Mapping Homogeneous Graphs on Linear Arrays

I. V. RAMAKRISHNAN, D. S. FUSSELL, AND ABRAHAM SILBERSCHATZ

Abstract — This paper presents a formal model of linear array processors suitable for VLSI implementation as well as graph representations of programs suitable for execution on such a model. A distinction is made between correct mapping and correct execution of such graphs on this model and the structure of correctly mappable graphs are examined. The formalism developed is used to synthesize algorithms for this model.

Index Terms — Array processors, graphs, mapping, parallel processing, synthesis, VLSI.

I. INTRODUCTION

In conventional Von Neumann architectures the processor receives data and instructions from a memory unit and returns the results of its computation to the memory unit. The operation rate that is realizable in such systems is limited by the bandwidth of the processor-memory communication link, commonly referred to as the “Von Neumann” bottleneck [1].

Kung [13] introduced systolic arrays as an elegant and cost-effective architectural solution using VLSI technology to overcome such a bandwidth limitation. A systolic array consists of a collection of processing elements, either all of the same type or a mixture of a few different types, each capable of performing a simple operation such as addition, multiplication, comparison, etc. The processing elements are interconnected to form a linear array, a rectangular mesh, a hexagonal mesh or a tree. Once a data item is retrieved from memory by the processor array, the data item passes through several processing elements in the array before returning to memory. This feature of using a datum from memory many times over without having to store and retrieve intermediate results gives rise to high computation throughput. In a typical application, such arrays would be attached as peripheral devices to a host computer which inserts input values into them and extracts output values from them. Simplicity and uniformity of the processing elements in a systolic array and the regularity of their interconnection make them suitable for VLSI implementation at minimal design costs. Many computationally demanding problems, especially in signal and image processing, require multiple operations to be performed on each data item in a repetitive manner. Systolic arrays (referred to as VLSI arrays from now on) are a cost-effective and efficient alternative for handling such problems. The processing elements in these arrays operate synchronously and in a typical application such an array would be attached as a peripheral device to a host computer which would insert input values and extract output values from it.

In a VLSI array, input data and (partial) results may flow in multiple directions at different speeds. Fig. 1, which is taken from [16], is a systolic algorithm (henceforth referred to as an array algorithm) developed by Kung and Leiserson to multiply an \( n \times n \) band matrix (of bandwidth four) by a vector of \( n \) elements on a linear array.

Each processing element (referred to as a cell from now on) in the array performs one multiplication step (that is, a multiplication and an addition). The \( x_i \)'s and \( y_i \)'s travel in opposite directions along the horizontal axis and the \( a_{ij} \)'s march downward along the vertical axis. The \( x_i \)'s, \( y_i \)'s, and \( a_{ij} \)'s stay in any cell for one clock tick before being pumped out of it. The time interval between insertion of \( x_i \), \( x_{i+1} \) and \( y_i \), \( y_{i+1} \), and \( a_{ij} \), \( a_{i,j+1} \) is two ticks. Observe also that the computation and I/O are overlapped.

All such array algorithms exhibit the following common feature. They are composed of “streams” of data traveling in multiple directions at different speeds. Each cell receives data from each of the streams, performs one of the primitive operations and pumps them out (possibly updated). The linear array algorithm shown in Fig. 1 is comprised of three streams. Two of them (one comprised of \( x_i \)'s and the other comprised of \( y_i \)'s) travel in opposite directions along the horizontal axis while the third stream (comprised of \( a_{ij} \)'s) moves down along the vertical axis. An element (which we will refer to as a token from now on) in each of these streams travels at a velocity of one cell per tick.

Elegance and cost-effectiveness of VLSI arrays has led to a flurry of research in designing algorithms on them with considerable success (see [13] for an extensive bibliography of such algorithms).

An important problem then is the automatic design of such algorithms from some high-level description of a computation. The design must transform such a description into a “correct” array algorithm for the computation and this would involve forming the requisite number of data streams, arranging the tokens and their speeds in such a way that the right data arrives at the right cell at the right time. For instance, in Fig. 1, consecutive \( x_i \)'s, \( y_i \)'s, and consecutive \( a_{ij} \)'s in a diagonal are separated by two time units. \( x_i \) follows \( x_{i+1} \), \( y_{i+1} \) follows \( y_i \), and, on a diagonal, \( a_{i,j+1} \) follows \( a_{i,j} \). Such an arrangement ensures that when \( y_i \) emerges from the array it has accumulated all the correct terms. (For details see [16].) The design must also take into consideration such fac-
tors as computational rate, area, time, communication topology, etc.

In the past some attempts had been made at systematic design of such array algorithms. Johnson and Cohen [11] and Weiser and Davis [26] have developed an imaginative methodology which transforms certain simple arithmetic expressions into computational networks. By explicitly representing time in these expressions via a delay operator they show how it can be manipulated to obtain different computational networks with differing performances. However the types of arithmetic expressions that can be so manipulated were not characterized. Besides, as their methodology is nonconstructive, it is of limited utility for mechanical design.

Independently and in parallel to our efforts, Moldovan [20] and Miranker and Winkler [21] have attempted to synthesize VLSI array algorithms. Their methods extract the dependencies between the variables of a program considered as points in euclidean space. Then they look for a linear transformation that maps these points onto an array. However, neither methods give a systematic way to find this transformation, that is, they are again nonconstructive. Recent attempts have been those of Capello and Stieglicz [2] and Kung and Lin [17]. Capello and Stieglicz have shown that array algorithms for the convolution problem described in [13] are related to one another by affine transforms. Given an array algorithm for a computation, affine transforms succinctly describe space/time rearrangements. However, they do not provide any guidelines for designing array algorithms. Kung and Lin extend the work of Johnsson and Cohen and Weiser and Davis. They have developed an algebra for manipulating array algorithms enabling them to construct various algorithms with differing performances.

The types of computation that can be realized on VLSI arrays exhibit a great deal of "regularity" in their communication requirements. Our research that is described in this paper has largely been focussed on gaining some insight into the regularity imposed by these arrays on the computations that can be realized by them. Such an understanding would be useful in developing mechanical transformations of high-level computational descriptions into array algorithms. To accomplish this two important tasks require prime consideration.

First, we need a formalization of VLSI arrays. Our formalization in this paper captures the intuitive model that has been used by other researchers. Our focus has been on linear arrays as they represent the simplest and also the most fundamental geometry for connecting processors. Besides, in practice, linear arrays appear more attractive than rectangular and hexagonal arrays. Among them are the following: linear arrays have bounded I/O requirements [13]; in a wafer containing faulty cells, a large percentage of nonfaulty cells can be efficiently reconfigured into a linear array with constant wire length between adjacent cells in the linear array [8], [18]; synchronization between cells in a linear array can be achieved by a simple global clock whose rate is independent of the size of the array [5]; linear arrays that have been built recently include the TRW array [12] and the WARP machine [14].

Second, we need a formal specification of high-level descriptions of computations. Such a specification must be amenable for formal analysis and must also enhance a designer's intuition. Program graphs are very useful in this regard. A program graph is a directed acyclic graph describing a computation. The edges represent values and the vertices represent computation of a function whose arguments are the values represented by the incoming edges. In this paper we formalize a class of program graphs to describe computations that are appropriate for VLSI arrays. The program graphs that we consider can be used to describe a variety of interesting computations including sorting, merging, priority-queue operations, FIR filtering, DFT, linear recurrences, multidimensional convolution, matrix computations, dynamic programming, transitive closure, and relational database operations.

We distinguish between correct mapping and correct execution of such program graphs on our linear array model. For a large class of such program graphs we provide a set of necessary and sufficient conditions that any graph in this class must satisfy in order that is can be mapped onto a linear array. (For such graphs we construct such a mapping and thus our method is constructive.) Almost all the program graphs describing computations for which linear-array algorithms have been described in the literature satisfy our characterization. We also discuss the importance of using some semantic knowledge (that is, some property of the function represented by the nodes in the graph) to correctly execute the graph.

The rest of this paper is organized as follows. In Section II we formalize the linear array and program graph models appropriate for execution on the linear array. We also provide precise formulations for correct mapping. In Section III we examine the structural properties of correctly mappable program graphs. We illustrate the theory developed in this paper by synthesizing a few published and some new array algorithms.

Since the proofs of the theorems are quite lengthy and since the reader need not understand it in order to proceed, the details of the proofs are deferred to the Appendix.

II. COMPUTATIONAL MODELS

We will now describe our linear array and program models. We begin with the linear array model.

A. Linear Array Model

For simplicity we restrict ourselves to a linear array made up of cells that do not have any decision-making ability. The cells therefore can only execute straight-line programs. Such a restriction precludes the flow of control signals in the array. Almost all the linear-array algorithms reported in the literature are devoid of any such control signals. Although the
theory described in this paper will address the design of such algorithms we will later on discuss possible extensions of this theory to handle programs with branching instructions.

Recall that a linear array is typically made up of cells that are all "uniform" and an algorithm on this array is comprised of streams and each stream consists of tokens that travel at a fixed velocity (cells/clock tick). We model uniformity of cells by requiring that they all execute the same program (that is, compute the same function) in every tick. As the cells do not have any decision-making ability the programs executed by these cells do not have any branching instructions.

In order to model streams, we label the communication links and assign directions to them. A stream in our model then, is a directed path in the array through cells and links having the same label. (The vertices in the directed path are the cells in the array and edges are the directed communication links.) To model the fixed velocity of tokens in a stream, we introduce a queue of buffers in the communication links. The buffer lengths are the same in all links with the same label and may differ in links with different labels.

The array is driven either by a single-phase or a two-phase global clock. A phase can be viewed as the instruction cycle of a cell. In a single-phase clocking scheme all cells are activated in every phase, whereas in two-phase clocking adjacent cells are activated during opposite phases of the clock. Cells execute their instructions during their activated phase.

Fig. 2 illustrates our linear-array model. The array has four streams labeled $l_1$, $l_2$, $l_3$, and $l_4$. $I_{1k}$/$O_{1k}$, $I_{2k}$/$O_{2k}$, and $I_{3k}$/$O_{4k}$ are the external input/output ports for the three streams $l_1$, $l_2$, and $l_4$, respectively, through which tokens are inserted into the array.

We now capture our linear-array model (described intuitively earlier) formally as follows. A linear array is a 3-tuple $A_r = (n, L_{A_r}, \Psi_{A_r})$ where:

1) $N$ is a sequence of identical cells with indices ranging from $1, 2, \cdots, N$.
2) $L_{A_r} = \{l_1, l_2, \cdots, l_k\}$ is a set of labels.
3) Every cell has $k$ pairs of input/output ports and each such pair is assigned a unique label from $L_{A_r}$.
4) $\Psi_{A_r}$ is the $k$-ary function computed by every cell in its active phase.

We next formally capture the notion of a stream which was intuitively described earlier as a directed path in the array and also the notion of tokens traveling in a stream at a constant velocity. Thus, a linear array has the following communication features.

1) If a cell communicates with other cells or with itself through a link labeled $l_j$ then a neighborhood constant $n_{l_j}$, which is only one of $\{1, -1, 0\}$, is associated with $l_j$ such that the outport labeled $l_j$ of any cell $s$ is connected by a link labeled $l_j$ to the input port labeled $l_j$ of cell $s + n_{l_j}$. For instance, in Fig. 2, $n_{l_1} = 1$ as every cell communicates with their neighbors to the right through links labeled $l_1$ and $n_{l_2} = -1$ as they use links labeled $l_2$ to communicate with their neighbors to the left. $n_{l_4} = 0$ as cells communicate with themselves through links labeled $l_4$. However, links labeled $l_4$ are only used for communication with the host.

2) A nonzero, positive delay constant $d_{l_j}$ is associated with every link labeled $l_j$ such that the $j$th component $O_j$ of the output of cell $s + n_{l_j}$ is the $j$-tuple input/output computed by cell $s$ in clock cycle $t$ then the $j$th component $O_j$ of this output is available at the input port labeled $l_j$ of cell $s + n_{l_j}$ in clock cycle $t + d_{l_j}$.

For instance, $\square$ and $\lozenge$ denote delay constants $d_{l_1}$ and $d_{l_2}$ on links labeled $l_1$ and $l_2$ respectively.

3) External communication with the host takes place through certain ports designated as follows:

If $n_{l_3} = 1$ or $n_{l_4} = -1$ then the ports labeled $l_j$ at either ends communicate with the host ($I_{1k}/O_{1k}$, $I_{2k}/O_{2k}$ in Fig. 2).

If $n_{l_2} = 0$ then a register in every cell serves as its port labeled $l_j$. A value is preloaded in this register before starting the computation and the result value (the preloaded value may be updated as computation progresses) is retrieved from this register after the computation terminates. The host cannot modify the contents of any of these registers when the computation is in progress.

Lastly, if no cell uses any link labeled $l_j$ to communicate with any other cell (like $I_{2k}/O_{4k}$ in Fig. 2) then each of these "vertical" ports labeled $l_j$ serves as an external port. A token inserted by the host into any such port in a clock tick is used by the cell for computation in that tick alone and is then discarded at the end of the tick.

B. Program Model

Our model of a program is motivated by two primary considerations. First, since such programs are to be transformed into array algorithms, it must enhance an algorithm designer's intuition. Second, for characterization purposes, it must be amenable to formal analysis.

Graphs provide a great deal of intuitive insight into the underlying system they represent and are also interesting combinatorial objects in their own right. For these reasons and also because a computation typically exhibits data dependencies we use DAG's (directed acyclic graphs) as high-level descriptions of a computation. The matrix-multiplication algorithm recently developed by the first author [25] mainly resulted from the insight developed by representing the data dependencies in the multiplication problem as a DAG. The vertices in the DAG typically denote functions to be computed and a directed edge between a pair of vertices denotes that a value computed by the function represented by the vertex at the edge bottom is required in computing the function represented by the vertex at the edge top.

Intuitively, a transformation of a DAG into a linear-array algorithm is concerned with assigning the vertices in the DAG to the cells in the array at appropriate clock ticks. Such an assignment must be performed within the communication
constraints imposed by the array. The “homogeneity” of the cells in our model impose certain restrictions on the types of DAG’s that are appropriate for execution on it. We are therefore led to the following formalization of programs appropriate for execution on linear arrays.

A homogeneous program graph $G = (V, E, L_G)$ is a labeled DAG where:

1. $V = V_G \cup SO_G \cup SI_G$ and $V_G$, $SO_G$ and $SI_G$ are three disjoint sets of vertices with $SO_G$ the set of source vertices, $SI_G$ the set of sink vertices, and $V_G$ the set of remaining vertices, which we shall call computation vertices.

2. $L_G$ is a set of labels and let $|L_G| = k$.

3. Every vertex in $V_G$ has $k$ incoming edges and $k$ outgoing edges and each pair of incident and outgoing edges is assigned a unique label from $L_G$. A source vertex has only one outgoing edge (henceforth referred to as an input edge) that is directed from it into some computation vertex and has no incoming edges. Similarly, a sink vertex also has only one incoming edge (henceforth referred to as an output edge) that is directed from it to some computation vertex and has no outgoing edges.

