, hexagonal connection pattern can also be used. The array is controlled by a global clock which is an integer fraction of the pixel rate.

Each PE is capable of performing two basic types of operations: functional and connective. Functional operations, as shown on Figure 20, include simple arithmetic, logical and shift operations. Arithmetic operations have two outcoming busses, one transfers the result while the other transfers the flag bits: carry, zero,..etc. The purpose is to allow processing on larger word size than connection width. Shift operators also produce their results on two ports, each port holds one half of the result.

Connective operations include data splitting and path intersection. Splitting operation is necessary since the fanout of each PE is limited to the number of available buses. Crossing over of paths is also necessary for general graph embedding. After each operation the PEs store the result(s) in the boundary cells to be accessible to the neighbors on the next instruction cycle.

The following are some issues which affect the array architecture:

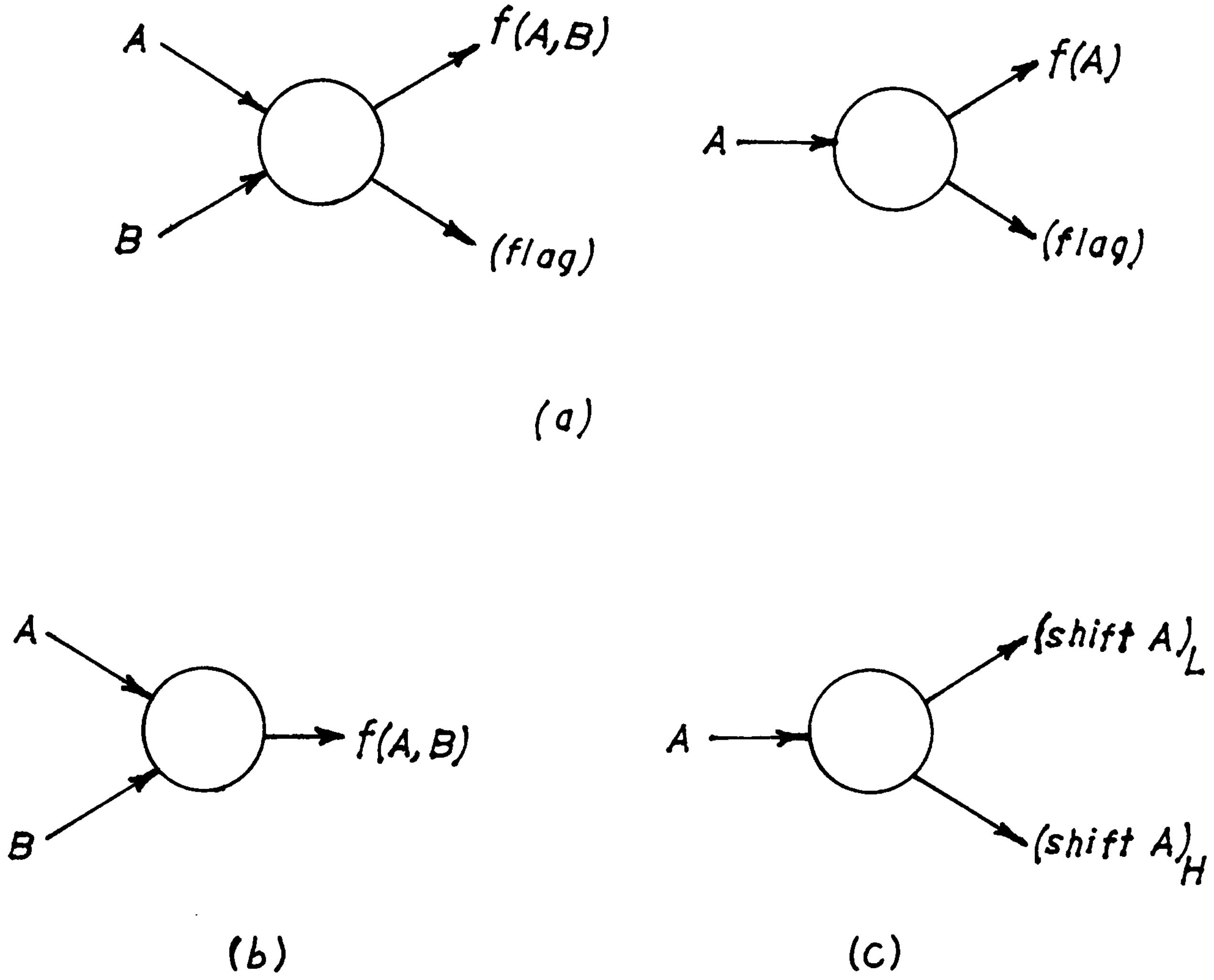


Figure 20. Functional PE operations.: (a)arithmetic (b)logical. (c) logical and arithmetic shift

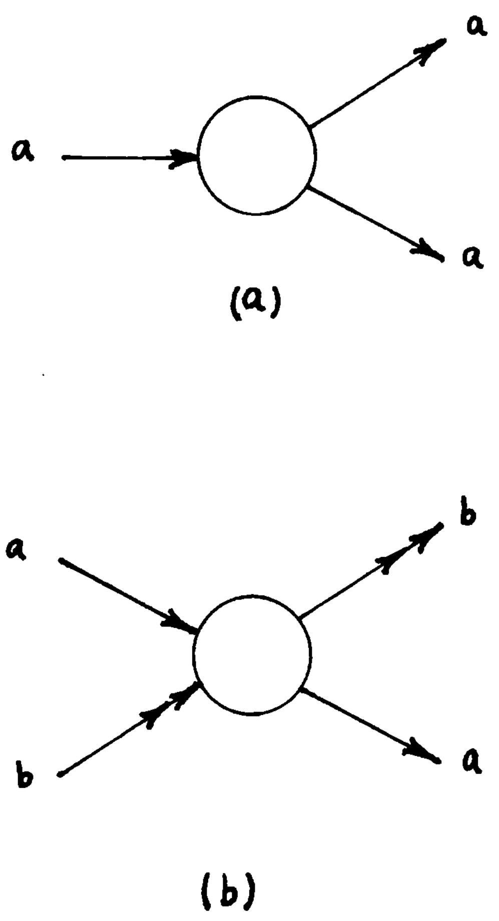


Figure 21. Connective PE operations.: (a)spliting (b)cross-overing paths.

1. Each PE has a fixed program throughout the life of a given algorithm. It repeatedly executes the same program on different data sets; this implies a need for local program memory.

2. Each PE executes a different program; this implies a distributed control rather than a global control unit.

3. Since the processing is performed in pipeline fashion, then all instructions are register to register or immediate instruction and there is no need for a data memory except for lookup tables.

4. To utilize the hardware efficiently, only simple instructions which can be executed in a single clock cycle should be implemented by each PE. Such processors are known as RISC (Reduced Instruction Set Computer) machines [52] . Complex instructions such as, multiplication and division, can always be represented by their dependency graph which will be embedded in the array.

3.4 The Architecture of the PEs

The proposed PE structure is shown in Figure 22. It comprises four basic modules: a programmable functional unit (FU), a Program Memory (PM), a Local Control (LC) and a Master Control unit (MC). The four modules are surrounded by a bus system. Each PE has 8 boundary registers (6 in hexagonal array) where it stores the results after each operation.

The Functional Unit (FU)

Includes an ALU (Arithmetic and Logical Unit) and a Shifter to perform simple arithmetic, logical and shift instructions. It receives its operand(s) from the adjacent neighbors and stores the result(s) in selected boundary registers.

Local Control Unit (LC)

Hardwired control unit which decodes the instruction received from the memory and sends control signals to both the functional units and the bus system.

Program Memory (PM)

Is loaded prior the start of the algorithm execution and remaines unchanged throughout the life of the algorithm. It will be repeatedly executed on different data sets.

Master Control unit (MC)

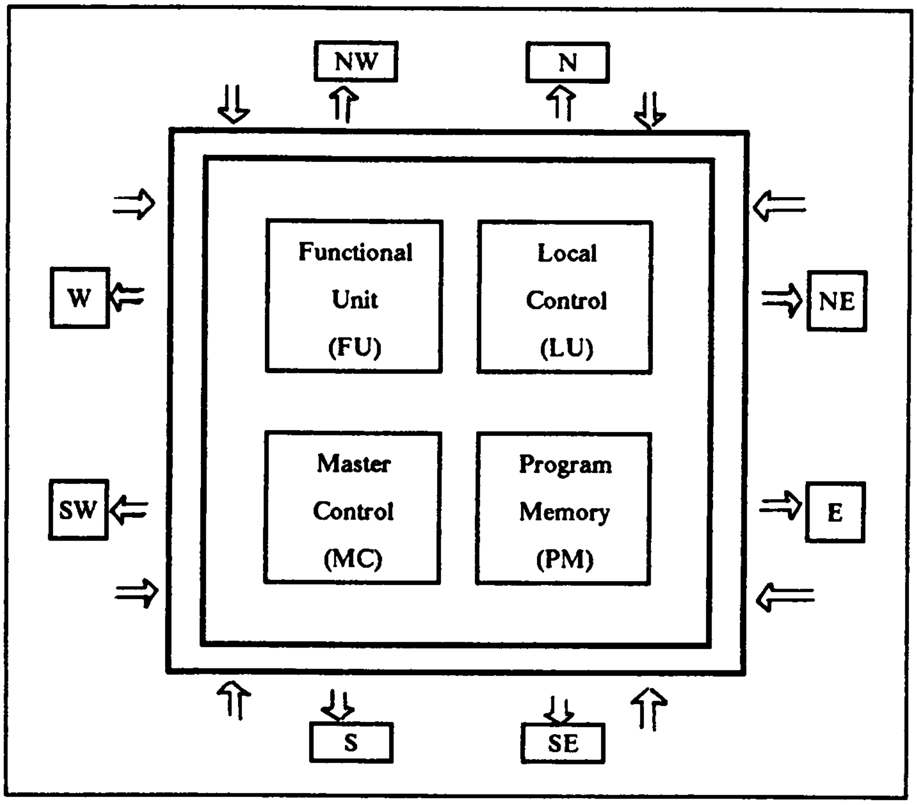


Figure 22. The proposed PE block diagram.

Loads the program memory, transfer programs to the adjacent cell (to the East) or gives control to the local controller to execute instructions stored in the program memory

The four modules are surrounded by a bus system which comprises two or more buses. These buses performs the following operations:

1. Provide the input(s) to the FU and transfers the output(s) to the neighbors.

2. Connect two PEs (neighbors to the one it resides in).

3. Split a data item by generating two data with the same value.

4. Bypass a faulty PE.

The number of simultaneous operations which can be carried by the PE depends on the number of available buses. In its simplest form, the connection unit is composed of two busses. Therefore it can either provide inputs to the FU, connect two different paths (crossing over or not), or split a data item into two. The major drawback of this form is that the FU will be idle whenever the PE is connecting two paths or splitting data.

On the other extreme, a PE which is directly connected to K neighbors could use up to $K/2$ buses in its CN unit. In this case the FU can be kept busy while the CN is connecting paths or splitting data. Obviously, such PE will occupy larger area and longer instruction length.

3.5 Programming the Array

In this section a new scheme for programming the array is presented. The objective is to avoid extra communication links in addition to those already existing between the PEs, as much as possible.

The host computer programs the array by sending two parallel streams: host instructions and processor instructions (node assignment). The host instructions are processed by the Master Control in the PEs, while the processor instructions are stored in the corresponding cell. The processors instruction stream flows through the East registers from one cell to the next, so it does not require any extra wiring. The host instruction stream flows through a dedicated path and registers used for programming only; according to the proposed host instruction set only 2-bit connection is needed . The cell interconnections during the programming stage are shown on Figure 23.

The host loads all the rows in parallel. Each row is loaded sequentially starting from the closest cell to the host, therefore no cell address generation is needed.

The two extra pieces of hardware needed to program the array in this fashion are a one-bit flag register (F) and a host instruction register (HI). The flag register is set whenever the PE finishes loading its program and is used by the MC to determine whether the incoming program instructions are to be stored inside the cell or to be passed to the next one. The HI register holds the host instruction. There are four basic host instructions:

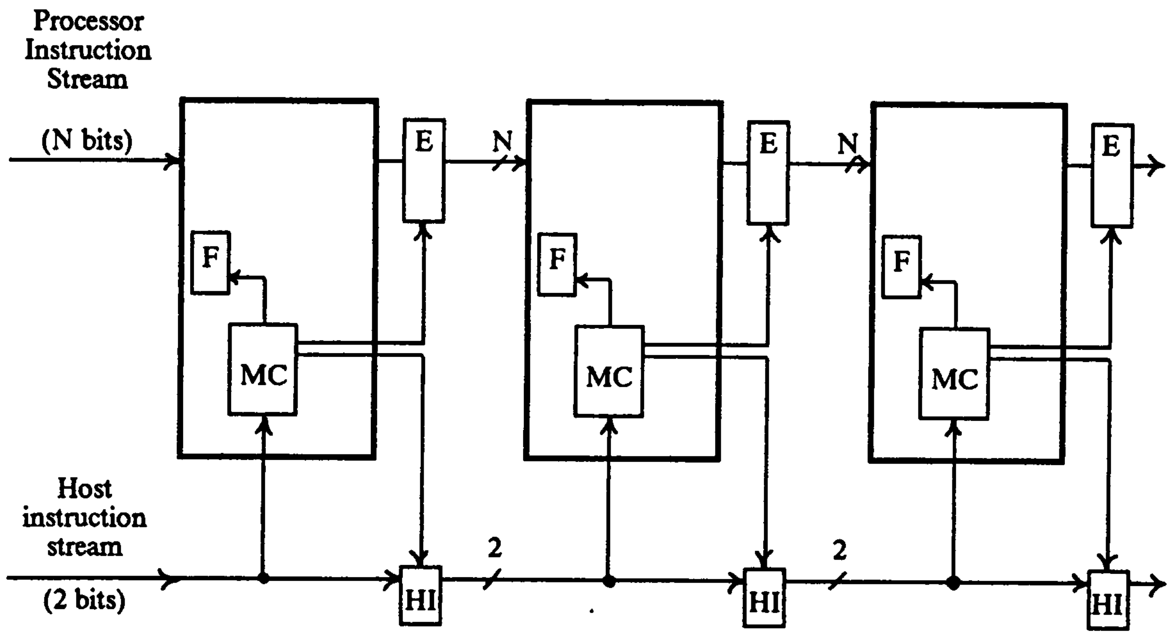


Figure 23. Cell interconnection during program loading.

Host instruction	enable HI	enable E
ResetF	yes	X
SetF	no	X
Load	yes if $F=1$	yes
	no if $F=0$	X
Execute	yes	set by LC

Table 2. Control signals for HI, E registers during array programming

1. ***ResetF***: (F = 0)sent by the host to announces the begining of reloading a new program to the array.

2. ***SetF*** : (F = 1)initiated by the host at the end of each node program, it informs the last cell which has been loading a program to stop doing so and start transferring the incoming instructions (host and processor) to the next cell.

3. ***Load***: when the host initiates this instruction it also sends along a processor instruction to be loaded in to the first cell which is not yet full.

4. ***Execute*** : initiates the start of executing the loaded program in each cell.

While ResetF and Execute instructions should always be passed to the following cells, the SetF instruction should not propagate to the cells which have not loaded their programs yet. The Load instruction should be passed only if the cell is full, i.e. F = 1. Enabling or disabling the contents of the HI and the E registers to flow to the next cell is controlled by the Master controller according to the last host instruction it received and the state of F register.

Figure 24shows an example of the instruction streams sent by the host to load a row of N cellss assuming that the cell interconnection is one byte wide. When the first cell receives a ResetF instruction it will clear its F register and store the instruction (i.e., ResetF) in the HI register to be used by the next cell on the next cycle. When the first cell receives the Load instruction, it will store the processor instruction in its program memory, and since register F is still clear, the contents of HI register will not change and the Load instruction will be passed to the next cell. The first cell will keep loading the incoming processor instructions until it receives a SetF instruction which will set the flag register and prevent any more loading into the cell's memory. All the incoming Load and SetF instruction after this point will be passed to the next cell until a ResetF or Execute instruction is received. If a cell does not have a node assignment then the host will send a SetF instruction only for that node.

In this programming scheme, each cell can start executing its local program immediately after it receives an Execute instruction and does not need to wait until the other cells finish loading their programs.

3.6 Comparison with Other Architectures.

To the author's knowledge the only machines which support pipelining and parallelism simultaneously are the systolic/wavefront arrays [9] and the multipipeline networks proposed recently by Hwang and Xu [17] for vector processing.

Systolic/wavefront arrays are algorithm-oriented, modular and locally connected. Usually, a systolic/wavefront array is built to process a certain algorithm, therefore each cell has a fixed program and the array has fixed inter-cell interconnections.

Multipipeline networks, as proposed in [17], have general purpose applications. They are constructed from multiple pipelined functional units, a register file and two crossbar networks. The network is programmed by mapping the

HI-stream	E-stream
Reset F	x
Load	first byte of cell 1 program
Load	second byte of cell 1 program
.	.
.	.
.	.
Load	last byte of cell 1 program
Set F	x
Load	first byte of cell 2 program
Load	second byte of cell 2 program
.	.
.	.
.	.
Set F	x
Load	first byte of cell n program
Load	second byte of cell n program
.	.
.	.
.	.
Set F	x
Execute	x

where x = don't care.

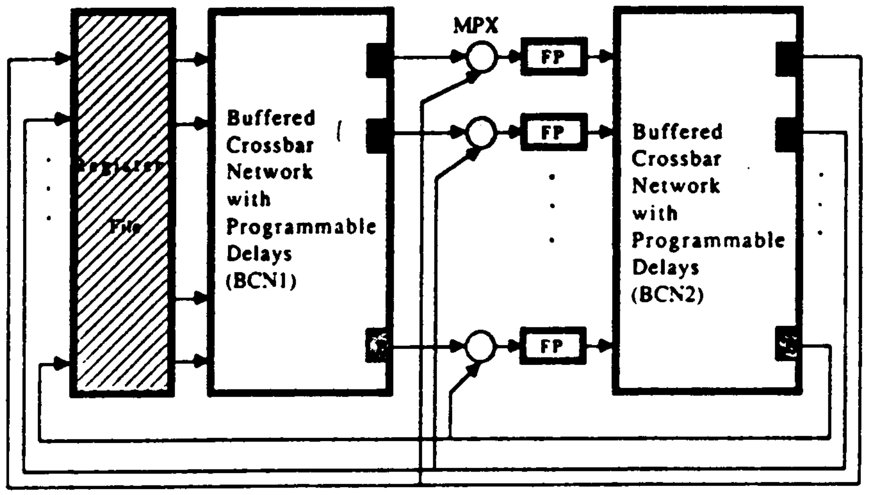
Figure 24. Programming a row of N cells.

nodes of the problem graph into the functional units and interconnecting data dependent nodes through the crossbar network.

Both the multipipeline network and the parallel pipelined array have general applications. The difference is in their scalability. Multipipelined networks are not scalable; problem with larger node numbers than the functional units have to be partitioned into subgraphs, where one subgraph is processed at a time. On the other hand, the machine described here is strongly scalable since it has a single cell type. Large problems are processed in the same fashion as small problems by the use of extra hardware modules.

3.7 Conclusions

The architecture of a universal VLSI array capable of processing any problem graph has been outlined. In addition a scheme to program the array has been proposed. This scheme does not require direct cell-host communication but rather utilizes the inter-PE connections as much as possible.


MPX
Register File
Buffered Crossbar Network with Programmable Delays (BCN1)
FP
FP
FP
Buffered Crossbar Network with Programmable Delays (BCN2)

B - Buffer (Programmable Delays)
FP - Functional Pipeline
MPX- Multiplexer

Figure 25. The multipipelined network [17].

Table 3. Comparison with Systolic arrays and Pipelined networks.

Characteristics	Systolic Array ([20],[9])	Parallel Pipelined Array (in this research)	Multipipelined Array (Wang and Xu [17])
Structure	modular	modular	not modular
Connection	local	local	local or global
Application	special	general	general
Topology	fixed	configurable	configurable
Programmability	fixed	variable	variable

Chapter 4: Embedding Problem Graphs in Processor Arrays

4.1 Introduction

This chapter considers embedding problem graphs, discussed in Chapter 2, in the processor array model presented in Chapter 3.

Two basic approaches have been used for embedding problem graphs into processor arrays: layout algorithms and heuristic procedures. Layout algorithms are suitable for regularly connected graphs, where certain patterns can be defined for embedding low order graphs. These patterns are used recursively to embed higher order graphs. Layout algorithms have been used extensively for embedding complete binary trees.

Heuristic procedures are more suitable for embedding general graphs with irregular topology. Since optimal graph embedding is one of the intractable problems, it is most unlikely that a polynomial time deterministic algorithm can ever be found for such problem.

In Chapter 2, it was shown that pipelined recursive operators are represented by cyclic digraphs with fixed cycle latency. Therefore, when embedding such operators the same cycle length has to be maintained. Such type of embedding is known as rigid embedding and will not be considered here.

On the other hand, sequential operators are represented by acyclic digraphs. Pipelining the processing of such operators requires synchronizing the arrival of data at each node in the digraph. Such constraint has to be satisfied when the operator is embedded. Contrary to recursive operators, sequential operators do not require fixed path lengths. In fact, the longest path in the digraph, henceforth refered to as its *diameter*, can be "stretched" to any desired length as long as the synchronization condition is satisfied.

In Section 4.2 basic definitions related to the embedding problem will be introduced. Section 4.3 will summarize the previous work in this area. Section 4.4 will consider embedding problem digraphs in nearest-neighbor connected arrays. The embedding problem will be directly related to VLSI implementations. Lower and upper bounds on the area required for such embedding will be discussed. Heuristic procedures for embedding acyclic digraphs and their implementation will be presented in sections 4.5 and 4.6. Monte Carlo experimental results will be analysed in section 4.7.

