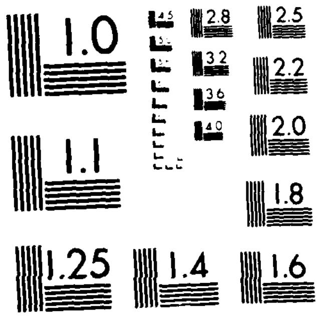
AD-A139 306 ON MAPPING CUBE GRAPHS ON VLSI ARRAY AND TREE ARCHITECTURES..(U) MARYLAND UNIV COLLEGE PARK CENTER FOR AUTOMATION RESEARCH I V RAMAKRISHNAN ET AL. DEC 83 1/1

UNCLASSIFIED CAR-TR-40 AFOSR-TR-84-0173 F/G 9/2 NL

END
DATE
FILMED
4 -84
DTIC


1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

3

CAR-TR-40
CS-TR-1358
December 1983

ON MAPPING CUBE GRAPHS ON VLSI ARRAY AND TREE ARCHITECTURES

I. V. Ramakrishnan
Department of Computer Science
University of Maryland
College Park, MD 20742

P. J. Varman
Department of Electrical Engineering
Rice University
Houston, TX 77001

COMPUTER VISION LABORATORY

CENTER FOR AUTOMATION RESEARCH

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

3

CAR-TR-40 December 1983
CS-TR-1358

ON MAPPING CUBE GRAPHS ON VLSI ARRAY AND TREE ARCHITECTURES

I. V. Ramakrishnan
Department of Computer Science
University of Maryland
College Park, MD 20742

P. J. Varman
Department of Electrical Engineering
Rice University
Houston, TX 77001

ABSTRACT

We formalize a model of array architectures suitable for VLSI implementation. A formal model of an arbitrarily structured tree machine is also presented. A mathematical framework is developed to transform cube graphs, which are data-flow descriptions of certain matrix computations, onto the array and tree models. All published algorithms for these computations can be obtained using the mathematical framework. In addition, novel linear-array algorithms for matrix multiplication are obtained. More importantly, the algorithms obtained for the tree model are of special significance. Besides their novelty, the independence of the tree algorithms from a specific inter-processor communication geometry make them robust to hardware faults as opposed to algorithms that are based on specific interconnection requirements.

Publication of this report was supported by the Air Force Office of Scientific Research under Contract F49620-83-C-0082. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

1. Introduction

Specialized array processors have been proposed as a means of handling compute-bound problems in a cost-effective and efficient manner [4,5,6]. These array processors are typically made up of simple, identical processing elements that operate in synchrony. Several array structures have been proposed that include linear arrays, rectangular arrays and hexagonal arrays. Simplicity and regularity of linear, rectangular and hexagonal array processors render them suitable for VLSI implementation. High performance is achieved by extensive use of pipelining and multiprocessing. 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.

A variety of algorithms have been designed for such arrays [1, 2, 5, 7, 10]. All these algorithms *exhibit the following feature*. They are composed of streams of data travelling in multiple directions at multiple speeds. Each processing element receives data from each of the streams, performs some simple operation and pumps them out (possibly updated). We will refer to such algorithms as "array algorithms". The array is typically comprised of *identical processors*, that is, they all execute the same set of instructions in every instruction cycle and they do not have any control unit. The array is driven by a single-phase or two-phase global clock [9].

A few methodologies have been proposed for transforming high-level specifications onto array algorithms [3,8,14]. Though these methodologies are imaginative they lack a mathematical basis. In this paper we outline a mathematical framework for transforming data-flow descriptions of matrix and related computations onto array machines and tree machines. The research presented herein is an extension to our work on transformation of high-level specifications onto linear-array algorithms [11]. In this paper we generalize the formal model of a linear array developed in [11] to include two-dimensional arrays (rectangular and hexagonal arrays). We also formalize another important model, namely, tree machines having arbitrary structure. Algorithms on such tree machines are particularly important. Any connected set of processors (that is, any two processors in the set can communicate with each other either directly or indirectly through other processors and communication links in the set) with no a priori topological restrictions can be used to execute these algorithms by forming a spanning tree of the connected set of processors. The connected set of processors could be the non-faulty processors in an underlying host machine (like a rectangular or a hexagonal array) which has both faulty and non-faulty processors and communication links. The hardware details of reconfiguring such a connected component of processors are provided in [12,13].

This paper is organized as follows. In Section 2 we formalize the array and tree machine models. We also introduce cube graphs which are data-flow descriptions of some matrix and related computations. In Section 3 we provide a precise formulation of correctly transforming cube graphs (referred to as *mapping* cube graphs in our terminology) onto the array and tree models. Algorithms for mapping cube graphs onto the array and tree models are also presented in Section 3. In the Appendix we provide a proof that the algorithms in Section 3 correctly transform a cube graph into array and tree algorithms.

2. Computational Models

We now formalize the array model, the tree model and cube graphs. We begin with a formal definition of an array processor.

2.1. Array Machine Model

Let $I_1, I_2, .., I_z$ be z sets of sequences of integers where each I_j ranges from 1 to m_j. Let $I \subseteq I_1 \times I_2 \times .. \times I_z$.

Definition 2.1: An *array machine* Ar is a 4-tuple $<N_{Ar}, T_{Ar}, \delta_{Ar}, \psi_{Ar}>$ where:

1. N_{Ar} is a set of identical processors.
2. $T_{Ar}=\{l1, l2, .., lk\}$ is the set of labels.
3. $\delta_{Ar}:N_{Ar}$ is a one-one function that assigns coordinates to every processor in the Euclidean z-space.
4. Every processor in the array has k input ports and k output ports. with each input port and output port assigned a unique label from T_{Ar}.
5. 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 processor. In a single-phase clocking scheme all processors are activated in every phase and every processor computes a k-ary function ψ_{Ar}. In a two-phase clocking scheme, adjacent processors are activated during opposite phases of the clock and every processor computes ψ_{Ar} in the phase it is active.

The value of z and the communication geometry determine the structure of the array processor. In this paper we will be examining three types of array processors, namely, linear, mesh and hexagonal arrays which are well-suited for VLSI implementation [4,9]. We now formalize these three arrays. Our definition captures the "nearest-neighbor" interconnection of these arrays and also the intuitive notion of a data stream used earlier in the description of array algorithms. $\forall l j \in T_{Ar}$, let n_{lj} be the *neighborhood constant* associated with label lj.

Definition 2.2: A *linear array* L_{Ar} is an array processor with $z=1$, that is, $I \subseteq I_1$. Besides the linear array has the following communication features. Let p be a processor index. Then, $\forall l j \in T_{Ar}$, the output port labelled lj of p is connected to the input port labelled lj of $p+n_{lj}$ where $n_{lj} \in \{1,-1,0\}$.

Let L_H, L_V and L_O be three disjoint sets of labels such that $L_H \bigcup L_V \bigcup L_O = T_{Ar}$.

Definition 2.3: A *mesh array* M_{Ar} is an array processor with $z=2$, that is, $I \subseteq I_1 \times I_2$. Besides, the mesh array has the following communication features. Let $<p,q>$ denote the coordinate of any processor in the mesh array. Then,

1. $\forall l j \in L_O$, the output port labelled lj of $<p,q>$ is connected to its own input port labelled lj, that is, $n_{lj}=0$.
2. $\forall l j \in L_H$, the output port labelled lj is conected to the input port labelled lj of $<p+n_{lj},q>$ where $n_{lj} \in \{1,-1\}$.
3. $\forall l j \in L_V$, the output port labelled lj is connected to the input port labelled lj of $<p,q+n_{lj}>$ where $n_{lj} \in \{1,-1\}$.

Let $L_H \bigcup L_V \bigcup L_O \bigcup L_T = T_{Ar}$ be four disjoint sets of labels.

Definition 2.4: Let $c \in \{1,-1\}$ denote the hexagonal array constant. A *hexagonal array* H_{Ar} is similar to a mesh array with the additional communication feature that $\forall l j \in L_T$, the output port labelled lj of $<p,q>$ is connected to the input port labelled lj of $<p+n_{lj}, q+n_{lj}c>$ where $n_{lj} \in \{1,-1\}$.

Fig. 2.1, Fig. 2.2 and Fig. 2.3 illustrate a linear, mesh and hexagonal array processors. In the figures I_1, I_2 and I_3 denote external input ports and O_1, O_2 and O_3 denote external output ports.

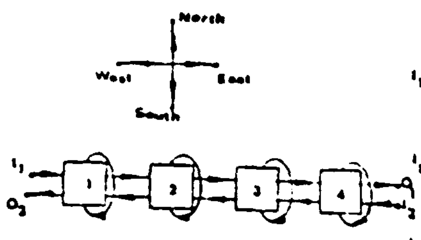


Figure 2·1

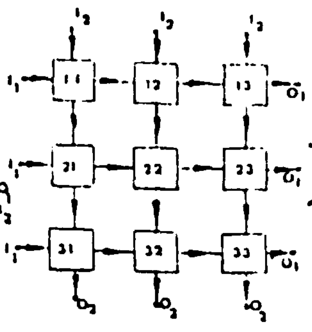


Figure 2·2

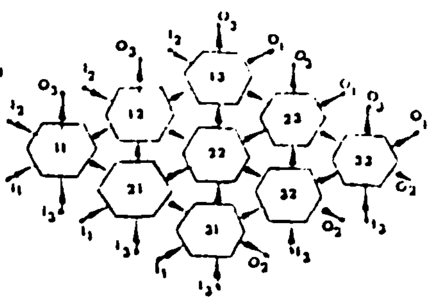


Figure 2·3

In Fig. 2.1, the links between processors directed from west to east are labelled $\ell 1$ and those directed from east to west are labelled $\ell 2$. The links connecting a processor back to itself are labelled $\ell 3$. The neighborhood constants are $n_{\ell 1} = 0$, $n_{\ell 2} = -1$, and $n_{\ell 3} = 0$.

In Fig. 2.2, the links directed from west to east are labelled $\ell 1$ and the links directed from north to south are labelled $\ell 2$. $L_H = \{\ell 1\}$ and $L_V = \{\ell 2\}$ and $n_{\ell 1} = n_{\ell 2} = 1$.