In any execution of $G$ on a linear array, every computation vertex in $G$ is a single instance of a function computed by a cell in a clock tick. Recall that all these cells execute identical straight-line programs and hence the functions represented by the computation vertices must also be identical straight-line programs. We can view the $k$ incoming edges of a computation vertex as representing the $k$-tuple input value that is required to compute the function represented by it and the $k$ outgoing edges as the $k$-tuple output value that is produced when the function is evaluated. A source vertex therefore denotes an input value and a sink vertex denotes an output value. As every computation vertex represents the same function we refer to these program graphs as homogeneous graphs.

Fig. 3 illustrates a homogeneous graph. The solid and dashed horizontal edges are labeled $l_1$ and $l_2$. The vertical edges are labeled $l_3$. The “doubly-linked” oblique edges are labeled $l_4$. X’s denote source or sink vertices and O’s denote computation vertices.

Although homogeneous graphs are a limited class of program graphs, they are useful in describing a number of interesting computations for VLSI arrays. These include sorting, merging, priority-queue operations, FIR filtering, DFT, linear recurrences, multidimensional convolution, matrix computations, dynamic programming, transitive closure and relational database operations. We will henceforth assume that $G$ is a homogeneous graph.

C. Mapping Homogeneous Graphs

We now precisely formulate correct mapping of homogeneous graphs. Intuitively, mapping of $G$ onto a linear array assigns each computation vertex of $G$ to a cell in the array at a particular time step and also assigns neighborhood and delay constants for the labels in the graph. In any mapping, the function computed by a cell in any tick must be identical to the function represented by any computation vertex in the graph. Assuming discrete time steps, let $T = \{0, 1, 2, \cdots\}$ be the sequence of natural numbers representing the progress of a computation from its start at tick 0.

A mapping of $G$ onto a linear array $A_r$ is a 4-tuple $(PA, TA, NA, DA)$ where:

1. $PA: V_G \rightarrow N$ and $TA: V_G \rightarrow T$ are many-one functions mapping computation vertices onto cells and time steps, respectively.

2. Let $I^*$ be a set of positive nonzero integers. Then, $NA: L_G \rightarrow \{1, -1, 0\}$ and $DA: L_G \rightarrow I^*$ are many-one functions assigning neighborhood and delay constants, respectively, to labels.

We denote $NA(lj)$ and $DA(lj)$ as $n_l$ and $d_l$ respectively. We are now ready to formalize a correct mapping.

Definition 1: A mapping is syntactically correct iff

1. for any label $lj$ and for any pair of computation vertices $v_i$ and $v_j$, if there is an edge labeled $lj$ directed from $v_i$ to $v_j$ then $PA(v_j) = PA(v_i) + n_l$ and $TA(v_j) = TA(v_i) + d_l$,

2. no two tokens can appear simultaneously at the same input port of any cell.

The first condition for correctness of mapping formally captures the two intuitive constraints imposed by our model, namely, that only neighboring cells communicate with one another (hence adjacent vertices are mapped on neighboring cells) and that tokens travel at a constant velocity in a stream (hence the time difference when adjacent vertices are mapped is a constant that depends on the label of the edge connecting them).

A final concern is our notion of correct execution of homogeneous graphs. Our motivation for such a distinction between correct mapping and execution is best illustrated by the following scenario depicted in Fig. 4.

In Fig. 4, $v_i$ is a computation vertex and $s_1$ is a source vertex linked to $v_i$ by an edge labeled $l$. Let the neighborhood and delay constants associated with label $l$ be 1 and $d_l$ respectively. Let $v_i$ be mapped onto cell $p$ at tick $t_i$. In order to compute the function represented by $v_i$, cell $p$ requires the input value, say, $i$ denoted by the source $s_1$. A token initialized to $i$ must therefore be inserted into the external port $I_i$ at tick $t - pd_l$. This token traverses cells $1, 2, \cdots, p - 1$ through links labeled $l$ before reaching its final destination (cell $p$) where it is consumed. Recall that our linear array has no control signals flowing through it. Such signals aid the cells in selectively updating tokens passing through them. In the absence of such signals, the initial value of the token inserted into $I_i$ at tick $t - pd_l$ is likely to have changed when it reaches cell $p$. For correctness of execution, we therefore require that the initial value of this token when it was inserted at $I_i$ (its entry time) and its value at cell $p$ (its consumption...
time) must remain the same. Analogously, we also require that the final value of the token (denoting the output value represented by $s_j$) that is computed by cell $q$ when $v_j$ is mapped on it at tick $t_2$ (the token’s production time) and its value when it emerges from $O_i$ (its exit time) be identical.

We are now in a position to introduce the notion of correct execution of homogeneous graphs.

**Definition 2:** $G$ is correctly executed on a linear array if and only if the following conditions hold:

1) The mapping is syntactically correct.

2) Every token’s value at its entry (exit) and consumption (production) times must be identical.

### III. SYNTACTIC CHARACTERIZATION

We will now identify the structure of homogeneous graphs for which there exist syntactically correct mappings on linear arrays. We will then construct linear-array algorithms for such graphs.

For notational simplicity we will be using the following conventions. We will refer to computation vertices simply as vertices. Both directed and undirected paths (obtained by ignoring direction on edges) comprise of a sequence of distinct vertices (except in cycles). We will denote a directed path from $v_i$ to $v_n$ by $v_i, v_2, \cdots, v_{n-1}$ by $[v_i, v_j, \cdots, v_{n-1}, v_n]$ (for $[v_i, v_n]$ therefore denotes an edge directed from $v_i$ to $v_n$) and will refer to an undirected path simply as a path.

For a syntactic characterization we only need to require certain relevant structural elements of a homogeneous graph $G$.

**Definition 3:** For any label $lj$ in $G$, a major path labeled $lj$ is a directed path from a source vertex $v_i$ to a sink vertex $v_j$ such that the label of all edges in this path is $lj$.

The path label of a major path is the label of the edges in the path.

For any label $lj$, let $E_lj = \{major paths having the same path label lj\}$. Some of these $E_lj$’s so formed are not relevant for syntactic characterization. These are as follows:

First, we will not require $E_lj$’s solely comprised of major paths that have only one computation vertex in them ($E_lj$ in Fig. 3). For every source vertex in these “dangling” major paths, an input token initialized to the value represented by the source vertex is inserted into a cell from its “vertical” port (see Fig. 1). It is used by the cell in the tick it is inserted and then emerges from the cell at the end of the tick. We will see later on (in our mapping algorithm) how these ticks are determined.

Second, if some of these sets, say, $E_{l1}, E_{l2}, \cdots, E_{ln}$ are identical (that is, the major paths in each of them after excluding the source and sink vertices are the same) then we choose only one of them. Suppose we choose $E_{l1}$ from these

sets and we assign the value $u$ to the neighborhood constant $n_{l1}$ and the value $v$ to the delay constant $d_{l1}$ of label $lj$. We will then assign $u$ to each of $n_{l1}, n_{l2}, \cdots, n_{lj-1}, n_{lj+1}, \cdots, n_{ln}$ and $w$ to each of $d_{l1}, d_{l2}, \cdots, d_{lj-1}, d_{lj+1}, \cdots, d_{ln}$.

We will therefore assume that our homogeneous graph is comprised of nonidentical sets of major paths and that no set is made up of dangling major paths alone. For instance, in Fig. 3, $E_{l1}$ and $E_{l2}$ are identical. $E_{l1}$ is made up of dangling major paths alone. Thus, we will choose either $E_{l1}$ and $E_{l2}$ or $E_{l2}$ and $E_{l}$ for our structural analysis.

An orthogonal set of labels in such a homogeneous graph consists of labels such that the subgraph, induced by all the computation vertices along with all the edges that are labeled using these labels alone, is connected. Besides, removing all the edges of the same label disconnects the subgraph. Let $SG$ denote such a subgraph and $L_SG$ denote the orthogonal set of labels. Note, however, that the labels in $L_SG$ need not be unique.

We have now developed the appropriate machinery to undertake a systematic analysis of the structure of homogeneous graphs and we begin by examining graphs that have only one label in their orthogonal set. Let $\mu_l$ denote such a label. This means that there exists a path between any pair of vertices through edges labeled $\mu_l$. Since $G$ is homogeneous, there is only one such pair of incoming and outgoing edge labeled $\mu_l$ in any vertex. Consequently, there exists only one major path labeled $\mu_l$ in $G$ and mapping such a graph is straightforward.

### A. $\Theta$ Graphs

We next examine graphs that have only two labels in their orthogonal set. We will denote the class of such graphs as $\Theta$ graphs. $\Theta$ is a large class that includes computational problems like sorting [4], convolution, linear recurrence, filtering [13], vector operations on matrices [16], and pattern matching [7].

Let $G$ be a homogeneous graph in $\Theta$ and let $\mu_l$ and $\nu_l$ denote the two labels in its orthogonal set. So $L_SG = \{\mu_l, \nu_l\}$ and there exists a path between any pair of computation vertices in $G$ through edges that are either labeled $\mu_l$ or $\nu_l$.

The structure imposed on $SG$ by any correct mapping has a clean formalization which is as follows:

**Definition 4:** Let each computation vertex in $SG$ be assigned a unique coordinate in 2-D euclidean space. Let $(x_{\mu_l}, y_{\mu_l})$ and $(x_{\nu_l}, y_{\nu_l})$ be the coordinates of computation vertices $\mu_l$ and $\nu_l$, respectively. Then $SG$ is a mesh graph iff for any label $l \in \{\mu_l, \nu_l\}$, there exists a directed path of length $d$ from $v_{\mu_l}$ to $v_{\nu_l}$ comprised only of edges labeled $l$ iff $y_{\mu_l} - x_{\mu_l} = d$.

Fig. 5 is an example of a mesh graph wherein horizontal and vertical major paths are labeled $\mu_l$ and $\nu_l$, respectively. We can now relate the structure of $SG$ to the existence of a syntactically correct mapping as follows:

**Theorem 1:** Let $G \in \Theta$. If there exists a syntactically correct mapping for $G$ then $SG$ must be a mesh graph.

**Proof:** See Appendix.

When $G$ is finally mapped onto a linear array, the computation vertices in $G$ may be partitioned into disjoint sets that comprise vertices which are mapped onto the same cell.
We will see later on (in Theorem 2) that such a partition is useful in expressing the structure of correctly mappable graphs in a simple way. We define such a partitioning of vertices as follows. Recall that \( (x_{\mu}, y_{\mu}) \) and \( (y_{\mu}, y_{\mu}) \) denote the first (horizontal) and second (vertical) coordinates of vertices \( v_1 \) and \( v_2 \), respectively.

Definition 5: A diagonalization of \( SG \) is a plane which intersects vertices \( (a, b) \) and \( (a + 1, b + 1) \) or \( (a, b) \) and \( (a - 1, b + 1) \) or \( (a, b) \) and \( (a, b + 1) \) or \( (a, b) \) and \( (a + 1, b) \). Formally, let \( w_1 \) and \( w_2 \) be two constants such that the pair \( (w_1, w_2) \in \{1, 1, (1, -1), (0, 1), (1, 0)\} \). Then a diagonal is a line in such a plane and is comprised of vertices \( v_1 \) and \( v_2 \) such that \( w_1 x_{\mu} + w_2 y_{\mu} = w_1 y_{\mu} + w_2 x_{\mu} \). If \( v_i \) is in a diagonal then the weighted sum \( w_i x_{\mu} + w_2 y_{\mu} \) is the weight of the diagonal.

We assign consecutive indexes to the diagonals (obtained by diagonalization) in increasing order of their weights with the diagonal having the least index assigned index 1.

Let \( D = \{D_1, D_2, \ldots, D_m\} \) denote the set of ordered diagonals. Diagonal \( D_i \) in \( D \) is assigned the index \( i \). Note that \( D_1 \cup D_2 \cup \ldots \cup D_m = \{v\} \) and these diagonals are pair-wise disjoint. We will refer to \( D \) as the set of main diagonals and to the pair \((w_1, w_2)\) as the main diagonalization factor.

Fig. 6 illustrates the four sets of main diagonals that are obtained for the four different values of \((w_1, w_2)\). The vertices on a dotted line belong to the same diagonal.

We will show later on (in our mapping procedure) that all the vertices in diagonal \( D_i \) are mapped onto cell \( i \). We therefore do not permit \( w_1 \) and \( w_2 \) to be simultaneously zero as this would then create only one diagonal and we will have to map all these vertices onto one cell alone.

Our mapping procedure is also concerned with fixing the time steps at which the computation vertices have to be mapped onto the cells. Recall that the difference in time when adjacent vertices are mapped, is a constant that is dependent on the label of the edge connecting the vertices. Intuitively, such edges would correspond to lines of constant “slope” in the plane comprised of main diagonals that run parallel to the vertical axis. These slopes may differ for edges with different labels. In order to capture this notion of slopes formally, we require another “axis” orthogonal to the main diagonals. We therefore introduce orthogonal diagonals which is again a plane passing through vertices \((a, b)\) and \((a + 1, b)\) if the main diagonals do not pass through them. Otherwise they are planes passing through vertices \((a, b)\) and \((a + 1, b)\).

Formally, let \( Dc = \{Dc_1, Dc_2, \ldots, Dc_n\} \) denote the set of orthogonal diagonals obtained by diagonalizing with the orthogonal diagonalization factor \((0, 1)\) or \((1, 0)\). We select \((0, 1)\) if \((w_1, w_2) \neq (0, 1)\) else we pick \((1, 0)\) as the factor. The orthogonal diagonals for Fig. 5 are shown in Fig. 7 below.

We are now ready to provide a complete syntactic characterization of graphs in \( \Theta \) that have a correct mapping.

First, we divide the graphs in \( \Theta \) into two classes \( \Theta_1 \) and \( \Theta_2 \). Graphs in \( \Theta_1 \) are diagonalized with \((w_1, w_2) \in \{(1, 0), (0, 1), (1, -1)\}\) and those in \( \Theta_2 \) with \((w_1, w_2) = (1, 1)\).

Second, let \( v_s \) and \( v_t \) be two computation vertices that lie on main diagonals whose indexes are \( p \) and \( q \), respectively, and on orthogonal diagonals whose indexes are \( s \) and \( r \), respectively. Let \( \Delta_0(v_s, v_t) \) denote \( q - p \) and \( \Delta_p(v_s, v_t) \) denote \( r - s \).

Theorem 2: Let \( G \in \Theta_1 \). Then, there exists a syntactically correct mapping for \( G \) if and only if there exists a set \( D \) of main diagonals and a set \( Dc \) of orthogonal diagonals such that each of the following conditions is satisfied.

1) If there is an edge labeled \( lj \) directed from \( v_i \) to \( v_j \) then \( \Delta_0(v_i, v_j) \) must be a constant which is one of \( \{1, -1, 0\} \). Let \( m_{ij} \) denote such a constant.

2) If there is an edge labeled \( lj \) directed from \( v_i \) to \( v_j \) then \( \Delta_p(v_i, v_j) \) must be a constant. Let \( c_{ij} \) denote such a constant.