4.2. Definitions

In this section some basic definitions related to the embedding problem will be presented. For graph related terms, the reader is refered to Appendix A.

A ***grid of processors*** (or a grid) G_k is a regular graph of degree k, i.e., each node in the graph has a degree k. It is usually convenient to think of grids as being themselves embedded in Euclidean 2-space [46], therefore each node of the grid is identified by its x and y coordinates. Based on this definition, G_4 will denote the square grid, G_6 will denote the hexagonal grid and G_8 will denote the nearest-neighbor grid.

A grid can either be planar or nonplanar. In a ***planar grid*** edges are not allowed to cross over; moreover, each PE is not allowed to connect more than one pair of its incident edges simultaneously. Such restriction is relaxed in ***nonplanar grids*** where edges are allowed to cross over and/or each PE can connect more than one pair of edges simultaneously. Therefore, one can say that G_8 is inherently nonplanar since inter-processing connections cross over, while G_4 and G_6 can be either planar or nonplanar depending on their PE capabilities. Each PE in a G_k grid can connect up to $k/2$ different paths simultaneously.

A ***directed grid*** D_k is a regular digraph of degree k. Therefore ***indegree***(v) = ***outdegree***$(v) = k$ for each node v in the digraph. A nonplanar directed grid D_k can connect up to k different paths at each node.

Definition 4.2.1: A ***grid-embedding*** of graph (digraph) $G = < V_G , E_G >$ in a grid (directed grid) G_k is a finite subgraph $M = < V_M , E_M >$ such that:

1. $M \subseteq G_k$

2. There exists a one-to-one mapping function f, which maps V_G *into* V_M

3. If $e_l = (v_i, v_j)$ is an edge (arc) in G then there exist a unique $f(v_i) - f(v_j)$ path P_l in M where:

$$P_l : f(v_i), (f(v_i), v_{m1}), v_{m1}, \ldots\ldots, (v_{mn}, f(v_j)), f(v_j)$$

4. If e_l, e_w are distinct edges (arcs) in G, then P_l, P_w are edge-disjoint (arc-disjoint) paths in M.

Definition 4.2.2: A ***rigid-embedding*** of graph (digraph) G in a grid (directed grid) G_k is a grid embedding where:

1. $|V_G| = |V_M|$

2. if $e = (v_i, v_j) \in E_G$ then $(f(v_i), f(v_j)) \in E_M$

Some authors refer to rigid embedding as the mapping problem or subgraph isomorphism.

A ***planar embedding*** is a grid-embedding where paths established to embed edges do not cross over, i.e., if $e_i, e_j \in E_G$ then P_i, P_j are node-dispoint paths except possibly at the end. Such embedding is sometimes referred to as node embedding. On the other hand, in ***nonplanar embedding***, or sometimes referred to as edge-embedding, paths are allowed to cross over.

Throughout this chapter, a vertex in the embedded graph will be called a ***node*** if it is an image of a node in the original graph, i.e., if $v \in V_M$ then $f^{-1}(v) \in V_G$. It will be called a ***connector*** if it is used to connect a path between distant nodes. In this case, $v \in V_M$ while $f(v)^{-1} \notin V_D$. A connector has indegree of one and outdegree of one. Nonplanar embeddings allow more than one connector to be embedded on the same PE .

4.3. Previous Work

In this section a summary of the previous work in this area is given. Most the work have been devoted to undirected graphs, more specifically to binary trees and planar graphs.

4.3.1 Embedding Binary Trees

Due to the several applications of binary tree machines, many researchers have extensively studied embedding complete binary trees in processor arrays. See for example, [9], [33], [34], [43]-[49], [51]. A complete binary tree is a binary tree in which every internal node has exactly two children nodes and all the paths from the root to the leaf nodes have the same length [47]. Different layouts have been proposed for embedding binary trees in hexagonal-, square-, and nearest neighbor-connected arrays. The following are different goals which have been set for the embedding:

Minimum area embedding: obviously, the optimal area required to embed a complete binary tree of n vertices is also O(*n*) area. A well known layout scheme which occupies such area is the H-tree layout [9], [33], [34], [47], [49]. Here, the embedding is performed recursively using pattern that look like the letter "H". Figure 26 and Figure 27 show the H-tree layout for a complete binary tree in square and hexagonal connected arrays, respectively.

Another scheme which also occupies an O(*n*) area, was proposed by [43]. Here the embedding is constructed recursively by defining basic tile types for both hexagonal- and square-connected arrays. In this scheme 100 percent utilization of the PE's within the area required for embedding is achieved (a PE is utilized if its computational power is used; obviously this is not the case when the PE is used as a connector). Such utilization is achieved by allowing the PE to act as a node and as a connector simultaneously.

Access to all leaves: such criterion is important in many processing arrays where the I/O ports lie on the boundary of the chip. In [32] the authors proved that a complete binary tree with *n* leaves can be layered in an O(*n log n*) area if the leaves are on the boundary of the chip, separated by a unit distance, and the boundary is convex. They also proved that it is impossible to embed a binary tree with all leaves on the boundary with O(*n*) area.

Minimum edge length: which implies reducing the number of PE's used to connect distant nodes, consequently, reducing the time required to traverse the tree. In

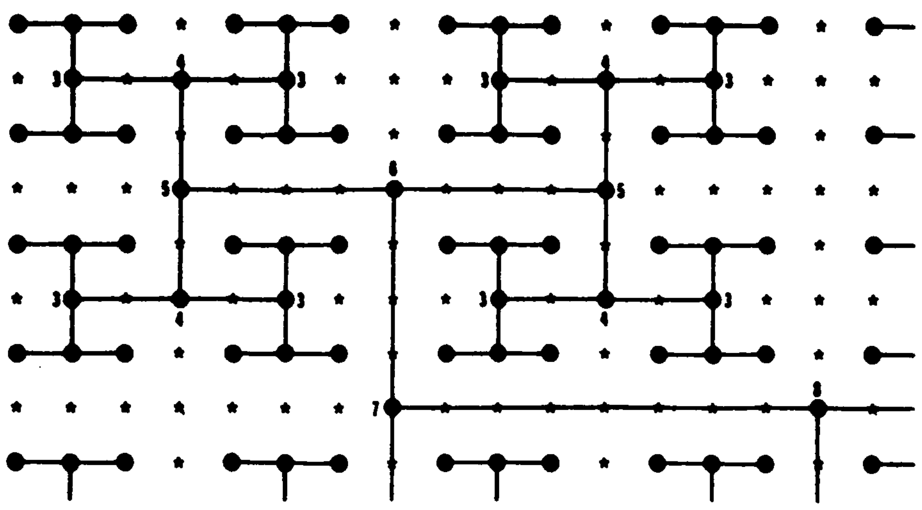


Figure 26. H-tree layout in square-connected array.

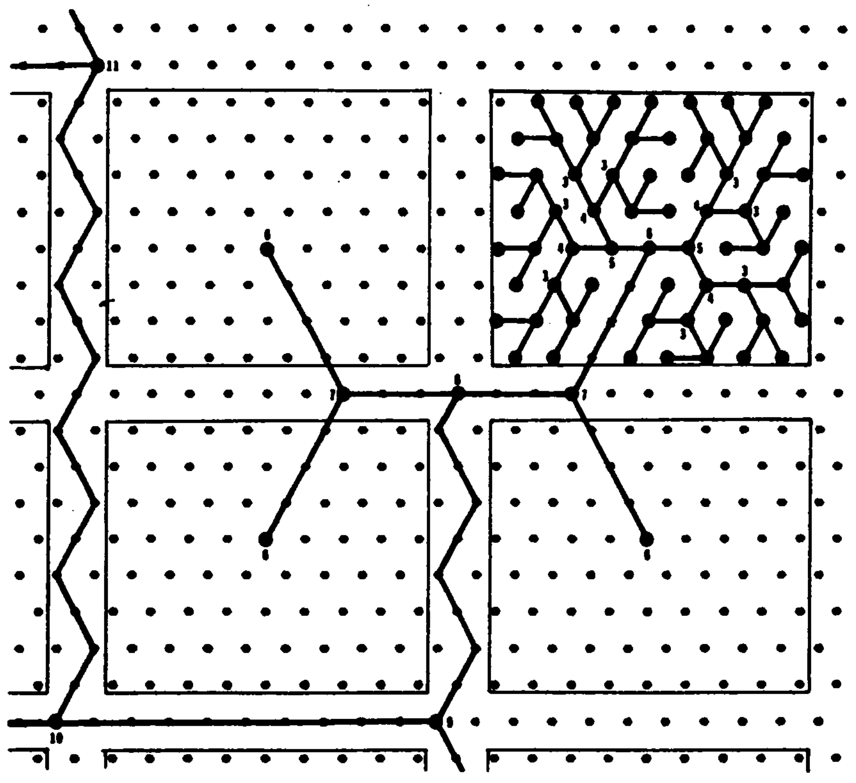


Figure 27. H-tree layout in hexagonal-connected array.

[51] a "hyper H- tree" was used to obtain a minimum edge length of $O(\sqrt{n}/\log n)$ for an arbitrary binary tree.

4.3.2 Embedding Planar Graphs

Planar and nonplanar embedding of planar graphs have both been considered by several researchers [38], [46], [47], [50]. Three cost measures have been used in embedding planar graphs: the rectangular area used for embedding, the total edge length, and the node-cost measure (which is the minimum number of grid points needed for embedding the graph).

Valiant [46] has studied planar embedding of planar graphs in square connected grid. He has proven that for the class of planar graphs of degree three or four, if $A(n)$ is the square area required to embed a graph of order n, then there exist c_1, $c_2 > 0$ such that $c_1 n^2 \leq A(n) \leq c_2 n^2$ Furthermore, he has proven that the upper bound on the area is $9n^2$.

Storer [50] has studied the same problem and proved that there is a tighter upper bound which is n^2. He also proved that the lower bound is $(n-2)^2$ for planar graphs of degree four and $(n/2)^2$ for planar graphs of degree three. He developed three heuristic strategies for minimal-node-cost embeddings.

Nonplanar embedding or embedding with cross overs has also been considered [46], [47]. Using the planar separation theorem,[4] it can be proven that

[4] The theorem states that for a planar graph of order n, one can find a set of no more than $4\sqrt{n}$ nodes whose removal disconnects the graph into two parts, neither of which has more than $2n/3$ nodes.

nonplanar embedding of planar graphs of degree four requires an O(n log$^2 n$) area. The details of the proof are shown in [46]. It is worth to mention that in this proof the number of cross overs at any point is not specified nor the local connectivity is preserved.

Bokhari [38] considered the mapping problem (rigid-embedding) where problem graphs are mapped into nearest-neighbor-connected arrays (the Finite Element Machine). In addition to the local connections between the processors, the machine has a time-shared bus which can connect any pair of processors that are not adjacent. The author developed a heuristic procedure which maps the problem graph into the processor array, with the objective of increasing the number of graph edges which are mapped to local connections. The optimum solution would be reached if such condition would be satisfied for all graph edges, in this case the time-shared bus would not be used and the machine would reach its greatest efficiency.

4.4 Synchronous Nonplanar Embedding of DAG's

Storer [50] has proven that embedding planar graphs with minimum area is an NP-hard problem, whether the embedding is planar or not. Therefore, it is most unlikely that a deterministic polynomial time algorithm can be found for such problem. Before discussing heuristic procedures we will consider some practical issues which have to be considered in VLSI applications and will impose some constraints on the embedding problem. The author feels that many of these issues have been ignored by most researchers who worked in this area. Upper and lower bounds on the area will be discussed taking into account these issues.

Some practical considerations:

1. The time needed to traverse a wire of length L is $\theta(L)$ (the unit length is the distance between any adjacent points on the grid). Therefore, in a clocked system if the indegree of a node is two the embedding should guarantee that the two data will arrive at the same time. Such embedding will be called ***synchronous embedding***.

2. The most expensive cost for embedding problem graphs into processor arrays is the rectangular area which contains the embedding. The number of holes within the rectangle is irrelevant. Therefore, minimizing the area will be one of the objective criterion considered here. The area will be represented as: h.w where h is the height and w is its width, as shown in Figure 28.

3. The number of crossed over paths at any point on the grid is limited and has to be prespecified at design time, since it corresponds to either the number of wire layers or to the number of buses. The embedding problem considered here will limit the number of cross overs to the minimum which is two.

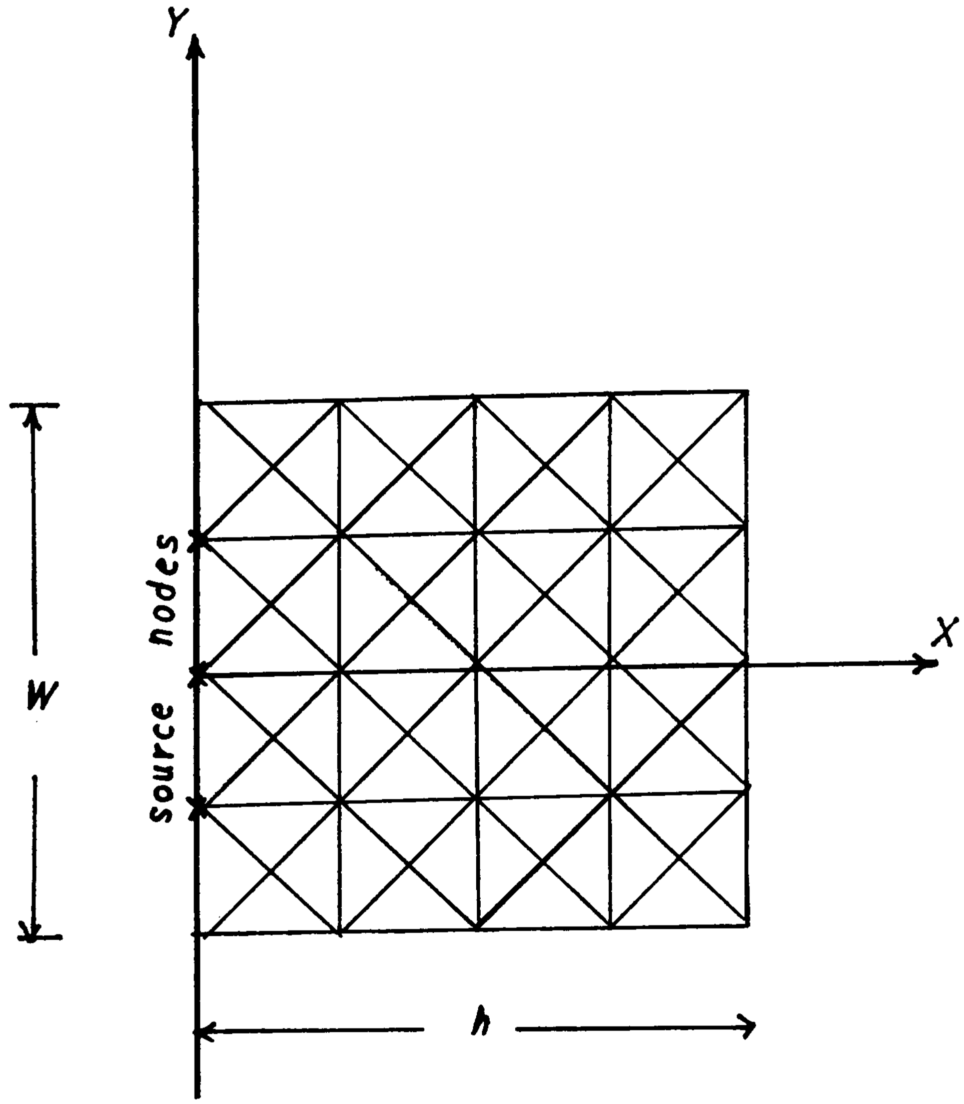


Figure 28. X, Y coordinates of the grid.

4. PE's used to embed source and the sink nodes of the problem digraph should be accessible to the outside world. In processor arrays where I/O ports lie on the boundary of the chip, source and sink nodes should be accessible to these ports.

5. In applications where the array is directly interfaced to a data source without a memory stage, as in real-time image processing systems, the locations of the PE's which perform the I/O are prespecified since they are hardwired to the peripheral devices. Hence, throughout this work, it will be assumed that the source nodes lie on one side of the grid, one source node per PE. Their order is known, they are not interchangeable but can be spread apart. Although the acyclic problem digraphs discussed in Chapter 2 are planar, requiring all the source nodes to be on one side of the array will force the embedding to be nonplanar.

Definition 4.4.1: An (N,N) binary bipartite digraph is a digraph with N source nodes with outdegree of two and N sink nodes of indegree of two. All the arcs are directed from the source to the sink nodes.

Theorem 4.4.1: Any (N,N) binary bipartite digraph can be embedded synchronously in a nearest-neighbor-connected grid in $(2N)^2$ area with no more than two edges crossing at any point on the grid.

Proof: Let all the sink nodes be embedded on one side of the grid in column one. We want to map the arcs of the digraph into equal length paths directed from the source to the sink nodes, such that each path is unique, and no more than two paths intersect at any point.

Each source node has two sink nodes as its immediate successors. If the node i is located on row R_i, then rows R_i, $R_i - 1$ will be reserved for paths outgoing from node i as shown in Figure 29(a). Since each sink node also has two source nodes as its immediate processor, we will choose to locate the sink node on one of the outgoing paths from the upper source node. The path from the lower source node is established by following one of its outgoing paths then a diagonal path which intersects the upper path at the sink node, as shown in Figure 29(b). Here node i has two successors s_1, s_2 which are also the successors of nodes j, k. Figure 30 shows the paths when two nodes have two common successors.

Following the above procedure then $2N$ rows are needed for the embedding of N source nodes, and since it is possible that the upper and the lower source nodes would have the same sink node then $2N$ columns are also needed in order to have equal path lengths.

If we let all the sink nodes lie on the same column at $2N$ grid distance from the source nodes then all the diagonal lines will be parallel, hence each path connecting a source to a sink is unique. Moreover, since only north and north-east grid segments have been followed, no more than two paths will intersect at any point.

Figure 31 shows a complete embedding of a (4,4) binary bipartite digraph.

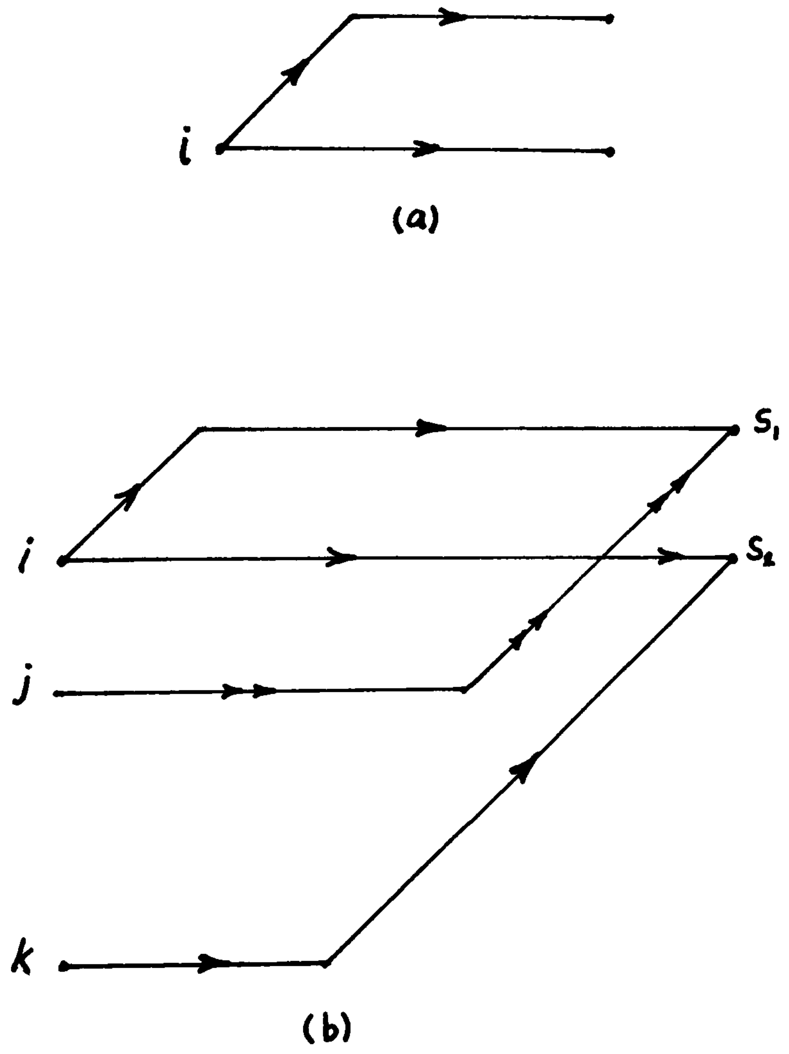
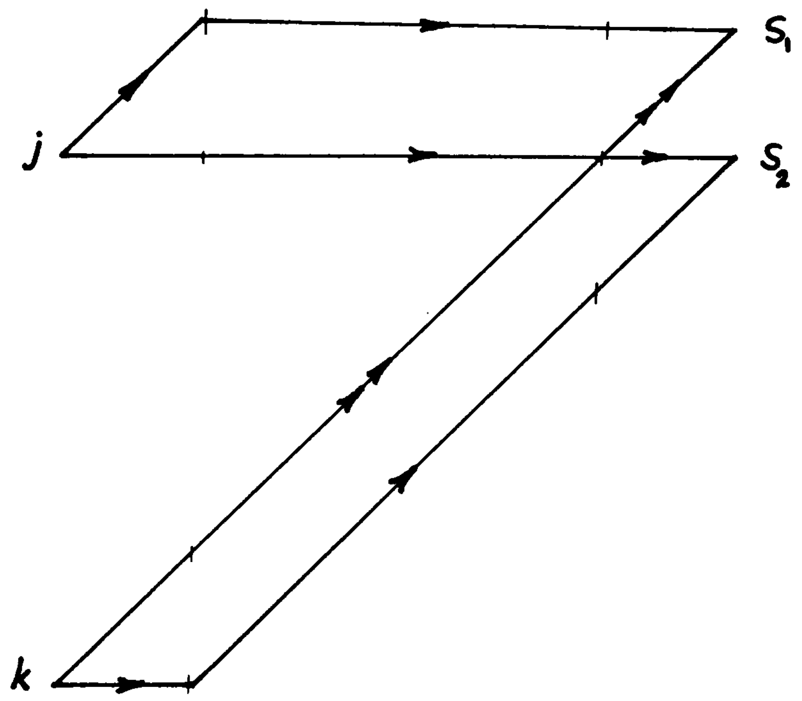


Figure 29. Synchronous embedding of bipartite digraph: (a) outgoing paths from each source. (b) the paths when the successors of a certain node are also the successors for other two nodes.



