M.Tech. (Computer Science) Dissertation Report

SEMI-AUTOMATIC CONVERSION FROM SEQUENTIAL TO PARALLEL ALGORITHMS

By

Amit Kumar Rakshit

Under the Supervision of

Prof. Bhabani Prasad Sinha

Advanced Computing and Microelectronics Unit
Indian Statistical Institute
203, Barrackpore Trunk Road
Calcutta - 700035

JULY, 1999

Certificate of Approval

This is to certify that the thesis entitled *Semi-Automatic Conversion from Sequential to Parallel Algorithms* submitted by *Amit Kumar Rakshit*, in partial fulfillment of the requirements for M.Tech. in Computer Science degree of the Indian Statistical Institute, Calcutta, is an acceptable work for the award of the degree.

[illegible]

Date: 26-7-95

(Supervisor)

Prof. Bhabani Prasad Sinha
Head of the Department
Advanced Computing and Microelectronics Unit
Indian Statistical Institute
203, Barrackpore Trunk Road
Calcutta - 700035

Date:

(External Examiner)

Acknowledgements

The very first person whom I would like to pay my sincerest gratitude and whom I feel myself honored to have worked with is my supervisor *Prof. Bhabani Prasad Sinha* for his guidance, advice, enthusiasm and support throughout the course of this dissertation.

I would like to take this opportunity to thank *Prof. Bhargab B. Bhattacharya, Prof. Aditya Bagchi, Dr. Nabanita Das, Dr. Bhabatosh Chanda* for the excellent and valuable courses they have offered in M.Tech. which help me in this work.

I would also like to thank to all C.S.S.C. and A.C.M.U. staffs for helping me to avail the computing facilities and also giving their valuable advises, whenever I have faced any problem.

Lastly, I should thank my classmates *Mr. Asimava Bera, Mr. P. Swamy Dhoss, Mr. Piyush Srivastava, Mr. Anjan Sarkar, Mr. Susanta Paul, Mr. Kaushik Majumdar, Mr. Arnab Nandi* for their helpful suggestions throughout the project and also my other classmates for sharing those moments of joy and frustration and made my two years stay at I.S.I. enjoyable.

I.S.I., Calcutta,
July 1999

(Amit Kumar Rakshit)

Abstract

Determination of parallel algorithms from computation intensive sequential algorithms, consisting of nested for-loops, and their implementation on special-purpose VLSI arrays is a subject of lot of importance. The mapping procedure is based on the mathematical transformations of index set and data dependence vectors, but it preserves the order imposed on the index set by the data dependencies. Also a proposed methodology is implemented here for parallel program generator.

Contents

Chapter 1

INTRODUCTION

1.1 Trends Towards High Performance Computation

Over the past four decades the computer industry has experienced four generations of development, physically marked by rapid changing of building blocks from relays and vacuum tubes (1940 – 1950s), to discrete diode transistors (1950 – 1960s), to small- and medium-scale integrated (SSI/MSI) circuits (1960 – 1970s), and to large- and very-large-scale integrated (LSI/VLSI) devices (1970s and beyond). Increase in device speed, and reliability and reduction in hardware cost and physical size have greatly enhanced the computer performance. However, better devices are not the sole factor, contributing to high performance. Ever since the stored-program concept of Von Neumann, the computer has been recognized as more than just a hardware organization problem. A modern computer system is really a composite of such items as processors, memories, functional units, interconnection networks, compilers, operating systems, peripheral devices, communication channels, and database banks. The more successful people are in making use of computers, the more computer performance they find they need. What was, in the last decade, largely preserve of academic and industrial research has become a significant segment of the high-performance computing market. It is the fact that the number of operations that a computer can perform per unit time has roughly doubled in every two years for the last four decades. Fast and efficient computers are in high demand in many scientific, engineering, medical, and research areas such as numerical weather prediction, computational aerodynamics, artificial intelligence and automation, remote sensing, nuclear reactor factory, weapon research and defense, computer-assisted tomography, seismic exploration etc. Large-scale computations are often performed in these application areas. For example, cross-sectional computer-assisted tomography images used to take 6 to 10 minutes to generate on a conventional computer. Using dedicated array processor, the processing time can be reduced to 5 to 20 seconds. The 100 megaflops machine, like a CRAY-1, would take 24 hours to complete the 24-hour forecast. But it is not uncommon for us to expect the forecast for the next day.