3) For any label \( lj \), if \( c_{ij} \Delta_0(v_i, v_j) = m_{ij} \Delta_p(v_i, v_j) \) then there must be a major path labeled \( lj \) passing through \( v_s \) and \( v_t \).

Intuitively, if vertices on consecutively indexed main diagonals are mapped onto neighboring cells then adjacent vertices must be on neighboring main diagonals. Condition 1 captures such a requirement.

Recall that in order for tokens to travel at a constant velocity in a stream, edges must be lines of constant slope in the plane comprised of main diagonals that are parallel to the vertical axis and orthogonal diagonals that are parallel to the horizontal axis. Identically labeled edges become lines of constant slopes in such a plane if the vertices they connect lie on main and orthogonal diagonals whose indexes differ by a constant. Condition 1 ensures that adjacent vertices lie on neighboring main diagonals whose indexes differ by a constant. We therefore require that they also lie on orthogonal diagonals whose indexes also differ by a constant and this is captured by Condition 2.

Lastly, two or more major paths with the same path label cannot lie on the same line in the plane. Otherwise, tokens
(whose values denote those represented by these paths) will appear simultaneously at the same input port of every cell. Condition 3 insists that major paths lying on the same line and having identical labels must be a single continuous path. We sketch the construction used in the sufficiency proof as it will be used to synthesize linear-array algorithms in our examples.

Proof: (Only If): see Appendix for details.

(If Part): Let \( D = \{ D_1, D_2, \ldots, D_n \} \) be the set of main diagonals where \( i \) denotes the index of diagonal \( D_i \). Construct a linear array \( L_n \) with \( n \) cells. Next construct a mapping as follows.

1) If there exists a transitive edge (if \( (v_i, v_j), (v_j, v_k) \) and \( (v_j, v_l) \) are directed edges then the edge \( (v_i, v_l) \) is a transitive edge) labeled \( l_j \) such that \( m_{i,j} = 0 \) then choose two-phase clocking else select a single-phase clocking scheme.

2) Let \( v_i \) be a computation vertex that lies on diagonal \( D_i \). Then map \( v_i \) on cell \( q \), that is let \( PA(v_i) = q \). This assigns computation vertices onto cells.

3) Next assign delay and neighborhood constants to the labels as follows. Let \( n_0 = m_0 \) and let \( d_a \) and \( d_b \) be two constants. If \( (w_1, w_2) = (1, -1) \) or there exists a transitive edge labeled \( l_j \) such that \( m_{i,j} = 0 \) then choose \( d_a \) to be 2 else let it be 1. Let \( c_{\min} \) be the minimum among all the \( c_{y_i} \)'s. If \( c_{\min} > 0 \) then choose \( d_b \) to be 1 else let \( d_b = 1 + |c_{\min}|d_a \). Finally, let \( d_0 = m_0d_a + c_i d_a \).

4) Construct the function \( TA \) which assigns computation vertices to time steps. Let \( v_i \) be the computation vertex that lies on main and orthogonal diagonals whose indexes are 1. Let \( TA(v_i) = t_0 \). Let \( v_i \) lie on the main diagonal indexed \( p \) and orthogonal diagonal indexed \( q \). Then, let \( TA(v_i) = t_0 + (q - 1)d_a + (p - 1)d_b \).

Steps 1–4 described above completes the construction of a correct mapping (see the Appendix for its proof of correctness). The three conditions of Theorem 2 are necessary but not sufficient for the existence of syntactically correct mappings for graphs in \( \Theta_2 \). However, in the next theorem we show that in certain cases it is both necessary and sufficient. Let \( C \) denote the set of all \( c_{y_i} \)'s excluding those for the labels \( l_0 \) and \( l_v \) in the orthogonal set.

Theorem 3: If all the \( c_{y_i} \)'s in \( C \) are positive or all of them are negative then there exists a syntactic: lly correct mapping for graphs in \( \Theta_2 \) if and only if the three conditions in Theorem 2 are satisfied.

Proof: Similar to Theorem 2 except in the construction of the delays for the labels. If \( c_{y_i} > 0 \) then set \( d_a = 2, d_b = 1, d_{l_a} = 1 \) and \( d_{l_v} = 3 \). If \( c_{y_i} < 0 \) then set \( d_a = -2, d_b = 3, d_{l_a} = 3 \) and \( d_{l_v} = 1 \). In the Appendix we have shown that such a construction always yields nonzero and positive delays for labels.

The construction used in the proof of Theorem 2 is the transformation technique that converts homogeneous graphs in \( \Theta \) into linear-array algorithms. Recall that for correct execution of a homogeneous graph we are required to ensure that every token’s value at its entry (exit) and consumption (production) time must be identical. Thus, in Fig. 4, the input token (initialized to \( i \)) that is inserted at \( t = pd_i \) into \( l_i \) must arrive unchanged at cell \( p \) after traversing cells 1, 2, \( \cdots, p - 1 \) through links labeled \( l \). Control signals, which could aid cells in selectively updating tokens passing through it, are absent and hence the value of this token when it reaches cell \( p \) is dependent on the function computed by the cells. In particular, suppose the array in Fig. 4 is used to compute the multiplication of a band matrix by a vector (see Fig. 1) and the \( y_i \)'s traverse through links labeled \( l \). Then, by inserting the value 0 through the vertical ports of each of the cells 1, 2, \( \cdots, p - 1 \) during those times when this token passes through them, we can ensure that its initial value \( i \) remains unchanged when it reaches cell \( p \).

In general, we will be required to use some properties of the function computed by the cells along with the timing information generated by the mapping in order to design a correct execution of the graph.

In [23] we have shown that without using any such properties, graphs having only one major path alone can be executed correctly.

We begin our transformation of a graph in \( \Theta \) into an array algorithm by choosing an appropriate diagonalization factor. First, such a choice should result in main and orthogonal diagonals that satisfy the conditions in Theorem 2 or Theorem 3. Second, if there exists more than one such diagonalization factor then our choice will be influenced by other factors including minimizing the number of cells, increasing throughput rate and decreasing response time. We will illustrate the influence of such criteria in our examples.

We now apply the results described above to synthesize linear-array algorithms for computing the vector multiplication of band matrices, convolution and sorting.

Example 1: Consider multiplication of a band matrix \( M \) by a vector \( X \) as shown below.

\[
\begin{bmatrix}
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5 \\
x_6 
\end{bmatrix} =
\begin{bmatrix}
\alpha_{11} & \alpha_{12} \\
\alpha_{21} & \alpha_{22} & \alpha_{23} \\
\alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\
\alpha_{42} & \alpha_{43} & \alpha_{44} & \alpha_{45} & \alpha_{46} \\
\alpha_{53} & \alpha_{54} & \alpha_{55} & \alpha_{56} & \alpha_{57} & \alpha_{58} \\
\alpha_{64} & \alpha_{65} & \alpha_{66} & \alpha_{67} & \alpha_{68} & \alpha_{69} & \alpha_{70} 
\end{bmatrix}
\begin{bmatrix}
y_1 \\
y_2 \\
y_3 \\
y_4 \\
y_5 \\
y_6 
\end{bmatrix}
\]

Fig. 8 is a program graph representing this computation. In Fig. 8, \( v_0 \) denotes a computation vertex. The horizontal, vertical, and oblique edges are labeled \( l_1, l_2 \) and \( l_3 \), respectively. If \( a, b, \) and \( c \) are the input values represented by the horizontal, vertical, and oblique input edges of \( v_i \), then the output values represented by the outgoing horizontal, vertical, and oblique edges of \( v_i \) are the inner-product step \( a + bc, b, \) and \( c \), respectively. Each horizontal source vertex denotes the value 0. Let \( E_H = \{ \text{horizontal major paths} \} \), \( E_V = \{ \text{vertical major paths} \} \), and \( E_O = \{ \text{oblique major paths} \} \). The dangling major paths in \( E_O \) are not relevant for our mapping as they all have only one computation vertex. It can be verified that \( \{ l_1, l_2 \} \) constitute the labels in the orthogonal set and that the graph (without the major paths in \( E_O \)) is in \( \Theta \). The connected component \( SG \) comprised of edges labeled \( l_1 \) and \( l_2 \) is shown in Fig. 9 below. (In fact, \( SG \) is the same as the graph without the major paths in \( E_O \).)
For purposes of clarity SG has been drawn without the source and sink vertices. Now diagonalize SG by the pair \((1, -1)\) to form the set of main diagonals \(D\). It can be verified that \(D\) is comprised of the four diagonals \(D_1, D_2, D_3,\) and \(D_4\) where

\[
D_1 = \{v_{31}, v_{42}, v_{53}, v_{64}\}, \quad D_2 = \{v_{21}, v_{32}, v_{43}, v_{54}, v_{65}\}, \\
D_3 = \{v_{11}, v_{22}, v_{33}, v_{44}, v_{55}\}, \quad D_4 = \{v_{12}, v_{23}, v_{34}, v_{45}\}.
\]

Next, diagonalize by the pair \((0, 1)\) to form the set \(D_c\) of orthogonal diagonals. It can be verified that \(D_c\) is comprised of the six diagonals \(D_{c1}, D_{c2}, D_{c3}, D_{c4}, D_{c5},\) and \(D_{c6}\) where

\[
D_{c1} = \{v_{11}, v_{12}\}, \quad D_{c2} = \{v_{21}, v_{22}, v_{23}\}, \quad D_{c3} = \{v_{31}, v_{32}, v_{33}\}, \\
D_{c4} = \{v_{42}, v_{43}, v_{44}\}, \quad D_{c5} = \{v_{53}, v_{54}, v_{55}\}, \quad D_{c6} = \{v_{64}, v_{65}\}.
\]

In Fig. 9 all the computation vertices belonging to the same diagonal in \(D\) lie on the same dashed line. Similarly, all the computation vertices belonging to the same diagonal in \(D_c\) lie on one horizontal major path.

Observe that \(m_{l1} = 1, m_{l2} = -1, c_{l1} = 0\) and \(c_{l2} = 1\) and that the graph satisfies Theorem 2. Next, using the construction in Theorem 2 we synthesize the linear-array algorithm in [16] (illustrated in Fig. 1). \(|D| = 4\) and hence the linear array has 4 cells indexed from 1 to 4. The computation vertices in \(D_1, D_2, D_3,\) and \(D_4\) are mapped onto cells 1, 2, 3, and 4, respectively. The neighborhood and delay constants are as follows: \(n_{l1} = 1, n_{l2} = -1, d_{l1} = 1,\) and \(d_{l2} = 1\). Recall (see the beginning of Section III) that since \(E_0\) has only dangling major paths, we associate a vertical port with each cell through which the tokens initialized to the value represented by each source (sink) vertex in \(E_0\) are inserted (extracted).

Let \(a, b,\) and \(c\) denote the inputs at the input ports labeled \(l1, l2,\) and \(l3,\) respectively, of a cell \(s\) in the array at some tick, say, \(t\). Then the output computed by this cell at \(t\) is the triple \((a + bc, b, c)\). Mapping of the graph in its entirety is shown in Fig. 10. The time at which a computation vertex is mapped is indicated by the side of the vertex in Fig. 10. For instance, the computation vertex on \(D_3\) and \(Dc_2\) is mapped at tick \(t + 2\).

Observe that only those tokens that traverse the array through links labeled \(l1\) are subject to updates. Let \(i_1\) and \(i_2\) denote the tokens initialized to the values represented by the source vertices \(ih_1\) and \(ih_2\), respectively. Similarly, let \(o_1\) and \(o_2\) be the tokens whose final values denote those represented by the sink vertices \(oh_3\) and \(oh_4\), respectively.

\(i_1\) and \(i_2\) are the only tokens not consumed at cell 1. They are however inserted into cell 1 and hence we need to preserve their values till they reach their final destination where they will be consumed (cell 3 for \(i_1\) and cell 2 for \(i_2\)). To do this we need the times at which they traverse the intermediate cells. Table I gives the times at which \(i_1\) appears at the input port labeled \(l1\) of cells 1 and 2 and \(i_2\) appears at the input port labeled \(l1\) of cell 1.

Consider some row of Table I, say, 2. The entry in column 1 indicates that \(i_2\) appears at the input port labeled \(l1\) of cell 1 at tick \(t\). Observe that a cell computes an inner-product step, and so by inserting 0 into the vertical port of cell 1 at tick \(t\) the invariance of \(i_2\)'s value at its entry and consumption time can be maintained. Similarly, by inserting 0 into the vertical port of cell 1 at tick \(t - 2\) and that of cell 2 at tick \(t - 1\) invariance of \(i_1\)'s value at its entry and consumption times can also be maintained.

\(o_1\) and \(o_2\) are the only tokens whose final values are not produced in cell 4. Hence, we are again required to preserve their values till they reach cell 4. This is done as follows. The production times for \(o_1\) and \(o_2\) are \(t + 9\) and \(t + 10\), respectively. Table II gives the times at which \(o_1\) appears at the input port labeled \(l1\) of cell 4 and \(o_2\) appears at the input ports labeled \(l1\) of cells 3 and 4.

The entries in Table II are interpreted in the same way as the entries in Table I. From Table II it is seen that by inserting 0 into the vertical ports of cell 3 at tick \(t + 10\) and cell 4 at \(t + 9\) and \(t + 1\) we can preserve the final values of tokens \(o_1\) and \(o_2\) till they emerge from cell 4.

Note that we need not be concerned about those tokens that traverse through links labeled \(l2\) as they are never updated by any cell.

Another closely related problem which requires cells that compute an inner-product step is the convolution problem defined as follows. Given the sequence of weights \(\{w_1, w_2, \cdots, w_k\}\) and the input sequence \(\{x_1, x_2, \cdots, x_r\}\) compute the output sequence \(\{y_1, y_2, \cdots, y_{n+1-k}\}\) defined by

\[
y_t = \sum_{j=1}^{n+1-k} w_j x_{t+j-1}.
\]

We illustrate the convolution problem on \(n = 5\) and \(k = 3\). The computation of the convolution problem for \(n = 5\) and \(k = 3\) is represented by the program graph of Fig. 11.

The horizontal, vertical and oblique edges are labeled \(l1, l2,\) and \(l3,\) respectively. If \(a, b,\) and \(c\) are the input values represented by the horizontal, vertical, and oblique input edges of \(v_y\) then the output values represented by the out-
going horizontal, vertical, and oblique edges of \( v_{ij} \) are \( a + bc, b, \) and \( c \), respectively.

Let \( E_\text{h} = \{ \text{horizontal major paths} \} \), \( E_\text{v} = \{ \text{vertical major paths} \} \), and \( E_\text{d} = \{ \text{oblique major paths} \} \). Observe that any pair of labels can be chosen to form the orthogonal pair. Let us choose \( l1 \) and \( l2 \). Then the connected component \( SG \) that is formed by removing all the edges labeled \( l3 \) is shown in Fig. 12.