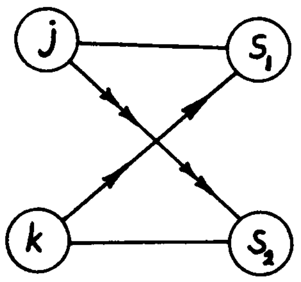


Figure 30. **Synchronous embedding of bipartite digraphs.:** the paths when two nodes have common successors.

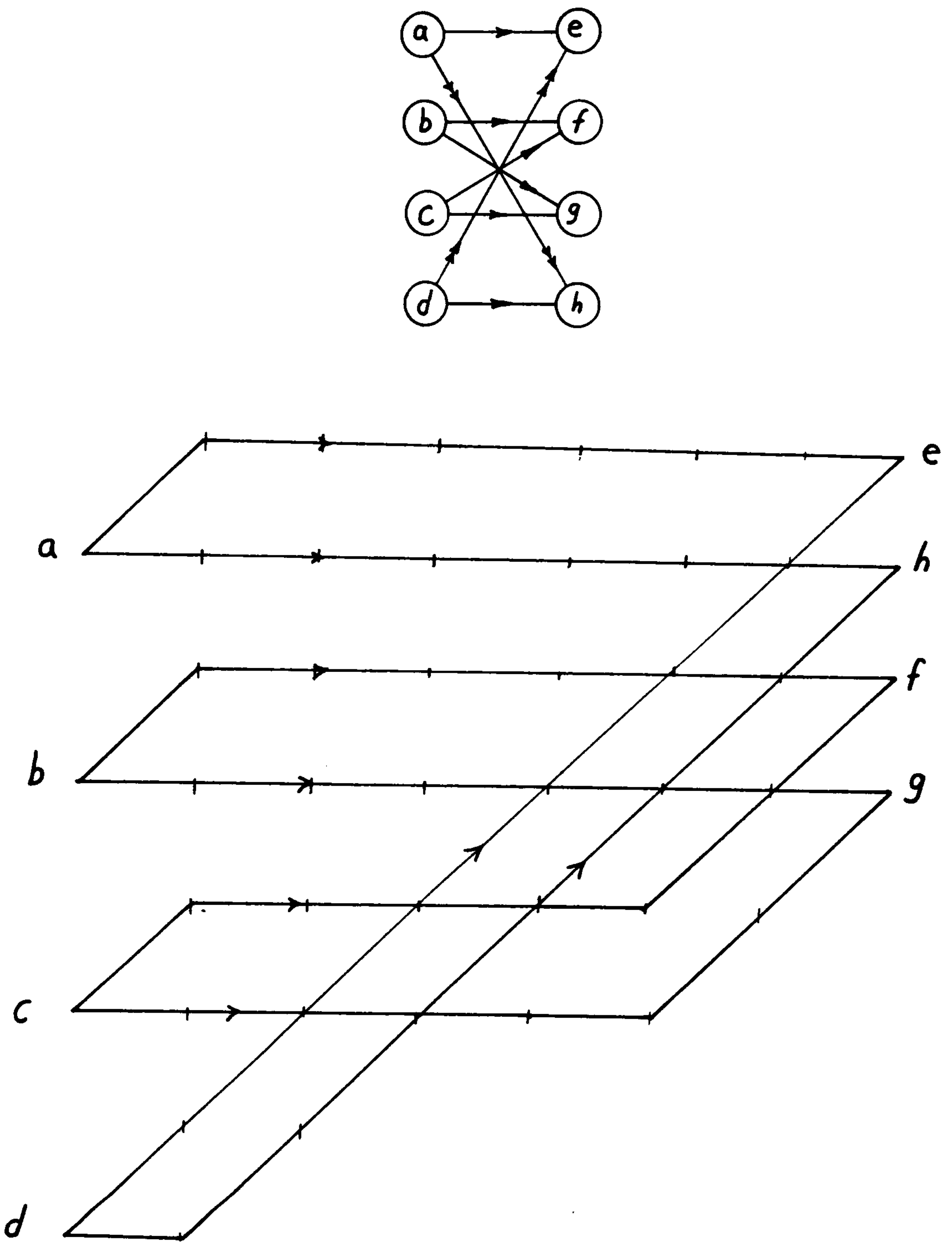


Figure 31. A complete embedding of (4,4) binary bipartite digraph.

Definition 4.4.2: A *$(n, n_p, d)DAG$* is a directed acyclic graph with order n, maximum number of nodes at any level equal to n_p, and a diameter d.

Theorem 4.4.2: Any (n, n_p, d) DAG with indegree ≤ 2 and outdegree ≤ 2 for each node, requires $(2n_p)(4n_p d)$ area to be embedded synchronously in G_8.

Proof: For synchronous embedding all the paths should be of equal length. A directed acyclic digraph can be viewed as d-cascaded bipartite digraphs such that the sink nodes for one stage are the source nodes for the next stage. The maximum possible size for any stage would be (n_p, n_p).

In theorem 4.4.1 it was proven that any (n_p, n_p) bipartite binary digraph can be embedded in $(2n_p)^2$ area, provided that the source nodes are placed two unit distance apart on one side of the grid. Hence, after embedding each stage we need a network which spreads the sink nodes of every stage before embedding the next stage. An $(n_p)^2$ grid area is needed for such network. It will be needed between every pair of successive stages as shown in Figure 32.

Theorem 4.4.3: There are many $(n,3,d)$ and $(n,2,d)$ DAG's which can be embedded synchronously in O(n) area in G_8.

Proof: This can be seen easily by refering to Figure 33 where a digraph of each class is drawn when d=4. Note that these digraphs are inherently synchronous,

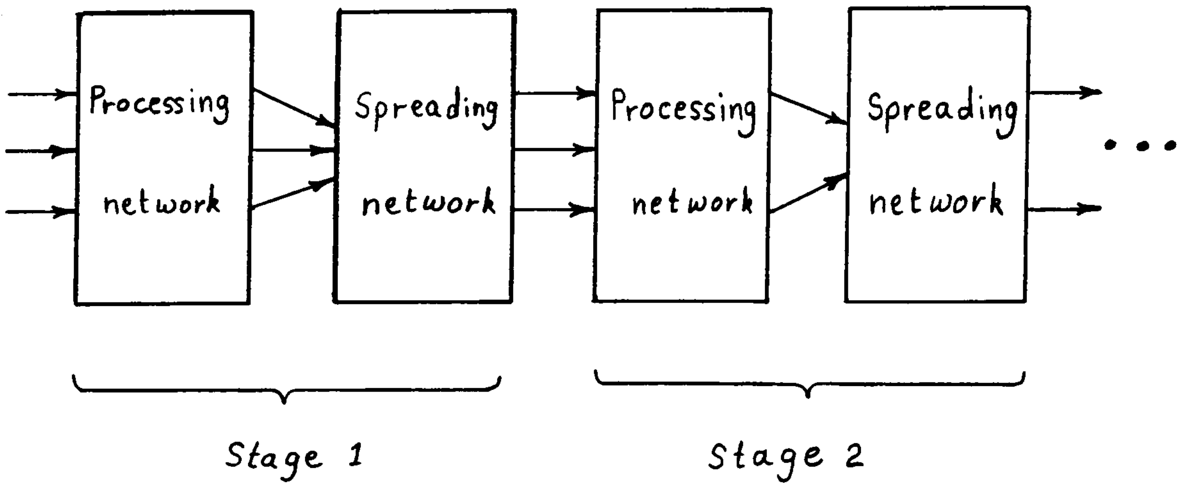


Figure 32. Synchronous embedding of d-stage DAG.

therefore no extra synchronization nodes are needed. Also, $n = n_p.d$ for these graphs.

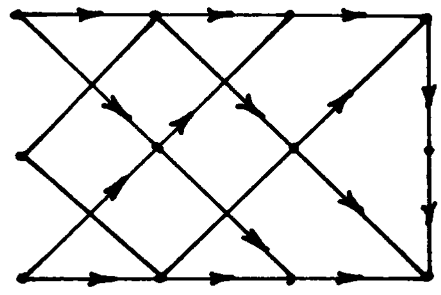
Theorem 4.4.4

A complete binary tree with n leaf nodes requires $n \log n$ area to be embedded synchronously in G_8.

4.5 Heuristic Procedures for Embedding DAG's

This section presents heuristic procedures for synchronous embedding of DAG. In addition to the constraints discussed in the previous section, the following are some assumptions which have been made:

1. All the digraphs considered are bilaterally connected.
2. All the nodes of the graph are computational or switching nodes; no separator nodes are assumed. There is no loss of generality as long as the problem (i.e., the algorithm) is shift invariant. In this case, the shift can be done before performing the algorithm.
3. It is assumed that the embedding is performed on a fault free array; fault tolerance is not considered.



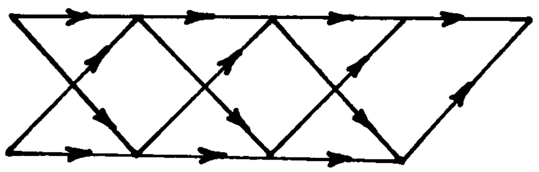


Figure 33. Digraphs which require area of order n.

4.5.1 Breadth-First Greedy (BFG) Heuristics

In breadth-first embedding, the embedding is started at the source nodes and then one level of vertices is embedded at a time until the sink nodes are reached. At each level, it tries to embed the maximum possible number of graph nodes at one grid distance from the previous level. The heuristic tries to achieve this goal regardless of the consequences encountered when embedding the rest of the graph. Therefore it will be described as a greedy procedure. The vertices at each level are either nodes of the graph or connectors establishing paths to embed edges. Note that the vertices at each level are specified dynamically as we perform the embedding, also the level of each node in the embedded graph is not necessarily the same as in the original graph since connectors are inserted between the graph nodes to insure local connectivity.

At any stage of the embedding, the last set of vertices which have been embedded will be called the parent set, while the set of vertices which will be embedded next will be called the children set. The general structure of a BFG heuristic which embeds a graph of order n will be as follows:

```
step 1: area-height ← number of source nodes
step 2: parent-set ← source nodes; embed source nodes.
step 3: While no-embedded-nodes < n Do
        (a) find children-set
        (b) embed the children-set
```

(c) parent-set ← children-set

Finding the children-set

In step (1) of the above procedure the children-set is constructed. As we said earlier it is the set of all vertices (nodes or connectors) which can be embedded at one grid distance from the vertices in the parent set . The elements of the children set are determined by examining the successor(s) nodes of each parent in the parent set. Doing so, one of the following will occur:

1. The indegree of the successor is one: in this case the successor will be included in the children set and it can be embedded at any available vacant location (grid point), one edge distance from its parent.

2. The indegree of the successor is two and the other parent has not been embedded yet: in this case a synchronization node need to be inserted between the parent and the successor and it will be added to the children set. The synchronization node can be embedded at any vacant location which is one-grid distance from its parent.

3. The successor has indegree of two and both parents have been embedded: the next step will be to examine the locations of the parents. If the two locations are close enough so that there is a vacant location which is one grid distance

from both parents, then the successor will be considered in the children set. Otherwise, a connector node will be inserted between each parent and the successor. When embedding these connectors, each should be embedded towards the other parent so that the successor node can be eventually embedded on the next level(s).

Embedding the children set

For each element in the children set there exist one or more possible location for embedding, many of these locations are common. We want to select one location per child such that the solution set satisfies the following:

1. Each node of the problem graph occupies a grid point exclusively.

2. Two connectors can occupy the same grid point if they are incident from different directions.

3. The solution set contains one location per row on the grid, i.e., no two locations lie on the same horizontal line.

4. Since the graphs are acyclic, only the moves shown on Figure 34 will be considered.

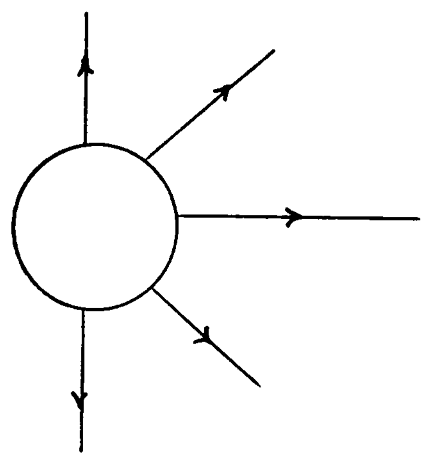


Figure 34. Possible moves from any point on the grid.

Conditions (1) and (2) are due to the properties of the processor array discussed earlier. Conditions (3) and (4) have been added to guarantee that there is at least one outgoing path (the east edge) from each child to the next level.

Several techniques can be used to solve for the locations of the children set which satisfies the above constraints. In implementing the heuristics which will be described in this section, backtracking procedure [42] has been selected to search for a feasible solution in a given solution space. The choice seems to be reasonable since the possible locations per child are finite. The search space is described in a table, one row per child. It lists all possible vacant locations for embedding that child. The locations in each row are ordered such that best location are listed first. Moreover, the rows of the table are sorted such that those with least number of possible locations are placed on the top of the table. This will speed up the search time, since less backtracking will be needed. Before going any further, let's consider the following example which illustrates the stages of the BFG heuristics.

Example 4.5.1: Consider embedding the digraph shown on Figure 35, where nodes 1 through 4 are source nodes, and node 12 is the sink node. The sequence of stages of embedding the digraph are shown below. At each stage, the newly created connectors are listed. The possible vacant locations to embed each child are given. If the child has more than one possible location, then the underlined location is the one chosen by the backtrack search procedure. The embedded digraph is shown on Figure 36.

Stage 1

Source	location
1	(0, 2)
2	(0, 1)
3	(0, 0)
4	(0, -1)

Stage 2

connector 13 has been inserted between 4 and 12.

Child	Possible locations
6	(1, 1)
7	(1, 0)
5	$\underline{(1,\ 2)}$ (1, 1)
13	$\underline{(1,-1)}$ (1, 0)

Stage 3

connector 14 has been inserted between 7 and 9.
connector 15 has been inserted between 13 and 12.

Child	Possible locations
8	$\underline{(2,\ 1)}$ (2, 2)
10	$\underline{(2,\ 2)}$ (2, 1)
15	$\underline{(2,-1)}$ (2, 0)
14	$\underline{(1,-1)}$ (2, 0) (2, 1) (2, -1)

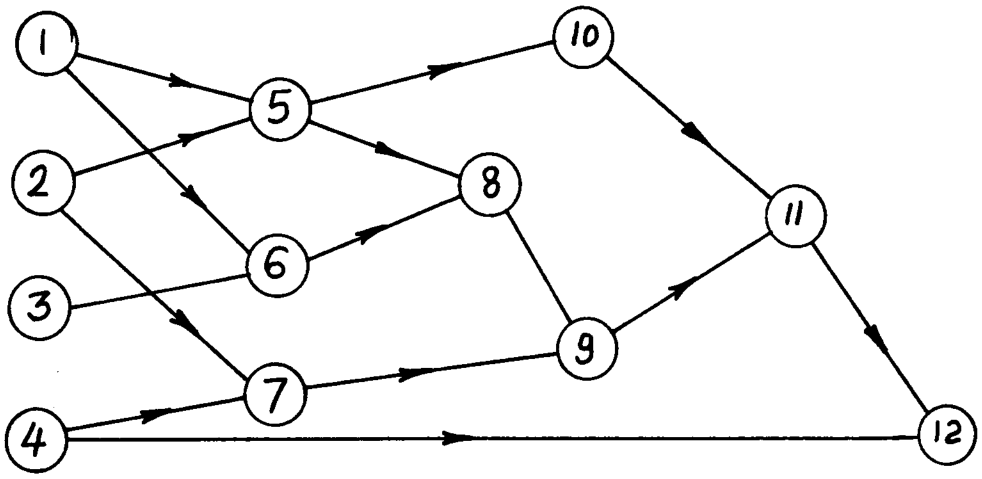


Figure 35. Example 4.5.1

Stage 4

connector 16 has been inserted between 10 and 11.
connector 17 has been inserted between 15 and 12.

Child	Possible locations
9	$\underline{(2,\ 0)}$
16	$\underline{(3,\ 2)}$ (3, 1)
17	(2, 0) $\underline{(3,\ -1)}$ (3, 0)

Stage 5

connector 18 has been inserted between 17 and 12.

Child	Possible locations
11	$\underline{(3,\ 1)}$
18	$\underline{(3,\ 0)}$ (4, -1) (4, 0)

Stage 6

Child	Possible locations
12	$\underline{(4,1)}$ (4,0)

The above procedure is not complete, since there is always the possibility that the embedding of the children nodes at any level will fail, even if we allow the height of the area to increase as we perform the embedding, and even if the cardinality of the children set is less than the area height. The heuristic procedure shown on Figure 37 considers this situation. Here, whenever the embedding of the children set fails the heuristic returns one level backward and re-embeds the the parent set, hoping that with the new parent locations the children set can be embedded. This process is repeated until either the children set can be embedded with the new parent locations, or parent re-embedding fails, i.e., all the solutions

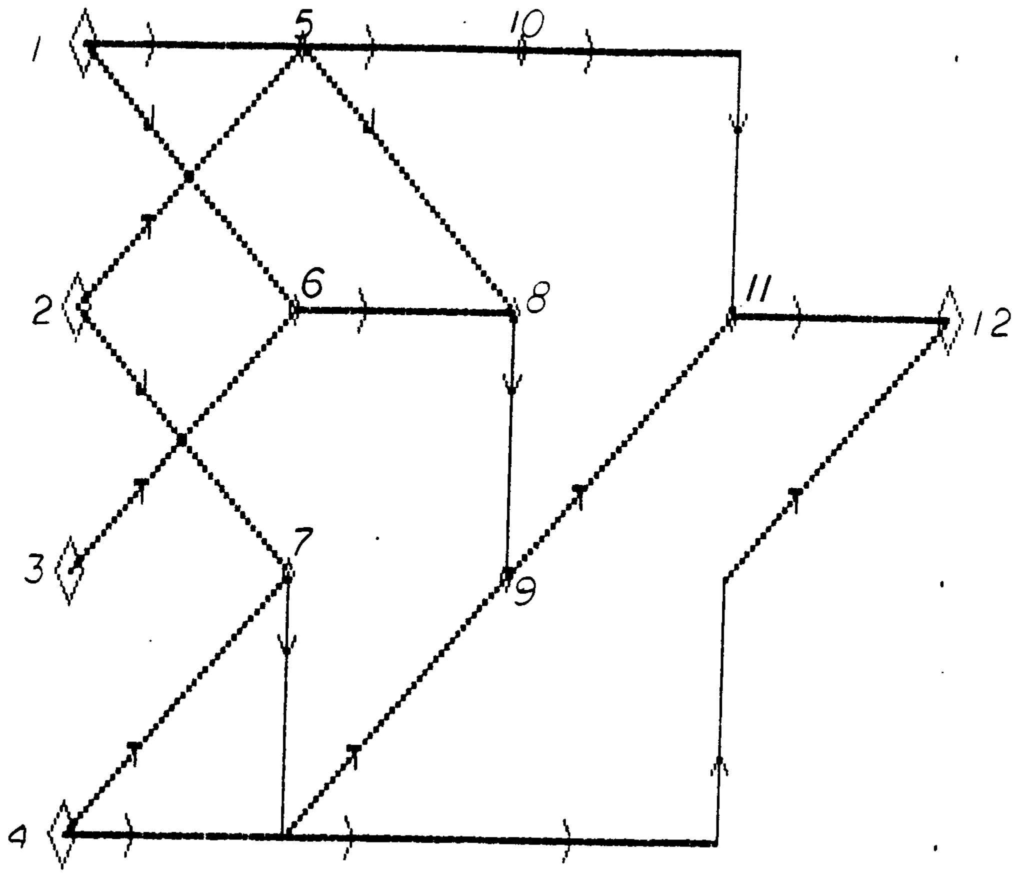


Figure 36. Embedded digraph of example 4.5.1

for the parent set have been exhausted without success to embed their children. If the parent re-embedding fails, then the procedure dislocates the embedding, increments the height of the rectangular area by one, and restart the embedding from the beginning.

This procedure brings up two new issues: how to embed the source nodes when the area height is greater than the number of the source nodes, and re-embedding the parent set.

Embedding the source nodes: In general, when the area height is made greater than the number of the source nodes, the source nodes cannot always be equally spaced, and the problem of distributing the spacing among the source nodes a rises. A possible solution to this problem would be to learn from the previously embedded portion of the graph which could not be pursued. Since the cardinality of the outreach set for each source node is proportional to the number of paths which will descend from that node, the outreach degree for each source node[5] will be chosen as basis for allocating the extra spacing around the source nodes. The extra spacing will be allocated to the source nodes in descending order according to their outreach degree.

Re-embedding the parent set: Re-embedding the parent set means selecting a new solution set to embed the parents. Here again the backtracking search proves to

[5] it will be defined as the ratio between the cardinality of the outreach set for that source and the sum of the cardinalities of outreach sets of all source nodes.

```
input:  DAG of order n.
step1:  area-height ← number of source nodes.
step2:  parent-set ←  source nodes; embed source nodes.
step3:  While no-embedded-nodes < n  Do
        (a) find children-set
        (b) embed the children-set
           If fails then re-embed parent-set
                    if o.k. then  return to (a).
                          else increment area-height
                              and restart at (2)
                else  parent-set ← children-set
```

Figure 37. One generation lookahead BFG heuristic.

be a good choice since it allows the search to be continued for other solution sets, if there is any. The new solution set should not have been considered before.