In Fig. 2.3, the links pointing northeast are labelled $\ell 1$, the links pointing southeast are labelled $\ell 2$ and the links directed from south to north are labelled $\ell 3$. $L_H = \{\ell 1\}$, $L_V = \{\ell 2\}$ and $L_T = \{\ell 3\}$. $n_{\ell 1} = n_{\ell 2} = 1$ and $n_{\ell 3} = -1$. The hexagonal array constant $c = -1$.

We will refer to the processor whose input port labelled ℓj is connected to the output port labelled ℓj of processor p as its *neighbor with respect to label* ℓj. If a processor q is the neighbor of p with respect to label ℓj then q can only receive data from p on the link labelled ℓj connecting them. Similarly p can only send data to q on the same link. The links connecting any two processors are unidirectional. Impose a direction on the links such that the sender is at the tail end and the receiver at the other end. A stream then, is a directed path through processors and links having the same label.

We model the speed of data in streams by associating a queue of buffers in the communication links. More precisely, let s be a processor in the array. Let $si_t=<si_t^1, si_t^2, .., si_t^k>$ denote the k-tuple input to processor s at time t where si_t^j is the value at the input port labelled lj of processor s at time t. Let $so_t=<so_t^1, so_t^2, .., so_t^k>$ denote the k-tuple output computed by processor s at time t, that is, $\psi_{Ar}(si_t)=so_t$. Elements in a data stream travel at a constant velocity, and hence a non-zero positive *delay constant* d_{lj} is associated with every label lj in T_{Ar} such that so_t^j appears at the output port labelled lj of s at time $t+d_{lj}$. The delay d_{lj} can be implemented as a queue using a shift register of length d_{lj}-1.

2.2. Tree Machine Model

We are now in a position to formalize a tree machine as follows.

Definition 2.5: A *tree machine* Γ_{Ar} is a set of processors in a tree that are indexed by some depth-first traversal of the tree. Besides it has the following communication features. Let p be a proceesor index. Then, $\forall lj \in T_{Ar}$, the output port labelled lj of p is connected to the input port labelled lj of $p+n_{lj}$ where $n_{lj} \in \{1,-1,0\}$. Besides, for every label $lj \in T_{Ar}$, and for every communication link between the output port of a processor indexed p and the input port of the processor indexed $p+n_{lj}$, associate a delay $\delta(lj,p)=d_{lj}+\Delta(lj,p)$ where d_{lj} is the delay constant associated with any communication link labelled lj, and $\Delta(lj,p)$ is the perturbation delay between processor p and $p+n_{lj}$.

A tree machine is a generalization of the linear array model (see definition 2.2). The term *tree machine* signifies that the interconnection between processors in the array can be represented by an arbitrary tree, the vertices of which represent processors and edges represent the adjacency relation between the processors. The corresponding representation for a linear array could be a special case of a tree forming a path graph.

On any such tree of processors it is possible to simulate the data flow through a linear array by routing the data streams through a closed path around the periphery of the tree (see Fig. 2.4).

closed path:
(abc ... hij)

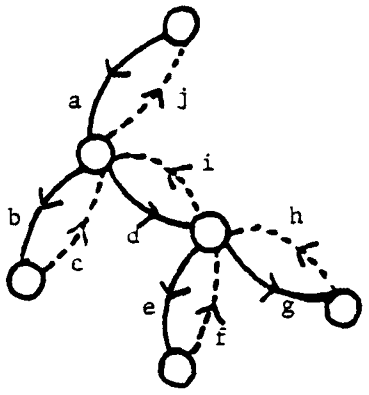


Figure 2.4

The major difference between this "logical pipeline" in a tree machine and a "physical pipeline" in the linear array model is that in the former, logically adjacent processors (i.e., the pair indexed i and i+1) need not be physically adjacent. Since all the data streams flow through the array at a finite velocity, the implication of this physical separation is that the delay encountered by a data element in traversing the array from processors i to i+1 (or vice versa) is a function of both the delay constant associated with the stream to which that element belongs and of the physical separation between the processors.

Our tree machine model (definition 2.5) is motivated by this idea. The delay for a data stream l_j between processors indexed p and $p+n_{l_j}$ is represented by

$\delta(lj,p) = d_{lj}+\Delta(lj,p)$. The first quantity is the delay constant associated with any link labelled lj and the second quantity is the perturbation in this delay caused by the non-adjacent physical arrangement of the logically adjacent processors p and $p+n_{lj}$.

2.3. Cube Graphs

We now provide a formal definition of graphs that we will be mapping later on onto linear, mesh and hexagonal arrays and tree machines.

Let $G=<V,E,L_G>$ be a labelled DAG where:

1. $V=V_G \bigcup SO_G \bigcup SI_G$, 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. $L_G=\{l1,l2,l3\}$ is a set of labels.
3. Every vertex in V_G has three incident edges and three outgoing edges, where each incident and outgoing edge is assigned a unique label from L_G.

In any execution of G on the array or the tree, every computation vertex in G is a single instance of a function evaluation that is performed in a cycle by a processor in the array or the tree. As all processors compute the same function, every computation vertex also represents the same function.

We can view the k incoming edges to a computation vertex v_x as representing the k-tuple input value to the processor that evaluates v_x. Similarly, we can view the k outgoing edges from v_x as the k-tuple output value that is computed by the processor on evaluating v_x. Throughout the rest of this paper we will adopt the terminology that a source vertex represents an input value and a sink vertex represents an output value.

Let J_1, J_2 and J_3 be three sequences of integers ranging from 0 to h_1, 0 to h_2 and 0 to h_3 respectively. Let $J \subseteq J_1 \times J_2 \times J_3$.

Definition 2.6: G is a *Cube Graph* iff there exists a one-one function $F: V_G \rightarrow J$ that satisfies the following: Let F_{l1}, F_{l2} and F_{l3} be three projection functions of F, that is, if $F(v_x) = <c_1, c_2, c_3>$ then $F_{l1}(v_x) = c_1$, $F_{l2}(v_x) = c_2$ and $F_{l3}(v_x) = c_3$. Let v_x and v_y be any two computation vertices in V_G. Then, for any label $lj \in L_G$, there exists a path comprised only of edges labelled lj passing through v_x and v_y such that the distance from v_x to v_y is d iff $F_{lj}(v_y) = F_{lj}(v_x) + d$ and $\forall li \in L_G - \{lj\}$, $F_{li}(v_y) = F$ [illegible] $_x$).

Henceforth, throughout the rest of this paper G will denote a cube [illegible]ph. A cube graph is an object in Euclidean 3-Space and we will refer to the 3 axes [illegible], l2nd and l3rd axes. $h_1 \geq 1$, $h_2 \geq 1$ and $h_3 \geq 1$ are the maximum dimensions along l1st, l2nd and l3rd axes respectively. If v_x is a computation vertex in a cube graph then we will denote $F_{l1}(v_x)$, $F_{l2}(v_x)$ and $F_{l3}(v_x)$ by x_{l1}, x_{l2} and x_{l3} respectively. Let v_0 denote the vertex whose coordinates are $<0,0,0>$.

3. Mapping Cube Graphs on Arrays and Trees

Intuitively mapping of G onto an array or tree machine assigns each computation vertex of G to a processor in the machine at a particular time step and also fixes the delay and neighborhood constant for every label in L_G. Assuming discrete time steps, let $T = \{0,1,2,..\}$ be the sequence of natural numbers representing the progress of a computation from its start at time 0.

Definition 3.1: A mapping of G onto a linear, rectangular, hexagonal or tree machines is a 4-tuple <PA,TA,NA,DA> where:

1. $T_{A_r} = L_G$

2. $PA: V_G \rightarrow I$ and $TA: V_G \rightarrow T$ are many-one functions mapping computation vertices onto processors and time steps respectively.

3. Let I^+ be a set of positive non-zero integers. $NA: L_G \rightarrow \{1,-1,0\}$ and $DA: L_G \rightarrow I^+$ are many-one functions assigning neighborhood constants and delay constants respectively.

[Note: $NA(lj) = n_{lj}$ and $DA(lj) = d_{lj}$]

We next formalize a correct mapping.

Definition 3.2: A mapping is *syntactically correct* iff

1. $\forall lj \in L_G$ and for any pair of computation vertices v_x and v_y, if there is an edge labelled lj directed from v_x to v_y, then $PA(v_y)$ is the neighbor of $PA(V_x)$ with respect to label lj and $TA(v_y) = TA(v_x) + d_{lj}$.

2. No two values appear simultaneously at the same input port of any processor.

3.1. Linear Array Mapping

We now describe the algorithm to map G onto a linear array L_{A_r}. We begin by developing some appropriate terminology for describing the algorithm.

Let $w_L = <w_1, w_2, w_3>$ be a triple where $w_1 = 1, w_2 \in \{1,-1\}$ and $w_3 \in \{1,-1\}$.

Definition 3.3: A *linear diagonalization* D_L of a cube graph is a pair $<D,w>$ with the following properties.

1. $D=\{D_1, D_2, .., D_k\}$ is a family of sets of computation vertices and $D_1 \bigcup D_2 \bigcup .. \bigcup D_k = V_G$.

2. $\forall D_p \in D$, if v_x and v_y are in D_p then $w_1x_{l1}+w_2x_{l2}+w_3x_{l3} = w_1y_{l1}+w_2y_{l2}+w_3y_{l3}$.

3. $\forall D_p \in D$ and $\forall D_q \in D$, $p<q$ iff $\forall v_x$ in D_p and $\forall v_y$ in D_q, $\sum_{i=1}^{i=3} w_i x_{li} < \sum_{i=1}^{i=3} w_i y_{li}$.

We will refer to w_L as the *linear diagonalization factor* of a cube graph and to any $D_p \in D$ as a *linear diagonal*. If v_x is in D_p then we will refer to $\sum_{i=1}^{i=3} w_i x_{li}$ as the weight of D_p.

We assign consecutive indices to the diagonals in D in increasing order of their weights with the diagonal having the least weight assigned index 1.

Algorithm

We are now in a position to describe the linear array mapping algorithm.