For purposes of clarity again, \( SG \) has been drawn without the source and sink vertices. It can be seen that the program graph in Fig. 11 is in \( \Theta \). Now diagonalize \( SG \) by \((1, 0)\) to form the set \( D \) of main diagonals. \( D \) is comprised of three diagonals \( D_1, D_2, \) and \( D_3 \) where \( D_1 = \{ v_{11}, v_{21}, v_{31} \} \), \( D_2 = \{ v_{12}, v_{22}, v_{32} \} \), and \( D_3 = \{ v_{13}, v_{23}, v_{33} \} \). Next diagonalize \( SG \) using \((0, 1)\) to form the set \( Dc \) of orthogonal diagonals. \( Dc \) is also comprised of three diagonals \( Dc_1, Dc_2, \) and \( Dc_3 \) where \( Dc_1 = \{ v_{11}, v_{12}, v_{13} \} \), \( Dc_2 = \{ v_{21}, v_{22}, v_{23} \} \), \( Dc_3 = \{ v_{31}, v_{32}, v_{33} \} \). In Fig. 12 all computation vertices belonging to a single diagonal in \( D \) lie on the same vertical major path. Similarly, all vertices belonging to a single diagonal in \( Dc \) lie on the same horizontal major path.

Observe that \( m_1 = 1, m_2 = 0, m_3 = -1, c_1 = 0, c_2 = 1, \) and \( c_3 = 1 \). It can be verified that Theorem 2 is satisfied by the graph. We next synthesize the linear-array algorithm in [13]. \(|D| = 3\) and hence the linear array has 3 cells indexed from 1 to 3. Using the construction in Theorem 2, we obtain \( n_1 = 1, n_2 = 0, \) and \( n_3 = 1 \). We also obtain \( d_1 = 1, \) and \( d_3 = 1 \). The computation vertices in \( D_1, D_2, \) and \( D_3 \) are mapped onto cells 1, 2, and 3, respectively.

\( m_2 = 0 \) and there exist transitive edges labeled \( l2 \). Hence, we use two-phase clocking to drive the array. Each cell has two pairs of I/O ports labeled \( l1 \) and \( l3 \). A register in each cell serves as the I/O port labeled \( l2 \). The values represented by the source \( iv_1, iv_2, \) and \( iv_3 \) are preloaded into the registers of cells 1, 2, and 3, respectively. The resulting mapped graph is shown in Fig. 13.

The tokens in the streams labeled \( l2 \) and \( l3 \) are never updated by any cell. Besides, since the tokens traveling in the stream labeled \( l1 \) are all consumed in cell 1, their entry and consumption times are identical. Similarly, their production and exit times are identical as they are all produced in cell 3.

Suppose we had picked \((1, 1)\) as the diagonalization factor for the convolution problem. In such a case we will obtain 5 main diagonals and hence we will be required to use 5 cells compared to the 3 cells required in the above solution.

**Example 2:** We wish to sort the set of elements \( \{2, 10, 5, 6\} \). A program graph that performs sorting is shown in Fig. 14. Each computation vertex represents the computation of the minimum and maximum of the two input elements denoted by the incoming horizontal and vertical edges. The outgoing horizontal and vertical edges denote the minimum and maximum, respectively. The horizontal edges are labeled \( l1 \) and the vertical edges are labeled \( l2 \). The initial values represented by the source vertices \( ik_1, ik_2, ik_3, \) and \( ik_4 \) are \( 2, 10, 5, \) and \( 6, \) respectively, and the initial values represented by the source vertices \( ih_1, ih_2, ih_3, \) and \( ih_4 \) are \( \infty \). It can be verified that the final values represented by the sink vertices \( oh_1, oh_2, oh_3, \) and \( oh_4 \) are \( 2, 5, 6, \) and \( 10, \) respectively.

We synthesize the algorithm popularly known as the “rebound sorter” [4].

Let \( E_\text{h} = \{ \text{horizontal paths} \} \) and \( E_\text{v} = \{ \text{vertical paths} \} \). This graph belongs to \( \Theta \) and \( \{l1, l2\} \) constitute its orthogonal set. Now use \((1, -1)\) to form the main diagonals. Let \( D_1 = \{v_{11}, v_{22}, v_{33}, v_{44}\}, D_2 = \{v_{12}, v_{23}, v_{34}\}, D_3 = \{v_{13}, v_{24}\} \) and \( D_4 = \{v_{14}\} \). Next use \((0, 1)\) to form the orthogonal diagonals. Let \( Dc_1 = \{v_{11}, v_{12}, v_{13}, v_{14}\}, Dc_2 = \{v_{21}, v_{22}, v_{23}\}, Dc_3 = \{v_{31}, v_{32}\}, \) and \( Dc_4 = \{v_{41}\} \). Observe that \( m_1 = 1, m_2 = -1, c_1 = 0 \) and \( c_2 = 1 \). We next map this graph as follows. First, we choose \( n_1 = 1, n_2 = 1, d_1 = 1 \) and \( d_2 = 1 \). Next the computation vertices in \( D_1, D_2, D_3, \) and \( D_4 \)
are mapped onto cells 1, 2, 3, and 4, respectively. The resulting mapped graph is shown in Fig. 15.

Each cell is comprised of 2 pairs of I/O ports labeled I1 and I2, respectively. Let a and b denote the inputs at the ports labeled I1 and I2, respectively, of cell s at tick t. Then the output computed by cell s is the pair \(\langle \min(a, b), \max(a, b) \rangle\).

Let \(i_1, i_2, i_3, \) and \(i_4\) denote the tokens initialized to 2, 10, 6, and 5, respectively. Similarly, let \(o_1, o_2, o_3, \) and \(o_4\) denote the tokens whose final values are 2, 5, 6, and 10, respectively. The tokens \(i_1, i_2, i_3, \) and \(i_4\) are inserted from cell 4 and we need to preserve their initial values as they traverse through intermediate cells to their destination cell where they are consumed. Similarly, the final values in each of the tokens \(o_1, o_2, o_3, \) and \(o_4\) must again be preserved without any changes till they emerge from cell 4. To do so, we proceed as follows. Tables III and IV are the times at which these tokens travel through intermediate cells before reaching their final destination.

Consider some row, say row 2 in Table III and Table IV. The entries \(t + 6\) and \(t + 7\) in columns 3 and 4 of Table IV denote the ticks at which \(o_3\) appears at the input port labeled I1 of cells 3 and 4, respectively. Similarly, the entries \(t + 6\) and \(t + 7\) in columns 3 and 4 of Table IV denote the ticks at which \(o_3\) appears at the input port labeled I1 of cells 3 and 4, respectively. Note that \(\min(x, -\infty) = -\infty\) and \(\max(x, \infty) = \infty\). Using this property we will now construct a correct execution of the sorting graph. Consider row 2 of Table III and Table IV again. If \(-\infty\) appears at the input port labeled I1 of cells 3 and 4 at ticks \(t - 1\) and \(t - 2\), respectively, then the value of \(i_2\) is preserved. Similarly, if \(\infty\) appears at the input port labeled I2 of cells 3 and 4 at ticks \(t + 6\) and \(t + 7\), respectively, then the final value of \(o_3\) is preserved. For every entry in Table IV we compute the times at which \(-\infty\) must be inserted into \(i_1\) and this is tabulated in Table V. Similarly, for every entry in Table IV we compute the times at which \(\infty\) must be inserted into \(i_2\) and this is tabulated in Table VI.

Consider some row, say, row 2, in Table V and Table VI. The entry \(t - 4\) in column 3 of Table V indicates that for \(-\infty\) to appear at the input port labeled I1 of cell 3 at tick \(t - 2\), it must be inserted into \(i_1\) at time \(t - 4\). Similarly, the entry \(t + 5\) in column 3 of Table VI indicates that for \(\infty\) to appear at the input port labeled I2 of cell 3 at tick \(t + 6\), it must be inserted into \(i_2\) at time \(t + 5\). From Table V we observe that it suffices to insert \(-\infty\) into \(i_1\) at ticks \(t - 6, t - 4,\) and \(t - 2\). Similarly, from Table VI we observe that is suffices to insert \(\infty\) into \(i_2\) at ticks \(t + 5, t + 7,\) and \(t + 9\).

**B. Cube Graphs**

A natural generalization of graphs in \(\Theta\) are homogeneous graphs that have more than two labels in their orthogonal set. Important computational problems like matrix multiplication [16], \(lu\)-decomposition of matrices [16] and operations on relational databases [15] can be described by such graphs. A complete characterization of such graphs seems difficult and we will now examine a subset of such graphs and describe a technique to map such graphs onto a linear array. Let \(\{I_1, I_2, I_3\}\) be the three labels in a homogeneous graph \(G\) which are also the labels in its orthogonal set.

**Definition 6:** Let each computation vertex in \(G\) be assigned a unique coordinate in 3-D Euclidean space. Let \(\langle x_1, y_1, z_1 \rangle\) and \(\langle x_2, y_2, z_2 \rangle\) be the coordinates of computation vertices \(v_i\) and \(v_j\), respectively. Then \(G\) is a cube graph iff for every label \(l_j\) \((j = 1, 2, 3)\) there exists a directed path of length \(d\) from \(v_i\) to \(v_j\), comprised only of edges labeled \(l_j\) iff \(y_j - x_j = d\).

Analogous to mesh graphs we define diagonalization of cube graphs as follows. Let the diagonalization factor of a
cube graph be the triple \((w_1, w_2, w_3) \in \{(1, 1, 1), (1, 1, -1), (1, -1, 1), (1, -1, -1)\}.

**Definition 7**: A diagonalization of a cube graph is a hyperplane which intersects vertices \((a, b, c)\) and \((a + 1, b + 1, c + 1)\) or \((a, b, c)\) and \((a + 1, b - 1, c + 1)\) or \((a, b, c)\) and \((a + 1, b + 1, c - 1)\). A diagonal is a plane and is comprised of vertices \(v_i\) and \(v_j\) such that \(w_1x_1 + w_2x_2 + w_3x_3 = w_1y_1 + w_2y_2 + w_3y_3\). If \(v_i\) is in a diagonal then the weighted sum \(w_1x_1 + w_2x_2 + w_3x_3\) is the weight of the diagonal.

Let us choose some arbitrary label from among the three labels and denote it as \(lj\). Next remove all the edges labeled \(lj\) from the cube graph and let \(MG = \{MG_1, MG_2, \cdots, MG_a\}\) be the set of connected components so formed. Observe that these \(MG_i\)'s are mesh graphs. We next combine the mesh graphs in \(MG\) into classes as follows. Let \(CG = \{CG_1, CG_2, \cdots, CG_b\}\) be a family of sets of mesh graphs such that \(CG_i = \{MG_i \in MG \mid \text{if } v_i \text{ is a computation vertex in } MG_i \text{ then its } j\text{th component of coordinate is } i\}.

We will now describe the algorithm to map a cube graph onto a linear array. Without loss of generality, let \(lj = l3\). So the mesh graphs in \(CG\) are comprised of edges whose labels are either \(l1\) or \(l2\). First pick some diagonalization factor and let \(B\) be the set of diagonals so formed. Next, if \(|D| = m\) then use an array with \(m\) cells to map the graph. The algorithm to do so has three phases. In the first phase we fix the neighborhood constants \(n_1, n_2, n_3\) and also construct the function \(PA\) that maps computation vertices onto cells. In the second phase we fix the delays \(d_1\) and \(d_2\) and also map the mesh graphs that are in \(CG\). In the third phase we fix \(d_3\). We also construct the function \(TA\) that maps computation vertices onto time steps by composing the mappings of the mesh graphs constructed in phase two. Let \(h_1, h_2, h_3\) and \(h_4\) denote the maximum values that a vertex can assume as its first, second, and third coordinates, respectively.

**Phase One**: Let \(n_1 = w_1, n_2 = w_2\) and \(n_3 = w_3\). Map all the vertices in \(D_1\) onto cell \(i\).

**Phase Two**:  
1) Choose \(d_1\) to be 1. If \(n_2 = 1\), then choose \(d_2\) to be 2 else pick 1.

2) For every \(CG_i\) do the following: 

a) let \(v_i\) denote the computation vertex whose coordinates are \((0, 0, i)\) and let \(TA(v_i) = t_i\) (we will show in phase three how to determine \(t_i\)).

b) if \(v_j\) is a computation vertex in any mesh graph in \(CG_i\), let \(TA(v_j) = t_j + x_1d_1 + x_2d_2\).

**Phase Three**: We first fix the delay for \(13\) as follows: 
1) if \(n_1 = n_2\) then if \(h_1 - h_2 + n_3 \neq 0\) then pick \(d_3\) to be \(h_1 + 1 + 2n_3\) else pick \(h_2 + 1 + n_3\),

2) if \(n_1 \neq n_2\) then if \(h_1 - h_2 + n_3 \geq 0\) then choose \(2h_1 + 1 + n_3\) as the delay for \(d_3\) else pick \(2h_2 + 1 - n_3\).

Next, to construct the function \(TA\) we compose the mappings of the mesh graphs in \(CG\), be choosing \(t_i\) to be \(t_j + id_3\).

In the appendix we have shown that this mapping is syntactically correct. Using this mapping algorithm we will now synthesize a new linear array algorithms for multiplying two dense matrices. These algorithms multiply two \(n \times n\) matrices using \(O(n)\) cells in \(O(n^2)\) time steps. The cells require no control, no addressable memory and the array requires no loading and unloading circuitry.

**Example 3**: Let \(A\) and \(B\) be the two matrices (shown below) that are to be multiplied.

\[
\begin{bmatrix}
a_{11} & a_{12} \\
da_{21} & a_{22}
\end{bmatrix}
\begin{bmatrix}
b_{11} & b_{12} & b_{13} \\
b_{21} & b_{22} & b_{23}
\end{bmatrix}
= 
\begin{bmatrix}
c_{11} & c_{12} & c_{13} \\
c_{21} & c_{22} & c_{23}
\end{bmatrix}.
\]

A program for computing this multiplication is given by the following recurrence.

\[
c_{i-1}^{(k+1)} = n_{i-1}^{(0)} + a_{ib}b_{ij}, \quad 1 \leq i, k \leq 2, 1 \leq j \leq 3
\]

\(c_{i}^{(3)} = 0\).

The program graph in Fig. 16 describes this computation. In Fig. 16, \(p_{ij}\) and \(q_{ij}\) denote computation vertices. The horizontal, vertical and oblique incident edges of \(p_{ij}\) are labeled \(l1\), \(l2\), and \(l3\), respectively. Similarly the horizontal, vertical and oblique outgoing edges of \(q_{ij}\) are labeled \(l1\), \(l2\), and \(l3\), respectively. If the horizontal, vertical and oblique incident edges of \(p_{ij}\) or \(q_{ij}\) represent the values \(a, b, c\), respectively, then the horizontal, vertical, and oblique outgoing edges of \(p_{ij}\) or \(q_{ij}\) represent the values \(a, b, c + ab\), respectively. Fig. 16, the oblique input edge incident on \(p_{ij}\) represents the value \(c_{ij}^{(0)}\) which is 0. The oblique outgoing edge from \(q_{ij}\) represents the final (output) value \(c_{ij}^{(3)}\) of \(c_{ij}\), that is, \(a_ib_j + a_2b_2j\).