4.5.2 Two-generation Lookahead BFG Heuristic

So far, one generation lookahead has been considered. Figure 38 shows the outline for a two generation lookahead embedding. The procedure is similar to the previous one except that after the set of children nodes is found, we search for pairs of mate nodes among them.[6] When embedding the children nodes, Mate nodes are brought as close as possible. This will reduce the length of the path to their child node. Although this procedure examines two generations ahead from the current set of parent nodes, it is still a greedy heuristic since it tries to embed maximum possible number of nodes on the next level, at the best location .

4.5.3 Modified BFG Heuristic.

The two heuristics discussed earlier try to maximize the number of graph nodes embeded at each level, using as much area height as needed to achieve this goal. As a result, their performance can be described as towards minimizing the diameter of the embedded graph rather than minimizing the area required for embedding. The heuristic described on Figure 39 will also try to embed the

[6] two nodes are said to be mates if they have a common child node.

```
Input:   DAG of order n.
Step1:   area-height ← number of source nodes.
Step2:   parent-set ← source nodes, embed source nodes.
Step3:   While no-embedded-nodes < n Do
           (a) find children-set and recognize the mates.
           (b) embed the children-set
               If fails then re-embed parent set
                   if o.k. then return to (a).
                   else increment area-height and restart at (2)
               else parent-set ← children-set
```

Figure 38. Two generation lookahead BFG heuristic.

```
Input: DAG of order $n$.
Step1: area-height  ←  number of source nodes.
Step2: parent-set ← source nodes, embed source nodes.
Step3: While no-embedded-nodes < n  Do
    (a) find children-set  and recognize the mates.
    (b) embed the children-set
        If fails then re-embed parent set
                if o.k. then  return to (a).
                    else
                     Shift k parents.
                     If not possible then increment
                        area-height and restart at(2)
            else  parent-set ← children-set
```

Figure 39. Modified BFG heuristic.

maximum possible number of nodes at each level. But in case such goal cannot be achieved, the procedure tries one more option before incrementing the area height. It re-examines the set of parent nodes to know whether it is possible to shift one or more parent node to the next level, such that the resulting children set can be embedded without increasing the area height. By shifting a parent node to the next level, we mean that it will no longer belong to the parent set but rather to their children set. A delay node will be added to the parent set and will act as its predecessor. Figure 40 on page 101 shows an example.

To guarantee that shifting a parent node to the next level reduce the number of children node, the following have to be satisfied:

1. The parent node has to be of indegree one and outdegree two, since shifting every such node will reduce the number of children by one.

2. The children of that parent node should be exclusive successors of that node, i.e., both children nodes should have indegree of one. Otherwise shifting the parent will result in another children set with the same size as the previous one.

3. Shifting the parent node should not cause an indefinite delay in embedding the graph. Such situation will occur if all the parents are delay nodes and/or connectors travelling in parallel paths.

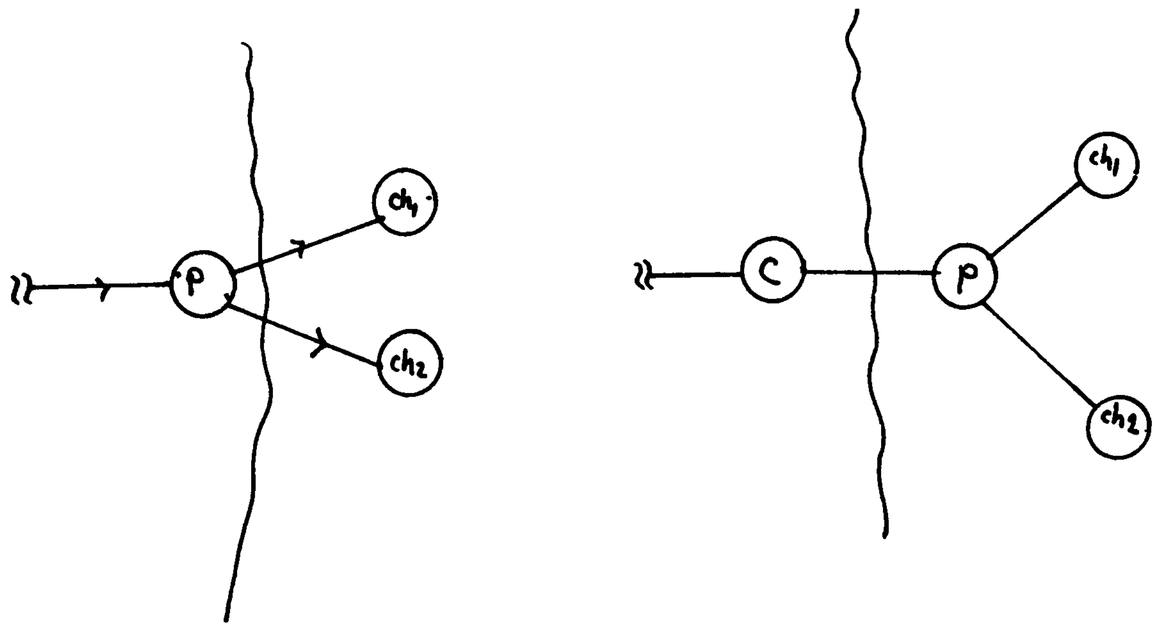


Figure 40. Shifting a parent node to the next level

4.6 Implementation and Examples

The three heuristics described in the previous section have been implemented in Turbo Pascal on IBM PC. The digraph to be embedded can either be fed to the program from a disk file, or can be self-generated. In the former case, the digraph is described as a single list. Only the successors of each node are specified.

For self-generation of a graph, only the number of source, sink and intermediate nodes are specified. A pseudo-random procedure has been developed to generate an acyclic digraph with the specified order.

The embedded digraph is given graphically as well as in doubly-linked-list form.

Example 4.5.2: Consider the digraph shown on Figure 41. Its embedding under the three heuristics are shown on Figure 42 through Figure 44. For this example, the two-generation BFG gives the better area and diameter.

4.7 Monte Carlo Experimental Results

To analyze the performance of the three heuristics, a pseudo-random graph generator, described in Appendix B, has been developed. It can randomly generate an acyclic graph given the number of source, sink and intermediate nodes.

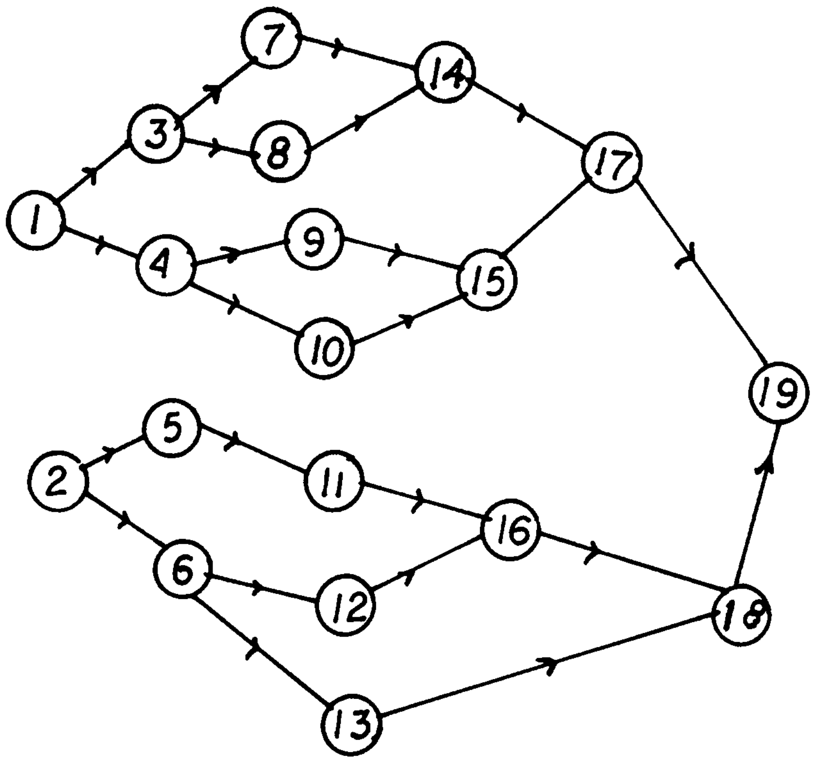


Figure 41. Example 4.5.2

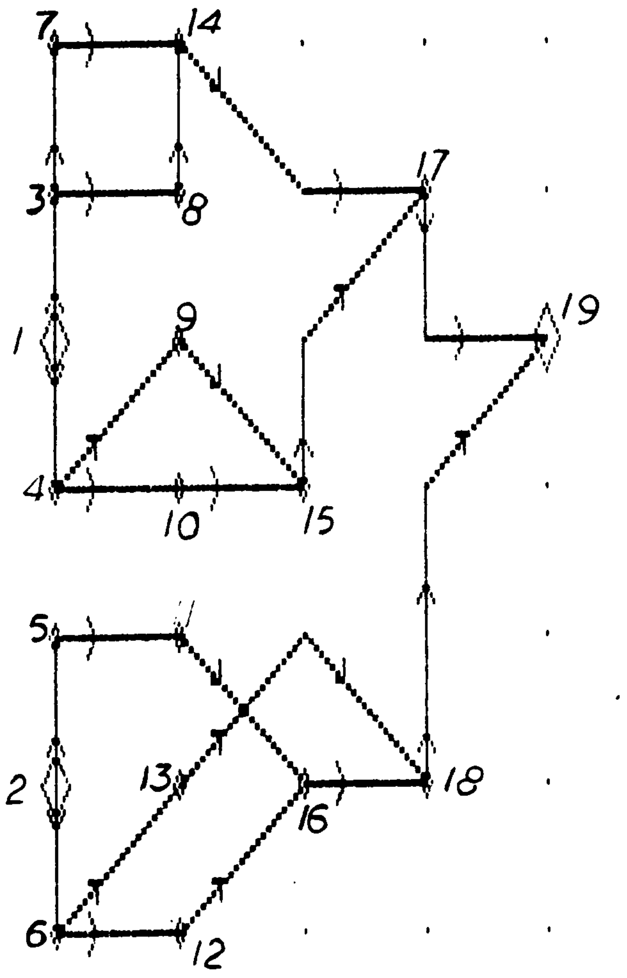


Figure 42. Embedding example 5.4.2 under one-generation BFG heuristic.

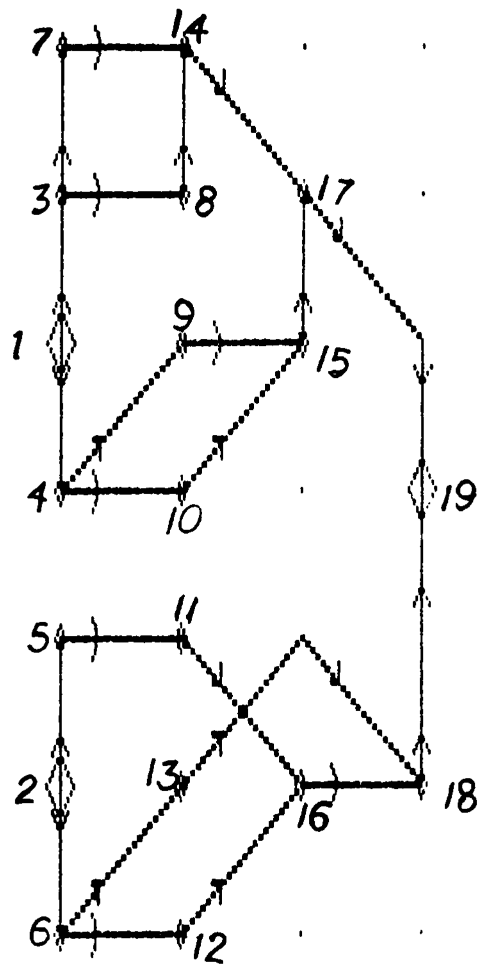


Figure 43. Embedding example 5.4.2 under two-generation BFG heuristic.

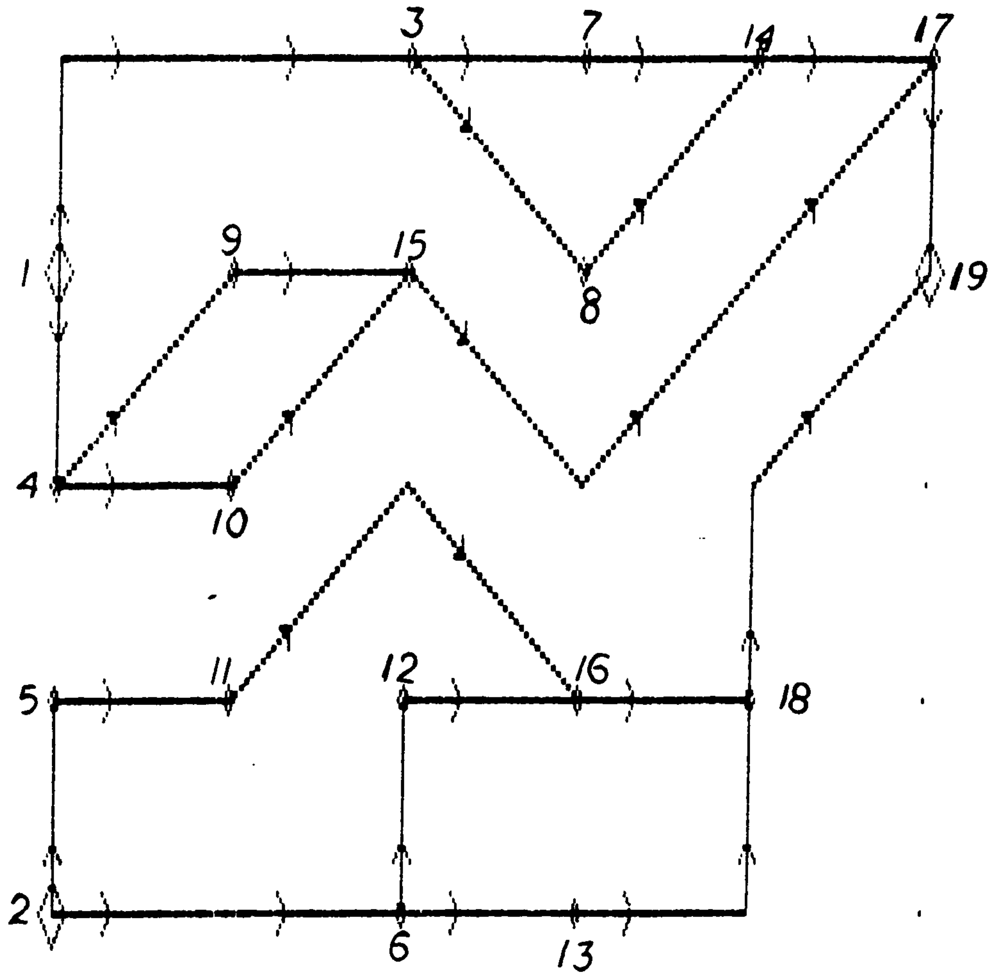


Figure 20. Embedding example 5.4.2 under modified BFG heuristic.

Procedure	Area	Diameter
One generation BFG	6x4 = 24	7
Two generation BFG	6x3 = 18	6
Modified BFG	4x5 = 20	6

Table 4. Comparison between the three heuristics applied to EXAMPLE 4.5.1

The three heuristics have been tested over 600 randomly generated graphs. The diameter of the graph and the rectangular area used for embedding have been selected as basis for comparison.

Figure 45 shows the results for one class of graphs which have thirty operational nodes, including one sink node and nine source nodes. Each graph has been embedded by the three heuristics. The horizontal axis corresponds to the number of nodes in the synchronized graph (the original graph before embedding) which is the sum of the number of operational nodes and the number of delay nodes needed for synchronization. Figure 46 shows the lower bound on the area (equal to the total number of nodes) and the area used for embedding by the three heuristics, for the same experiment.

Comparing the performance of the three heuristics according to the diameter of the embedded graph, indicates that the second and the third heuristics give the same diameter for most the cases. Also this diameter is shorter than the one used by the first heuristic. The third heuristic does not seem to provide a better solution than the second as expected.

Analyzing the area performance, it appears that although the second and the third heuristics need less area than the first heuristic, this area is much higher than the lower bound for most the cases.

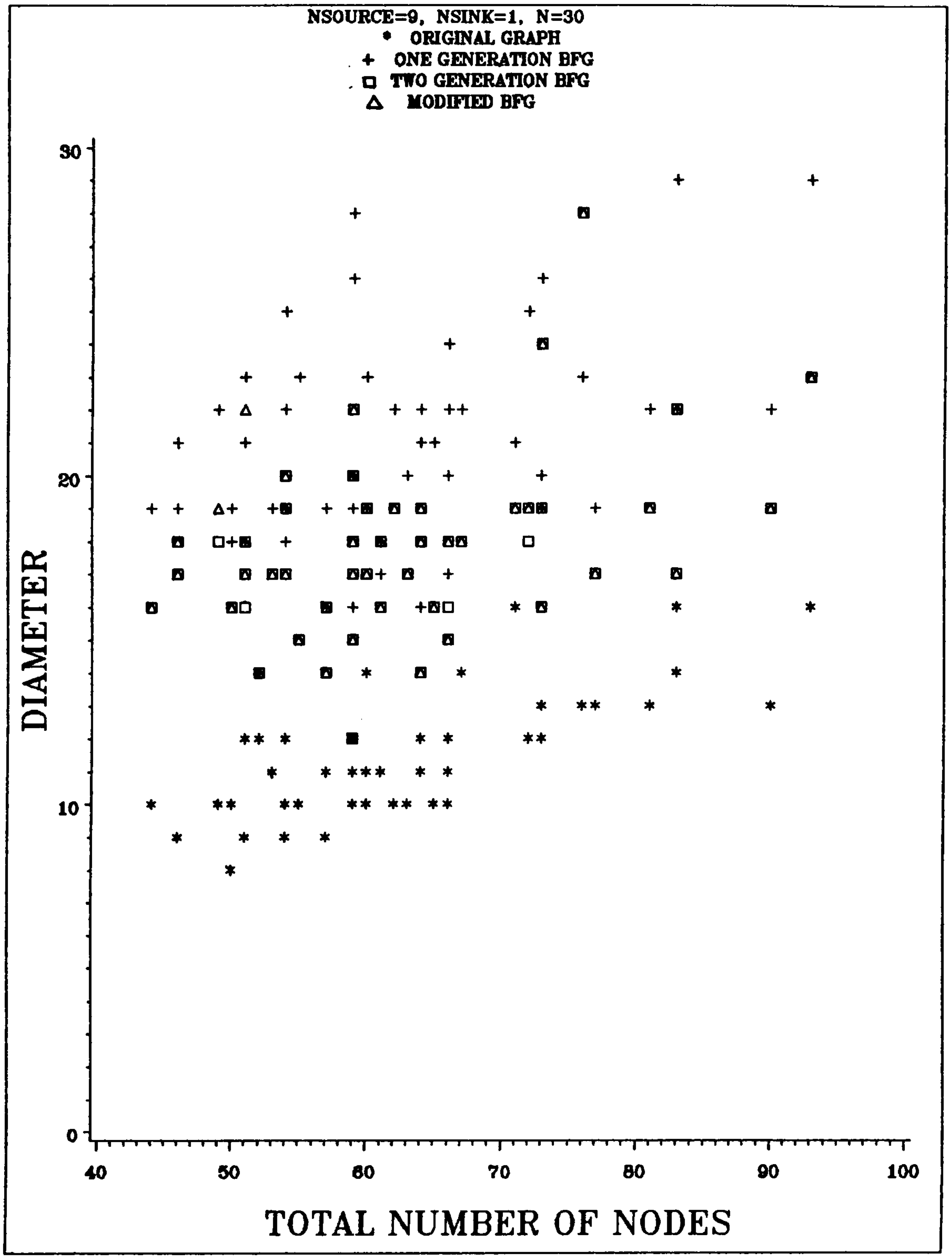


Figure 45. Diameter verses number of nodes for the three heuristics.