1. Perform a linear diagonalization $D_L=<D,w_L>$ of the cube graph. For every $D_p \in D$ assign a proceesor indexed p.

2. Choose $n_{l1}=w_1$, $n_{l2}=w_2$ and $n_{l3}=w_3$. This fixes the neighborhood constants of the labels.

3. Choose $d_{l1}=1$. If $n_{l2}=1$ then choose $d_{l2}=2$ else choose $d_{l2}=1$. Choose d_{l3} as follows.

   If $n_{l2}=1$ then if $h_1-h_2+n_{l3} \geq 0$ then choose $d_{l3}=h_1+1+2n_{l3}$ else choose $d_{l3}=h_2+1+n_{l3}$

   If $n_{l2}=-1$ then if $h_2-h_1+n_{l3} \geq 0$ then choose $d_{l3}=2h_2+1+n_{l3}$ else choose $d_{l3}=2h_1+1-n_{l3}$.

4. Map vertices in D_i onto processor i, that is, $\forall v_x$ in D_i, let $PA(v_x)=i$.

5. Let $TA(v_x)=\sum_{i=1}^{i=3} x_{li}d_{li} + t_1$ where $TA(v_0)=t_1$

3.2. Mesh Array Mapping

We next describe the algorithm to map G onto a mesh array M_{Ar}.

Let $w_m=<w_1,w_2,w_3>$ be a triple where $w_1=1$, $w_2\in\{1,-1\}$, and $w_3=1$. Let $L_G = L_H \bigcup L_V$. Let $l1\in L_H$ and $l3\in L_V$.

Definition 3.4: A ***mesh diagonalization*** D_M of a cube graph is a pair $<D,w_M>$ with the following properties.

1. $D=\{D_{<1,1>}, D_{<1,2>}, \ldots, D_{<m,s>}\}$ is a family of sets of computation vertices and $D_{<1,1>}\bigcup D_{<1,2>}\bigcup \cdots \bigcup D_{<m,s>}=V_G$.

2. For any $D_{<p,q>}\in D$, if v_x and v_y are in $D_{<p,q>}$ then $\forall li\in L_H$ and $\forall lj\in L_V$, $\sum_{li} w_{li}x_{li} = \sum_{li} w_{li}y_{li}$ and $\sum_{lj} w_{lj}x_{lj} = \sum_{lj} w_{lj}y_{lj}$

3. $\forall D_{<p,q>}\in D$ and $\forall D_{<r,s>}\in D$, $p<r$ iff $\forall v_x$ in $D_{<p,q>}$ and $\forall v_y$ in $D_{<r,s>}$, and $\forall li\in L_H$, $\sum_{li} w_{li}x_{li} < \sum_{li} w_{li}y_{li}$. Similarly, $q<s$ iff $\forall lj\in L_V$, $\sum_{lj} w_{lj}x_{lj} < \sum_{lj} w_{lj}y_{lj}$.

We will refer to w_M as the ***mesh diagonalization factor*** of a cube graph and to any $D_{<p,q>}\in D$ as a ***mesh diagonal***. If v_x is in $D_{<p,q>}$ then we will refer to $\sum_{li} w_{li}x_{li}$ where $li\in L_H$ as the ***horizontal weight*** and $\sum_{lj} w_{lj}x_{lj}$ where $lj\in L_V$ as the ***vertical weight*** of $D_{<p,q>}$ respectively. p and q will denote the ***horizontal*** and ***vertical*** indices respectively.

We assign consecutive horizontal indices to the diagonals in increasing order of their horizontal weights with the diagonal having the least horizontal weight assigned the horizontal index 1. Similarly, we assign consecutive vertical indices to the diagonals in increasing order of their vertical weights with the diagonals having the least vertical weight assigned the vertical index 1.

Algorithm

We are now in a position to describe the mesh array mapping algorithm.

1. Perform a mesh diagonalization $D_M = <D,w_M>$ of the cube graph. For every $D_{<p,q>} \in D$ assign a processor to the pth row and qth column of a mesh.
2. Choose $n_{l1}=w_1$, $n_{l2}=w_2$ and $n_{l3}=w_3$. This fixes the neighborhood constants of the labels.
3. Choose $d_{l1}=1$, $d_{l3}=1$. If $w_2=1$ then choose $d_{l2}=2$ else choose $d_{l2}=1$.
4. Map vertices on $D_{<p,q>}$ onto the processor in the pth row and qth column, that is, $\forall v_x$ in $D_{<p,q>}$, let $PA(v_x)=<p,q>$.
5. Let $TA(v_x)=\sum_{i=1}^{i=3} x_{li}d_{li} + t_1$ where $TA(v_0)=t_1$.

3.3. Hexagonal Array Mapping

We describe the algorithm to map G onto a hexagonal array H_{Ar}. Let $w_H=<w_1,w_2,w_3>$ be a triple where $w_1=1, w_2=1$ and $w_3 \in \{1,-1\}$. Let the hexagonal array constant $c \in \{1,-1\}$. Let $L_G = L_H \bigcup L_V \bigcup L_T$ and let $l1 \in L_H$, $l2 \in L_V$, and $l3 \in L_T$.

Definition 3.4: A *hexagonal diagonalization* D_H of a cube graph is a pair $<D,w_H>$ with the following properties.

1. $D=\{D_{<1,1>}, D_{<1,2>}, .., D_{<m,n>}\}$ is a family of sets of computation vertices and $D_{<1,1>} \bigcup D_{<1,2>} \bigcup .. \bigcup D_{<m,n>} = V_G$.

2. For any $D_{<p,q>} \in D$, if v_x and v_y are in $D_{<p,q>}$ then $w_1 x_{l1} + w_3 x_{l3} = w_1 y_{l1} + w_3 y_{l3}$ and $w_2 x_{l2} + w_3 x_{l3} c = w_2 y_{l2} + w_3 y_{l3} c$.

3. $\forall D_{<p,q>} \in D$ and $\forall D_{<r,s>} \in D$, $p<r$ iff $\forall v_x$ in $D_{<p,q>}$ and $\forall v_y$ in $D_{<r,s>}$ $w_1 x_{l1} + w_3 x_{l3} < w_1 y_{l1} + w_3 y_{l3}$. Similarly, $q<s$ iff $w_2 x_{l2} + w_3 x_{l3} c < w_2 y_{l2} + w_3 y_{l3} c$.

We will refer to w_H as the ***hexagonal diagonalization factor*** of a cube graph and to any $D_{<p,q>} \in D$ as a ***hexagonal diagonal***. If v_x is in $D_{<p,q>}$ then we will refer to $w_1 x_{l1} + w_3 x_{l3}$ as the ***horizontal weight*** and $w_2 x_{l2} + w_3 x_{l3} c$ as the ***vertical weight*** of $D_{<p,q>}$ respectively. p and q will denote the ***horizontal*** and ***vertical*** indices respectively.

We assign consecutive horizontal indices to the diagonals in increasing order of their horizontal weights with the diagonal having the least horizontal weight assigned the horizontal index 1. Similarly, we assign consecutive vertical indices to the diagonals in increasing order of their vertical weights with the diagonals having the least vertical weight assigned the vertical index 1.

Algorithm

We now describe the hexagonal array mapping algorithm.

1. Perform a hexagonal diagonalization $D_H = <D, w_H>$ of the cube graph. For every $D_{<p,q>} \in D$ assign a processor to the pth row and qth column of a mesh.

2. Choose $n_{l1}=w_1$, $n_{l2}=w_2$ and $n_{l3}=w_3$. This fixes the neighborhood constants of the labels.

3. Choose $d_{l1}=1$, $d_{l2}=1$ and $d_{l3}=1$.

4. Map vertices on $D_{<p,q>}$ onto the processor in the pth row and qth column, that is, $\forall v_x$ in $D_{<p,q>}$, let $PA(v_x)=<p,q>$.

5. Let $TA(v_x)=\sum_{i=1}^{i=3} x_{li}d_{li} + t_1$ where $TA(v_0)=t_1$.

3.4. Mapping on Tree Machines

Unlike in the linear-array mapping we are required to constrain the choice of w_1, w_2 and w_3. Let $<w_1,w_2,w_3>\in\{<1,1,1>, <1,-1,-1>\}$. A linear diagonalization is performed on the cube graph before being mapped onto the tree amchine. The first four steps involved in mapping a cube graph on a tree processor is the same as the first four steps in mapping cube graphs onto linear arrays. An additional step is involved for fixing the perturbation delays as follows. Let p be a processor index in the tree. (Recall that indexing is done by a depth-first traversal of the tree.)

case 1: If $<w_1,w_2,w_3>=<1,1,1>$ then $\Delta(l1,p)=\Delta(l2,p)=\Delta(l3,p)$.

case 2: If $<w_1,w_2,w_3>=<1,-1,-1>$ then $\Delta(l1,p)=-\Delta(l2,p+1)=-\Delta(l3,p+1)$.

The final step involves fixing the times at which the vertices are mapped. Let $v_x\in D_p$. Then $TA(v_x)=t_1+\sum_{i=1}^{i=3} x_{li}d_{li}+\sum_{j=1}^{p-1}\Delta(l1,j)$ where $TA(v_0)=t_1$.

The constraints on the delay perturbations (cases 1 and 2 above) are motivated by the following discussion. Let T be an arbitrary tree whose vertices are numbered by some depth-first traversal of the tree as shown in Fig. 3.1. The vertex numbered i will be referred to as v_i. Now replace each edge in the tree by a pair of edges between the two vertices and consider a closed path in this graph from v_1 back to itself that visits all the vertices in the order $v_1, v_2, \ldots, v_n$ as shown in Fig. 3.2.