This program graph is a cube graph as illustrated in Fig. 17. The cube graph is shown without the source and sink vertices for purposes of clarity.

The maximum dimensions on the first, second, and third axes is 2, 1, and 1, respectively, that is, \(h_1 = 2, h_2 = 1\), and \(h_3 = 1\). We next map this graph onto a linear array using the mapping algorithm for a cube graph.

Pick the triple \((1, 1, 1)\) to diagonalize the graph. Let \(D_1, D_2, D_3, D_4\) and \(D_5\) be the diagonals where \(D_1 = \{p_{11}\}\), \(D_2 = \{p_{12}, p_{21}, q_{11}\}\), \(D_3 = \{p_{13}, p_{22}, q_{12}, q_{21}\}\), \(D_4 = \{p_{33}, q_{31}, q_{22}\}\), and \(D_5 = \{q_{53}\}\). Each cell is comprised of 3 pairs of I/O ports labeled \(l1, l2\), and \(l3\), respectively. Let \(a, b, c\) denote the inputs at the input ports labeled \(l1, l2\), and \(l3\), respectively, of cell \(s\) at time \(t\). Then cell \(s\) computes the triple \((a, b, c + ab)\) at tick \(t\).

From phase one, we obtain \(n_1 = 1, n_2 = 1\) and \(n_3 = 1\). Also all the computation vertices in \(D_1\) are mapped onto cell

<table>
<thead>
<tr>
<th>TABLE V</th>
</tr>
</thead>
<tbody>
<tr>
<td>[</td>
</tr>
<tr>
<td>t-1</td>
</tr>
<tr>
<td>t-2</td>
</tr>
<tr>
<td>t-1</td>
</tr>
<tr>
<td>t-1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE VI</th>
</tr>
</thead>
<tbody>
<tr>
<td>[</td>
</tr>
<tr>
<td>t-5</td>
</tr>
<tr>
<td>t-6</td>
</tr>
<tr>
<td>t-7</td>
</tr>
<tr>
<td>t-7</td>
</tr>
</tbody>
</table>
necessary and sufficient conditions on the structure of graphs in this class for the existence of syntactically correct mappings on linear arrays. As a practical consequence of our characterization, we developed a transformation technique that constructs linear-array algorithms for programs described by homogeneous graphs.

Subsequently, we examined cube graphs which are more general than graphs in $\Theta$ and developed a technique to correctly map such graphs onto linear arrays. We used this technique to synthesize some new linear-array matrix multiplication algorithms.

The characterization developed in this paper sheds some insight into the regularity of computations that can be mapped on VLSI arrays. Although the theory developed in this paper is inadequate to synthesize the algorithm described in [25], we believe that it would have been difficult to construct this algorithm without the insight gained by the method outlined in this paper. We would also like to mention that more recent research similar in spirit to the one described in this paper appears in [3], [6], [10], [19], and [22].

The next important step is to extend the characterization developed in this paper to other models like mesh arrays and hexagonal arrays. Preliminary results for these models appear in [24]. The model used in this paper precluded control signals. Arrays that allow control signals are useful in handling graphs that do not have the requisite structure for mapping (like the graphs that describe dynamic programming and transitive closure in [9]). For such graphs it would be necessary to "syntactically restructure" them before mapping and such restructuring would require some semantic properties of the computation represented by the graph. A systematic framework for such semantic-driven mapping of graphs is an interesting open problem.

**APPENDIX**

Theorems 1, 2, and 3 characterize the structure of graphs in $\Theta$. We will now develop the proofs for these theorems. In addition we will also show that the algorithm to map a cube graph on a linear array is syntactically correct.

We first establish certain fundamental results on major paths and mesh graphs which we will use later on in the proofs of the theorems.

We will continue to follow the notational conventions adopted in the beginning of Section III. Additionally, for any two computational vertices $v_i$ and $v_j$, we will be using $\Delta_\ell(v_i, v_j)$ and $\Delta_\ell(v_i, v_j)$ to denote $PA(v_i) - PA(v_j)$ and $TA(v_i) - TA(v_j)$, respectively. We will again assume that $G$ is comprised of nonidentical paths and that no set is made of dangling major paths alone.

**A. Properties of Major Paths**

A major path specifies some transformation that a data item undergoes and a correct mapping of a program graph preserves the transformations of all the major paths in the graph. The value represented by a major path will be either the input value represented by the source vertex or the output value represented by the sink vertex or an intermediate value represented by an edge between two pairs of computation.

---

**Fig. 16.**

**Fig. 17.** Cube graph.

**Fig. 18.**
vertices in the major path. All major paths in a program graph are unique as we have not assumed any properties of the function represented by the computation vertices in the graph. So a value represented by a major path is also unique. We use uniqueness to mean that the value represented by a major path is distinguishable from the value represented by any other major path.

The cell model that we have used in the linear array does not have any branching ability. This imposes certain restrictions on major paths labeled $lj$ in mappings where the neighborhood constant $n_j$ is 0. These restrictions are captured in the following lemma.

**Lemma 1:** Let $v_j$ and $v_k$ be any two computation vertices and $lj$ be any label in $G$. Then, in any mapping, if $n_j = 0$ and $PA(v_j) = PA(v_k)$ (i.e., the neighborhood constant of label $lj$ is 0 and $v_j$ and $v_k$ are mapped on the same cell) then $v_j$ and $v_k$ must be in the same major path labeled $lj$.

**Proof:** If $n_j = 0$ then in every cell a register serves as the cell’s I/O port labeled $lj$ and a value is preloaded into this register, and so if $v_j$ and $v_k$ are in different major paths labeled $lj$ then two registers would be needed — one to hold the value of the first major path and the second to hold the value of the second major path. The cell would then require branching to choose one of the two registers whenever it is in active phase.

In the following lemma we relate the vertices and edges in a path to the cells and time steps at which they are mapped.

**Lemma 2:** Let $v_j$ and $v_k$ be any pair of vertices in $G$. Consider any path $p$ from $v_j$ to $v_k$. For any label $lj$ let $k^j_1$ and $k^j_2$ denote the number of edges labeled $lj$ in $p$ whose directions are consistent and not consistent, respectively, with the directed path from $v_j$ to $v_k$ through the same sequence of vertices as in $p$. Then in any correct mapping of $G$, $\Sigma_{v_j} (k^j_1 - k^j_2)n_j = \Delta_p(v_j, v_k)$ and $\Sigma_{v_j} (k^j_1 - k^j_2)d_j = \Delta_p(v_j, v_k)$. $\Box$

**Proof:** Let $\Sigma_{v_j} (k^j_1 - k^j_2) = n$. So $n$ is the path length. The lemma is easily established by induction on $n$. $\Box$

From the Lemma 2 the following result on major paths is immediate.

**Lemma 3:** Consider any major path labeled $lj$ and let $v_j$ and $v_k$ be any two vertices in this major path. Then in any correct mapping of $G$, $\Delta_p(v_j, v_k)d_j = \Delta_p(v_j, v_k)n_j$.

**Proof:** Immediate from Lemma 2. $\Box$

We next show that if two major paths have the same set of computation vertices then they must be identical.

**Lemma 4:** Let $q_1$ and $q_2$ be two major paths and let $V_l$ and $V_r$ be the sets of computation vertices in $q_1$ and $q_2$, respectively. If $V_l = V_r$ and there exists a correct mapping for $G$ then $q_1$ and $q_2$ must be identical.

**Proof:** Suppose $q_1$ and $q_2$ are not identical. Then there must exist two computation vertices $v_j$ and $v_k$ in $q_1$ and $q_2$ such that $v_j$ precedes $v_k$ in $q_1$ and $v_k$ precedes $v_j$ in $q_2$. Now consider any correct mapping of $G$. $v_j$ precedes $v_k$ in $q_1$ and so $\Delta_p(v_j, v_k) > 0$. Likewise, $v_j$ precedes $v_k$ in $q_2$ and so $\Delta_p(v_j, v_k) > 0$ — a contradiction.

We are now in a position to show that there can be at most one label whose neighborhood constant can be 0.

**Lemma 5:** Let $li$ and $lj$ be any two labels. Then in any correct mapping of $G$, if $n_l = n_j$ then $n_l \in \{1, -1\}$.

**Proof:** Note that the sets of major paths in $G$ are not identical. Since $li$ and $lj$ are in $G$, this implies that there exists a major path $q_i$ labeled $li$ that is not identical to any of the major paths labeled $lj$. This implies that there exists a major path $q_i$ labeled $lj$ and,

1) either the computation vertices in $q_i$ and $q_j$ are the same,

2) or the computation vertices in $q_i$ are a subset of the computation vertices in $q_j$,

3) or the computation vertices in $q_i$ are a subset of the computation vertices in $q_j$.

Consider the first case. By Lemma 4, $q_i$ and $q_j$ must be identical. Next consider the second case. $q_i$ passes through a subset of the vertices in $q_j$. Let $v_j$ and $v_k$ be two vertices in $q_j$ such that $v_j$ is in this subset and $v_k$ is not. Clearly then, there is a major path $q_i$ labeled $lj$ (distinct from $q_j$) that passes through $v_j$ as illustrated in Fig. 19.

Now assume $n_j = n_j = 0$. So $PA(v_j) = PA(v_k)$ and $q_j$ and $q_i$ are distinct major paths labeled $lj$, violating Lemma 1. So $n_l = n_j \neq 0$. We can similarly show that $n_l = n_i \neq 0$ in the third case as well. $\Box$

A correct mapping must ensure that no two tokens appear simultaneously at the same input port of any cell. As we will see in the next lemma, this forces certain restrictions on the structure of major paths.

**Lemma 6:** For any label $lj$ and for any pair of vertices $v_j$ and $v_k$, if $\Delta_p(v_j, v_k)d_j = \Delta_p(v_j, v_k)n_j$ in any correct mapping of $G$ then there must be a major path labeled $lj$ passing through $v_j$ and $v_k$.

**Proof:** Assume that in a correct mapping there exists a pair of vertices $v_j$ and $v_k$ and a label $lj$ such that $\Delta_p(v_j, v_k)d_j = \Delta_p(v_j, v_k)n_j$ and $v_j$ and $v_k$ are in different major paths labeled $lj$. Let $q_1$ and $q_2$ be the two major paths such that $v_j$ is in $q_1$ and $v_k$ is in $q_2$. Using Lemma 3 it can be easily shown that for any pair of vertices $v_j$ in $q_1$ and $v_k$ in $q_2$, if $\Delta_p(v_j, v_k)d_j = \Delta_p(v_j, v_k)n_j$ then $\Delta_p(v_j, v_k)d_j = \Delta_p(v_j, v_k)n_j$. So assume without loss of generality, that $v_j$ and $v_k$ are the first computation vertices (the vertices adjacent to source vertices) in $q_1$ and $q_2$, respectively. Now $n_j \in \{1, -1, 0\}$. We will arrive at a contradiction for each of the three values that $n_j$ assumes.

Case 1: $n_j = 0$. So $\Delta_p(v_j, v_k) = 0$ as $d_j > 0$. Hence, by Lemma 1 there must be a major path labeled $lj$ passing through $v_j$ and $v_k$ — a contradiction.

Case 2: $n_j = 1$. Now $\Delta_p(v_j, v_k)$ must be either 0, positive or negative. Let $PA(v_j) = s_1$, $PA(v_k) = s_2$, $TA(v_j) = t_1$ and $TA(v_k) = t_2$. Let $i_1$ and $i_2$ be two tokens that are initialized to the input values represented by the source vertices in $q_1$ and $q_2$, respectively.

1) $\Delta_p(v_j, v_k) = 0$. So $s_1 = s_2$. Now $n_j \neq 0$ and so $\Delta_p(v_j, v_k) = 0$. Hence, $i_1$ and $i_2$ appear simultaneously at the same input port labeled $lj$ of $s_1$ — a contradiction.

2) $\Delta_p(v_j, v_k) > 0$. So $s_2 > s_1$. Now $n_j = 1$ and so $\Delta_p(v_j, v_k) > 0$ and hence $t_2 > t_1$, $i_1$ appears at the input port labeled $lj$ of $s_1$ at time $t_2 - (s_2 - s_1)d_j$. This reduces to $t_2 - \Delta_p(v_j, v_k)n_j$ which is $t_2 - (t_2 - t_1)$. So $i_1$ and $i_2$ appear simultaneously at the same input port of $s_1$ at time $t_1$ — a contradiction.

3) $\Delta_p(v_j, v_k) < 0$. So $s_1 > s_2$. Now $n_j = 1$ and so
\[ \Delta_F(v_i, v_j) < 0 \] and so \( t_1 > t_2 \). \( i_1 \) appears at the input port labelled \( l_j \) of \( s_2 \) at time \( t_1 - (s_1 - s_2) d_{y} \). This reduces to \( t_1 + \Delta_F(v_i, v_j) n_0 \) which is \( t_1 + (t_2 - t_1) \). Hence, \( i_1 \) and \( t_2 \) appear simultaneously at the same input port of \( s_2 \) at time \( t_2 \) — a contradiction.

**Case 3:** \( n_y = -1 \). The proof is along lines similar to that of Case 2.

We next show that if the neighborhood constants of any two labels are equal then their delays cannot be the same.

**Lemma 7:** Let \( l_i \) and \( l_j \) be any two labels. In any correct mapping of \( G \) if \( n_l = n_y \) then \( d_{l_i} \neq d_{l_j} \).

**Proof:** As \( G \) has only sets of nonidentical major paths, there exists a major path \( q_s \) labeled \( l_i \) that is not identical to any of the major paths labeled \( l_j \). This implies that there exists a major path \( q_{s'} \) labeled \( l_j \) and,

1. either the computation vertices in \( q_{s'} \) and \( q_s \) are the same,
2. or the computation vertices in \( q_{s'} \) are a subset of the computation vertices in \( q_s \),
3. or the computation vertices in \( q_{s'} \) are a subset of the computation vertices in \( q_s \).

Consider the first case. By Lemma 4, \( q_s \) and \( q_{s'} \) must be identical — a contradiction. Next consider the second case. \( q_{s'} \) passes through a subset of the vertices in \( q_s \). Let \( v_s \) and \( v_{s'} \) be the two vertices in \( q_s \) such that \( v_s \) is in the subset and \( v_{s'} \) is not. Then there is a major path \( q_{s'} \) labeled \( l_j \) distinct from \( q_s \) that passes through \( v_s \) as illustrated in Fig. 20. By Lemma 3, \( \Delta_F(v_s, v_{s'}) = c \) and, hence, \( \Delta_F(v_s, v_{s'}) = d_{l_j} \) by Lemma 6. \( v_s \) and \( v_{s'} \) must be in the same major path labeled \( l_j \) — a contradiction. We can similarly show that if \( n_u = n_y \) then \( d_{l_u} \neq d_{l_j} \) for the third case also.