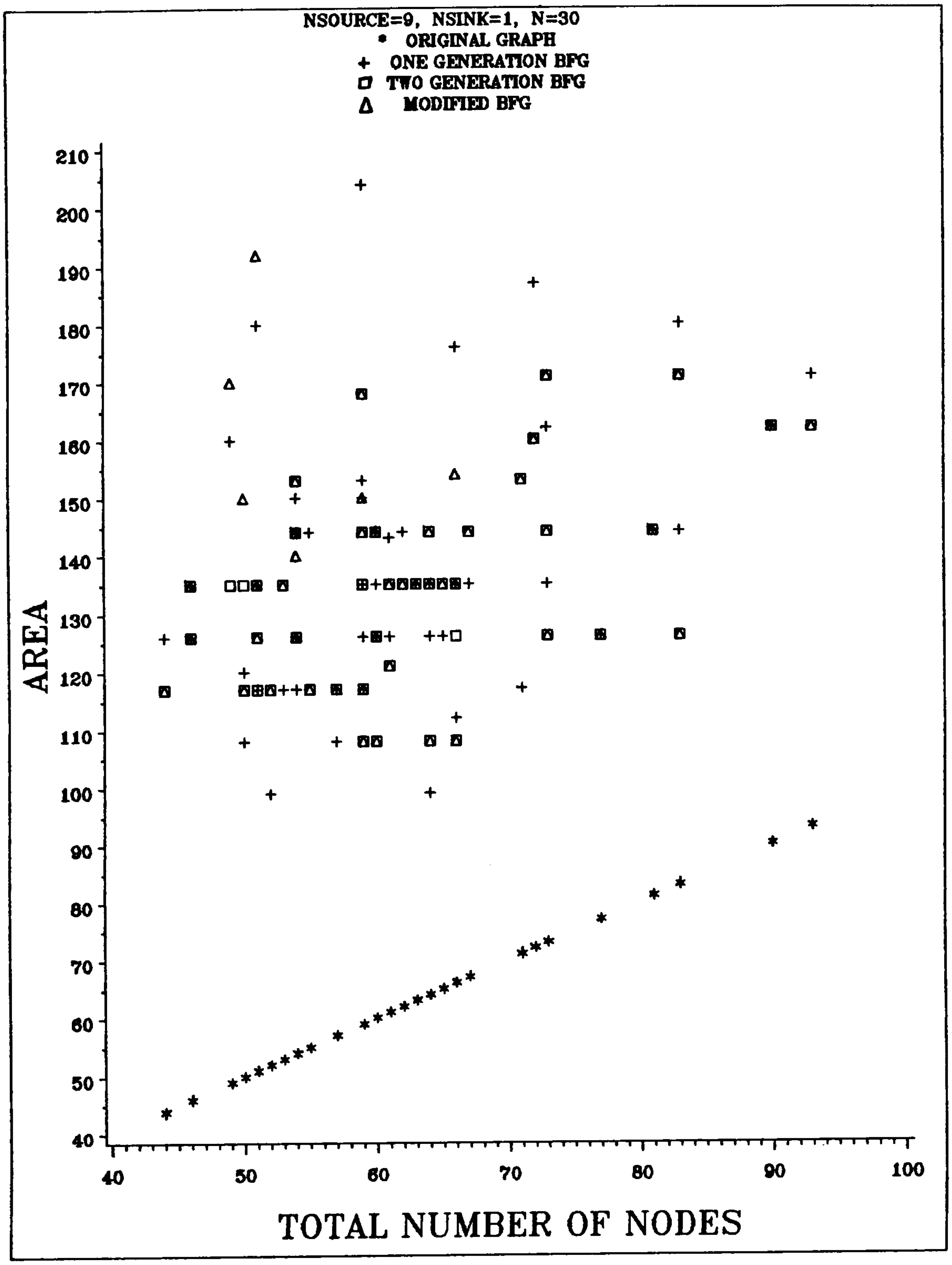


Figure 46. Area verses number of nodes for the three heuristics.

4.8 Conclusions

In this chapter, embedding acyclic digraphs in nearest-neighbor array have been considered. Three heuristic procedures have been developed and tested. The results show that the two generation BFG and modified BFG perform better than the one generation BFG for a large class of graphs.

Chapter 5: Conclusions and future work

In this research a new architecture has been proposed. It utilizes both functional parallelism and data pipelining to achieve high speed computation. Although the objective of the research was a design for real-time image processing, the proposed architecture is suitable for any application which involves large data sets. Vector processing is an example of such applications. This architecture has the following advantages for real-time image processing:

1. It can be directly interfaced with the sensor data without extensive buffering or data formatting.

2. The throughput of the system can be as high as one instruction cycle per data period in a RISC processor

3. The graph is embedded statically, therefore there is no runtime overhead.

4. The architecture does not impose any penalty for processing large images since the throughput of the array is independent of the size of the image.

5. The architecture is suitable for Wafer Scale Integration (WSI) and is highly fault tolerant.

Future Work

Beside designing the array, the following potential issues can be pursued:

1. Consider separation nodes and cyclic graphs in the embedding procedure.

2. Improve the heuristic procedures developed in this research. In the author's opinion, the next step may be to develop an iterative heuristic which starts with the solution given here and try to reduce the area.

3. Study the tradeoff between the complexity of the PE's (hence their area) and the area needed to embed the graph. In this work, PE's were selected to be as simple as possible. It turned out that the area is large compared to the graph order. Arrays with more complex PEs (those which can connect and compute at the same time) are expected to embed the graphs with less area.

References

1. P. Narendra, " VLSI architecture for real-time image processing," Proceedings of Compcon 81, Feb. 1981, pp. 303-307.

2. J. I. Potter, " Image processing on massively parallel processor," Computer, Jan. 1983, pp. 62-67.

3. A. Rosenfeld and A. Kak, Digital Picture Processing, Academic Press, 1976.

4. R. Chellappa and A. Sawchuk, Digital Image Processing and Analysis: Volume 1: Digital Image Processing, IEEE Computer Society Press, 1985.

5. W. K. Pratt, Digital Image Processing, New York: Wiley-interscience, 1978.

6. Haralick, R. M., "Some neighborhood operators, " pp. 11-35, in Real-Time/ Parallel Computing, by Onoe, Preston and Rosenfeld, Plenum Press, 1981.

7. P. Kogge, The Architecture of Pipelined Computers, McGraw-Hill, 1981.

8. D. Siewiorek, C. Bell and A. Newell, Computer Structures: principles and Examples, McGraw-Hill, 1982.

9. Systolic arrays, special issue in Computer, July 1987.

10. R. Offen, VLSI Image Processing, Collins professional and technical books, 1985.

11. T. Pavlidis, Algorithms for Graphics and Image Processing, Computer science press, 1982.

12. J. Kittler and M. Duff, Image Processing System Architecture, Research Studies Press, 1985.

13. A. Reeves, "Survey: parallel computer architecture for image processing," Computer Vision, Graphics and Image Processing 25, 1984.

14. K. Doshi and P. Varman, "Modular systolic architecture for image convolutions," Proc. 14 th Annual Symposium on Computer Architecture 3, June 1987, pp. 56-6.

15. M. Flynn, "Very high speed computer systems," Proc IEEE Vol. 54, Dec. 1966, pp. 1901-1909

16. D. Ballard, C. Brown, Computer Vision, Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

17. K. Hwang and Z. Xu, "Multipipeline networking for compound vector processing," IEEE trans. computers, vol. C-37, 1988, pp. 33-47.

18. G. Lipovski and M. Malek, Parallel Computing: Theory and Comparisons, John Wiley and Sons, 1987.

19. W. Hillis, The Connection Machine, MIT press, 1985.

20. H. T. Kung, "Why systolic architecture?," Computer, Jan 1982, pp. 37-46.

21. R. M. Lougheed, " A high recirculating neighborhood processing architecture," Proc. SPIE, vol. 534, pp22-33.

22. R. M. Lougheed, D. L. McCubbrey, " The CytoComputer: A practical pipelined image processor," Proc. of the 7th Annual International Symposium on Computer Architecture, 1981, pp 271-277.

23. K. E. Batcher, " Architecture of a massively parallel Processor ," Proceedings of the 7th Annual International Symposium on Computer Architecture, 1981, pp. 168-173.

24. A. L. Fisher, H. T. Kung, L. M. Monie and H. Walker," Design of the PSC: A programmable systolic chip," Proceeding of the 3rd Caltech Conference on VLSI, California Institute Of Technology, March 1983, pp. 287-302.

25. I. Koren, " A reconfigurable and fault-tolerant VLSI multiprocessor array," Proceeding of the 8th. Symp. Computer Architecture, 1981, pp425-442.

26. D. Gajiski and J. Peir, "Essential issues in multiprocessor systems," IEEE Computer,vol. 18 ,June 1985, pp. 9-28.

27. P. Patton, "Multiprocessors: Architecture and applications," IEEE Computer, vol. 18, June 1985, pp. 29-42.

28. K. Hwang, "Multiprocessor supercomputers for scientific/engineering applications," IEEE Computer, vol. 18, June 1985, pp. 57-75.

29. J. Dennis, "Data flow supercomputers," IEEE Computer, vol. 13, Nov. 1980, pp48-56.

30. K. Kavi, B. Buckles and U. Bhat, " A formal definition of data flow graph Models," IEEE Trans. Computer, Nov. 1986, pp. 940-947.

31. I. Watson and J.Gurd, " A practical data flow computer," IEEE Computer, vol. 15, no. 2, Feb. 1982, pp. 51-57.

32. L. Johnsson and D. Cohen, "A mathematical approach to modeling the flow of data and control in computational networks," In H. Kung, B. Sproul and G. Steel, "VLSI systems and computations," pp. 213-225, Computer Science Press, 1981.

33. L. Snyder, "Introduction to the configurable highly parallel Computer," IEEE Computer, Jan. 1982, pp. 47-56.

34. C. Mead and L.Conway , Introduction To VLSI Systems, Addison-Wesley, 1980.

35. R. Negrini, M. Sami, R. Stefanelli, "Fault tolerance techniques for array structures used in supercomputing," Computer, Feb. 1986, pp78-87.

36. S. Levialdi, Integrated Technology for Parallel Image Processing, Academic Press, 1985.

37. P. Schnck, D.Austin, S. Squires, J. Lehmann, D. Mizell and K. Wallgren,"Parallel processor programs in the federal goverment," IEEE Computer, June 1985, pp. 43-56.

38. S. Bokhari, "On the mapping problem," IEEE Trans. on Computer, vol. C-30, March 1981, pp. 207- 214.

39. J. Gaudiot, R. Vedder, G. Tucker, D. Finn and M. Campell," A distributed VLSI architecture for efficient signal and data processing," IEEE Trans. on Computer, vol. C-34, Dec. 1985, pp. 1072-1087.

40. M.Behzad, G.Chartland and L. Lesniak-Foster, Graphs & Digraphs, Wadsworth International Group, California, 1981.

41. D. Robinson and L. Foulds, Digraphs: Theory and Techniques, Gordon and Breach Science Publisher, 1980.

42. S. Goodman and Hedetniemi, Introduction to the Design and Analysis of Algorithms, McGraw-Hill, 1977.

43. D.Gordon, "Efficient embedding of binary trees in VLSI arrays," IEEE Trans. on Computer, vol. C-36, Sep. 1987, pp. 1009-1018.

44. M.Chughtai,"Complete binary spanning trees of eight neighbor array," IEEE Trans. on Computer, vol. C-34, June 1985, pp. 547-549.

45. R. Bert and T. Kung, "On the area of binary tree layout," Inform. Proc. Lett., vol. 11, pp. 46-48, 1980.

46. L. Valiant, "Universality consideration in VLSI circuits," IEEE Trans. on Computer, vol. C-30, Feb. 1981, pp. 135-140.

47. J.D. Ullman, Computational Aspects of VLSI, Rockville, Md.: Computer Science Press, 1984.

48. M. Gary and D. S. Johnson, Computers and Intractability : A guide to the theory of NP completeness, Freeman, San Francisco, 1979.

49. D. Gordon, I.Koren and G. Silberman, "Embedding tree structure in VLSI hexagonal array," IEEE Trans. Computer, vol. C-33, pp. 104-107, 1984.

50. J. Storer, "On minimal-node-cost planar embedding," Networks, vol. 14, 1984, pp. 181-212.

51. W. Ruzzo and L. Snyder, "Minimum edge length of planar embedding of trees," in VLSI Systems and Computations, H. Kung, R. Sproul and G. Steele, Eds. Rockville, MD: Computer Science Press, 1981, pp. 119-123.

52. D. Patterson and C. Sequin, " A VLSI RISC," IEEE Computer, Sep. 1982, pp. 8-21.

53. H. Jagadish, R. Mathews, T. Kailath, and J. Newkirk, "A study of pipelining in computer arrays," IEEE Trans., vol. C-35, no. 5, May 1896, pp. 431-439.

Appendix A. Some Definitions in Graph Theory

A ***Graph*** G = < V,E > is a finite, nonemty set V together with a set E (disjoint from V). Each element in V is called a ***vertex*** while each element of E is called an ***edge***. Each edge is a two-element subset of elements of V.

A graph G = < V,E > is ***Bipartite*** if it is possible to partition V into two disjoint sets V_1, V_2 such that each edge in E joins two vertices from different sets.

The ***degree of a vertex v (deg v)*** in a graph G is the number of edges incident with v. The ***degree of the graph*** is the maximum degree of its node. A graph G is called ***r-regular graph*** if for each vertex v in G, $v = r$.

A ***walk W*** from v_i to v_j in a graph G is a finite alternating sequence: W: v_i, (v_i, v_k), v_k,...., v_l, (v_l, v_j), v_j of vertices and edges beginning with vertex v_i and ending with vertex v_j. A $v_i - v_j$ ***path*** is a walk whose vertices are distinct.

A graph is ***connected*** if their exist a path connecting every two of its vertices.

A ***Tree*** is a connected graph that contains no cycles.

A ***directed graph*** or ***Digraph*** D = < V,E > is a finite, nonempty set of vertices V together with a set E of ***arcs***. Each arc is an ordered pair of distinct elements of V.

The ***indegree(v)*** of a vertex *v*in a digraph D is the number of arcs incident to the vertex *v*. The ***outdegree(v)*** of a vertex *v* in a digraph D is the number of incident arcs from *v*. The ***degree*** of a vertex *v* in a digraph D is the sum of its outdegree and its indegree. A ***sink*** in a digraph D is a vertex with zero outdegree. A ***source*** in a digraph D is a vertex with zero indegree.

A digraph D is called ***r-regular digraph*** if $outdegree(v) = indegree(v) = r$ for every vertex v in D. If $u, v \in V$ and $(u,v) \in E$, in a digraph D, then u is said to be a ***predecessor*** of v , and v is said to be the successor of u.

The ***order*** of the graph G (or digraph D) is the cardinality of the set V. The ***Size*** of the graph G (or digraph D)is the cardinality of the set E.

A ***walk W*** from v_i to v_j in a digraph D is a finite alternating sequence: $W: v_i, (v_i, v_k), v_k, \ldots, v_l, (v_l, v_j), v_j$ of vertices and arcs beginning with vertex v_i and ending with vertex v_j. A $v_i - v_j$ ***path*** is a walk whose vertices are distinct. A ***cycle*** is a path whose endpoints are the same.Two paths are called ***edge-disjoint paths*** in a digraph D if they have no arcs in common.

A vertex v is said to be ***reachable*** from the vertex u in a digraph D if their exist a walk (equivalently a path) from u to v in D. A digraph D is called ***unilaterally connected*** if for every two distinct vertices of D, at least one of them is reachable from the other. The ***outreach set OR(v)*** and the ***inreach set IR(v)*** of vertex v in digraph D are the sets:

$$OR(v) = \{u\colon u \textit{ is reachable from } v\}$$

$$IR(v) = \{w\colon v \textit{ is reachable from } w\}$$

A ***Diameter*** of a digraph D is the length of the longest path in D.

The ***level(v)*** of a vertex v in D is the length of the longest path in D ending at *v*.

A ***Directed Acyclic Graph (DAG)*** is a directed graph which contains no cycles.

Appendix B. Pseudo-Random Graph Generator.

In this appendix an algorithm to randomly generate an acyclic digraph is described. Although the given algorithm guarantees that there is no isolated nodes, it does not guarantee that the graph is unilaterally connected.

Definition B.1:A ***logical numbering*** of a digraph $D = < V_d E_d >$ of order n, assigns a unique integer value $1 \leq L(v) \leq n$ with each node $v \in V_d$ such that if $(a,b) \in E_d$ then $L(a) < L(b)$.

Definition B.2: The adjacency matrix (or vertex-to-vertex incidence matrix) of a digraph $D = < V_d E_d >$ of order n , is an nxn matrix T such that:

$$T(i,j) = \begin{matrix} 1 & if\,(v_i, v_j) \in E_d \\ 0 & \text{otherwise} \end{matrix}$$

If the nodes of an acyclic digraph D are logically numbered, then the rows and the columns can be interchanged so that the adjacency matrix T will be as shown in Figure 47, where N_s, N_k, N_i , represent the number of source, sink and intermediate nodes, respectively. The zero entries in the shown matrix are necessary to have an acyclic digraph, while the x entries can be replaced with 1's and 0's such that:

1. $1 \le \sum_{i=1}^{n} T(i,j) \le k2$

2. $1 \le \sum_{j=1}^{n} T(i,j) \le k1$

Where K_1, K_2 are The desired indegree and outdegree for the generated graph.

The following algorithm generates an acyclic digraph given the number of source, sink and intermediate nodes. Only the non-zero $(N_s + N_i)$ by $(N_i + N_k)$ matrix will be generated and stored. The algorithm uses a pseudo-random function, $Random(x)$, which selects a positive integer less or equal to x.

Algorithm B.1: Each entry $T(i,j)$ will be denoted with one of the following: "*Connection*" if there is an arc between nodes i and j. "*Noconnection*" if it is impossible to establish an arc which connects them since one of the nodes has reached its maximum degree. "*Undecided*" if there is no connection yet but could be possibly established in the future, and "*Forbidden*" if it is impossible to connect the two nodes, i.e these entries which are filled with zeros in Figure 47

1. Input: N_s, N_k, N_i.
2. Initialize: $N_r = N_s + N_i$, $N_c = N_i + N_k$. Let T be $N_r x N_c$ matrix. Set

   $T(i,j) = Forbidden$ for $i > N_s$ and $j \le (i - N_s)$, otherwise $T(i,j) = Undecided.$

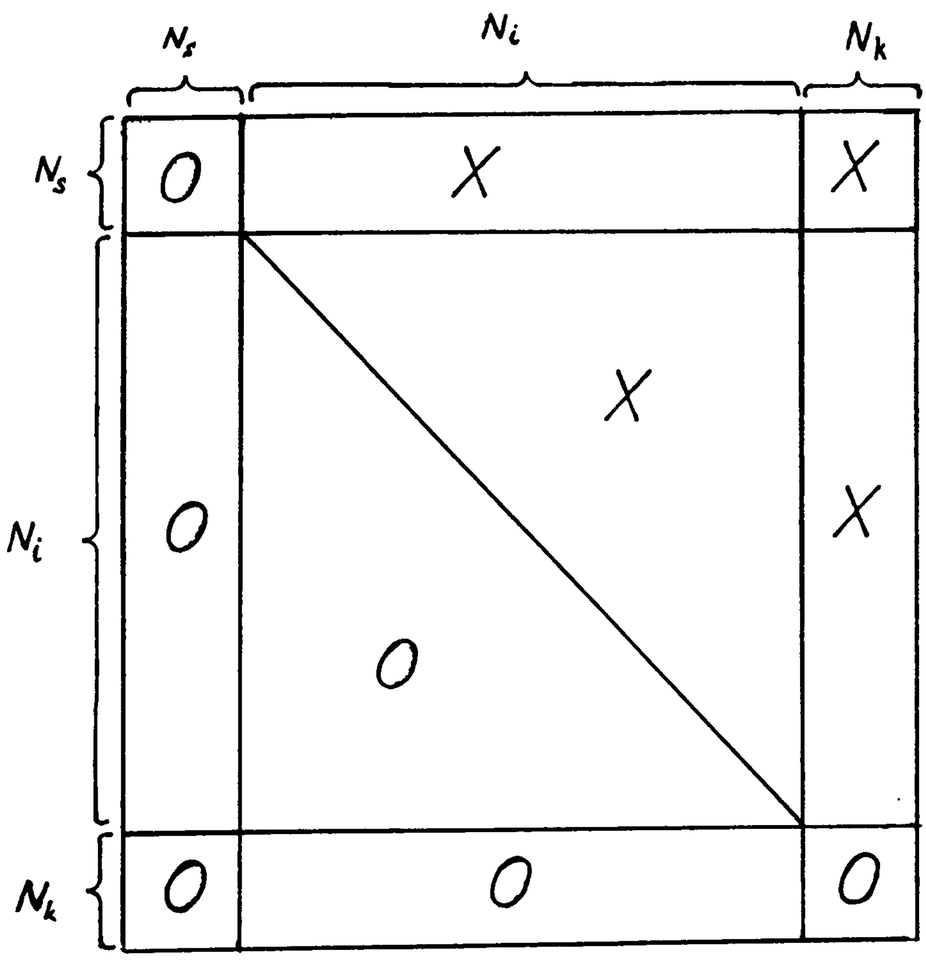


Figure 47. The adjacency matrix for an acyclic digraph.