These kind of applications are possible due to an exponential growth in VLSI technology and due to seamless research and development in designing more and more efficient processor architectures as well as more and more efficient algorithms. Higher performance of processors has been possible to achieve by innovating improved architectural features, e.g., RISC architectures, pipelined and vector processing, systolic architectures, etc., and also by using parallel and distributed computing. For parallel and distributed computing, efficient algorithms are designed which are mapped on suitable architectures. The combined effort of innovating improved processor designs along with improved parallel and distributed algorithms for various real-life problems leads us towards high performance computing.

1.2 The Parallel Computation

A parallel computing system consists of parallel hardware and parallel software to run on it. A parallel processing system is a collection of processing elements that executes programs concurrently, as well as communicate and cooperate with each other to solve large problem fast. One of the simplest ways to use parallelism is the task farm. In this approach, a master process produces a number off independent tasks which are farmed out to slave processes. Once these have been processed, the results are collected by a third process, which write them to disk or displays them on the screen. Typically, the master and each slave reside on different processors. This arrangement provides good load balancing for many types of problem — if one task takes a long time to complete, the other processors in the system can get on with processing other tasks in parallel. A problem is only suitable for task farm parallelism if it can be broken down into a large number of tasks which can be executed independently of each other. In order to maximize the throughput of a task farm the number of tasks should be much greater than the number of processors available. A typical application is ray tracing on a MIMD machine.

Another effective form of parallelism is grid decomposition. In this case, the application must be based on underlying grid, as in cellular automata and image processing applications. This grid is divided into patches and one patch put on each processor. Each processor updates the values in its part of grid, then swaps boundary values with its neighbors. This approach yields good results when a local operator must be applied to each point on the grid. The operations on the grid can be carried out in parallel on each of the sub-grids except in the boundary regions.

A third form of parallelism, algorithmic parallelism, is usually more difficult to implement well. In algorithmic parallelism the different functions within the application are on different processors. For example, the transformation, clipping, shading, and Z-buffering stages of a graphics pipeline can each be put on separate processors. The problem with pipelines of this sort is that they are susceptible to bottlenecks. If one stage of the pipe is more compute-intensive than the others, its performance will limit that of the system as a whole.

Several strategies are being developed to tackle the problem of parallelism. One approach is based on machine dependent parallel programming notions, which take the form of new programming languages as is available with today's parallel computers. But this adds to the problem of non-portability and consumes a lot of man-hours in learning languages which are architecture dependent and as such complicated. Moreover, the rate of growth in VLSI technology indicates that today's software for parallel computer will be redundant in tomorrow.

The second approach is to use problem solving environments that generate efficient parallel programs from high level specification. This approach is much more promising as it is suitable for software engineers as the architecture dependency will be taken care of by the environment. But this necessitates writing parallel algorithms as well as developing powerful translators that take sequential algorithms as input and translate it to its equivalent parallel algorithm which will be the input to the problem-solving environment. There are several reasons why this translator is required. The most frequently mentioned reason is that there are many sequential algorithms, which would be convenient to execute on parallel computers. Also, sequential algorithms are easier to develop and since majority of existing computers are sequential in nature sequential algorithm development will continue till parallel computation becomes cost effective.

Our work ultimately aims for the above mentioned translator development. We have started our work in developing a ***parallel program generator***. It takes a high level of *sequential iterative algorithm* as input and generates an equivalent *pipelined version* of the same which can be implemented on *systolic array* architecture. The communication links to be used as well as the geometry of arrangement of processors are also taken care of. It also gives the option to specify target architecture and to test whether it is possible to execute a parallel version of the given algorithm on it.

Another alternative approach, which is being developed in Artificial Intelligence field, is to use machine learning strategy. In this scheme, the intelligent computer is fed with different sequential algorithms along with their parallel versions which are available for different architectures. The machine thus has a knowledge base, and after studying the pattern of the input sequential algorithm, it will be capable to generate the corresponding parallel version using it's knowledge-base. But this kind of approach is still in its infant stage and in future, it may open a new horizon in the area of parallel computation.