**B. Connected Components**

We will now examine the relationship between correct mapping and connected components. In particular let \( l_{\mu} \) and \( l_{\nu} \) be any two labels in \( G \) and let \( S \) be a connected component that is obtained by removing all the edges whose labels are neither \( l_{\mu} \) nor \( l_{\nu} \). In general several such components may result and \( S \) is one such component.

Let \( S_{\mu} = \{ \text{major paths labeled } l_{\mu} \text{ in } S \} \) and \( S_{\nu} = \{ \text{major paths labeled } l_{\nu} \text{ in } S \} \).

Let \( G_r^s = (V_r^s, E_r^s) \) and \( G_r = (V_r^s, E_r^s) \) be two directed graphs and \( F_r^s \) and \( F_r \) be two one-one functions such that:

1. \( F_r^s : S_{\mu} \rightarrow V_r^s \) (the major paths in \( S_{\mu} \) are represented by the vertices in \( V_r^s \)).
2. \( E_r = \{ (q_{s}, q_s) \} \) \( q_{s} \in S_{\mu}, q_s \in S_{\mu} \) and there exists a directed edge labeled \( l_{\nu} \) from some computation vertex in \( q_{s} \) to some computation vertex in \( q_s \).
3. \( F_r : S_{\mu} \rightarrow V_r^s \) (the major paths in \( S_{\mu} \) are represented by the vertices in \( V_r^s \)).

**Case 3:** \( n_y = -1 \). The proof is along lines similar to that of Case 2.

We next show that if the neighborhood constants of any two labels are equal then their delays cannot be the same.

**Lemma 7:** Let \( l_i \) and \( l_j \) be any two labels. In any correct mapping of \( G \) if \( n_l = n_y \) then \( d_{l_i} \neq d_{l_j} \).

**Proof:** As \( G \) has only sets of nonidentical major paths, there exists a major path \( q_s \) labeled \( l_i \) that is not identical to any of the major paths labeled \( l_j \). This implies that there exists a major path \( q_{s'} \) labeled \( l_j \) and,

1. either the computation vertices in \( q_{s'} \) and \( q_s \) are the same,
2. or the computation vertices in \( q_{s'} \) are a subset of the computation vertices in \( q_s \),
3. or the computation vertices in \( q_{s'} \) are a subset of the computation vertices in \( q_s \).

Consider the first case. By Lemma 4, \( q_s \) and \( q_{s'} \) must be identical — a contradiction. Next consider the second case. \( q_{s'} \) passes through a subset of the vertices in \( q_s \). Let \( v_s \) and \( v_{s'} \) be the two vertices in \( q_s \) such that \( v_s \) is in the subset and \( v_{s'} \) is not. Then there is a major path \( q_{s'} \) labeled \( l_j \) distinct from \( q_s \) that passes through \( v_s \) as illustrated in Fig. 20. By Lemma 3, \( \Delta_F(v_s, v_{s'}) = c \) and, hence, \( \Delta_F(v_s, v_{s'}) = d_{l_j} \) by Lemma 6. \( v_s \) and \( v_{s'} \) must be in the same major path labeled \( l_j \) — a contradiction. We can similarly show that if \( n_u = n_y \) then \( d_{l_u} \neq d_{l_j} \) for the third case also.

By Lemma 2

\[ \Delta_r(v_s, v_{s'}) = n_{l_{\mu}}h + n_{l_{\nu}}m \]

and

\[ \Delta_r(v_s, v_{s'}) = d_{l_{\mu}}h + d_{l_{\nu}}m . \]

Let the distance between \( v_s \) and \( v_{s'} \) in the major path \( q_{s} \) be \( k \). Hence,

\[ \Delta_r(v_s, v_{s'}) = n_{l_{\mu}}k \]

and

\[ \Delta_r(v_s, v_{s'}) = d_{l_{\mu}}k \]

and so

\[ n_{l_{\mu}}k = n_{l_{\mu}}h + n_{l_{\nu}}m \]

(a)

\[ d_{l_{\mu}}k = d_{l_{\mu}}h + d_{l_{\nu}}m \]  

(b)

By Lemma 5, if \( n_{l_{\mu}} = n_{l_{\nu}} \), then \( n_{l_{\mu}} = 0 \), and hence the possible values that the pair \( (n_{l_{\mu}}, n_{l_{\mu}}) \) can assume are \( (1,0) \),
\((1, 1), (-1, -1), (1, 0), (-1, 1), (0, 1), (0, -1)\). We will arrive at a contradiction for each of these values that \((n_{1w}, n_{1v})\) assumes.

1) Consider the set of values \((1, 1)\) and \((-1, -1)\), that is, \(n_{1w} = n_{1v}\). From (a) and (b) \(d_{1w} = d_{1v}\) — a contradiction since by Lemma 7, \(d_{1w} \neq d_{1v}\).

2) Consider the set of values \((0, 1)\) and \((0, -1)\), that is, \(n_{1w} = 0\) and \(n_{1v} = 0\). From (a) \(n_{1v} = 0\) — a contradiction as \(n_{1v} \neq 0\).

3) Consider the set of values \((1, 0)\) and \((-1, 0)\), that is, \(n_{1w} = \{1, -1\}\) and \(n_{1v} = 0\). From (a) and (b) \(d_{1w} = 0\) — a contradiction as \(d_{1w} > 0\).

4) Consider the set of values \((-1, 1)\) and \((1, -1)\), that is, \(n_{1w} = \{1, -1\}\) and \(n_{1v} = \{1, -1\}\). From (a) and (b) \(d_{1w} = -d_{1v}\) — a contradiction as \(d_{1w} > 0\) and \(d_{1v} > 0\).

So we have arrived at contradictions when \(G^p_i\) has a cycle, and hence \(G^p_i\) must be acyclic.

We next show that there must be a directed path between any pair of vertices in \(V^p_i\). Suppose not. Then let \(q_t\) and \(q_r\) be two vertices in \(V^p_i\) that do not have a directed path between them. Now \(G^p_i\) is connected and so there must be a \(q_k\) in \(V^p_i\) such that one of the following two cases must occur:

1) There are two vertex disjoint directed paths; one from \(q_t\) to \(q_k\) and the other from \(q_k\) to \(q_r\).

2) There are two vertex disjoint directed paths; one from \(q_t\) to \(q_k\) and the other from \(q_k\) to \(q_r\).

We will only consider the first case and the proof for the second case will be similar. Let \(q_m\) be the vertex adjacent to \(q_t\) in the directed path from \(q_t\) to \(q_k\) and \(q_n\) be the vertex adjacent to \(q_k\) in the directed path from \(q_k\) to \(q_r\) as shown in Fig. 23. Now \(q_m, q_n, q_t, q_r\) and \(q_k\) are all major paths labeled \(l\mu\) in \(G\). Existence of a directed edge from \(q_m\) to \(q_k\) in \(G^p_i\) in Fig. 23 implies that there exists computation vertices \(v_{1m}\) in \(q_m\) and \(v_{1n}\) in \(q_n\) and a directed edge \(e_{m}\) labeled \(l\nu\) from \(v_{1m}\) to \(v_{1n}\). Similarly, existence of a directed edge from \(q_t\) to \(q_k\) in \(G^p_i\) implies that there exists computation vertices \(v_{1t}\) in \(q_t\) and \(v_{1r}\) in \(q_r\) and a directed edge \(e_{t}\) labeled \(l\nu\) from \(v_{1t}\) to \(v_{1r}\) as illustrated in Fig. 24. In Fig. 24 each of the shaded boxes denote a major path labeled \(l\mu\). Let the distance between \(v_{1w}\) and \(v_{1u}\) in \(q_k\) be \(h\) and hence in any correct mapping

\[
\Delta_{r}(v_{1w}, v_{1u}) = n_{1w}h
\]

and

\[
\Delta_{r}(v_{1w}, v_{1u}) = d_{1w}h.
\]

As there is a directed edge from \(v_{1w}\) to \(v_{1u}\)

\[
\Delta_{r}(v_{1w}, v_{1u}) = n_{1w}h
\]

and

\[
\Delta_{r}(v_{1w}, v_{1u}) = d_{1w}h.
\]

Also, as there is a directed edge from \(v_{1t}\) to \(v_{1u}\)

\[
\Delta_{r}(v_{1t}, v_{1u}) = n_{1w}h
\]

and

\[
\Delta_{r}(v_{1t}, v_{1u}) = d_{1w}h.
\]

From the above equations we obtain

\[
\Delta_{r}(v_{1t}, v_{1u}) = n_{1w}h
\]

and

\[
\Delta_{r}(v_{1t}, v_{1u}) = d_{1w}h.
\]

Now by Lemma 6, \(v_{1w}\) and \(v_{1u}\) must be in the same major path labeled \(l\mu\). But \(q_m\) and \(q_n\) are distinct — a contradiction.

Lastly, we must show that the directed path between any pair of vertices in \(G^p_i\) is unique. Suppose not. Let \(q_m\) and \(q_n\) be two vertices in \(G^p_i\) such that there are two distinct directed paths from \(q_m\) to \(q_n\). Let \(\{q_m, q_{m1}, \cdots, q_{mi}, q_r, q_{r1}, \cdots, q_{rj}\}\) and \(\{q_m, q_{m1}, \cdots, q_{mi}, q_r, q_{r1}, \cdots, q_{rj}\}\) be the two sequences of vertices traversed by the first and second directed paths respectively. Let \(q_t\) and \(q_r\) be distinct. So the two sequences differ after \(q_t\). We have already shown that there must be a directed path between any pair of vertices in \(G^p_i\). Without loss of generality let there be a directed path from \(q_t\) to \(q_r\). So now there are two directed paths from \(q_t\) to \(q_r\). The first directed path is a directed edge from \(q_t\) to \(q_k\) and the second directed path is through the sequence of vertices \(\{q_t, q_{t1}, \cdots, q_{tk}\}\). These two directed paths imply the existence of computation vertices \(v_{1k}\) and \(v_{1r}\) in \(q_k\) and \(q_r\), respectively, and two paths \(p_1\) and \(p_2\) between them as shown in Fig. 25.
The first path $p_1$ between $v_s$ and $v_t$ traverses the edge $e_a$ labeled $lv$. The second path $p_2$ is through computation vertices in $q_1, q_2, \ldots, q_j$. Let $k_1^p$ and $k_2^p$ be the number of edges labeled $l\mu$ in $p_1$ whose directions are consistent and not consistent, respectively, with the direction imposed on them by the directed path from $v_s$ to $v_t$ passing through the same sequence of vertices as in $p_1$. Let $k_1^p = k_2^p = h_1$. For this path, from Lemma 2, we obtain

$$\Delta_{v_1}(v_s, v_t) = h_1 n_{l\mu} + n_{l\nu}$$

and

$$\Delta_{v_2}(v_s, v_t) = h_2 d_{l\mu} + d_{l\nu}.$$  

Let $k_1^p$ and $k_2^p$ be the number of edges labeled $l\mu$ in $p_2$ whose directions are consistent and not consistent, respectively, with the direction imposed on them by the directed path from $v_s$ to $v_t$ passing through the same sequence of vertices as in $p_2$. Let $k_1^p = k_2^p = h_2$. Also let $k_1^p$ and $k_2^p$ be the number of edges labeled $l\nu$ in $p_2$ whose directions are consistent and not consistent respectively with the direction imposed on them by the directed path from $v_s$ to $v_t$ passing through the same sequence of vertices as in $p_2$. Let $k_1^p - k_2^p = m$. The distance from $q_s$ to $q_t$ must be at least 1 and so $m > 1$. For the second path $p_2$, from Lemma 2 again, we obtain

$$\Delta_{v_1}(v_s, v_t) = h_2 n_{l\mu} + n_{l\nu} m$$

and

$$\Delta_{v_2}(v_s, v_t) = h_2 d_{l\mu} + d_{l\nu} m$$

and so

$$(h_1 - h_2) n_{l\mu} = (m - 1) n_{l\nu} \tag{c}$$

and

$$(h_1 - h_2) d_{l\mu} = (m - 1) d_{l\nu} \tag{d}.$$  

Now (c) and (d) are similar to (a) and (b) that resulted from a cycle in $G^p$. Hence solution to (c) and (d) would lead to contradictions and hence the directed path between any pair of vertices in $G^p$ must be unique.

We establish the link between mesh graphs and $S$ through the following lemma.

**Lemma 9:** $S$ is a mesh graph if and only if the following conditions are satisfied:

1) $G^p$ is a cyclic, and there must exist a unique directed path between any pair of vertices in $V^p$.

2) $G^p$ is acyclic, and there must exist a unique directed path between any pair of vertices in $V^p$.

**Proof:** (Only If): Simple.

(If Part): Let $V_S$ be the set of computation vertices in $S$. Topologically sort the vertices of $G^p$ and $G^p$. Assign indexes ranging from 0 to $|V^p| - 1$ to the topologically sorted vertices in $V^p$. Similarly, assign indexes ranging from 0 to $|V^p| - 1$ to the topologically sorted vertices in $V^p$. We next assign coordinates to the vertices as follows: let $q_m$ and $q_n$ be any two vertices in $V^p$ and $V^p$, respectively. Now $q_m$ and $q_n$ are major paths labeled $l\mu$ and $l\upsilon$, respectively, in $S$. Let $v_s$ be the computation vertex in $q_m$ and $q_n$. Let a and $b$ be the indexes assigned to $q_m$ and $q_n$, respectively, by the topological sort of vertices in $V^p$ and $V^p$, respectively. Now assign the coordinate $(a, b)$ to $v_s$. Using conditions 1) and 2) of Lemma 9, it can be easily shown that such an assignment transforms $S$ into a mesh graph.

We are now in a position to establish our fundamental result relating $S$, mesh graphs, and a correct mapping.

**Theorem 4:** If there exists a syntactically correct mapping for $G$ then $S$ must be a mesh graph.

**Proof:** Straightforward from Lemma 8 and Lemma 9.

Observe that Theorem 1 is now an immediate consequence of the above theorem.

**C. Properties of Mesh Graphs**

We examine some properties of mesh graphs that we will be using later on. For purposes of examining these properties alone we will assume that the connected component $S$ is a mesh graph.

Let $v_s$ and $v_t$ be any two computation vertices in $S$ whose coordinates are $(x_{l\mu}, x_{l\nu})$ and $(y_{l\mu}, y_{l\nu})$, respectively. Consider any path $p$ between $v_s$ and $v_t$. Let $k_1^p$ and $k_2^p$ be the number of edges labeled $l\mu$ in $p$ whose directions are consistent and not consistent, respectively, with the direction induced on them by the directed path from $v_s$ to $v_t$ through the same sequence of vertices as in $p$. Similarly, let $k_1^p$ and $k_2^p$ be the number of edges labeled $l\nu$ in $p$ whose directions are consistent and not consistent, respectively, with the direction induced on them by the directed path from $v_s$ to $v_t$ through the same sequence of vertices as in $p$. In the following lemma we relate $(x_{l\mu}, x_{l\nu})$ and $(y_{l\mu}, y_{l\nu})$ to $k_1^p$, $k_2^p$, $k_1^p$, and $k_2^p$.