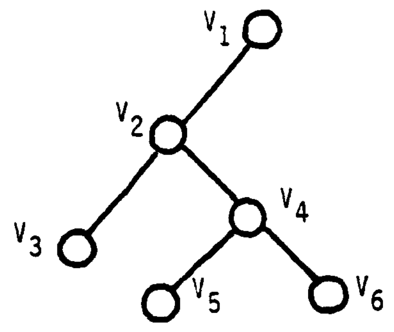


Figure 3.1

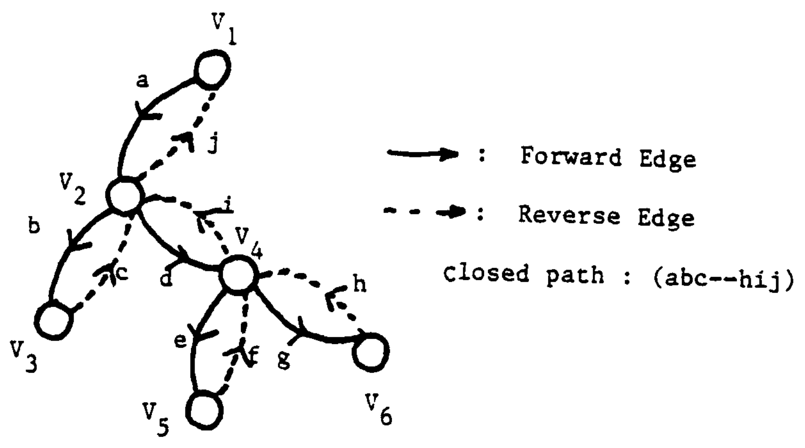
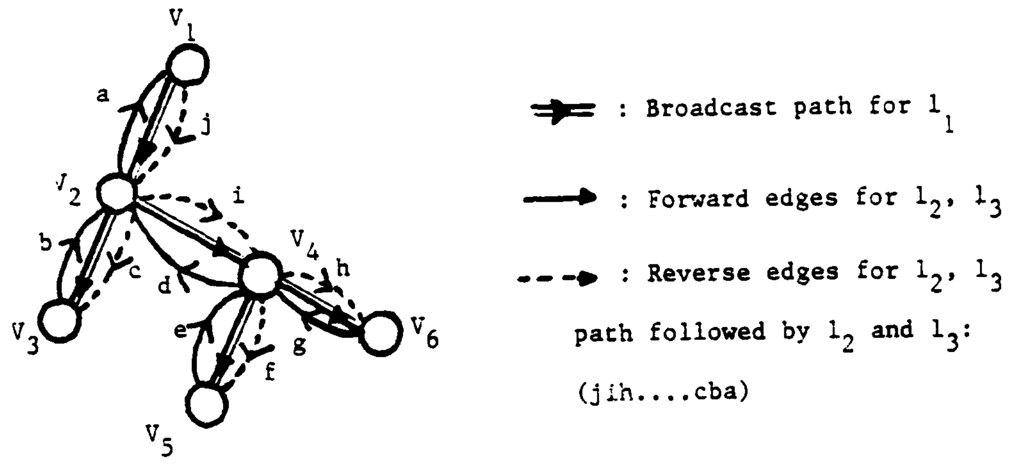


Figure 3.2

Such a path is composed of *forward* edges (those encountered while traversing from v_i to v_j, $i<j$) and *reverse* edges (those used to backtrack over previously visited vertices). Each reverse edge is assumed to have a constant delay d associated with it; a forward edge has a delay (d_{l1}, d_{l2} or d_{l3}) which depends on the label ($l1, l2$ or $l3$) of the stream traversing the edge.

In case 1, all the three streams $l1$, $l2$ and $l3$ traverse the closed path mentioned above. If there are x_p reverse edges in this path between v_p and v_{p+1} (note $x_p \geq 0$), then the effective delay for a stream labelled lj in traversing between v_p and v_{p+1} is $\delta(lj,p)=d_{lj}+x_p d$, corresponding to a delay perturbation $x_p d$. Note that the perturbation delay between v_p and v_{p+1} for any p, is the same for all labels.

In case 2, elements of stream $l1$ propagate from v_1 to the leaf vertices in a series of local broadcast steps. An element at v_p is broadcast to all vertices v_q, $q>p$, that are adjacent to v_p in the tree as shown in Fig. 3.3.



V_1
V_2
V_3
V_4
V_5
V_6
a
b
c
d
e
f
g
h
i
j
: Broadcast path for l_1
: Forward edges for l_2, l_3
: Reverse edges for l_2, l_3
path followed by l_2 and l_3:
(jih....cba)

Figure 3.3

The elements encounter a delay d_{l1} in moving from v_p to v_q. Owing to the depth-first numbering scheme, the difference between the time at which the values of a data element reaches v_{p+1} and the time at which it reaches v_p is $(x_p-1)d_{l1}$, where x_p is the number of reverse edges between v_p and v_{p+1}. Note however, that the element does not traverse these reverse edges, but a copy of its value reaches v_{p+1} by the direct broadcast path. Thus if $x_p=0$ (i.e.,v_p and v_{p+1} are physically adjacent in the tree) then the element will reach v_{p+1}, d_{l1} cycles later than it reaches v_p; else it will reach v_{p+1} at the same or earlier time than it reaches v_p. The effective delay encountered between v_p and v_{p+1} is $\delta(l1,p)=-(x_p-1)d_{l1}$, corresponding to a perturbation $\Delta(l1,p)=-x_pd_{l1}$.

Elements of streams of $l1$ and $l2$ traverse a closed path around the tree as before, but in the direction opposite to that in case 1, that is, in the direction $v_n,v_{n-1},..,v_1$. The effective delay for either of these streams (say $l2$) between v_{p+1} and v_p is $d_{l2}+x_pd$, corresponding to a perturbation $\Delta(l2,p+1)=x_pd=\Delta(l3,p+1)$. The conditions in case 2 can be satisfied by choosing $d=d_{l1}$.

In the appendix we have shown that the mapping algorithms for tree machines, linear, mesh and hexagonal arrays correctly map a cube graph.

Recall that the host machine inserts input values and extracts the result values from the array. We now describe the evaluation of the times at which insertion and extraction must be done. Also recall that the source vertex represents an initial value and the sink vertex represents a final value. Without loss of generality, let v_x be the computation vertex connected to a source (sink) vertex by an edge labelled l. The delays in the links having identical labels are all the same. Hence, if the distance of the processor (onto which v_x is mapped) from the external input (output) port is k then the input (output) value represented by the source (sink) vertex must be inserted (extracted) into (from) the array by the host at time t-k n_l (t+k n_l).

We next synthesize three algorithms to illustrate our mapping techniques. The first involves synthesis of a novel linear-array matrix multiplication algorithm that we first reported in [10]. We will then synthesize another matrix multiplication algorithm on the tree machine. In our final example we will synthesize an algorithm for multiplication of band matrices on a hexagonal array that appeared in [5].

Example 3.1 Consider multiplication of two dense matrices A and B as shown below.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}$$

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

$$c_{ij}^{(k+1)}=c_{ij}^{(k)}+a_{ik}b_{kj},\ 1\leq i,k\leq 2 \text{ and } 1\leq j\leq 3$$
$$c_{ij}^{(1)}=0$$

The data-flow description of this computation is shown in Fig. 3.4.

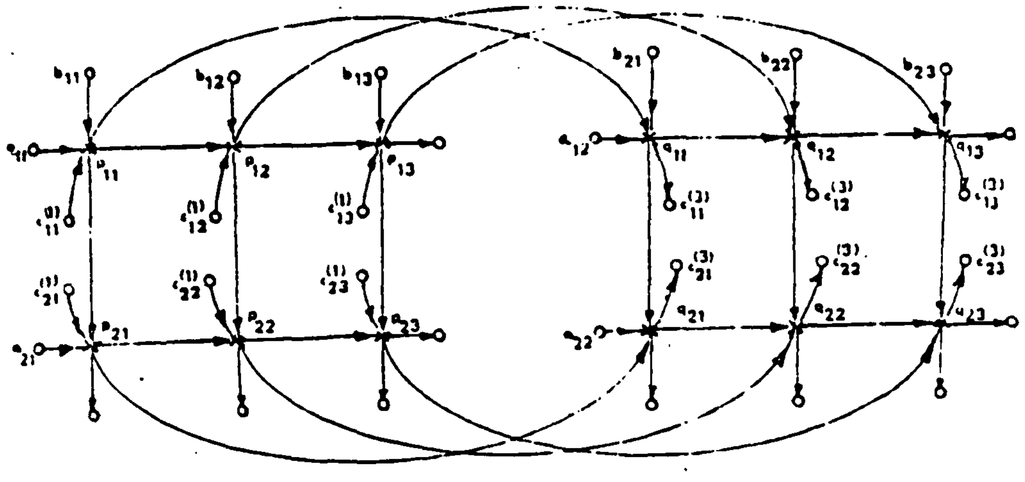


Figure 3.4

In Fig. 3.4, p_{ij} and q_{ij} denote computation vertices. The horizontal, vertical and oblique incident edges of p_{ij} are labelled $l1$, $l2$ and $l3$ respectively. Similarly the horizontal, vertical and oblique outgoing edges of p_{ij} are labelled $l1$, $l2$ and $l3$ respectively. If the horizontal, vertical and oblique incident edges of p_{ij} or q_{ij} represent the values a, b and c respectively then the horizontal, vertical and oblique outgoing edges of p_{ij} or q_{ij} represent the values a, b and c+ab respectively. In Fig. 3.4, the oblique input edge incident on p_{ij} represents the value $c_{ij}^{(1)}$ which is 0. The oblique outgoing edge from q_{ij} reresents the final (output) value $c_{ij}^{(3)}$ of c_{ij}, i.e., $a_{i1}b_{ij}+a_{i2}b_{2j}$.

The graph in Fig. 3.4 is a cube graph as illustrated in Fig. 3.5. The cube graph is shown without the source and sink vertices for purposes of clarity. The maximum dimensions of $l1^{th}$,$l2^{nd}$ and $l3^{rd}$ axes is 2, 1 and 1 respectively, i.e., $h_1=2$, $h_2=1$ and $h_3=1$.