Chapter 2

VLSI COMPUTING STRUCTURE

2.1 Parallel Computation in VLSI Architecture

Highly parallel computing structures promise to be a major application area for the million-transistor chips. Such computing systems have structural properties that are suitable for VLSI implementation. Parallel structures imply a basic computational element repeated perhaps hundred or thousands of times. This architectural style immediately reduces the design problem by similar order of magnitude.

The key attributes offered by VLSI technology are : large amount of hardware available at very low cost, reduced power consumption and physical size, and increased reliability at circuit level. Additionally, the high level of integration can conceivably eliminate the need to physically separate processors from memory, thus eliminating the bottleneck among them. Parallelism and pipelining are two classical concepts without which the efficient utilization of the large hardware resources offered by VLSI is not possible. Parallelism implies the operation of many units at the same time. Pipelining also requires a multitude of resources, but in contrast with parallelism, the resources work in a chain allowing data to flow only from one unit to next one. Both, parallelism and pipelining, can be seen at different logic levels. The *first level* of parallelism is offered by partitioning the computational task into smaller computational modules. The *second level* of parallelism is found within each computational module. The *last level* of parallelism is offered by the simultaneous processing of all the bits in a word; and this level is present in almost all computers. We fix our focus at the second level.

The exploitation of parallelism at the first level is often necessary because computational problems are larger than a single VLSI device can process at a time. If a parallel algorithm is structured as a network of smaller computational modules, then the modules can be assigned to different VLSI devices. The communications among these modules and their operation control dictates the structure of the VLSI system and its performance. In *Fig. 1*, a simplistic organization of a computer system consisting of several VLSI devices, main memory, and an interconnection network is shown. Each VLSI device has a number of processors working in parallel.