3. Randomly select to fill the matrix top-down or bottom-up. only the upper triangular matrix can be filled.
4. For each row R, find all columns C which satisfy $T(R,C) = Undecided$, suppose that their is N_c of these columns.
5. If $N_c > 0$ then
   a. Od = Minimum (N_c , Random(k sub 2)) {set output degree}
   b. Randomly select Od columns from the available N_c columns. For each selected column C, Do
      1) Set $T(R,C) = Connection$
      2) If the number of connections in column C is k_1, then change every $T(i,C) = Undecided$, to $T(i,C) = Noconnection$ for $1 \leq i \leq N_r$

   Else

   a. Od = 1;
   b. for each column C with $T(R,C) = Undecided$, find every rows i, such that $T(i,C) = Connection$ and the number of 1's in row i is more than one.
   c. Randomly select one position (i,C) from the available ones. set $T(R,C) = Connection$ and $T(i,C) = Noconnection$

Appendix C. Program Listings

```
{
 Embeddig an Acyclic Graph in an Eight-Connected Processor Array


 Langauge: Turbo Pascal
 System: IBM PC.

 Written by: Faridah Ali
 Last update: March 1988.

}
CONST
    {grid size 100*100}
    miny = -50;
    maxy = 50;
    minx = 0;
    maxx = 100;

    maxnodes = 1000;        { maximum no of graph nodes to be embedded.}
    nparallelNodes= 150;  { maximum no of parallel nodes.}
    Maxdiam=50;
    maxsucc = 5;          { maximum no of successor PE's for each PE
                        in 8-connected system.}
    blank = ' ';

    { node types}
    connector='C';
    source='S';
    sink='K';
    Operator='O';


    Unary = 1;
    Binary = 2;


TYPE
    { doubly linked list }
    DblLnk = array [1 .. maxnodes] of record
                  ntype : char;  {source, sink, operator or connector.}
                  nchildren ,
                  ch1,
                  ch2,
                  nparents,
                  par1,
                  par2: Integer;
                  Visit:boolean;  { a flag to identify whether the
                               node has been embedded or not.}
                  X,Y:integer;    { location of the embedded node.}
                  Level:integer;
```

```
                End;

     movedir = (up, down, anydir);

     PointStatus = (Empty, Full, HalfFull);

     Grids = array [minx .. maxx, miny .. maxy] of PointStatus;
     neighbors = array [1.. maxsucc] of Record
                          X,Y:Integer;
                          END;

     CutPoints = array [miny .. maxy] of PointStatus;

     SearchTable = array [1 .. nparallelnodes] of record
               Inode:integer;      { node ID }
               Nloc :integer;      { number of possible locations }
               loc  : neighbors;   { x,y coordinates of the locations}
               end;

     IndexArray = array [1.. nparallelNodes] Of Integer;

     Connectors = array [1.. nparallelNodes] of Record
               Par, Child, Connector:Integer;
               End;
     Edges= array [1.. maxNodes, 1..4] Of Integer;
     Mates= array [1..nparallelnodes] of record
            Nmate: Integer;  {no. of sibling of the node}
            S: array[1..2] of Integer;  {sibling indices in the}
            End;                        {search table.}


VAR
     Graph: DblLnk;
     SourceNodes:IndexArray;

     Nsource,NactiveEdges,
     Ngraphnodes,NembedNodes,
     MaxId,PreMaxId: integer;  {maximum integer values used to identify}
                         { nodes of the grpah }

     Nsync,GrphDiameter,Np:Integer;

     Grid : grids;
     RecentEdges:Edges;

     TopRow,BottomRow,
     AreaWidth,Areaheight: integer;  { Current Area used for embedding.}


     Nprespar,
     NFirstChildren,
    Npreschild,
    NpresConnector: Integer;

    Prespar,Preschild: SearchTable;
    PresConnector: Connectors;
    ParIndex,ChildIndex: IndexArray;
    ParCut,ChildCut: CutPoints;
    Parmate,ChildMate: Mates;


    ParID,ParID1,ParID2,ChildID,ConnectID:Integer;

    GraphFile,Outputfile: Text;
    FileName: String[14];
    GraphId: String[10];
    Option:char;


    {dummy variables}
    I,k,J,JJ,KK,XX,YY,Iforward,MW: integer;
    cc:char;
    IA:IndexArray;

    { Flags }
    Found,DlySuccd,
    ParLastSol,          { no more solutions exist foe embedding the parents}
    ParEmbed,ChildEmbed,    { Parents/Children have been successfully embedded }
    LocAvailable:Boolean;
{----------------------------------------------------------------------}
{$R+}
Procedure SetFiles;

Begin

  writeln; Write ('  GraphID: ');
  Readln (GraphId);
  FileName:=Concat(GraphId,'.Inp');
  Assign (GraphFile,FileName);


  FileName:=Concat(graphId,'.OUT');
  Assign(OutputFile,FileName);
  ReWrite(OutputFile);

End;{setFIles}
{----------------------------------------------------------------------}

PROCEDURE INC (var I1: integer);
Begin
     I1:=I1+1;
End;
```

```
{---------------------------------------------------------------------}

Procedure CopyPoints(N:Integer; Source:Neighbors; Var Destination: Neighbors);

{ copies a set of points from one array to another }
Var I:Integer;

Begin
   For I:=1 TO N DO
      Begin
        Destination[I].X:=Source[I].X;
        Destination[I].Y:=Source[I].Y;
      End;
End;
{---------------------------------------------------------------------}

Procedure DefineConnector (ParID, ChildID: Integer; VAR ConnectorId: Integer);
{ creates a new connector node }

Begin
  INC (NpresConnector);
  INC (MaxID);
  ConnectorId:=MaxId;
  PresConnector[Npresconnector].par:=ParID;
  PresConnector[Npresconnector].Child:=ChildID;
  PresConnector[NpresConnector].Connector:=ConnectorID;
End;

{---------------------------------------------------------------------}

Procedure Insertnode (ParId,ChildId,NodeId: Integer;NodeType: Char);
{ Inserts a node between the parent and the child nodes in the Graph.
 A zero value for ParId or ChildId indicates that the new node will
 proceed the child or follow the parent, respectively.}

Begin

 Graph[NodeID].nType:=NodeType;
 Graph[NodeID].Visit:=False;
 Graph[NodeId].Par1:=0;
 Graph[NodeID].Ch1:=0;
 Graph[NodeId].Par2:=0;
 Graph[NodeID].Ch2:=0;

 If ParId <>0 Then
            Begin
              Graph[NodeID].nParents:=1;
              Graph[NodeID].Par1:=ParID;
            End
         Else Begin
```

```
                 Graph[NodeID].nParents: = 0;
                 Graph[ChildID].nparents: = 1;
                End;
 If ChildId  < > 0 Then
               Begin
                 Graph[NodeId].Nchildren: = 1;
                 Graph[NodeId].Ch1: = ChildId;
               End
           Else
              Begin
                Graph[NodeId].Nchildren: = 0;
                Graph[ParID].nchildren: = 1;
              End;

 If ParID  < > 0 Then
 { connect parent to the new node}
 If (Graph[ParID].Ch1 = ChildID) OR(ChildId = 0)
    Then Graph[ParID].Ch1: = NodeID
    Else Graph[ParID].Ch2: = NodeID;

 If ChildId < > 0 Then
 { connect the new node to the child node}
 IF (Graph[ChildID].Par1 = ParID) OR (ParID = 0)
    Then Graph[ChildID].Par1: = NodeID
    Else Graph[ChildID].Par2: = NodeID;
    {Writeln; writeln(' Connect Nodes ', ChildID:6,' , ',ParID:6,
       'by bode ',NodeID);}
End; {InsertNode}
{------------------------------------------------------------------------}
Procedure DeleteConnector (ParID,ChildId,ConnectID: Integer);

Begin
  Graph[ConnectId].visit: = False;
  Graph[ConnectId].par1: = 0;
  Graph[ConnectId].Ch1: = 0;

  {delete the link between the parent and the connector}
   If ParID  < > 0 Then
      If Graph[ParID].ch1 = ConnectID
         Then Graph[ParId].ch1: = ChildID
         Else Graph[ParID].ch2: = ChildID
      Else Graph[ChildId].nparents: = 0;

   If ChildID  < > 0 Then
      If Graph[ChildID].par1 = ConnectID
          Then Graph[ChildID].par1: = ParID
          Else Graph[ChildID].par2: = ParID
      Else Graph[PArId].nchildren: = 0;
```

```
End; {deleteConnector}
{---------------------------------------------------------------------------}
Procedure CountSyncNodes(Var Nsync,Diameter,Np: Integer);

{ Nsync: no. of nodes needed to synchronize the graph}
{ Diameter: graph diameter}
{ Np: maximum number of nodes at any level}

Type Nparray =array[1..maxdiam] of Integer;
Var  Level1,Level2,MaxLev12,MaxLevel,I: Integer;
     MoreNodes:Boolean;
     NperLev:Nparray;

Begin

  Nsync:=0;
  Np:=0;
  MaxLevel:=0;
  MoreNodes:=True;
  For I:=1 to Maxdiam Do NperLev[I]:=0;
  { a level value of zero indicates that the node has not been considered yet}
  For I:=1 To MaxId Do Graph[I].level:=0;
  For I:=1 to Nsource Do Graph[SourceNodes[I]].level:=1;

  While MoreNodes Do
  Begin
   MoreNodes:=False;
   For I:=1 To MaxID Do
     If Graph[I].level =0 Then
     Begin
       Case Graph[I].nparents of
       Unary:
               If Graph[Graph[I].Par1].level <>0 Then
                  Begin
                    Graph[I].level:=Graph[Graph[I].par1].level +1;
                    Inc(Nperlev[Graph[I].Level]);
                  End;
       Binary: Begin
                 Level1:=Graph[Graph[I].par1].level;
                 Level2:=Graph[Graph[I].par2].level;

                 If (Level1 <>0) And ( Level2<>0) Then
                 Begin
                      { count the no. of synchronization nodes needed. }
                      Nsync:=Nsync+Abs(Level1-level2);
                      MaxLev12:=Level1;
                      If Level2>MaxLev12 Then MaxLev12:=Level2;
                      Graph[I].level:=1+MaxLev12;
                      Inc(Nperlev[Graph[I].Level]);

                 End;
```

```
         End;{Binary}

      End;{case}
      MoreNodes:=True;

      IF Graph[I].Level> MaxLevel Then MaxLevel:=Graph[I].Level;
   End;
  End;
 Diameter:=MaxLevel-1;
 Np:=NperLev[1];
 For I:=2 to Diameter Do  If NperLev[I]> Np Then Np:=NperLev[I];

 writeln; write(' Nsync= ',Nsync,'  Diameter=',Diameter,
            'Np ',Np);
End; {CountSync}
{-------------------------------------------------------------------}

Procedure Intersect ( Np1,Np2: Integer;Points1,Points2: Neighbors;
                 Var Np: Integer;  Var Points: Neighbors);

{ The procedure finds the set of points 'Points', which is the intersection }
{ between the two sets: 'Points1' and 'Points2'. Np1, Np2, and Np are the  }
{ cardinality of the sets Points1, Ponts2, And Points respectively.        }
{                                                                 }
Var I,J:integer;
Begin
     Np:=0;
     For I:=1 to Np1 Do
        For J:=1 To Np2 DO
         If (Points1[I].x = Points2[J].x) AND (Points1[I].y=Points2[J].y)
             Then  Begin
                      Np:=Np+1;
                      Points[Np].x:=Points1[I].x ;
                      Points[Np].y:=Points1[I].y;
                   End;
      { writeln;
        write (' Procedure Intersect Np=',Np)  ;
        For j:=1 to Np Do Write(' (',Points[j].x,',',Points[J].Y,')'); }
End;

{----------------------------------------------------------------------}
Procedure OrderIndices (X1,Y1,X2,Y2: Integer; Var XO1,YO1,XO2,YO2: Integer);
Begin
  XO1:=X1;  YO1:=Y1;   XO2:=X2;   YO2:=Y2;
  IF (XO1>XO2) OR ( (XO1=XO2) AND (YO1>YO2) )
      Then  Begin   XO1:=X2;  YO1:=Y2;  XO2:=X1;  YO2:=Y1;  End
End;

{----------------------------------------------------------------------}
Procedure ReservEdge (X1,Y1,X2,Y2: Integer);
```

```
Var XO1,YO1,XO2,YO2: Integer;
Begin
  {writeln(' ResvEdge ',X1,Y1,X2,Y2);  }
  OrderIndices (X1,Y1,X2,Y2, XO1,YO1,XO2,YO2);
  INC(NactiveEdges);
  RecentEdges[NactiveEdges,1]:= XO1;
  RecentEdges[NactiveEdges,2]:= YO1;
  RecentEdges[NactiveEdges,3]:= XO2;
  RecentEdges[NactiveEdges,4]:= YO2;
End;

{----------------------------------------------------------------------}
Procedure TestEdge (X1,Y1,X2,Y2: Integer; Var Found:Boolean; Var Loc: Integer);

Var XO1,YO1,XO2,YO2: Integer;
Begin
  OrderIndices (X1,Y1,X2,Y2, XO1,YO1,XO2,YO2);
  Loc:= 1;   Found:= False;
  While (Loc < = NactiveEdges ) AND Not(Found) DO

    IF  (RecentEdges[Loc,1]= XO1) And
       (RecentEdges[Loc,2]= YO1) And
       (RecentEdges[Loc,3]= XO2) And
       (RecentEdges[Loc,4]= YO2)
     Then Found:= True
     Else Loc:= Loc+ 1;

End;
{----------------------------------------------------------------------}
Procedure DislocateEdge(X1,Y1,X2,Y2: Integer);

Var Loc,I: Integer; Found:Boolean;

Begin
 { writeln(' DisEdge ',X1,Y1,X2,Y2);  }
  TestEdge(X1,Y1,X2,Y2,Found,Loc);
  IF Not(Found) Then
      Begin  writeln( '**** Error in DislocateEdges **'); Readln; End;
  For I:= loc to NactiveEdges DO
    Begin
        RecentEdges[I,1]:= RecentEdges[I + 1,1];
        RecentEdges[I,2]:= RecentEdges[I + 1,2];
        RecentEdges[I,3]:= RecentEdges[I + 1,3];
        RecentEdges[I,4]:= RecentEdges[I + 1,4];
    End;

  NactiveEdges:= NactiveEdges-1;
End;
{----------------------------------------------------------------------}
Procedure DelOldEdges;
```

```
Var I,J: Integer;
Begin
 I:=1;
 While (RecentEdges[I,1] < (AreaWidth-2)) AND
      (I < NactiveEdges) Do  I:=I+1;
 NactiveEdges:=NactiveEdges-I+1;
 For J:=1 To NactiveEdges Do
   Begin
       RecentEdges[J,1]:=RecentEdges[J+I-1,1];
       RecentEdges[J,2]:=RecentEdges[J+I-1,2];
       RecentEdges[J,3]:=RecentEdges[J+I-1,3];
       RecentEdges[J,4]:=RecentEdges[J+I-1,4];
   End;

End;
{-----------------------------------------------------------------------}
Procedure possiblemoves ( Chld, X,Y : Integer ; Nodedir: Movedir ;
                        VAR NP: Integer; Var Points: Neighbors);


{ (X,y)  : location of a parent node.
  nodedir: the desired direction of the travel between the parent and the
          child, if any.
  NP     : no of possible locations to embed the child node.
  Points : the possible locations to embed the child.

  the procedure finds all locations, which are one-edge distant from the
  point (x,y) on the grid, and can be used to to embed the child node. }



TYPE
     ForwardDir = ( N, NE, E, SE, S );

VAR
     X1,Y1: Integer;


Procedure Insert ( Neighbordir: ForwardDir);
Begin
     Case Neighbordir of

          N   : Begin   X1:=X      ; Y1:=Y+1     ; end;

          NE  : Begin   X1:=X+1    ; Y1:=Y+1     ; end;

          E   : Begin   X1:=X+1    ; Y1:=Y       ; end;

          SE  : Begin   X1:=X+1    ; Y1:=Y-1     ; end;

          S   : Begin   X1:=X      ; Y1:=Y-1     ; end;
```

```
        END;

    If (Grid [x1, y1] = Empty) OR
      ((Grid[x1,y1]= HalfFull) AND (Graph[ChID].ntype= connector))
      Then
          Begin
            NP:= NP + 1;
            Points[Np].x := X1;
            Points[Np].y := Y1;
          End;

END; {procedure Insert}

Begin
   NP:= 0;
   IF X < AreaWidth
        Then    Case NodeDir of
                Up   : Begin Insert(NE); Insert(E);  End;
                Down : Begin Insert(SE); Insert(E);  End;
                AnyDir: Begin
                          Insert(E);
                          If Y < TopRow Then Insert(NE);
                          If Y > BottomRow Then Insert(SE);
                      End;
                End
        Else
                Case NodeDir of
                Up   : Begin Insert(N);  Insert(NE); Insert(E); End;
                Down : Begin Insert(S);  Insert(SE); Insert(E); End;
                AnyDir: Begin
                      If Y < TopRow Then Insert(N);
                      If Y > BottomRow Then Insert(S);
                      Insert(E);
                      If Y < TopRow Then Insert(NE);
                      If Y > BottomRow Then Insert(SE);
                      End;
                End;
{   If NP=0 Then Writeln (' No vacant location around (',x,',',Y,')');
writeln;
write(' possiblemoves forchildren of (',x,',',y,') Direction ');
For i:= 1 to Np DO write('(',Points[I].x,',',Points[i].y,') '); }

 End; {possiblemoves}
{---------------------------------------------------------------------}

Procedure SourceOutreach (Var Outreach: IndexArray);

{ From the embedded graph, so far, this procedure Counts the number of
 outreached node for each source node. }
```

```
Type IndexArray = Array [1.. 900] Of Integer;
Var Moreoutreach:boolean;
   P,C:IndexArray;
   I,J,K,Np,Nc:integer;

Begin
   For I:= 1 to Nsource Do
   Begin
    Np:= 1;   MoreOutreach:= True;
    P[1]:= SourceNodes[I]; Outreach[I]:= 0;

    While MoreOutreach Do
    Begin
      Nc:= 0; MoreOutreach:= False;
      For J:= 1 to Np DO
          If ( Graph[P[J]].visit) And
            ( Graph[P[J]].nchildren > 0 ) Then
          Begin
           MoreOutreach:= True;
           Inc(Nc);
           C[Nc]:= Graph[P[j]].ch1;
           Inc(Outreach[I]);
           If Graph[P[J]].nchildren = 2 Then
              Begin
                 Inc(Nc);  C[Nc]:= Graph[P[J]].ch2;
                 Inc(Outreach[I]);
              End;
          End;
     For J:= 1 to Nc DO  P[J]:= C[J];
     Np:= Nc;
    End;{while}
   End;

End; {SourceOutreach}
{--------------------------------------------------------------------------}
Procedure DivideHeight(Var Nposition: IndexArray);

{ Divides the given area height among the source nodes. Each node will be
will be given, at leat, one position. If the area height is grater than
the number of the source nodes, the number of positions assigned for
each node is propotional to its outreach degree.}

Var  Extrarows,OutreachSum,NpositionSum,
     MaxoutreachSource,I,J: Integer;
     Outreach: IndexArray;

Begin

  OutreachSum:= 0;  NpositionSum:= 0;
  For I:= 1 to Nsource Do Nposition[I]:= 1;
  If (AreaHeight > Nsource) Then
  Begin
   ExtraRows: = Areaheight-Nsource;
   SourceOutreach (Outreach);

   { write (' source outreach ');
   For I: = 1 to Nsource Do write(Outreach[I]:5);}

   For I: = 1 to Nsource DO Outreachsum: = Outreachsum + outreach[I];
   For I: = 1 to Nsource Do
      Nposition[I]: = Nposition[I] + Trunc(Outreach[I]*ExtraRows/OutreachSum);

   { if there is still extra rows distribute then among the source nodes
    in descending order according to cardinality of there reachout sets}

   NpositionSum: = 0;
   For I: = 1 to Nsource DO NpositionSum: = NpositionSum + Nposition[I];
   For I: = 1 to (AreaHeight-NpositionSum) Do
      Begin
        MaxOutreachSource: = 1;
        For J: = 2 to Nsource Do
           If Outreach[J] > Outreach[MaxOutreachSource]
                 Then MaxOutreachSource: = J;
        Outreach[MaxOutreachSource]: = 0;
        Inc(Nposition[MaxOutreachSource]);
      End;


  End;
End;{DivideHeight}
{-----------------------------------------------------------------------}
Procedure EmbedSource(DesiredHeight: Integer);
{ Embeds the source nodes on the first column distributed within the area}
{ height. Also, initializes necessary variables to start a new embedding for }
{ the graph in the given area }

VAR I,K,Child: Integer;
   Nposition:IndexArray;

Begin

AreaHeight: = DesiredHeight;
TopRow: = AreaHeight Div 2;
BottomRow: = TopRow-AreaHeight + 1;
AreaWidth: = 0;

writeln;write(' @@@ a new allocation is started  with',
                  '  AreaHeight = ' ,AreaHeight);
```

```
XX:=0;     YY:=TopRow;
DivideHeight(Nposition);

For I:=MinX to MaxX Do
   For J:=MinY TO MaxY DO
        Grid[I,J]:=Empty;

{ delete any leading connectors.}
For I:=1 to Nsource Do
  If Graph[SourceNodes[I]].ntype=connector Then
    Begin
        Child:=Graph[SourceNodes[I]].Ch1;
        While Graph[Child].ntype =connector Do
                                        Child:=Graph[Child].ch1;
        SourceNodes[I]:=Child;
        Graph[Child].nparents:=0;
    End;



For I:=1 to MaxID DO
   Begin
        Graph[I].visit:=False;
        Graph[I].x:=MaxX+1;
        Graph[I].y:=MaxY+1;
        If I>Ngraphnodes Then
          DeleteConnector(graph[I].par1,graph[I].Ch1,I);
   End;

For I:=1 to Nsource DO
    Begin
      ParID:=SourceNodes[I];
      Prespar[I].Inode:=ParID;
      ParIndex[I]:=1;
      Prespar[I].Nloc:=1;
      Prespar[I].loc[1].x:=XX;
      Prespar[I].loc[1].y:=YY;

      { center the node within the assigned positions for it.}