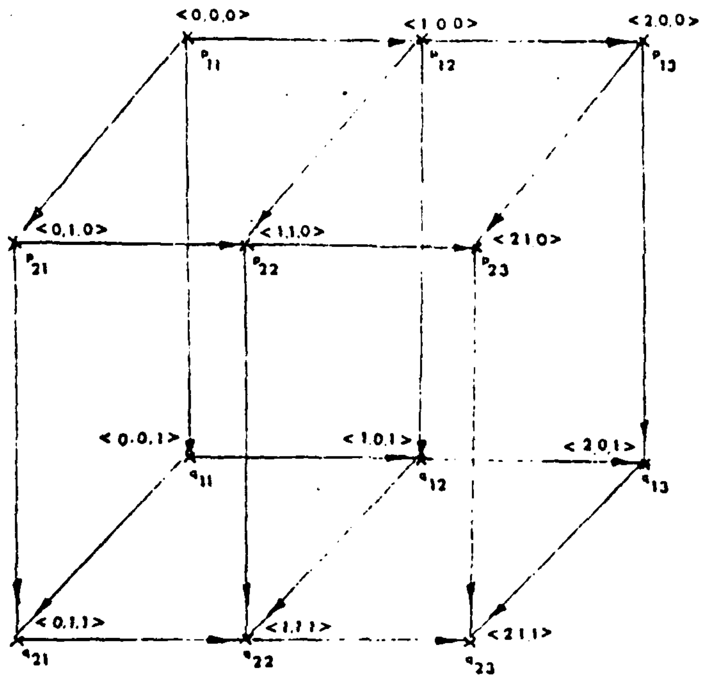


Figure 3.5

We next map this graph onto a linear array using the linear-array mapping algorithm.

Let $w_L=<w_1, w_2, w_3>=<1,1,-1>$. For this choice of w_L, the set D of diagonals is comprised of $D_1=\{ q_{11} \}$, $D_2=\{ p_{11}, q_{12}, q_{21} \}$, $D_3=\{ p_{12}, p_{21}, q_{13}, q_{22} \}$, $D_4=\{p_{13}, p_{22}, q_{23} \}$, $D_5=\{p_{23} \}$.

We use $|D|=5$ processors indexed from 1 to 5. The neighborhood constants for labels $l1$, $l2$ and $l3$ are $n_{l1}=1$, $n_{l2}=1$ and $n_{l3}=-1$. The vertices in D_i are mapped onto processor indexed i. The delays for the labels $l1$, $l2$ and $l3$ are $d_{l1}=1$, $d_{l2}=2$ and $d_{l3}=1$. The resulting mapping of the entire cube graph is shown in Fig. 3.6. The times at which a computation vertex is mapped is indicated by the side of the computaion vertex, for instance, p_{21} is mapped onto processor 3 at time t_1+2. If A and B were $n \times n$ matrices then the synthesized algorithm above would require $O(n)$ processors and will take $O(n^2)$ time steps to compute the result matrix.

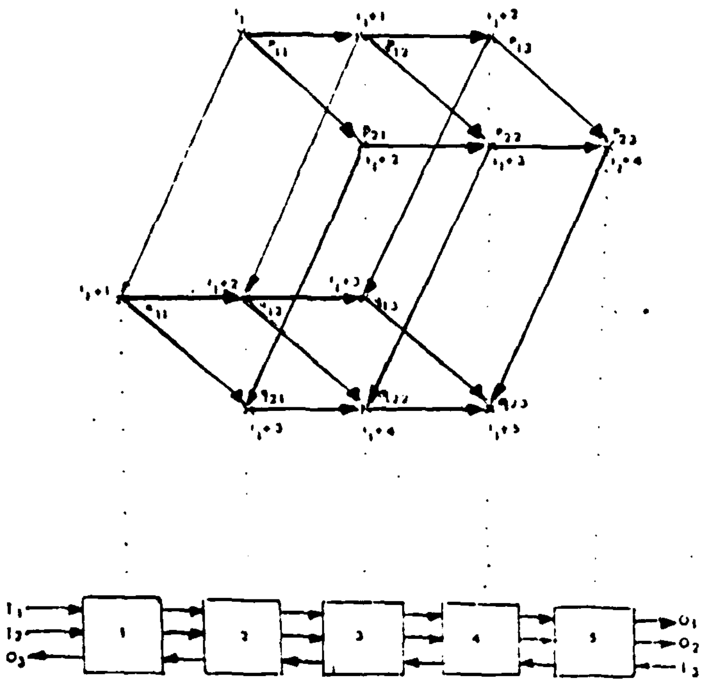


Figure 3-6

Example 3.2: Consider again multiplication of the two matrices in the previous example. We will synthesize a tree algorithm for multiplying the two matrices.

Let $w_L = \langle w_1, w_2, w_3 \rangle = \langle 1,-1,-1 \rangle$. For this choice of w_L, the set D of diagonals is comprised of $D_1 = \{ q_{21} \}$, $D_2 = \{ q_{22}, q_{11}, p_{21} \}$, $D_3 = \{q_{23}, q_{12}, p_{22}, p_{11}\}$, $D_4 = \{q_{13}, p_{23}, p_{12} \}$, $D_5 = \{p_{13} \}$.

We use $|D| = 5$ processors indexed from 1 to 5. The neighborhood constants for labels $l1$, $l2$ and $l3$ are $n_{l1} = 1$, $n_{l2} = n_{l3} = -1$. Vertices in D_i are all mapped onto processor indexed i. The delays for labels $l1$, $l2$ and $l3$ are $d_{l1}=1, d_{l2}=1$ and $d_{l3}=6$.

Let the five vertex tree representing the tree array be as shown in Fig. 3.7 below.

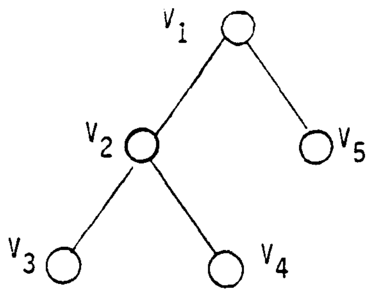
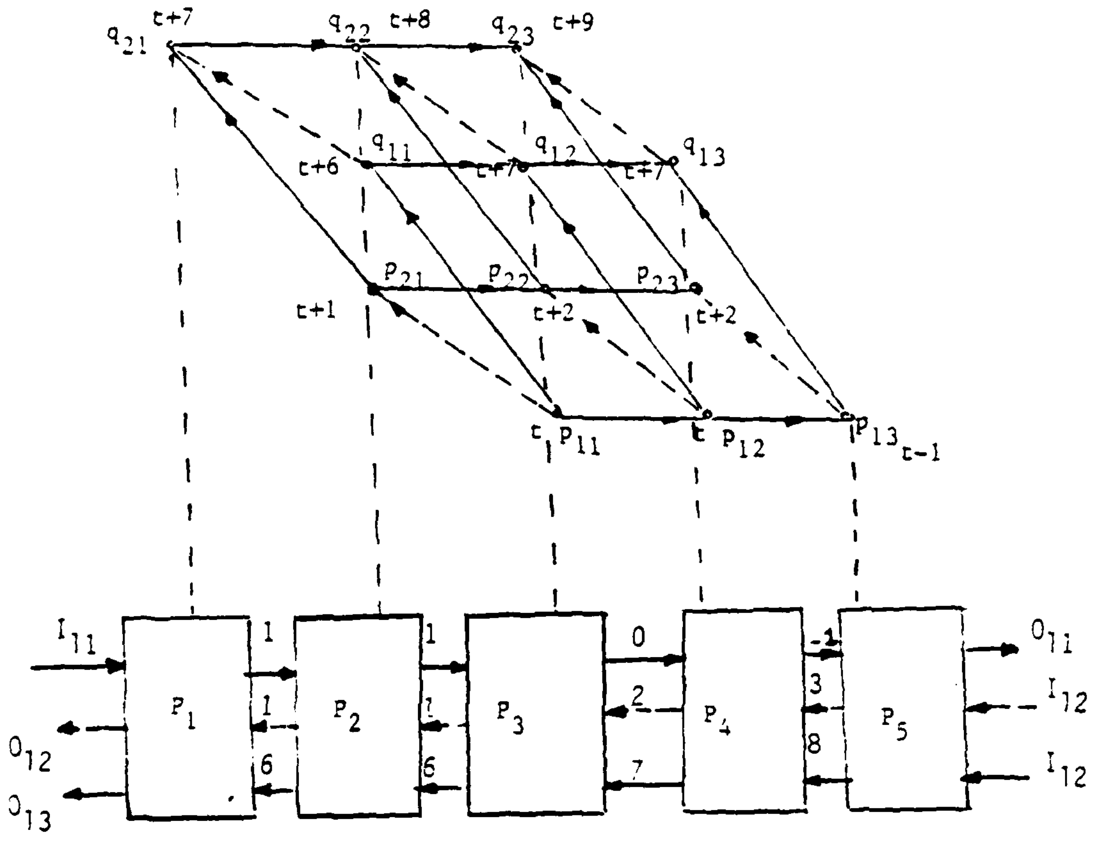


Figure 3.7

Since the choice of n_{l1}, n_{l2}, and n_{l3} satisfies case 2, we choose the delay d along reverse edges to be equal d_{l1}. The perturbations in the delay for $l1$ satisfy $\Delta(l1,1)=0$, $\Delta(l1,2)=0$, $\Delta(l1,3)=-1$ (there is one reverse edge between v_3 and v_4) and $\Delta(l1,4)=-2$. The perturbations for $l2$ and $l3$ satisfy $\Delta(l2,j)=\Delta(l3,j)=-\Delta(l1,j-1)$, j=2,..,5. The effective delay between logically adjacent processors (δ ' s) is shown in Fig. 3.8 for each stream. The resulting mapping of the cube graph is also shown in the Fig. 3.8. The times at which a computation vertex is mapped is calculated from the final step of the mapping algorithm for tree machines and is indicated by the side of the computation vertex. If A and B were $n \times n$ matrices then the tree algorithm will require $O(n)$ processors and interestingly, $O(n^2)$ time steps to compute the result matrix !!