**Lemma 10:** $k_1^p - k_2^p = y_{l\mu} - x_{l\mu}$ and $k_1^p - k_2^p = y_{l\nu} - x_{l\nu}$.

**Proof:** The proof is by induction on the path length. Let $n$ denote the path length. $v_s$ and $v_t$ are distinct and hence, $n > 0$.

**Basis Step:** $n = 1$, so the path consists of only one edge. Hence, only one of $k_1^p$, $k_2^p$, $k_1^p$, and $k_2^p$ can be 1 and the rest must be 0. Let $k_1^p = 1$. So $k_1^p = k_2^p = 0$. This implies that the path is a directed edge labeled $l\mu$ from $v_s$ to $v_t$. By definition of a mesh graph, $y_{l\mu} - x_{l\mu} = 1$ and $y_{l\nu} - x_{l\nu} = 0$. Similarly, we can also prove that the basis is true for the other three cases also.

**Induction Step:** Assume the lemma is true for paths of length $\leq n$. Consider any path from $v_s$ to $v_t$ of length $n + 1$. If $n + 1 = 1$ then Lemma 10 holds by basis. So assume $n + 1 > 1$ and let $v_1$ be any intermediate vertex in this path. Let $n_1$ and $n_2$ denote the path length from $v_s$ to $v_1$ and $v_1$ to $v_t$ in this path. Clearly $n_1 \leq n$ and $n_2 \leq n$. By applying the induction hypothesis to each of these two paths, it follows that the lemma is true for paths of length $\leq n + 1$.
Now consider any correct mapping of $S$ and let $v_i$ and $v_j$ be any two computation vertices in $S$. We relate the cells and the times at which they are mapped in the following lemma.

**Lemma 11:** Let $\Delta(t, v_i) = (y_{i_1} - x_{i_1})n_{i_1} + (y_{i_2} - x_{i_2})n_{i_2}$ and $\Delta(t, v_j) = (y_{j_1} - x_{j_1})n_{j_1} + (y_{j_2} - x_{j_2})n_{j_2}$.

**Proof:** Straightforward from Lemma 10 and Lemma 2.

We next establish a fundamental property of mesh graphs. This property relates the existence of a directed path between two computation vertices in a mesh graph to certain relationships between their coordinates. This is useful in the proof of Theorem 2 wherein we show that certain graphs in $\Theta$ cannot be mapped correctly. To prove this property the following lemma is useful.

**Lemma 12:** Let $l_i \in \{\mu, \nu\}$ and let $q_m, q_n,$ and $q_k$ be three distinct major paths labeled $l_i$. If the indexes $m, n,$ and $k$ of $q_m, q_n,$ and $q_k,$ respectively, are such that $m < k < n$, then any path between any computation vertex in $q_m$ and any computation vertex in $q_n$ must pass through a computation vertex in $q_k$.

**Proof:** Let $l_i = l\mu$, and let $v_m, v_n,$ and $v_k$ be any three computation vertices in $q_m, q_n,$ and $q_k$, respectively. Indexes of $q_m, q_n,$ and $q_k$ are $m, n,$ and $k$, respectively, and hence, $x_{i_m} = m, y_{i_n} = n,$ and $z_{i_k} = k$.

Now assume that the path does not pass through any computation vertex in $q_n$. Then the path must traverse an edge labeled $l\nu$ between two computation vertices in major paths $q$ and $q_k$, that are labeled $l\mu$ such that $s$ and $r$ are the indexes of $q$ and $q_k$, respectively, then $s < k < r$. By Lemma 10, the number of edges labeled $l\nu$ in any path from $q_i$ to $q_j$ is $r - s$. Since $s < k < r$ and $k, r$, and $s$ are integers, $r - s \geq 2$. But there is also an edge labeled $l\nu$ between a computation vertex in $q_m$ and a computation vertex in $q_n$ it follows from the definition of a mesh graph that $r - s = 1$ — a contradiction. Using similar arguments we can show that Lemma 12 is true for $l_i = l\nu$.

The following result is a straightforward consequence of the previous lemma.

**Corollary 1:** Let $l_i, l_j \in \{l\mu, l\nu\}$ and let $l_i \neq l_j$. Let $q_m$ and $q_n$ be two distinct major paths labeled $l_i$. If their indexes $m$ and $n$ differ by 1 then any path between a computation vertex in $q_m$ and a computation vertex in $q_n$ must traverse an edge labeled $l\nu$ between computation vertices in $q_m$ and $q_n$, respectively.

**Proof:** Without loss of generality let $m = n + 1$ where $m$ and $n$ are the indexes of $q_m$ and $q_n$, respectively. Now pick a path from some computation vertex in $q_m$, say $v_j$, to some computation vertex in $q_n$, say $v_j$, such that it does not traverse an edge between any pair of computation vertices in $q_m$ and $q_n$. Then there must be a computation vertex $v_j$ in this path distinct from $v_i$ and $v_j$. Let $v_j$ be in the major path $q_j$. Let $s$ be the index of $q_j$. If $s > m$ then the path from $v_i$ to $v_j$ violates Lemma 12 and if $s < m$, then the path from $v_i$ to $v_j$ violates Lemma 12.

We are now ready to establish a fundamental property of mesh graphs.

**Lemma 13:** Let $v_i$ and $v_j$ be any pair of computation vertices such that $y_{i_m} \geq x_{i_m}$ and $y_{j_n} \geq x_{j_n}$. Then there must exist a directed path from $v_i$ to $v_j$.

**Proof:** Let $y_{i_m} - x_{i_m} = m$ and $y_{j_n} - x_{j_n} = n$. The proof is an induction on $m$ and $n$.

**Basis Step:** We need to consider the case when $m = 0$ and $n \geq 0$ and the case when $m \geq 0$ and $n = 0$.

**Case 1:** $m = 0$ and $n \geq 0$. By the definition of a mesh graph, there must be a directed path from $v_i$ to $v_j$ in some major path labeled $l\nu$.

**Case 2:** $m \geq 0$ and $n = 0$. Again by definition of a mesh graph, there must be a directed path from $v_i$ to $v_j$ in some major path labeled $l\mu$.

**Induction Step:** Assume the lemma holds for any pair of vertices $v_k$ and $v_l$ such that $0 \leq y_{i_m} - x_{i_m} \leq m$ and $0 \leq y_{j_n} - x_{j_n} \leq n$. We will show that it holds for any $v_i$ and $v_j$ such that $0 \leq y_{i_m} - x_{i_m} \leq m + 1$ and $0 \leq y_{j_n} - x_{j_n} \leq n + 1$. To do this we have to consider the following cases:

1) $y_{i_m} - x_{i_m} \leq m + 1$ and $y_{j_n} - x_{j_n} \leq n$.

2) $y_{i_m} - x_{i_m} \leq m$ and $y_{j_n} - x_{j_n} = n + 1$.

3) $y_{i_m} - x_{i_m} = m + 1$ and $y_{j_n} - x_{j_n} = n + 1$.

The following geometric picture comes in useful in understanding the proof.

The lines $GH, IJ,$ and $KL$ denote major paths labeled $l\nu$. The index of $GH$ is $x_{i_m}$ and the indexes of $IJ$ and $KL$ are $m + x_{i_m}$ and $m + 1 + x_{i_m}$, respectively. The lines $AB, CD,$ and $EF$ denote major paths labeled $l\mu$. The index of $AB$ is $x_{i_m}$ and the indexes of $CD$ and $EF$ are $n + x_{j_n}$ and $n + 1 + x_{j_n}$.

The induction hypothesis holds for $v_i$ and any $v_j$ within the region enclosed by $AB, CD, GH,$ and $IJ$ which is the shaded region in Fig. 26.

We first proceed to establish that the lemma holds for any $v_i$ such that $y_{j_n} - x_{j_n} = m + 1$ and $0 \leq y_{i_m} - x_{i_m} \leq n$. Consider one such vertex $v_j$ as shown in Fig. 27.

From Corollary 1, any path from $v_i$ to $v_j$ must traverse an edge labeled $l\mu$ between vertices in $IJ$ and $KL$. Let $v_k$ and $v_w$ be the two vertices in $IJ$ and $KL$, respectively. Now $v_k$ and $v_w$ must appear in one of the three following regions in Fig. 27:

1) above $AB$,

2) within $AB$ and $MN$,

3) below $MN$.

Fig. 28 (a), (b), and (c) illustrates cases 1, 2, and 3, respectively.

**Case 1:** By the definition of a mesh graph, $v_i$ must exist in $AB$. Then there is a directed path from $v_i$ to $v_j$ and from $v_j$ to $v_k$.

**Case 2:** By the inductive assumption, there is a directed path from $v_k$ to $v_w$. The edge labeled $l\mu$ is directed from $v_j$ to $v_w$. By definition of mesh graph again, there is a directed path from $v_w$ to $v_j$.

**Case 3:** We now show that whenever $v_i$ and $v_w$ occurs below $MN$ then $v_i$ must always exist. Suppose not. Then there cannot be any vertex on $IJ$ above $MN$ and on $MN$ to the left of $IJ$ (by definition of mesh graph). Consider any path $p_1$ between $v_i$ and any vertex, say $v_s$, on $IJ$. $v_i$ must be below $MN$. Then by Lemma 12, there must exist a vertex, say $v_w$, on $MN$ in the path $p_1$ and $v_s$ precedes $v_w$ in $p_1$. As $v_i$ does not exist, $v_i$ must be to the right of $IJ$. Consider any path $p_2$ between $v_i$ and $v_j$. By Lemma 12 again, there must exist a
Proof: (of Theorem 3) (Only If Part): Consider a correct mapping of G. By Theorem 1, SG must be a mesh graph. Now diagonalize SG as follows: if the pair \((n_{i}, n_{j}) \in \{(1, 1), (1, -1), (1, 0), (0, 1)\}\), then let \((w_{1}, w_{2}) = (n_{i}, n_{j})\) else let \((w_{1}, w_{2}) = (-n_{i}, -n_{j})\). We will prove that the three conditions are necessary when \((n_{i}, n_{j}) = (1, -1)\) as the proof is similar for the other values that it can assume. For such a choice then, the orthogonal diagonalization factor is \((0, 1)\).

1) Consider an edge labeled \(l_j\) directed from \(v_i\) to \(v_j\). Now \(\Delta_r(v_i, v_j) = n_y \in \{1, -1, 0\}\). Also, by Lemma 11, \(\Delta_r(v_i, v_j) = (y_{i} - x_{i}) - (y_{j} - x_{j})\) and so \(\Delta_r(v_i, v_j) = \Delta_r(v_i - v_j)\), and hence is a constant. Let \(m_y\) denote this constant.

2) Now \(\Delta_r(v_i, v_j) = (y_{i} - x_{i}) - (y_{j} - x_{j})\)

and

\[\Delta_r(v_i, v_j) = m_y.\]

As \(m_y, d_{ij}, d_{ip}, d_{jp}\) are all constants, \(y_{i} - x_{i}\) is a constant. Since \((0, 1)\) is the orthogonal diagonalization factor, \(\Delta_r(v_i, v_j) = (y_{i} - x_{i})\), and hence is a constant. Let \(c_y\) denote this constant.

3) Let \(a = (y_{i} - x_{i})\) and \(b = (y_{j} - x_{j})\). It can be easily verified that \(d_{ij} = (m_{ij} + c_{ij})d_{ip} + c_{ij}d_{jp}\). From Lemma 11, \(\Delta_r(v_i, v_j) = (a - b)\) and \(\Delta_r(v_j, v_i) = d_{ij}a + d_{jp}b\). As \((0, 1)\) is the orthogonal diagonalization factor, \(\Delta_r(v_i, v_j) = b\). Also \(c_{ij}\Delta_r(v_i, v_j) = m_y\Delta_r(v_i, v_j)\) and so \(c_{ij}(a - b) = m_yb. Now,\)

\[\Delta_r(v_i, v_j)d_{ij} = (a - b)d_{ij}\]

\[= (m_{ij} + c_{ij})d_{ip} + c_{ij}d_{jp}\]

\[= m_y\Delta_r(v_i, v_j)\]

\[= d_{ij}a + d_{jp}b\]

\[= m_y\Delta_r(v_i, v_j)\]

and so from Lemma 6, there must be a major path labeled \(l_j\) passing through \(v_i\) and \(v_j\).

(If Part): Let \(D = \{D_1, D_2, \ldots, D_n\}\) be the set of main diagonals where \(i\) denotes the index of diagonal \(D_i\). Construct a linear array \(L_{di}\) with \(n\) cells. Next construct a mapping as follows.

1) If there exists a transitive edge (if \((v_i, v_j), (v_j, v_k)\), and \((v_k, v_l)\) are directed edges then the edge \((v_j, v_k)\) is a transitive edge) labeled \(l_j\) such that \(m_y = 0\) then choose two-phase clocking else select a single-phase clocking scheme.

2) Let \(v_i\) be a computation vertex that lies on the main diagonal \(D_q\). Then map \(v_i\) on cell \(q\), that is let \(PA(v_i) = q\). This assigns computation vertices onto cells.

3) Next assign delay and neighborhood constants to the labels as follows. Let \(n_y = m_y\) and let \(d_{ip}, d_{jp}\) be two constants. If \((w_1, w_2) = (1, -1)\) or there exists a transitive edge labeled \(l_j\) such that \(m_y = 0\) then choose \(d_j\) to be 2 else let it be 1. Let \(c_{min}\) be the minimum among all the \(c_j\)'s. If \(c_{min} > 0\), then choose \(d_j\) to be 1 else let \(d_j = 1 + |c_{min}|d_a\). Finally, let \(d_y = m_yd_b + c_yd_a\).
4) Construct the function \( TA \) which assigns computation vertices to time steps. Let \( v_i \) be the computation vertex that lies on main and orthogonal diagonals whose indexes are 1. Let \( TA(v_i) = t_0 \). Let \( v_i \) lie on the main diagonal indexed \( p \) and orthogonal diagonal indexed \( q \). Then, let \( TA(v_i) = t_0 + (q - 1)d_a + (p - 1)d_b \).

Steps 1)–4) transform a homogeneous graph in \( \Theta \) into an array algorithm. We now establish that the mapping is correct.

We begin by showing that \( n_{ij} \) and \( d_{ij} \) are constants. Consider an edge labeled \( lj \) from \( v_i \) to \( v_j \) and let \( n_{ij} \) lie on a main diagonal indexed \( p \) and an orthogonal diagonal indexed \( q \). Similarly, let \( v_j \) lie on a main diagonal indexed \( r \) and an orthogonal diagonal indexed \( s \).