      K:=Nposition[I] Div 2;
      If I=NpresPar Then  K:=(Nposition[I]-1) Div 2;
      YY:=YY-k;

      Graph[ParID].X:=XX;
      Graph[ParID].Y:=YY;
      Graph[ParID].Visit:=True;
      Grid[XX,YY]:=Full;

      { find the y coordinate  for the next node. }
      YY:=YY-(Nposition[I]-K);
   END;
NembedNodes:= Nsource;
MaxID:= NgraphNodes;
NpresPar:= Nsource;
NactiveEdges:= 0;
ParLastSol:= True;
END;{procedure Embedsource }

{-----------------------------------------------------------------}
Procedure SortChildren;

{ Sorts the children table in ascending order according to the number of
  possible locations for each node. }

Var Exchange:Boolean;
    Temp1,Temp2: Integer;   TempA:neighbors;

Begin
Exchange:= True;
While Exchange DO
Begin
   Exchange:= False;
   For I:= 1 to (NpresChild-1) Do
      If PresChild[I].Nloc > PresChild[I + 1].Nloc
        Then Begin  {exchange row I,I + 1}

              Temp1:= PresChild[I].Inode;
              Temp2:= PresChild[I].Nloc;
              CopyPoints(Temp2,PresChild[I].loc,TempA);

              PresChild[I].Inode:= PresChild[I + 1].Inode;
              PresChild[I].Nloc:= PresChild[I + 1].Nloc;
              CopyPoints(PresChild[I + 1].Nloc,PresChild[I + 1].loc,
                                          PresChild[I].loc);

              PresChild[I + 1].Inode:= Temp1;
              PresChild[I + 1].Nloc:= Temp2;
              CopyPoints(Temp2,TempA,PresChild[I + 1].loc);

              Exchange:= True;
              End; {Then}

END; { While}

{
Writeln;
Writeln(' Sorted Children table');
For j:= 1 to Npreschild do
 Begin
    Writeln;
    Write(PresChild[j].Inode:6);
   For k: = 1 to PresChild[j].Nloc Do
       Write('(',PresChild[j].Loc[k].x:3,',',PresChild[j].loc[k].y:3,')   ');
 End;
 }

END; {procedure SortChildren}
{---------------------------------------------------------------------------}
Procedure FindMates(Table: SearchTable; TableSize: Integer
                              ;Var NodeMate: Mates);


{ for each child in the search table, check if any of its mates exist
whithin the next entries of the tabe.}

Var Mate,Child,J,K,NodeId: Integer;
    Found:Boolean;


Begin
 For I: = 1 to TableSize Do
 Begin
   NodeId: = Table[I].Inode;
   NodeMate[I].Nmate: = 0;
   J: = 1;
   While (J < = Graph[NodeID].nchildren) Do
     Begin
       If J = Unary Then Child: = Graph[NodeID].Ch1
                 Else Child: = Graph[NodeID].Ch2;
       If Graph[Child].nparents > 1 Then
         Begin
           IF Graph[Child].par1  < > NodeID
               Then Mate: = Graph[Child].par1
               Else Mate: = Graph[Child].par2;

           K: = I + 1;   Found: = False;
           While (K < = TableSize) And Not(Found) Do
              If Table[K].Inode  < >  Mate Then K: = K + 1
                                Else Found: = True;

           If Found Then
              Begin
               Inc(NodeMate[I].Nmate);
               NodeMate[I].S[NodeMate[I].Nmate]: = K;
              End;
         End;
       J: = J + 1;
     End;
 End;{for I}
End;
{---------------------------------------------------------------------------}
Procedure MinHeight (Var MW: Integer);
```

```
{ Estimates the minimum height of the arrary processor required to embed
the current set of children nodes.}

Type ChildFlag= Array [1..NparallelNodes] Of Boolean;
Var PartnerExst:ChildFlag;
   Np: Integer;
   Points:Neighbors;

Begin
  MW:=0;
  For I:=1 to NpresChild Do   PartnerExst[I]:=False;
  For I:=1 To NpresChild Do
   If Not (PartnerExst[I]) Then
     Begin
       Inc(MW);
       ChildId:=PresChild[I].Inode;
       If Graph[ChildId].Ntype =connector Then
       Begin
       { check if another connector node can share the same grid
         location with the current node.}
         K:=I+1; Found:=False;
         While (K<= NpresChild ) And Not (Found) Do
           If (Graph[PresChild[K].Inode].Ntype <> connector) OR
              PartnerExst[k]
           Then Inc(K)
           Else Begin
                Intersect(PresChild[I].nloc,PresChild[k].nloc,
                          PresChild[I].loc,PresChild[K].loc,
                          Np,Points);
                If Np <>0 Then Begin PartnerExst[k]:=True;
                                     Found:=True;
                               End
                          Else Inc(K);
                End;
       End;
     End;

{ writeln(' NpresChild ',NpresChild, ' MinHeight ',MW);}
End;{Procedure MinHeight}
{-----------------------------------------------------------------------}
Procedure FindChildren(Var LocAvailable: Boolean; Var MW: Integer);

Var Np,Np1,Np2,temp1,temp2: Integer;
   Points,Points1,Points2:Neighbors;
Begin
{ Finds the set of children nodes which are the immediate successor of the
  present parent nodes and can be embedded at one edge distant from their parents.
  If the child node in the original graph can not be embedded next, a connector
  node is inserted between the parent and that child. The connector node is
  considered the immediate child of that parent.}
```

```
NpresChild: = 0;        Np1: = 1;       Np2: = 1;
NpresConnector: = 0;
LocAvailable: = True;

k: = 1;

While (K < = NpresPar) and (LocAvailable)
Begin
 J: = 1;  ParID: = PresPar[K].Inode;
 If Graph[ParId].nchildren = 0 Then
                    Begin
                    DefineConnector(ParId,0,ConnectId);
                    InsertNode(ParID,0,ConnectId,connector);
                    End;
 ChildID: = Graph[ParID].Ch1;

 While ((J < = Graph[ParID].Nchildren) And LOcAvailable) DO
  Begin
  If NOT (Graph[ChildID].Visit ) Then
   Case Graph[ChildID].Nparents Of

   Unary: Begin
         Graph[ChildID].Visit: = true;
         NpresChild: = NpresChild + 1;
         PresChild[NpresChild].Inode: = ChildID;

         { find all possible locations for embedding that child }
         { one edge distance from its parent node. }

         PossibleMoves(ChildID,Graph[ParID].X,Graph[ParID].Y,anydir,Np1,Points);
         PresChild[NpresChild].Nloc: = NP1;
         CopyPoints(Np1,Points,PresChild[NpresChild].loc);
         End; {Unary }

   Binary: Begin
          ParID1: = ParID;
          IF Graph[childId].Par1 = ParID1 Then   ParID2: = Graph[ChildID].Par2
                                  Else   ParID2: = Graph[ChildId].Par1;

          { Check if the other parent has already been embedded}
          JJ: = 1; Found: = False;
          While (JJ < = NpresPar) AND NOT(Found) DO
              IF PresPar[JJ].Inode = ParID2 Then Found: = True
                                    ELSE JJ: = JJ + 1;
          IF Found THEN { the other parent node has been embedded}
            Begin
            { Chech if the two Parents are close enough so that }
            { their mutual child node can be embedded one edge  }
            { distant from ech of them. }
```

```
PossibleMoves(ChildID,Graph[ParID1].x,Graph[ParID1].Y,anydir,NP1,Points1);
PossibleMoves(ChildId,Graph[ParID2].X,Graph[ParID2].Y,anydir,NP2,Points2);
Intersect(Np1,NP2,Points1,Points2,Np,Points);

If NP=0 Then { the parents are far apart and the child  }
        { node can not be Embedd at this points.}
        { Therefore, descend a connector  fron     }
        { each parent toward the other, to shorten }
        { the distance between them.}
     Begin
     { let par1 be the higher node}
     If Graph[ParID1].Y < Graph[ParID2].Y Then
                 Begin  Temp1:=ParID2;
                        ParID2:=ParID1;
                        ParID1:=Temp1;
                  ENd;
     For JJ:=1 to 2  DO
     Begin

        Case JJ of
           1: Begin
              DefineConnector(ParID1,ChildID,ConnectID);
              InsertNode(ParID1,ChildID,ConnectID,connector);
              PossibleMoves(ConnectID,Graph[ParID1].x,Graph[ParID1].y,
                        down,Np1,Points);
              End;
           2: Begin
              DefineConnector(ParID2,ChildID,ConnectID);
              InsertNode(ParID2,ChildID,ConnectID,connector);
              PossibleMoves(ConnectID,Graph[ParID2].x,Graph[ParID2].y,
                        up,Np1,Points);
              End;
           End; {Case 1,2}
        Graph[ConnectID].visit:=True;
        INC(NPresChild);
        PresChild[NpresChild].Inode:=ConnectID;
        PresChild[NpresChild].Nloc:=Np1;
        CopyPoints(NP1,Points,PresChild[NpresChild].loc);
      End;{ For JJ}
      End {np=0}

     ELSE
       Begin  { The Child node Can Be embedded }
        Graph [ChildID].Visit:=True;
        INC(NpresChild);
        PresChild[NpresChild].Inode:=ChildID;
        PresChild[NpresChild].Nloc:=Np;
        CopyPoints(Np,Points,PresChild[NpresChild].loc);
       ENd; { NP <> 0}
    END {THEN}
ELSE  Begin { the other parent node has not been embedded yet.}
```

```
                    { Therefore, Insert a connection node between the }
                    { Parent and the child node. }

                     DefineConnector(ParID1,ChildID,ConnectID);
                     InsertNode(ParID1,ChildID,ConnectID,connector);
                     Graph[ConnectId].visit: = true;
                     INC(NpresChild);
                     PresChild[NpresChild].Inode: =  ConnectID;

                     PossibleMoves(ConnectID,Graph[ParID1].x,Graph[ParID1].y,
                                   anydir,Np1,Points);
                     PresCHild[NpresChild].Nloc: = Np1;
                     CopyPoints(Np1,Points,PresChild[NpresChild].loc);


                END; {ELSE}
         End;{Binary}


     End;{Case Unary, Binary}

    If (Np1 = 0) Or (Np2 = 0) Then LocAvailable: = False;
    j: = j + 1;  ChildID: = Graph[ParID].Ch2;
    ENd; {while}

k: = K + 1;
End; {while K}

If LocAvailable Then
 Begin
   SortChildren;
   For J: = 1 to NpresChild Do ChildIndex[J]: = 1;
   For J: = BottomRow To TopRow Do ChildCut[J]: = Empty;
   FindMates (PresChild,NpresChild,ChildMate);
   MinHeight(MW);
 End;
{
Writeln;Writeln(' children table');
For j: = 1 to Npreschild do
 Begin
    Writeln;
    Write(PresChild[j].Inode:6);
    For k: = 1 to PresChild[j].Nloc Do
         Write('(',PresChild[j].Loc[k].x:3,',',PresChild[j].loc[k].y:3,')   ');
 End; }

End; {procedure FindChildren}

{------------------------------------------------------------------------}
Procedure Reorder(X,Y,K: Integer; Var Table: SearchTable);
```

```
{ Reorder the entries of row K in the search table, so that closer locations
to (X,Y) are listed first.}

Type Distances= Array[1..MaxSucc,1..2] of Integer;

Var  Nl, Smallest, Ntemp,
     Xi,Yi,Dis,L: Integer;
     LocDistance: Distances;
     Temp: Neighbors;

Begin
     Nl:=Table[K].nloc;
     Ntemp:=0;
     For L:=1 to Nl Do
     Begin
       Xi:=Table[K].Loc[L].x;
       Yi:=Table[K].loc[L].Y;
       Dis:= Abs(Xi-X)+ Abs(Yi-Y);
       If (Xi<>X) And (Yi<>Y) And
         ((Dis Mod 2) <>0) Then Dis:=Dis-1;

       LocDistance[L,1]:=L;
       LocDistance[L,2]:=Dis;
     End;


     {order the points in ascending order according to their distance
      from x,y}

     While NL>0 Do
     Begin
        Smallest:=1;
        For L:=1 to NL Do
           If LocDistance[L,2]<LocDistance[Smallest,2] Then Smallest:=L;
        Inc(Ntemp);
        Temp[Ntemp].x:=Table[K].loc[LocDistance[smallest,1]].x;
        Temp[Ntemp].y:=Table[K].loc[LocDistance[smallest,1]].y;
        For L:=Smallest to (NL-1) Do
           Begin
           LocDistance[L,1]:=LocDistance[L+1,1];
           LocDistance[L,2]:=LocDistance[L+1,2];
           End;
        NL:=NL-1;
     End;


     {restore the ordered points back into the table}
     For L:=1 to Ntemp Do
        Begin
        Table[K].loc[L].x:=Temp[L].x;
        Table[K].loc[L].y:= Temp[L].y;
        End;

End;
{----------------------------------------------------------------------}

Procedure BackTrackSearch (Var Table:SearchTable; TableSize, StartLocation: Integer;
                Var Index:IndexArray; Var RowStatus:CutPoints;
                Sib:Mates; Var Exist:Boolean);

Var I,J,K,NodeID,XX,YY,loc: Integer;
   Found, MorePoints, Occupied:Boolean;


{----------------------------------------------------------------------}

Procedure ForwardSolution( Var Found:Boolean);

{Tests whether the I'th child can be located at (XX,YY) }


Begin
   Found:= False;

   Case RowStatus[YY] of

    Empty: Begin
         If Graph[NodeId].ntype < > connector
          Then  Begin  Index[I]:=J;
                  Found:= True;
                  RowStatus[yy]:= Full;
              End
          Else  Begin
              TestEdge(Graph[ParId1].x,Graph[ParId1].y,
                           XX,YY,occupied,loc);
              If Not(Occupied)
                Then  Begin
                    ReservEdge(Graph[ParId1].X,Graph[ParId1].Y,
                                       XX,YY);
                    If Grid[XX,YY] = Empty
                      Then RowStatus[YY]:= HalfFull
                      Else RowStatus[YY]:= Full;
                    Index[I]:=J;
                    Found:= True;
                    End
                Else J:=J + 1;
                end
        End; {empty}

  HalfFull: { if a connector is already occupying a PE, then
          another connector can occupy the same PE as long
```

```
            as the path to that PE is free.}

           If  Graph[NodeId].ntype = connector Then
              Begin
                ParId1: = Graph[NodeID].Par1;
                TestEdge(Graph[ParId1].X,Graph[ParId1].Y,XX,YY,
                                      Occupied,Loc);
                IF Not(Occupied)
                  THEN Begin
                      Found: = True;
                      Index[i]: = J;
                      RowStatus[yy]: = Full;
                      ReservEdge(Graph[PArId1].X,Graph[ParId1].Y,XX,YY);
                      End
                   ELSE J: = J + 1;
              End
              Else J: = J + 1;
  Full:  J: = J + 1;

  End;{case}

End; {ForwardSolution}
{----------------------------------------------------------------------}
Begin

  Exist: = True;
  I: = Startlocation;
  While (1  < =  I)  AND ( I  < =  TableSize ) AND Exist Do
   Begin
   J: = Index[I];
   Found: = False;
   NodeID: =  Table[i].Inode;
   ParID1: = Graph[NodeID].par1;

   While (J  < = Table[I].Nloc) AND NOT(Found) Do
      Begin
        YY: =  Table[I].Loc[J].Y;
        XX: =  Table[I].Loc[J].X;
        ForwardSolution(Found);
      End;

  { If a possible location for that node has been found then advance}
  { to the next entry in the table. Otherwise, Backtrack}

  IF Found
    Then Begin
        For K: = 1 to Sib[I].Nmate Do  Reorder(XX,YY,Sib[I].S[K],Table);
        I: = I + 1;
        End

    ELSE Begin
```

```
        { dislocate the node, and backtrack to the node which still }
        { has other locations. }
        Iforward: = I;
        MorePoints: = False;
        Index[I]: = 1;
        If I = 1
        Then Exist: = False
        Else    Repeat

                I: = I-1;
                J: = Index[I];
                YY: = Table[I].loc[Index[I]].y;
                XX: = Table[I].loc[Index[I]].x;
                NodeId: = Table[I].Inode;
                ParId1: = Graph[NodeId].par1;
                If Graph[NodeID].ntype = connector Then
                  DislocateEdge(Graph[ParId1].x,Graph[ParId1].y,XX,YY);

                Case RowStatus[yy] of
                 HalfFull: RowStatus[YY] : = Empty;
                 Full     : If Graph[nodeID].ntype  =  connector
                            Then RowStatus [yy]: = HalfFull
                            Else RowStatus [yy]: = Empty;

                 End; {case}

                IF J < >  Table[I].Nloc
                          Then   Begin
                                  MorePoints: = True;
                                  INC(Index[I]);
                                 End
                          Else     Index[I]: = 1;
                IF (I = 1) AND (J = Table[1].Nloc )Then Exist: = False;

               Until MorePoints Or NOT(Exist);


              End; {Else}

     End; { while I}

ENd; {BackTrack}
{-----------------------------------------------------------------------}

Procedure SetParents;

{ Renames the present set of children nodes, as the new set of parent nodes. }

Begin
```

```
For I:=1 to NpresChild Do
   BEGIN
       ChildID:=PresChild[I].Inode;
       XX:=PresChild[I].loc[ChildIndex[I]].X;
       YY:=PresChild[I].Loc[ChildIndex[I]].y;
       Graph[ChildId].x:=XX;
       Graph[ChildID].y:=YY;

       If Graph[ChildID].ntype <> connector
         Then Grid[XX,YY]:=Full
         Else If Grid[XX,YY]=Empty Then Grid[XX,YY]:=HalfFull
                          Else Grid[XX,YY]:=Full;

       {Update area used for embedding}
       IF XX> AreaWidth Then AreaWidth:=XX;

       { copy to parent table}
       PresPar[I].Inode:=PresChild[I].Inode;
       PresPar[I].Nloc:=PresChild[I].Nloc;
       CopyPoints(PresChild[I].Nloc,
                PresChild[I].Loc,PresPar[i].Loc);

       If Graph[ChildID].Ntype <> connector
                 Then Inc(NembedNodes);

       ParIndex[I]:=ChildIndex[I];
       ParMate[I].Nmate:=ChildMate[I].Nmate;
       For J:=1 to ParMate[I].Nmate Do
              ParMate[I].S[J]:=ChildMate[I].S[J];
   End; { for I}

 For I:=BottomRow To TopRow DO
         ParCut[I]:=ChildCut[I];

 NpresPar:=NpresChild;
 NpresChild:=0;
 ParLastSol:=False;
 If NactiveEdges>0 Then DelOldEdges;

 writeln;writeln('NGraph  =',ngraphnodes,
                   ' Nembeded=  ',NembedNodes);

 End; {SetParents}

{--------------------------------------------------------------------}

Procedure ReEmbedParents(Var ParEmbed:Boolean);

{ ReEmbed the parent nodes in a new set of locations. ParEmbed is a flag
  which indicate if reallocation can be performed.}
```

```
Var GrndparId,Parloc,Startloc: Integer;

Begin
  writeln;writeln(' reallocating parent nodes');

 { delete all connectors inserted in the process of embedding the children}
 If Npresconnector > 0 Then
    Begin
        MaxID:= PreMaxId;
        For I:= 1 tO Npresconnector Do
              DeleteConnector(PresConnector[I].par,
              Presconnector[I].child,
              Presconnector[I].Connector);
    Npresconnector:= 0;
    End;

 For I:= 1 To NpresChild DO Graph[PresChild[I].Inode].Visit:= False;
 For I:= 1 TO NpresPar  DO
     Begin
       ParID:= PresPar[I].Inode;
       YY:= PresPar[I].Loc[ParIndex[I]].Y;
       XX:= PresPar[I].loc[ParIndex[I]].X;
       If Graph[ParId].ntype < > connector
         Then Grid[XX,YY]:= Empty
         Else If Grid[XX,YY]= Full Then Grid[XX,YY]:= HalfFull
                                  Else Grid[XX,YY]:= Empty;
     End;
  { starting from the bottom of the parent Table, find the first row
    which still has unconsidered points}

  Found:= False;
  I:= NPresPar;
  Repeat
    { writeln(' BackTracking Parent .. I= ',I);}
     YY:= PresPar[I].loc[ParIndex[I]].y;
     XX:= PresPar[I].loc[ParIndex[I]].x;
     ParId:= PresPar[I].Inode;

     Case ParCut[yy] of
       HalfFull: ParCut[YY] := Empty;
       Full:
            If Graph[ParID].ntype =  connector
               Then ParCut [yy]:= HalfFull
               Else ParCut [yy]:= Empty;

       End; {case}

     If Graph[ParId].ntype= connector
       Then Begin
            GrndparId:= Graph[ParId].par1;
         If Grndparld < > 0 Then
           DislocateEdge(Graph[GrndparlD].x,Graph[Grndparld].y,XX,YY)
         End;
   IF  ParIndex[I] < PresPar[I].Nloc Then
           Begin
               Inc (ParIndex[I]);
               Found: = True;
           End
           Else Begin
               ParIndex[I]: = 1;
               I: = I-1;
               End;
Until Found Or (I = 0) ;


If I  < > 0
       Then  BackTrackSearch(PresPar,NpresPar,I,ParIndex,ParCut,
                                      ParMate,ParEmbed)
       Else ParEmbed: = False;

IF ParEmbed
   Then  { An alternative solution is found for the parent nodes}