(Numbers on edges indicate the effective delay between logically adjacent processors for the tree of Figure 3.7)

Figure 3.8

Example 3.3 Consider multiplication of two band matrices A and B as shown below wherein a_{ij} and b_{ij} denote the $[ij]^{th}$ entries in A and B respectively.

$$\begin{bmatrix} a_{11} & a_{12} & & & \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & a_{43} & a_{44} & a_{45} \\ & & a_{53} & a_{54} & a_{55} \\ & & & a_{64} & a_{65} \end{bmatrix} \qquad \begin{bmatrix} b_{11} & b_{12} & b_{13} & & & \\ b_{21} & b_{22} & b_{23} & b_{24} & & \\ & b_{32} & b_{33} & b_{34} & b_{35} & \\ & & b_{43} & b_{44} & b_{45} & b_{46} \\ & & & b_{54} & b_{55} & b_{56} \end{bmatrix}$$

Let C=A$\times$B be the result matrix. The data-flow description in Fig. 3.9 represents multiplication of A$\times$B. The horizontal, lateral and vertical edges are labelled $l1$, $l2$ and $l3$ respectively. In Fig. 3.9, v_{ij}^{k+1} is the computation vertex at a vertical distance k from v_{ij}^{1}. Thus, v_{22}^{3} is the computation vertex at a vertical distance 2 from v_{22}^{1}. The program graph in Fig. 3.9 is a cube graph as illustrated in Fig. 3.10. We next map this graph on a hexagonal array using the hexagonal array mapping algorithm.

Let $w_H = <w_1, w_2, w_3> = <1,1,-1>$ and c=1. It can be verified that for this choice of w_H the set of diagonals D is comprised of $\{ D_{ij} \mid 1 \leq i,j \leq 4 \}$.

The hexagonal array is comprised of 4 rows and columns of processors which are identical to the procesors used in example 3.1. $L_H = \{l1\}$, $L_V = \{l2\}$ and $L_T = \{l3\}$. The neighborhood constants for the labels are $n_{l1} = n_{l2} = 1$ and $n_{l3} = -1$. The delays are $d_{l1} = d_{l2} = d_{l3} = 1$. The constant c for the array is 1. Fig. 3.11 iluustrates the mapping.

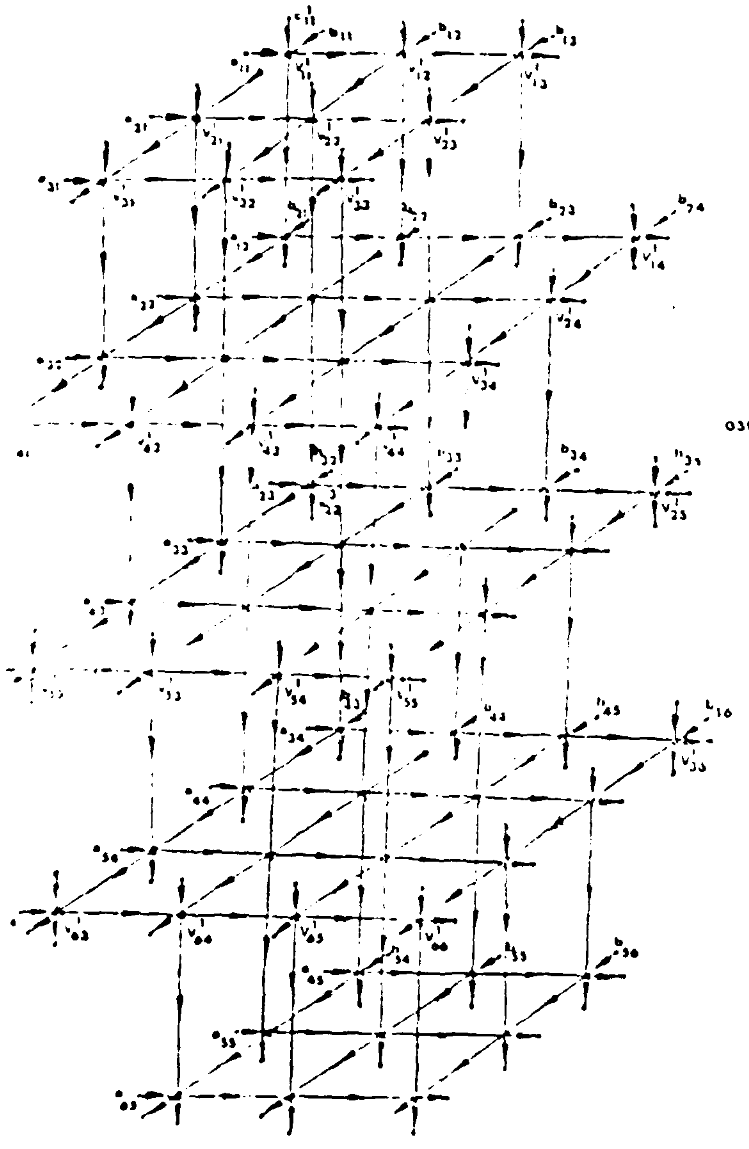


Figure 3.9

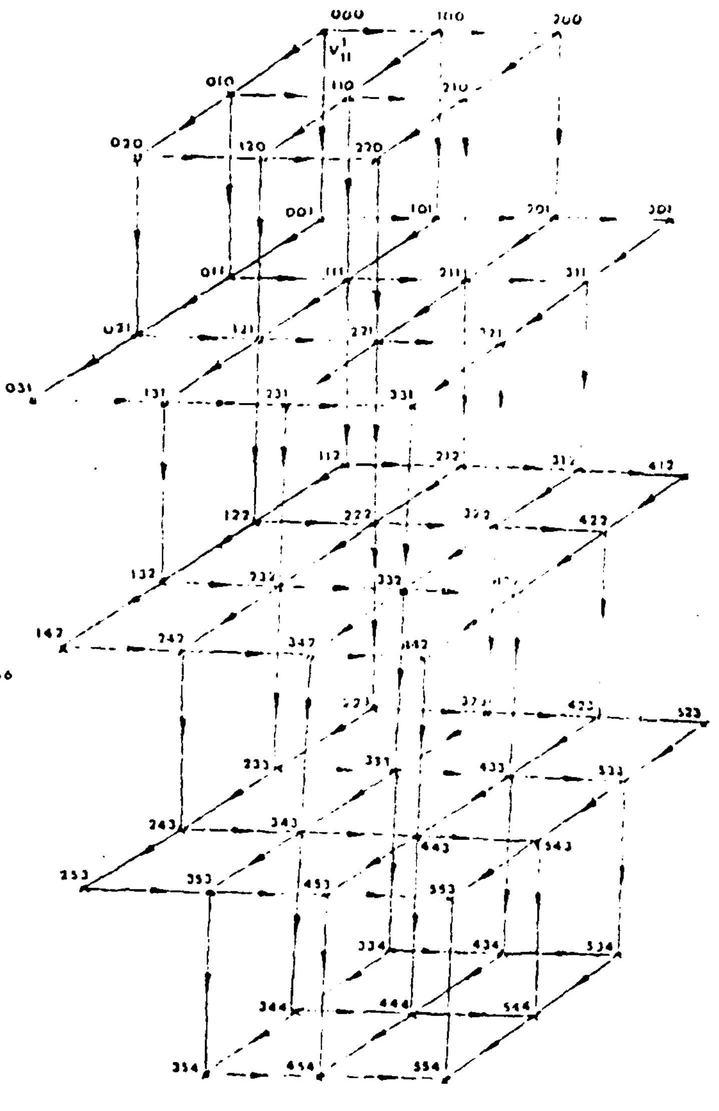


Figure 3.10

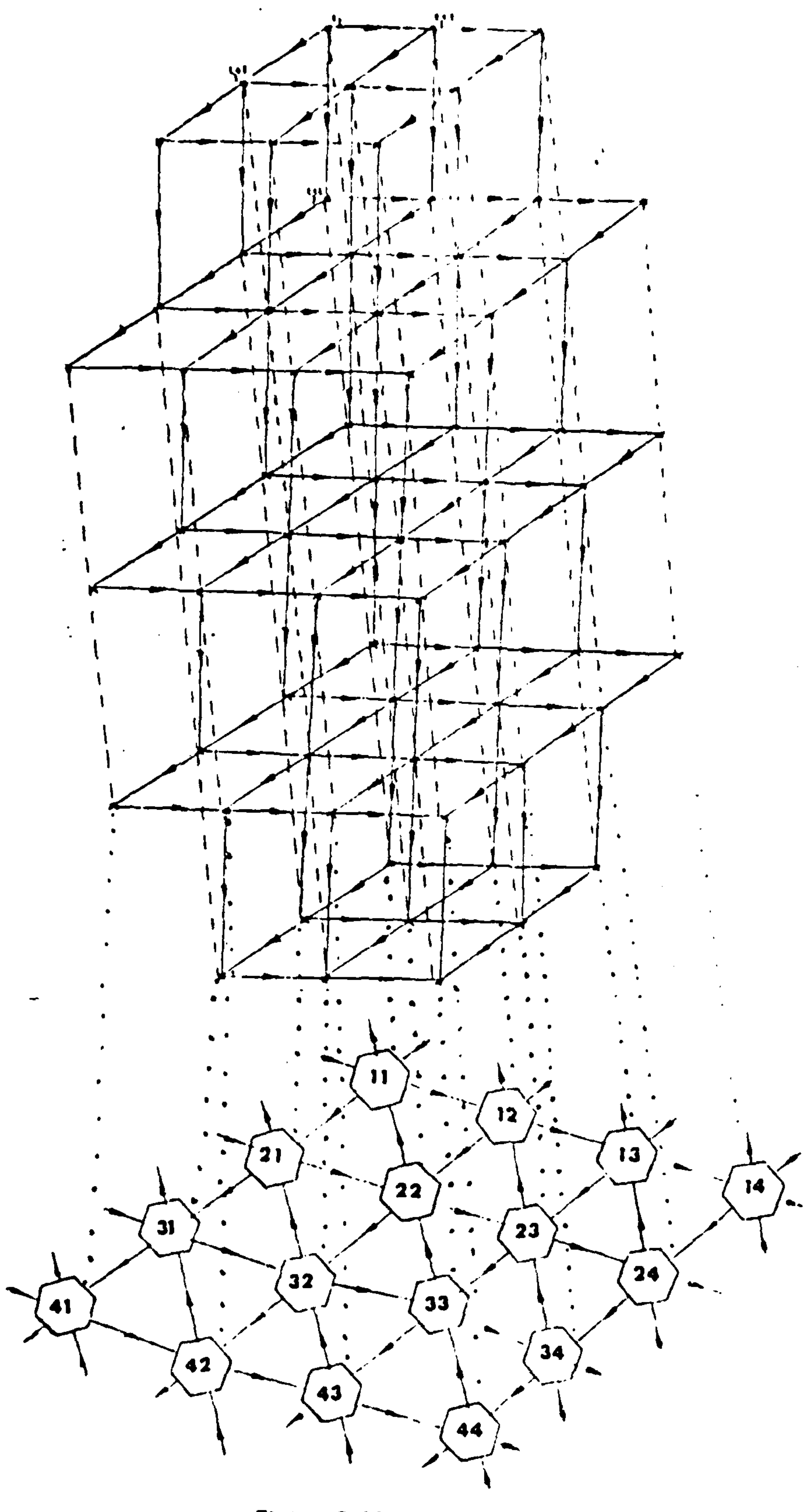


Figure 3.11

Conclusion

In this paper we formalized linear, mesh and hexagonal array processors suitable for VLSI implementation. We also presented a model of a tree machine. We then presented novel algorithms for dense matrix multiplication on a linear array and the tree machine. We also derived a hexagonal array algorithm for multiplying band matrices. Our linear-array algorithm for multiplying dense matrices is particularly useful in situations where the I/O bandwidth is limited as the algorithm requires only a constant (three) number of I/O ports for inserting the elements of A and B matrices and retrieving the result values. The tree algorithm has the same features as the linear-array algorithm. More importantly, the tree algorithm is robust to harware faults in the underlying host.