Now
\[
\Delta_0(v_i, v_j) = \Delta_p(v_i, v_j)
\]
\[
= r - p
\]
\[
= m_{ij} \in \{1, -1, 0\}
\]
\[
= n_{ij}.
\]

Next,
\[
\Delta_1(v_i, v_j) = (s - q)d_a + (r - p)d_b
\]
\[
= \Delta_{ci}(v_i, v_j)d_a + \Delta_{di}(v_i, v_j)d_b
\]
\[
= m_{ij}d_a + c_{ij}d_a
\]
\[
= d_{ij}.
\]

Next, we will show that if \( n_{ij} = 0 \), then all the vertices mapped onto the same cell belong to the same major path labeled \( lj \). Suppose \( n_{ij} = 0 \). Then \( m_{ij} = 0 \). Consider any \( v_j \), and \( v_j \) such that \( \Delta_0(v_i, v_j) = 0 \). Then \( \Delta_p(v_i, v_j) = 0 \) and so \( c_{ij} \Delta_0(v_i, v_j) = m_{ij} \Delta_0(v_i, v_j) \). But by condition 2) of the theorem there must be a major path labeled \( lj \) passing through \( v_i \) and \( v_j \). So, whenever \( n_{ij} = 0 \) and \( PA(v_i) = PA(v_j) \), there is always a major path labeled \( lj \) passing through \( v_i \) and \( v_j \).

We will now show that no two tokens appear simultaneously at the same input port of any cell. We have shown that for any label \( lj \), if \( n_{ij} = 0 \) then vertices mapped onto the same cell all belong to the same major path labeled \( lj \). So we need to consider only major paths whose neighborhood constants are either 1 or -1. Let \( q_1 \) and \( q_2 \) be two major paths labeled \( lj \). Let \( i_1 \) and \( i_2 \) be the two tokens that appear simultaneously at the same input port of some cell. Let \( i_1 \) and \( i_2 \) be initialized to the value represented by the source vertices in \( q_1 \) and \( q_2 \), respectively. Let \( n_{ij} = 1 \). Clearly, these two tokens must have been inserted at the same time and let \( t \) denote this time.

Let \( v_i \) and \( v_j \) be the first vertices of \( q_1 \) and \( q_2 \), respectively. The time taken by \( i_1 \) to reach \( PA(v_i) = t + PA(v_i)d_{ij} \) and the time taken by \( i_2 \) to reach \( PA(v_j) = t + PA(v_j)d_{ij} \). Without loss of generality, \( PA(v_i) \geq PA(v_j) \) and hence, \( \Delta_0(v_i, v_j) = \Delta_p(v_i, v_j)d_{ij} \) and so \( \Delta_0(v_i, v_j)n_{ij} = \Delta_0(v_i, v_j)d_{ij}n_{ij} \). Now \( m_{ij} = 0 \) and \( d_{ij} = m_{ij}d_a + c_{ij}d_a \) and so \( \Delta_0(v_i, v_j)d_{ij}n_{ij} = \Delta_0(v_i, v_j)d_{ij}n_{ij} \). But by condition 2) of the theorem, \( q_1 \) and \( q_2 \) must be the same major path labeled \( lj \). We can arrive at a similar contradiction when \( n_{ij} = -1 \).

Lastly, we will now show that \( d_{ij} > 0 \). Consider the case when \((w_1, w_2) = (1, -1)\). Then the orthogonal diagonalization factor is \((0, 1)\) and hence \( \Delta_0(v_i, v_j) = y_{12} - x_{12} \). By construction \( d_{ij} > 0 \) and \( d_{ij} > 0 \). Now \( d_{ij} = m_{ij}d_a + 2c_{ij} \). We will show that \( \forall lj, c_{ij} \geq 0 \). Let \( v_i \) and \( v_j \) be vertices such that there is an edge labeled \( lj \) from \( v_i \) to \( v_j \). So \( \Delta_0(v_i, v_j) = (y_{12} - x_{12}) = m_{ij} \).

Suppose \( c_{ij} < 0 \). Then \( y_{12} < x_{12} \). Now \( m_{ij} \in \{1, -1, 0\} \) and hence, \( y_{12} = x_{12} \) and so by Lemma 13, there must be a directed path from \( v_i \) to \( v_j \) causing a cycle. So \( \forall lj, c_{ij} \geq 0 \). Hence \( d_{ij} = 1 \) and \( d_{ij} = m_{ij} + 2c_{ij} \).

If \( c_{ij} = 0 \) then we will show that \( m_{ij} > 0 \). Suppose \( m_{ij} \leq 0 \). Then \( \Delta_0(v_i, v_j) \leq 0 \). \( c_{ij} = 0 \) and hence \( y_{12} = x_{12} \) and hence \( y_{12} = x_{12} \). So by Lemma 13, there must be a directed path from \( v_i \) to \( v_j \) causing a cycle. So \( m_{ij} > 0 \) and hence \( d_{ij} > 0 \). Lastly, if \( c_{ij} > 0 \) then \( d_{ij} \geq 1 \) as \( m_{ij} \in \{1, -1, 0\} \). For both these cases we can easily establish that \( d_{ij} > 0 \).

\[ \square \]

D. Correctness of Mapping Cube Graphs

We will now show that our algorithm to map a cube graph is syntactically correct. Recall that \( L_C \) denotes the set of three labels \( \{1, 12, 13\} \). Let \( l \in L_C \) and let \( n_l \) and \( d_l \) denote its neighborhood and delay constants, respectively. We begin by first showing that the mapping preserves the neighborhood constant of the labels.

**Theorem 5:** If \( v_i \) and \( v_j \) are a pair of computation vertices with an edge labeled \( l \) directed from \( v_i \) to \( v_j \) then \( PA(v_i) = PA(v_j) + n_l \).

**Proof:** Let \( v_i \) and \( v_j \) be the vertices in diagonals \( D_p \) and \( D_q \), respectively. Let \( w_p \) and \( w_q \) denote the weights of \( D_p \) and \( D_q \), respectively. Then,
\[
w_1x_{1j} + w_2x_{2j} + w_3x_{3j} = w_p \quad \text{and} \quad w_1y_{1j} + w_2y_{2j} + w_3y_{3j} = w_q.
\]

Assume \( l = 11 \) and let there be an edge labeled \( l \) directed from \( v_i \) to \( v_j \). From the definition of a cube graph we obtain \( y_{1j} = x_{1j} + 1, y_{2j} = x_{2j} \) and \( y_{3j} = x_{3j} \). Consequently, \( w_q - w_p = w_i = 1 \). Since the diagonals are indexed in order of their weights, it follows that the index of \( D_p \) must be one more than the index of \( D_p \), that is, \( q = p + 1 \). The mapping algorithm maps vertices in \( D_p \) onto cell \( p \) and those of \( D_q \) onto cell \( p + w_p \) and hence, \( PA(v_i) = PA(v_j) + w_p \). Also from the mapping algorithm \( n_1 = w_p \). The proof is similar for \( l = 12 \) and \( l = 13 \).

\[ \square \]

**Theorem 6:** Let \( v_i \) and \( v_j \) be a pair of vertices with an edge labeled \( l \) directed from \( v_i \) to \( v_j \). Then \( TA(v_i) = TA(v_j) + d_l \).

**Proof:** If the label of \( l \) is either \( 11 \) or \( 12 \) then \( v_i \) and \( v_j \) and the edge are all in the same mesh graph within the same set in \( CG \), say, \( CG_i \). So \( y_{1j} - x_{1j} = 0 \) and from the mapping algorithm we obtain \( TA(v_i) - TA(v_j) = (y_{1j} - x_{1j})d_{11} + (y_{1j} - x_{1j})d_{12} \). If \( l = 11 \) then \( y_{1j} - x_{1j} = 0 \) and \( y_{1j} - x_{1j} = 1 \) and hence, \( TA(v_i) - TA(v_j) = d_{11} \). If \( l = 12 \) then \( y_{1j} - x_{1j} = 0 \) and \( y_{1j} - x_{1j} = 1 \) and hence \( TA(v_i) - TA(v_j) = d_{12} \).
If \( l = l_3 \) then \( y_0 - x_0 = 1, y_0 - x_2 = 0 \) and \( y_0 - x_3 = 0 \). Let \( v_s \) be a vertex in \( CG_2 \). Clearly, \( v_s \) must be a vertex in some mesh graph within \( CG_{l+1} \). From phase 3 of the mapping algorithm, it can be shown that \( TA(v_s) - TA(v_s) = d_{\parallel} \). □

We have to next establish that no two tokens appear simultaneously at the same input port of any cell. The next lemma comes in useful to do so.

**Lemma 14:** Let \( n_i \in \{1, -1\} \). Let \( P_1 \) and \( P_2 \) be two distinct major paths labeled \( l \) in \( G \) and let \( v_i \) and \( v_s \) be the computation vertices adjacent to the source vertices in \( P_1 \) and \( P_2 \), respectively. Let \( PA(v_i) = s_1, PA(v_s) = s_2 \) where \( s_1 \leq s_2 \). Let \( TA(v_i) = t_1 \) and \( TA(v_s) = t_2 \). If the two tokens initialized to the values represented by the source vertices in \( P_1 \) and \( P_2 \), respectively, appear simultaneously at the input port labeled \( l \) of a cell then \( (t_2 - t_1)n_i = (s_2 - s_1)d_i \).

**Proof:** Let \( i_1 \) and \( i_2 \) denote these two tokens and assume that they appear simultaneously at the input port of a cell, say, \( s \).

1) Let \( n_1 = 1 \). Then \( i_1 \) travels through intermediate cells \( 1, 2, \ldots, s_1 \) and \( i_2 \) travels through cells \( 1, 2, \ldots, s_2 \). Equating the times taken by these two tokens to reach \( s \), we obtain \( t_2 - t_1 = (s_2 - s_1)d_i \) and hence, \( (t_2 - t_1)n_i = (s_2 - s_1)d_i \).

2) Let \( n_1 = -1 \) and let \( m \) denote the total number of cells in the array. Then the tokens traversing through links labeled \( l \) are inserted through cell \( m \). Equating the times taken by \( i_1 \) and \( i_2 \) to reach \( s \), we obtain \( t_2 - t_1 = (s_2 - s_1)d_i \) and hence, \( (t_2 - t_1)n_i = (s_2 - s_1)d_i \). □

We are now ready to prove that the mapping ensures that distinct tokens never appear simultaneously at the same input port of any cell.

**Theorem 7:** Let \( P_1 \) and \( P_2 \) be two distinct major paths labeled \( l \). The mapping ensures that the two tokens initialized to the input values represented by the source vertices \( P_1 \) and \( P_2 \), respectively, never appear simultaneously at the input port labeled \( l \) of any cell.

**Proof:** Recall that \( h_1, h_2, \) and \( h_3 \) are the maximum values that a vertex can assume as its first, second, and third coordinates, respectively. Let \( v_i \) and \( v_s \) be the vertices adjacent to the source vertices in \( P_1 \) and \( P_2 \), respectively. From the mapping algorithm we obtain

\[
PA(v_i) - PA(v_s) = \Delta(P) = \sum_{i=1}^{3} k_i n_i
\]

where

\[
k_i = y_i - x_i
\]

and \(-h_i \leq k_i \leq h_i\).

\[
TA(v_i) - TA(v_s) = \Delta(T) = \sum_{i=1}^{3} k_i d_i
\]

Now assume that the two tokens meet in some cell. Then, by Lemma 13, we have \( (\Delta(T)n_i = (\Delta(P)d_i) \) \(*\) We will show that \(*\) cannot be satisfied.

1) Let \( n_1 = 1 \) and so by the mapping algorithm, \( d_{\parallel} = 1 \) and \( d_{\parallel} = 2 \). \( P_1 \) and \( P_2 \) are distinct major paths labeled \( l \) and so \( k_2 = k_3 \neq 0 \).

a) Let \( h_1 - h_2 = n_0 \geq 0 \). So \( d_{\parallel} = h_1 + 1 + 2n_0 \) and \(*\) reduces to \( k_3(h_1 + 1 + n_0) = k_2 \). Now \( h_1 + 1 + n_0 \geq 1 \) and so \( k_2 = 0 \) and \( k_3 = 0 \). Besides, \( h_2 \leq h_1 + n_0 \) and \(-h_1 \leq k_3 \leq h_2 \) and so \(*\) cannot be satisfied.

b) Let \( h_1 - h_2 + n_0 < 0 \) and so \( d_{\parallel} = h_1 + n_0 \) and \(*\) reduces to \( k_3(h_2 + 1) = k_2 \). Now \( h_2 \geq 1 \), and so \( k_2 \neq 0 \) and \( k_3 = 0 \). Besides, \(-h_2 \leq k_2 \leq h_2 \), and so \(*\) cannot be satisfied.

2) Let \( n_2 = -1 \). So \( d_{\parallel} = 1 \) and \( d_{\parallel} = 1 \).

a) Let \( h_1 - h_2 + n_0 \geq 0 \). So \( d_{\parallel} = h_1 + 1 + 2n_0 \) and \(*\) reduces to \( k_3(h_1 + 1 + n_0) = k_2 \). Now \( h_1 + 1 + n_0 \geq 1 \) and so \( k_2 = 0 \) and \( k_3 = 0 \). Besides, \(-h_2 \leq k_2 \leq h_2 \) and \(*\) cannot be satisfied.

b) Let \( h_1 - h_2 + n_0 < 0 \) and so \( d_{\parallel} = h_1 + n_0 \) and \(*\) reduces to \( k_3(h_2 + 1) = k_2 \). Now \( h_2 \geq 1 \), and so \( k_2 \neq 0 \) and \( k_3 = 0 \). Besides, \(-h_2 \leq k_2 \leq h_2 \), and so \(*\) cannot be satisfied.

Using the inequality relationships between \( k_1, k_2, k_3 \) and \( h_1, h_2, h_3 \), we can similarly establish that the two equations, \( \Delta(P)d_{\parallel} = \Delta(T)n_{\parallel} \) and \( \Delta(P)d_{\parallel} = \Delta(T)n_{\parallel} \) cannot be satisfied.

□

**REFERENCES**


I. V. Ramakrishnan received the B. Tech degree in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur, India, in 1975, the M.E. degree in automation from the Indian Institute of Science, Bangalore, India, in 1977, and the Ph.D. degree in computer science from the University of Texas at Austin in 1983.

From 1977 to 1979 he was a Scientific Officer at the National Center for Software Development and Computing Techniques at the Tata Institute of Fundamental Research, Bombay, India, where he designed and implemented a capability-based operating system for a closely coupled network of multi-processors. From August 1983 to August 1985 he was an Assistant Professor in the Department of Computer Science at the University of Maryland, College Park. He is currently an Assistant Professor in the Department of Computer Science at the State University of New York, Stony Brook. His research interests include parallel processing and fault-tolerant computing.

D. S. Fussell, photograph and biography not available at the time of publication.

Abraham Silberschatz received the Ph.D. degree in computer science from the State University of New York, Stony Brook, in 1976. He is currently a Professor of Computer Science at the University of Texas, Austin, specializing in the area of concurrent processing. His research interests include operating systems, database systems, distributed systems, and programming languages. He has published numerous journal and conference papers on such topics as language-based access control schemes, interprocess communication facilities, and graph-based locking protocols. He is also the coauthor of a recently published textbook entitled Operating Systems Concepts.

Dr. Silberschatz received the 1978 IEEE Computer Society Outstanding Paper Award for the paper "Capability Manager," which appeared in IEEE TRANSACTIONS ON SOFTWARE ENGINEERING.