       { store the new locations.}

       For I: = 1 to NpresPar Do
       Begin
           ParId: = PresPar[I].Inode;
           XX: = PresPar[I].loc[ParIndex[I]].X;
           YY: = PresPar[I].Loc[ParIndex[I]].y;
           Graph[ParID].x: = XX;
           Graph[ParID].y: = YY;

           If Graph[ParID].ntype  < >  connector
             Then Grid[XX,YY]: = Full
             Else If Grid[XX,YY] = Empty Then Grid[XX,YY]: = HalfFull
                                  Else Grid[XX,YY]: = Full;

           {Update area used for embedding}
           IF XX >  AreaWidth Then AreaWidth: = XX;

       End  { For I}
    Else {restore the last obtained solution.}
       For I: = 1 to Nprespar Do
       Begin
           ParLastSol: = True;
           ParID: = PresPar[I].Inode;
           XX: = Graph[Parld].x;
           YY: = Graph[ParID].y;
           If Graph[ParID].ntype  < >  Connector
             Then Begin
                   Grid[XX,YY]: = Full;
```

```
                    ParCut[YY]:= Full;
                  End
                Else
                  Begin
                    If Grid[XX,YY]= Empty Then Grid[XX,YY]:=  HalfFull
                                Else Grid[XX,YY]:= Full;
                    If ParCut[YY]= Empty  Then ParCut[YY]:=  HalfFull
                                Else ParCut[YY]:= Full;
                    Grndparld:= Graph[ParId].par1;
                    If GrndparID <> 0 Then
                        ReservEdge(Graph[GrndparID].x,
                              Graph[Grndparld].y,XX,YY)
                 End;
        End;

 End;{reEmbed parents}

{--------------------------------------------------------------------------}
Procedure ShftNods(Nnodes: Integer; Var DlySuccd:Boolean);

Var GrndPar,Ndeg12: Integer; Dlypar:IndexArray;

Begin
 DlySuccd:= True;


  { find how many nodes of are candidate for delay }
  Ndeg12:= 0;
  For I:= 1 to NpresPar Do
  Begin
   ParId:= PresPar[I].Inode;
   If (Graph[ParId].nparents <= 1 ) AND (Graph[ParID].nchildren= 2)

      Then IF (Graph[Graph[ParId].Ch1].Nparents= 1) AND
            (Graph[Graph[ParId].Ch2].Nparents= 1) Then

     Begin
       {writeln(' a candidate for delay: ',ParID);}
       Inc(Ndeg12);
       DlyPar[Ndeg12]:= I;
     End;
  End;{for I}
  If Ndeg12 < Nnodes Then DlySuccd:= False;

 If DlySuccd Then
 Begin

   For I:= 1 to Npreschild Do
        Graph[Preschild[I].Inode].visit:= False;
```

```
{ delete all connectors inserted in the process of embedding the children}
If Npresconnector > 0 Then
Begin
      MaxID:=PreMaxId;
      For I:=1 tO Npresconnector Do
            DeleteConnector(PresConnector[I].par,
            Presconnector[I].child,
            Presconnector[I].Connector);
      NpresConnector:=0;
End;


For I:=1 to Nnodes Do
Begin
  ParId:=PresPar[DlyPar[I]].Inode;
 { writeln(' node to be delayed ',ParID);}

  Inc(MaxID);
  ConnectID:=maxID;
  If Graph[ParId].nparents=0 Then
     For J:=1 to Nsource DO
         If SourceNodes[J]=ParId Then SourceNodes[J]:=ConnectID;
  If Graph[ParID].nparents=0 Then Grndpar:=0
                       Else GrndPar:=Graph[ParId].par1;
  InsertNode(Grndpar,ParId,ConnectId,Connector);

  Prespar[DlyPar[I]].Inode:=ConnectId;

  Graph[ParId].visit:=False;
  Graph[ConnectId].visit:=True;

  XX:=Graph[ParID].x;
  YY:=Graph[ParID].y;
  Graph[ConnectId].x:=XX;
  Graph[ConnectId].y:=YY;
  Grid[XX,YY]:=HalfFull;
  ParCut[YY]:=HalfFull;
  If Grndpar<>0 Then
         ReservEdge(Graph[Grndpar].X,Graph[Grndpar].Y,XX,YY);

  NembedNodes:=NembedNodes-1;
End;

FindMates(Prespar,NpresPar,ParMate);


{check for indefinite delay}
DlySuccd:=False;
I:=1;
While Not(DlySuccd) And (I<=Nprespar) Do
Begin
```

```
      If Graph[Prespar[I].Inode].ntype < > connector Then DlySuccd: = True
         Else If ParMate[I].nmate < > 0 Then DlySuccd: = True;
      Inc(I);

  End
 End;


End; {ShftNods}
{-----------------------------------------------------------------------}
{                  I/O Routines                                         }
{-----------------------------------------------------------------------}


Procedure ReadGraph;
{ reads a graph from a file.}
Begin

SetFIles;
Reset(GraphFile);
{Initialize}
NgraphNodes: = 0;  NpresPar: = 0;       NpresChild: = 0;
NFirstChildren: = 0; Nsource: = 0;

For K: = 1 to MaxNodes Do
   Begin
     Graph[k].Ntype: = Blank;    Graph[k].Nchildren: = 0;    Graph[k].Nparents: = 0;
     Graph[k].Ch1: = 0;          Graph[k].Ch2: = 0;
     Graph[k].Par1: = 0;         Graph[k].Par2: = 0;
   End;

Readln(Graphfile,cc,I,Graph[I].ch1,Graph[I].Ch2);
While Not EOF (GraphFile) DO
 Begin
     INC(NGraphNodes);

     Graph[I].NType: = CC;

     If CC < > sink Then Graph[I].Nchildren: = 1;
     IF Graph[I].Ch2 < > 0 Then Graph[I].Nchildren : = 2;

     {connect the children to the parent node}
     JJ: = Graph[I].ch1;    K: = 1;
     While (JJ < > 0 ) AND (K < = 2) DO
         Begin
             Inc (Graph[JJ].Nparents);
             If Graph[jj].Nparents = 1 THEN Graph[JJ].Par1: = I
                                ELSE Graph[JJ].Par2: = I;
             JJ: = GRaph[I].Ch2;     K: = K + 1;
         END;
     If MaxID < I Then MaxID: = I;
```

```
    {list source nodes as the first set of parents}
    If CC = source THEN
           Begin  INC(Nsource);;
                SourceNodes[Nsource]: = I;
                NFirstChildren: = NFirstChildren + Graph[I].Nchildren;
           End;
    Readln(GraphFile,cc,I,Graph[I].Ch1,Graph[I].Ch2);
 End; {Eof}
 MaxID: = NgraphNodes;
END; {ReadGraph}
{---------------------------------------------------------------------}
Procedure PrintEmbedGraph;

{ prints the embedded graph on a disk file.}
Var NtotalNodes,Diameter: Integer;

Begin

 Writeln(OutputFile,'  GraphID : ',GraphID); writeln(OutputFile);
 Writeln(OutputFile,'  Area used For Embedding : ',AreaHeight, ' * ',
                (AreaWidth + 1));

 CountsyncNodes(Nsync,Diameter,np);
 Writeln(OutputFile,'  Embedded Graph Diameter : ',Diameter);

 NtotalNodes: = 0;
 For J: =  BottomRow To TopRow Do
    For I: = 0 To AreaWidth Do
       If Grid[I,J]  < >  Empty Then  Inc(NtotalNodes);
 Writeln(OutputFile,'  Total number of grid points used for embedding :',
               NtotalNodes);
 For I: = 1 to 3 Do Writeln(OutputFile);


 Writeln(OutputFile,'  Node  Type  Nch   Ch1   Ch2   Npar  Par1  Par2  X     Y ');
 Writeln(OutputFile,' _____________________________________________________________');

 For k: = 1 to MaxId Do
 begin
   Writeln(OutputFile,K:6,Graph[k].nType:6,Graph[K].Nchildren:6,Graph[k].ch1:6,
        Graph[k].ch2:6,Graph[k].Nparents:6,Graph[k].Par1:6,Graph[k].Par2:6,
        GRaph[k].X:6,Graph[k].Y:6);
 end;


 Close(OutputFile);
End; {PrintGraph}
{---------------------------------------------------------------------}
 Procedure PrntParTbl;
 Var j,k: Integer;
```

```
Begin

Writeln;
Writeln(' parent table');
writeln('NodeID ------------ possible locations ---------- selected loc');
For j:=1 to Nprespar Do
  Begin
     Writeln; write(PresPar[j].Inode:3,'  ');
     For k:=1 to PresPar[J].nloc Do
        write('(',Prespar[J].loc[k].x:3,',',
               Prespar[J].loc[k].y:3,') ');
     for k:=1 to (5-PresPar[j].nloc) Do write ('          ');
     write('     ',ParIndex[j]:1);
  End;
End;
{----------------------------------------------------------------------}
Procedure DrawGraph;

{Draws the embedded graph on the screen.}

Procedure Diamond (x,y,r: Integer);
{-------------------------------}
{Draws a diamond centered at point X,Y}

Begin
  Draw(x+r,y,x,y+r,1);
  Draw(x,y+r,x-r,y,1);
  Draw(x-r,y,x,y-r,1);
  Draw(x,y-r,x+r,y,1);
End;



Procedure Cross (x,y: Integer);
{----------------------------}
Var i: Integer;
Begin
  i:=1;
  Draw(x-i,y,x+i,y,1);
  Draw(x,y+i,x,y-i,1);
End;


Procedure Arrow(X1,Y1,X2,Y2,color: Integer);
{----------------------------------------}
{draws an arrow directed from x1,y1 to x2,y2. }
Var Xp,Yp,xa1,ya1,xa2,ya2,L: Integer;
   Theta: Integer;

Begin
   L:=3;
   Xp:=(2*X1+X2) Div 3;
   Yp:=(2*Y1+Y2) Div 3;
   If (X1=X2) Then If (Y1>Y2) Then Theta:=-90
                        Else Theta:=90
          Else  If (Y1=Y2) Then Theta:=0
          Else  If (Y1>Y2) Then Theta:=-45
          Else  Theta:=45;

   Case Theta Of
     -45: Begin xa1:=xp;   ya1:=yp+1; xa2:=xp-1;  ya2:=yp;   End;
      45: Begin xa1:=xp-1; ya1:=yp;   xa2:=xp;    ya2:=yp-1; End;
      0 : Begin xa1:=xp-1; ya1:=yp+1; xa2:=xp-1;  ya2:=yp-1; End;
     -90: Begin xa1:=xp+1; ya1:=yp+1; xa2:=xp-1;  ya2:=yp+1; End;
      90: Begin xa1:=xp-1; ya1:=yp-1; xa2:=xp+1;  ya2:=yp-1; End;
     End;{case}
   Draw(Xa1,Ya1,Xp,Yp,color);
   Draw(Xa2,Ya2,Xp,Yp,color);

End;



Const offset=5;

Var GridSize,temp,Ix,Iy,
    X1,Y1,X2,Y2: Integer;

Begin

   HiRes;
   HiResColor(LightGray);
   GraphWindow(5,5,635,195);
   gridsize:=310 Div (AreaWidth+1);
   temp:=180 Div Areaheight;
   If Temp<GridSize Then GridSize:=Temp;

   For I:=1 to MaxID Do
   If graph[I].visit Then
   Begin
    X1:=Offset+Graph[I].x*2*gridSize;
    Y1:=Offset+(TopRow-Graph[I].y)*gridSize;

    If Graph[I].Nchildren >=1 Then
      Begin
        x2:=Offset+(Graph[Graph[I].ch1].x)*2*gridSize;
        Y2:=Offset+(TopRow-Graph[Graph[I].ch1].y)*GridSize;
        Draw(x1,y1,x2,y2,1);  Arrow(x1,y1,x2,y2,1);
      End;
    If Graph[I].Nchildren =2 Then
      Begin
        x2:=Offset+(Graph[Graph[I].ch2].x)*2*gridSize;
        Y2:= Offset + (TopRow-Graph[Graph[I].ch2].y)*GridSize;
        Draw(x1,y1,x2,y2,1);    Arrow(x1,y1,x2,y2,1);
       End;

    If (Graph[I].ntype  =  source) OR (Graph[I].ntype = sink)
                    Then Diamond (X1,Y1,5)
                    Else IF Graph[I].ntype < >  connector Then Diamond(X1,Y1,2);

   End;
   For Ix:=0 to AreaWidth Do
    For Iy:=  0  to (Areaheight-1) Do
          Begin
               x1:= Offset + Ix*2*GridSize;
               Y1:= Offset + Iy*GridSize;
               Plot(X1,Y1,1);
          End;

End;{DrawGraph}
{------------------------------------------------------------------}
```

```
Procedure Rgraph ( Var Nsource, Ngraphnodes:Integer; Var Graph:DblLnk);
{ This routine generates an acyclic graph. The number of sink, source
  and intermediate nodes are given. The maximum input and output degrees
  of the nodes is prespecified. The program randomly connects the nodes
  to form a connected acyclic  digraph. }

{writeen By: Faridah Ali }
{Date:10/15/87          }

Const
   { entries of the adjacency matrix}
    Connection = 1;
    Noconnection = 0;
    undecided = -1;
    ForbidCon = 2;

    MaxInDegree = 2;
    MaxOutDegree = 2;

    MaxNodes= 100;



Type
    AdjacencyMatrix = Array [1..MaxNodes,1.. MaxNodes] of Integer;
    IndexArray = Array [1..MaxNodes] of Integer;
    NodesDegree =  Array[1..Maxnodes] of Integer;
Var
    Amatrix: AdjacencyMatrix;
    Xindex,Yindex: IndexArray;
    NodesIndegree,Nodesoutdegree: NodesDegree;

    Nsink,NinterMed,
    Nid,Nod: Integer;

    Nloc,I,J,K,L,N,Nx,Ny: Integer;
    MinEdges,MaxEdges: Integer;

    TopDown,BottomUp: Boolean;
    Finish,OtherEnd,DeadEnd: Boolean;

{---------------------------------------------------------------}

Procedure StoreGraph;
{stores the randomly generated graph in the array Graph and on a disk file.}

 Procedure Link(node:Integer);
  Var J,JJ,K:Integer;
  Begin
  { establish links between the node and its children}
  J:=1;

  While Amatrix[Node,J] <> Connection Do J:=J+1;
  Graph[Node].ch1 :=J+Nsource;
  If Graph[node].nchildren >1 Then
          Begin
               J:=J+1;
               while Amatrix[node,J] <> Connection Do J:=J+1;
               Graph[I].Ch2:=J+Nsource;
          End;

  {establish links to the parents}
  JJ:=Graph[node].ch1;   K:=1;
  While (JJ<>0) ANd (K<=2) Do
   Begin
      Graph[JJ].Nparents:=Graph[JJ].Nparents+1;
      If Graph[JJ].nparents=1 Then Graph[JJ].par1:=node
                             Else Graph[JJ].par2:=node;
      JJ:=Graph[I].ch2;   K:=K+1;
   End;

End;{link}

Begin
  ReWrite(GraphFile);
  NgraphNodes:=Nsource+NSink+NinterMed;
  MaxID:=Ngraphnodes;
  NfirstChildren:=0;
  For K:=1 to MaxNodes Do
   Begin
     Graph[K].Ntype:=Blank;   Graph[K].Nchildren:=0;  Graph[K].nparents:=0;
     Graph[K].ch1:=0;     Graph[k].Ch2:=0;
     Graph[K].Par1:=0;    Graph[K].par2:=0;
   End;

  For I:=1 to Nsource Do
   Begin
     SourceNodes[I]:=I;
     Graph[I].ntype:='S';
     Graph[I].Nchildren:=NodesOutdegree[I];
     NfirstChildren:=NfirstChildren+NodesOutdegree[I];
     Link(I);
     Writeln(GraphFIle,'S',I:4,Graph[I].Ch1:4,Graph[I].ch2:4);
   End;

  For I:=(Nsource+1) to (Nsource+NinterMed) DO
   Begin
     Graph[I].ntype:='O';
     Graph[I].nchildren:=NodesOutdegree[I];
     Link(I);
```

```
    Writeln(GraphFile,'O',I:4,Graph[I].ch1:4,Graph[I].ch2:4);
  End;

 For I:=(Nsource+NinterMed+1) To NgraphNodes Do
  Begin
    Graph[I].ntype:='K';
    writeln(GraphFile,'K',I:3,0:3,0:3);
  End;
  Writeln(GraphFile);
  Close(GraphFile);
End;{storeGraph}
{---------------------------------------------------------------------}
Begin
   SetFiles;

   Writeln(' Enter:  Nsource   Nsink    NinterMed');
   Readln(Nsource,Nsink,NinterMed);
   Writeln(outputfile,' ');
   Writeln(outputfile,' Nsource= ',Nsource,'  Nsink= ',
              Nsink,'  NinterMed= ',NinterMed);


   NX:=NinterMed+Nsink;
   NY:=Nsource+NinterMed;
   MinEdges:=(NinterMed+Nsource+Nsink)*MaxIndegree;
   MaxEdges:=0;
   Finish:=False;

   While Not(Finish) Do
   Begin
 { initialize In and Out Degree for each node and the Adjacency matrix}
    For I:=1 to Ny Do NodesOutdegree[I]:=0;
    For J:=1 to Nx Do NodesINdegree[J]:=0;
    For I:=1 to Ny Do
       For J:=1 to Nx DO
          If (I>Nsource) And (J <=(I-Nsource) )
            Then Amatrix[I,J]:=ForbidCon
            Else Amatrix[I,J]:=Undecided;

 { Randomly selest whether to fill the matrix from TopDowm or BottomUp}
   TopDown:=False;    BottomUp:=False;
   If Random(2)=0 Then TopDown:=True
               Else BottomUp:=True;
   If TopDown Then I:=1
            Else I:=Ny;

   OtherEnd:=False;    DeadEnd:=False;

   While Not(OtherEnd) And Not(DeadEnd)  Do
   Begin
     Nloc:=0;
```

```
For J:=  1 to Nx Do
   If Amatrix[I,J] = Undecided Then
            Begin
               Nloc: = Nloc + 1;
               XIndex[Nloc]: = J;
            End;
{ randomly set the outdegree of the node}
Nod: = 1;
If Nloc > Nod Then
         Nod: = 1 + Random(MaxOutDegree);


If Nloc> = Nod
  Then
       For K: = 1 to Nod Do
       Begin
        NodesOutdegree[I]: = Nod;
        N: = 1 + Random(Nloc);
        J: = XIndex[N];
        Amatrix[I,J]: = Connection;
        NodesIndegree[J]: = NodesIndegree[J] + 1;
        If NodesINdegree[J] = MaxIndegree Then
        {No more connection can sink in node J}
           For l: = 1 to Ny Do
              If Amatrix[l,J] = Undecided Then
                           Amatrix[l,J]: = Noconnection;

        For l: = N to Nloc Do  Xindex[l]: = Xindex[l + 1];
        Nloc: = Nloc-1;
       End {then}

   Else Begin
        Nloc: = 0;
        For J: = 1 to Nx Do
           IF Amatrix[I,J] = Noconnection Then
              For K: = 1 to Ny Do
                If (Amatrix[K,J] = connection) AND
                   (NodesOutdegree[K] = MaxoutDegree) Then
                    Begin
                      Nloc: = Nloc + 1;
                      Xindex[Nloc]: = J;
                      Yindex[Nloc]: = K;
                    End;
        If Nloc = 0
           Then DeadEnd: = True
           Else
              Begin
               l: = 1 + Random(Nloc);
               J: = Xindex[l];
               K: = Yindex[l];
               Amatrix[I,J]: = connection;
```

```
                Amatrix[K,J]: = Noconnection;
                NodesOutdegree[K]: = NodesOutdegree[K]-1;
                NodesOutDegree[I]: = NodesOutDegree[I] + 1;
               End;

        End;{Else}


 If TopDown Then
        Begin I: = I + 1;
            IF I = (Ny + 1) Then OtherEnd: = True;
        End
Else If BottomUp Then
        Begin
            I: = I-1;
            If I = 0 Then OtherEnd: = True;
        End;

End;{ while Not OtherEnd}

 { check for isolated nodes}
 {-------------------------}
 For I: = 1 to NX Do
  If NodesIndegree[I] = 0 Then
    Begin
     Nloc: = 0;
     For K: = 1 TO Ny DO
         If (NodesOutDegree[K] < MaxOutDegree )
           AND (Amatrix[K,I] = Undecided) Then
           Begin
            Nloc: = Nloc + 1;
            Yindex[Nloc]: = K
           End;
    If Nloc = 0 Then DeadEnd: = True
      Else Begin
           Nid: = 1;
           If Nloc > Nid Then Nid: = 1 + Random(MaxInDegree);
           NodesIndegree[I]: = Nid;
           For K: = 1 To Nid Do
              Begin
               N: = 1 + Random(Nloc);
               J: = Yindex[N];
               AMatrix[J,I]: = Connection;
               NodesOutdegree[J]: = NodesOutdegree[J] + 1;
               For L: = N to Nloc Do Yindex[L]: = Yindex[L + 1];
               Nloc: = Nloc-1;
              End;
         End;
    End;
 If Not(DeadEnd) Then Begin

                          StoreGraph;
                          Finish: = True;
                       End;

End; {Not finish}

End;
```

```
{-----------------------------------------------------------}
{ Two-Generation Lookahead BFG Heuristic                    }
{-----------------------------------------------------------}

{$I Elib.pas }
{$I Rgraph.pas}

{ M A I N   P R O G R A M }

Begin
Writeln(' OPtion: (R,G)   >:'); Readln(Option);
Case UpCase(Option) Of
  'R' :ReadGraph;
  'G' :Rgraph(Nsource,NgraphNodes,Graph);
  End;{case}

CountSyncNodes(Nsync,GrphDiameter,Np);
EmbedSource(NSource);

While NembedNodes < NgraphNodes Do
Begin
 {PrntParTbl;}

 PreMaxId:= MaxID;
 FindChildren(LocAvailable,MW);

 If MW  < =  AreaHeight Then
 Begin

 {Find next possible allocation of children nodes}
 IF LocAvailable  Then
       BackTrackSearch(PresChild,NpresChild,1,ChildIndex,ChildCut,
                                          ChildMate,ChildEmbed);


 If (LocAvailable And ChildEmbed)

     Then SetParents
     ELSE Begin
          If ParLastSol Then ParEmbed:= False
                       Else ReEmbedParents(Parembed);
          If Not Parembed Then
                       EmbedSource(AreaHeight + 1);

     End {if Not Child Embed}
 End
 Else
     EmbedSource(Areaheight + 1);

END;{ While NembedNodes  <  Ngraph }
```

```
PrintEmbedGraph;
DrawGraph;
END.
```

```
{-----------------------------------------------------}
{ Modified Two-Generation Lookahead BFG Heuristic       }
{-----------------------------------------------------}

{$I elib.pas }
{$I Rgraph.pas}

{ M A I N   P R O G R A M }

Begin
Writeln(' OPtion: (R,G)   >:'); Readln(Option);
Case UpCase(Option) Of
   'R' :ReadGraph;
   'G' :Rgraph(Nsource,NgraphNodes,Graph);
   End;{case}

CountSyncNodes(Nsync,GrphDiameter,Np);
EmbedSource(NSource);

While NembedNodes < NgraphNodes Do
Begin
 {PrntParTbl;}

 PreMaxId:= MaxID;
 FindChildren(LocAvailable,MW);

 If MW  < =  AreaHeight Then
 Begin

 {Find next possible allocation of children nodes}
 IF LocAvailable  Then
        BackTrackSearch(PresChild,NpresChild,1,ChildIndex,ChildCut,
                                       ChildMate,ChildEmbed);


 If (LocAvailable And ChildEmbed)

     Then SetParents
     ELSE Begin
          If ParLastSol Then ParEmbed:= False
                    Else ReEmbedParents(Parembed);
          If Not Parembed Then
            Begin
              ShftNods(1,Dlysuccd);
              If Not DlySuccd Then EmbedSource(AreaHeight+ 1);
            End;

     End {if Not Child Embed}
 End
 Else
   Begin
```

```
          If (MW-Areaheight) = NpresPar
                    Then  DlySuccd: = False
                    Else ShftNods(MW-areaheight,DlySuccd);

          If Not Dlysuccd Then EmbedSource(Areaheight + 1);
   End;

END;{ While NembedNodes  <  Ngraph }

PrintEmbedGraph;
DrawGraph;
END.
```