References

[1] T.C. Chen, V.Y. Lum, and C. Tung, "The Rebound Sorter: An efficient Sort Engine for Large Files," *Proc. of the* 4th Int'l Conf. on Very Large Data Bases , (1978) ,pp. 312-318.

[2] L.J. Guibas, and F.M. Liang, "Systolic Stacks, Queues and Counters," *Proc. MIT Conf. on Advanced Research in VLSI,* (January, 1982), pp. 155-164.

[3] L. Johnsson, and D. Cohen, "A Mathematical Approach to Modelling the Flow of Data and Control in Computational Networks," *VLSI Systems and Computations,* H.T. Kung, R.F. Sproull, and G.L. Steele, Jr., (editors), Computer Science Press, (1981), pp. 213-225.

[4] H.T. Kung, "Let's Design Algorithms for VLSI Systems," *Proc. Caltech Conf. on Very Large Scale Integration: Architecture, Design, Fabrication,* (January, 1979), pp. 65-90.

[5] H.T. Kung, and C.E. Leiserson, "Systolic Arrays (for VLSI)," *Sparse Matrix Proceedings 1978*, I.S. Duff, and G.W. Stewart, (editors), SIAM, (1979), pp. 256-282.

[6] H.T. Kung, "Why Systolic Architectures," *IEEE Computer 15(1)*, (January, 1980), pp. 37-46.

[7] H.T. Kung, and P.L. Lehman, "Systolic (VLSI) Arrays for Relational Database Operations," *Proc. SIGMOD*, (1980), pp. 105-116.

[8] S.Y. Kung, "VLSI Array Processor for Signal Processing," *Proc. MIT Conf. on Advanced Research in Integrated Circuits*, (January, 1980).

[9] C. Mead, and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, (1980).

[10] I.V. Ramakrishnan, D.S. Fussell, and A. Silberschatz, "Systolic Matrix Multiplication on a Linear Array," *Twentieth Annual Allerton Conf. on Computing, Control and Communication*, (October, 1982).

[11] I.V. Ramakrishnan, D.S. Fussell, and A. Silberschatz, "On Mapping Homogeneous Graphs on a Linear-Array Processor Model," *1983 International Conference on Parallel Processing*, (August, 1983).

[12] P.J. Varman, "Wafer-Scale Reconfiguration of Array Processors," Ph.D Dissertation, University of Texas at Austin, (August, 1983).

[13] P.J. Varman, and D.S. Fussell, "Design of Robust Systolic Algorithms," *1983 International Conference on Parallel Processing*, (August, 1983).

[14] U. Weiser, and A. Davis, "A Wavefront Notation Tool for VLSI Array Design," *VLSI Systems and Computations*, H.T. Kung, R.F. Sproull, and G.L. Steele, Jr., (editors), Computer Science Press, (1981), pp. 226-234.

Appendix

We first prove that the mapping algorithm for the tree machine correctly maps the cube graph. We begin by first showing that the mapping preserves the neighborhood constant of the labels.

Theorem A.1: Let $l \in L_G$ and let n_l and d_l be its neighborhood and delay constants respectively. If v_x and v_y are a pair of computation vertices with an edge labelled l directed from v_x to v_y then $PA(v_y)=PA(v_x)+n_l$.

Proof: Let v_x and v_y be the vertices in diagonals D_p and D_q respectively and w_p and w_q be the weights of D_p and D_q respectively. So,

$w_1x_{l1}+w_2x_{l2}+w_3x_{l3}=w_p$, and
$w_1y_{l1}+w_2y_{l2}+w_3y_{l3}=w_q$

We will show that the theorem holds for $l=l1$ as the proofs for $l=l2$ and $l=l3$ are similar.

Let e be the edge labelled l directed from v_x to v_y. From the definition of a cube graph we obtain $y_{l1}=x_{l1}+1$, $y_{l2}=x_{l2}$ and $y_{l3}=x_{l3}$. Consequently, $w_q-w_p=w_1=1$. Since the diagonals are indexed in order of their weights, it follows that index of D_q must be one more than the index of D_p, that is, $q=p+1$.

The mapping algorithm maps vertices in D_p onto processor p and those of D_q onto processor $p+w_1$ and hence $PA(v_y)=PA(v_x)+w_1$. Also from the mapping algorithm $n_{l1}=w_1$. So the theorem holds for $l=l1$. $\square$

We next show that the mapping preserves the delay constant of every label l.

Theorem A.2: Let $l \in L_G$ and let n_l and d_l be its neighborhood and delay constants respectively. Let v_x and v_y be a pair of vertices with an edge labelled l directed from v_x to v_y. If v_x is in diagonal D_p then $TA(v_y)=TA(v_x)+\delta(l,p)$.

Proof: We have to consider the two cases when $n_{l1}=n_{l2}=n_{l3}=1$ and $n_{l1}=1$, $n_{l2}=n_{l3}=-1$.

case 1: $n_{l1} = n_{l2} = n_{l3} = 1$.

Let $v_y \in D_q$ and $l=l1$ with no loss of generality. From the final step in the mapping algorithm for the tree machine we obtain:

$$TA(v_x)=t_1 + \sum_{i=1}^{3} x_{li}d_{li} + \sum_{j=1}^{p-1}\Delta(l1,j)$$
$$TA(v_y)=t_1 + \sum_{i=1}^{3} y_{li}d_{li} + \sum_{j=1}^{q-1}\Delta(l1,j)$$

By definition of a cube graph we have, $x_{l2} = y_{l2}$, $x_{l3} = y_{l3}$ and $y_{l1} = x_{l1} + 1$. From theorem A.1 we obtain $PA(v_y)=PA(v_x)+1$, i.e., $q=p+1$. Therefore,

$$TA(v_y)-TA(v_x)=d_{l1} + \sum_{j=1}^{q-1}\Delta(l1,j) - \sum_{j=1}^{p-1}\Delta(l1,j)$$
$$=d_{l1} + \sum_{j=p}^{q-1}\Delta(l1,j)=d_{l1}+\Delta(l1,p)=\delta(l1,p)$$

case 2: $n_{l1}=1$, $n_{l2}=n_{l3}=-1$.

If $l=l1$ then the proof is the same as that used in case 1. Else let $l=l2$ with no loss of generality. Again by definition of a cube graph we have, $x_{l1} = y_{l1}$, $x_{l3} = y_{l3}$ and $y_{l2} = x_{l2} + 1$. From theorem A.1 we obtain $PA(v_y)=PA(v_x)-1$, i.e., $q=p-1$. So,

$$TA(v_y)-TA(v_x)=d_{l2} + \sum_{j=1}^{q-1}\Delta(l1,j) - \sum_{j=1}^{p-1}\Delta(l1,j)$$
$$=d_{l2} -(\sum_{j=1}^{p-1}\Delta(l1,j)-\sum_{j=1}^{q-1}\Delta(l1,j))$$
$$=d_{l2} - \Delta(l1,q)=d_{l2} + \Delta(l2,q+1)=d_{l2} + \Delta(l2,p)$$

$=\delta(l2,p)$

$\square$

We have to next establish that no two values appear simultaneously at the input port of any processor and the following definition and lemma comes in handy for proving it.

Definition A.1 For any label $l \in L_G$, a *major path* labelled l in G is a directed path from a source vertex to a sink vertex such that all the edges in the path are labelled l.

Lemma A.1: Let $l \in L_G$ and $n_l \in \{1,-1\}$. Let P_1 and P_2 be two distinct major paths labelled l in G and let v_x and v_y be the computation vertices adjacent to the source vertices in P_1 and P_2 respectively. Let $PA(v_x) = s_1$, $PA(v_y) = s_2$ where $s_1 \leq s_2$. Let $TA(v_x) = t_1$ and $TA(v_y) = t_2$. If the input/output values represented by the source and sink vertices of P_1 and P_2 appear simultaneously at the input port of a processor then

$$(t_2 - t_1)n_l = (s_2 - s_1)d_l + n_l\left(\sum_{j=s_1}^{s_2-1} \Delta(l1,j)\right).$$

Proof: Again we need to consider the two cases when $n_{l1}=n_{l2}=n_{l3}$ and $n_{l1}=1, n_{l2}=n_{l3}=-1$.

case 1: $n_{l1} = n_{l2} = n_{l3} = 1$.

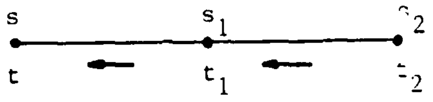
Since $PA(v_x) = s_1$ and $PA(v_y) = s_2$, we have $v_x \in D_{s_1}$ and $v_y \in D_{s_2}$. Assume without loss of generality that the input values represented by the source vertices of P_1 and P_2 appear simultaneously at the input port of processor s. Let $s \leq s_1 \leq s_2$ and the proof will be similar for other values of s. Let t be the time at which both the values appear at the input port labelled l of s. The time taken by the input value represented by the source vertex of P_1 to reach the input port labelled l of s_1 is $t + \sum_{j=s}^{s_1-1} \delta(l,j)$ which is $TA(v_x)$. Similarly, the time taken by the input value represented by the source vertex of P_2 to reach the input port labelled l of s_2 is $t+\sum_{j=s}^{s_2-1}\delta(l,j)$ which is $TA(v_y)$ and hence,

$$t_1 = TA(v_x)=t+(s_1-s)d_l+\sum_{j=s}^{s_1-1}\Delta(l1,j), \text{ and}$$
$$t_2 = TA(v_y)=t+(s_2-s)d_l+\sum_{j=s}^{s_2-1}\Delta(l1,j), \text{ and hence,}$$
$$t_2-t_1=(s_2-s_1)d_l+\sum_{j=s_1}^{s_2-1}\Delta(l1,j)$$

Since $n_l = 1$ by hypothesis, we obtain $(t_2-t_1)n_l=(s_2-s_1)d_l+n_l(\sum_{j=s_1}^{s_2-1}\Delta(l1,j))$.

case 2: $n_{l1} = 1$, $n_{l2} = n_{l3} = -1$.

If $l=l1$, same proof as case 1 holds else assume $l=l2$ with no loss of generality. $n_{l2}=-1$, and $s_2\geq s_1\geq s$. As illustrated in the figure below, if the two values have to meet at s at time t then $t_2\geq t_1\geq t$.



Now $t=t_1+\sum_{j=s+1}^{s_1}\delta(l2,j)=t_1+(s_1-s)d_{l2}+\sum_{j=s+1}^{s_1}\Delta(l2,j)$ is the time taken by the input value represented by the source vertex of P_1 to reach s,

and $t=t_2+\sum_{j=s+1}^{s_2}\delta(l2,j)=t_2+(s_2-s)d_{l2}+\sum_{j=s+1}^{s_2}\Delta(l2,j)$ is the time taken by the input value represented by the source vertex of P_2 to reach s.

Since the values meet at s, the time t is the same in both the equations and hence,

$$(t_2-t_1)=(s_1-s_2)d_{l2}+\sum_{j=s+1}^{s_1}\Delta(l2,j)-\sum_{j=s+1}^{s_2}\Delta(l2,j)$$
$$=(s_1-s_2)d_{l2}-(\sum_{j=s+1}^{s_2}\Delta(l2,j)-\sum_{j=s+1}^{s_1}\Delta(l2,j))$$
$$=(s_1-s_2)d_{l2}-\sum_{j=s_1+1}^{s_2}\Delta(l2,j)$$

Since $\Delta(l2,j)=-\Delta(l1,j-1)$ we have, $(t_2 - t_1)=(s_1 - s_2)d_{l2}+\sum_{k=s_1}^{s_2-1}\Delta(l1,k)$

Also as $n_{l2} = -1$, so $(t_2 - t_1)n_{l2}=(s_2 - s_1)d_{l2}+n_{l2}(\sum_{k=s_1}^{s_2-1}\Delta(l1,k))$. $\square$

We next show that the mapping ensures that no two input/output values appear simultaneously at the input port of any processor.

Theorem A.3 Let $l \in L_G$. Let P_1 and P_2 be two distinct major paths in G labelled l. The mapping ensures that the input/output value represented by the source/sink vertices of P_1 and P_2 never appear simultaneously at the input port labelled l of any processor.

Proof: Let v_x and v_y be the vertices adjacent to the source vertices in P_1 and P_2 respectively. From the mapping algorithm we obtain,

$$PA(v_y)-PA(v_x)=\Delta(P)=\sum_{i=1}^{3}k_i n_{li} \text{ where } k_i=y_{li}-x_{li} \text{ and } -h_i \leq k_i \leq h_i.$$

Let $v_x \in D_p$, $v_y \in D_q$ and $p \leq q$ with no loss of generality. From the mapping algorithm we also obtain,

$$TA(v_y)-TA(v_x)=\Delta T=\sum_{i=1}^{3}(y_{li} - x_{li})d_{li}+\sum_{j=1}^{q-1}\Delta(l1,j)-\sum_{j=1}^{p-1}\Delta(l1,j)$$
$$=\sum_{i=1}^{3}k_i d_{li}+\sum_{j=p}^{q-1}\Delta(l1,j)$$

Now assume that the input/output value represented by the source/sink vertices of P_1 and P_2 appear simultaneously at the input port labelled $l1$ of a processor. By lemma A.1 we have,

$$(\Delta T)n_{l1}=(\Delta P)d_{l1}+n_{l1}(\sum_{j=p}^{q-1}\Delta(li,j)) \text{ which is the same as}$$

$n_{l1}(\sum_{i=1}^{3} k_i d_{li}) + n_{l1}(\sum_{j=p}^{q-1} \Delta(l1,j)) = (\Delta P) d_{l1} + n_{l1}(\sum_{j=p}^{q-1} \Delta(l1,j))$ and hence,

$(\Delta P) d_{l1} = n_{l1}(\sum_{i=1}^{i=3} k_i d_{li})$ .....(*)..

We next show that (*) cannot be satisfied.

1. Let $n_{l2}=1$ and so by the mapping algorithm, $d_{l1}=1$ and $d_{l2}=2$. P_1 and P_2 are distinct major paths labelled $l1$ and so $k_2=k_3\neq 0$.

   a. Let $h_1-h_2+n_{l3}\geq 0$. So $d_{l3}=h_1+1+2n_{l3}$ and (*) reduces to $k_3(h_1+1+n_{l3})+k_2=0$. Now $h_1+1+n_{l3}\geq 1$ and so $k_2\neq 0$ and $k_3\neq 0$. Besides $h_2\leq h_1+n_{l3}$ and $-h_2\leq k_2\leq h_2$ and so (*) cannot be satisfied.

   b. Let $h_1-h_2+n_{l3}<0$ and so $d_{l3}=h_1+n_{l3}$ and (*) reduces to $k_3h_2+k_2=0$. Now $h_2\geq 1$ and so $k_2\neq 0$ and $k_3\neq 0$. Besides $-h_2\leq k_2\leq h_2$ and so (*) cannot be satisfied.

2. Let $n_{l2}=-1$. So $d_{l1}=1$ and $d_{l2}=1$.

   a. Let $h_2-h_1+n_{l3}\geq 0$ and so $d_{l3}=2h_2+1+n_{l3}$. So (*) reduces to $2k_2+k_3(2h_2+1)=0$. As $h_2\geq 1$, so $2h_2+1\geq 3$ and so $k_2\neq 0$ and $k_3\neq 0$. Besides $-h_2\leq k_2\leq h_2$ and so $-(2h_2+1)\leq 2k_2<2h_2+1$ and so (*) cannot be satisfied.

   b. Let $h_2-h_1+n_{l3}<0$ and so $d_{l3}=2h_1+1-n_{l3}$. So (*) reduces to $2k_2+k_3(2h_1+1-2n_{l3})=0$. Now $1\leq h_2<h_1-n_{l3}$. So $2h_1+1-2n_{l3}>1$ and hence $k_2\neq 0$ and $k_3\neq 0$. Besides $-h_2\leq k_2\leq h_2$ and so $-(2h_1+1-2n_{l3})<2k_2<2h_1+1-2n_{l3}$ and hence (*) 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 $\Delta P\, d_{l2}=(\sum_{i=1}^{i=3} k_i d_{li})\, n_{l2}$ and $\Delta P\, d_{l3}=(\sum_{i=1}^{i=3} k_i d_{li})\, n_{l3}$

cannot be satisfied and hence no two input/output values will appear simultaneously at the input port of any processor labelled $l2$ or $l3$. □

Proof that the linear-array mapping algorithm correctly maps a cube graph on a linear array follows immediately from the proof of correctness of mapping cube graphs onto tree machines by letting the perturbation delay δ's be zero in the above proofs.

It can be easily established that if v_x and v_y are two computation vertices connected by an edge labelled l then the mesh-array mapping algorithm maps the vertices on processors which are on the same horizontal row if $l \in L_H$ (like processors 11, 12 and 13 in Fig. 2.2) or on the same vertical column if $l \in L_V$ (like processors 11, 21 and 31 in Fig. 2.2).

It can be similarly established that the hexagonal-array mapping algorithm maps the two vertices on the same row of processors aligned in a north-easterly direction (like processors 11, 12 and 13 in Fig. 2.3) if $l \in L_H$. If $l \in L_V$ they are mapped on a row of processors aligned in a north-westerly direction (like processors 11, 21 and 31 in fig 3.3) and if $l \in L_T$ the vertices are mapped on the same column of processors (like processors 21 and 12 in Fig. 3.3). All these rows and columns constitute a linear array and hence the correctness proof used above can be used to establish that the mesh and hexagonal-array mapping algorithms also map cube graphs correctly.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 84-0173 / 2. GOVT ACCESSION NO. A139306	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ON MAPPING CUBE GRAPHS ON VLSI ARRAY AND TREE ARCHITECTURES	5. TYPE OF REPORT & PERIOD COVERED Technical Report
	6. PERFORMING ORG. REPORT NUMBER CAR-TR-40; CS-TR-1358
7. AUTHOR(s) I.V. Ramakrishnan, P. J. Varman	8. CONTRACT OR GRANT NUMBER(s) F49620-83-C-0082
9. PERFORMING ORGANIZATION NAME AND ADDRESS Dept. of Comp. Sci. University of MD College Park, MD 20742 / Dept. of Elec. Engin. Rice University Houston, TX 77001	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Math. & Info. Sciences, AFOSR/NM Bolling AFB Washington, DC 20332	12. REPORT DATE December 1983
	13. NUMBER OF PAGES 36
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer architectures
VLSI
Arrays
Trees
Cube graphs

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We formalize a model of array architectures suitable for VLSI implementation. A formal model of an arbitrarily structured tree machine is also presented. A mathematical framework is developed to transform cube graphs, which are data-flow descriptions of certain matrix computations, onto the array and tree models. All published algorithms for these computations can be obtained using the mathematical framework. In addition, novel linear-array algorithms for matrix multiplication are obtained. More importantly, the

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

algorithms obtained for the tree model are of special significance. Besides their novelty, the independence of the tree algorithms from a specific inter-processor communication geometry make them robust to hardware faults as opposed to algorithms that are based on specific interconnection requirements.

LMED

-8