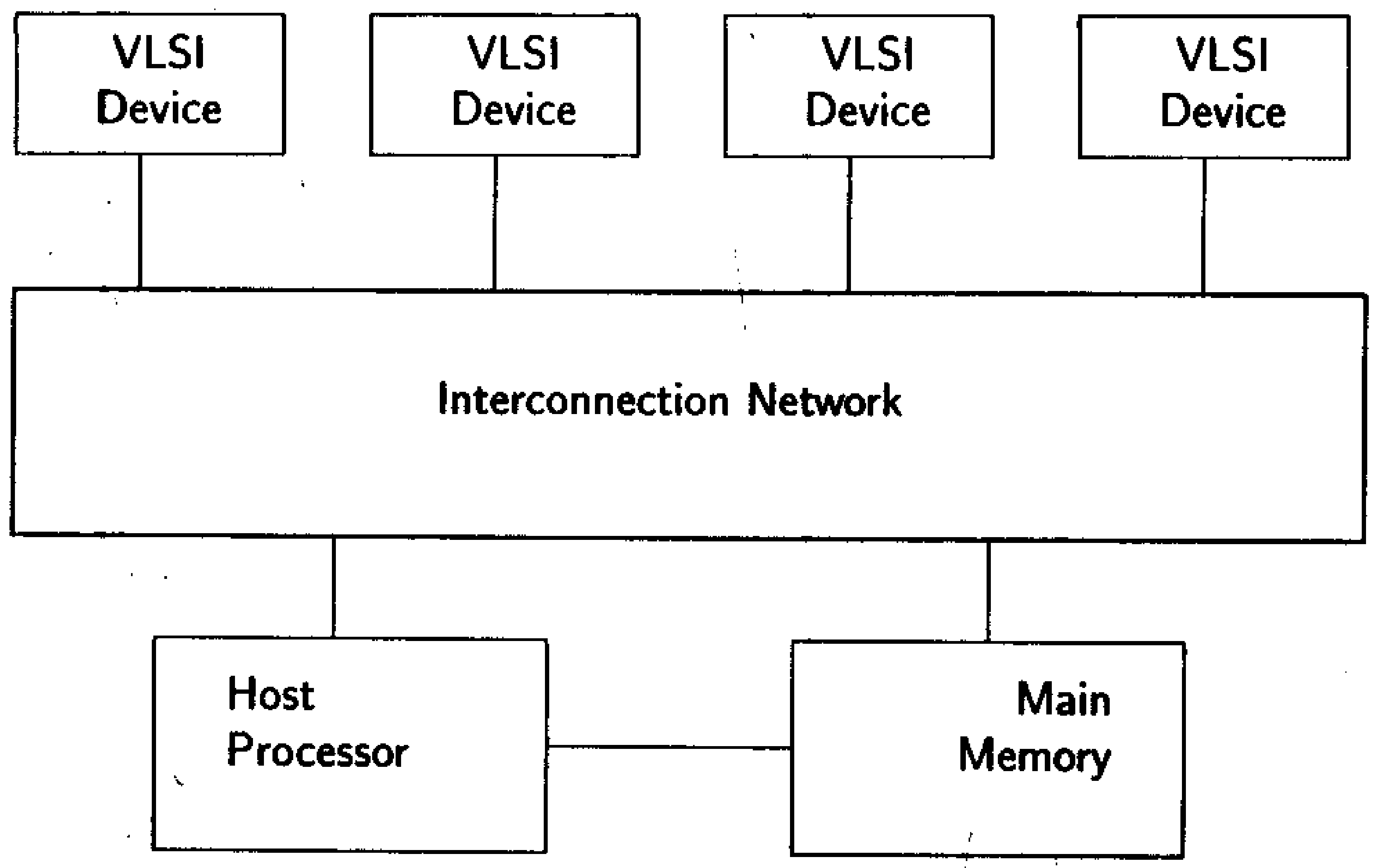

Fig. 1 *Organization of a computer system containing several special purpose VLSI processor arrays, interconnection network,host processor, and main memory.*

The I/O bottleneck problem in VLSI system imposed a serious restriction on the algorithm design. The parallel algorithms should be designed in such a manner that it can be partitioned into modules to reduce the communication among them. Moreover, data entering the VLSI device should be utilized exhaustedly before passing again through the I/O ports. Unorganized data communication within the VLSI devices often detoriates the performance. If the hardware contains local interconnections, the silicon area, time, and energy can be utilized efficiently. The solution to this problem is to design algorithms which, when mapped into VLSI hardware, require only local data transfer.

2.2 The Systolic Array Architecture

The choice of appropriate architecture for any electronic system is very closely related to the implementation of technology. Moreover, to increase the efficiency of computation, it is necessary to match the characteristic of the algorithms with that of the computer architecture. Properly designed parallel structures that need to communicate only with their nearest neighbors will gain most from the system. Precious time is lost when the modules that are far apart want to communicate. For example, the delay in crossing a chip on polysilicon, one of three primary interconnect layers on an NMOS chip, can be 10 to 50 times the delay of an individual gate.

The concept of ***systolic array architecture*** was developed by Kung and associates [5], [6]. A systolic system consists of a set of interconnected cells, each capable of performing some simple operations. Cells in systolic system are typically interconnected to form a systolic array or a systolic tree. Information in a systolic system flows between cells in a pipelined fashion and communication with the outside world occurs only at the "boundary" cells. For example, in a systolic array, only those cells on the array boundaries may be I/O ports for the system. By replacing a single processing element with an array of processing elements, a higher computation throughput can be achieved without increasing memory bandwidth. The crux of this approach is to ensure that once a data item is brought out from the memory, it can be used efficiently at each cell it passes. VLSI systolic arrays can assume many different structures for different compute-bound algorithms. Other advantages include modular expansionability, simple and regular data and control flows, use of simple and uniform cells, elimination of global broadcasting, limited fan-in and fast response time.

The major problem with a systolic array is still in its I/O barrier. The globally structured systolic array can speed-up computations only if the I/O bandwidth is high. With current IC technology, only a small number of I/O pins can be used for a VLSI chip. Of course, I/O port sharing and time-division multiplexing can be used to alleviate this problem.

2.3 A VLSI Model of Computation

A model of VLSI computing structure is needed in order to relate the features of an algorithm to the realities of hardware. Trade-offs are possible between various parameters of the VLSI device in order to improve the performance. The approach taken here is to distinguish between the operation of the systolic system at the array level and the activities taking place inside the processing cells. The array level is called the **global level**, and the processor level is called the **local level**. At both the levels, the operation should be examined in time and space.

Here the focus is set only on the step from the parallel algorithm to the global model. A model of the processing cell and the transition from global model to the local model can be found in [16]. The organization and the operation of the VLSI array can be described by the *network geometry* ***G***, the *functions* ***F*** performed by the processing cells and the *network timing* ***T***.

The assumptions about the VLSI systolic network are as follows:

- The network consists of a planar mesh connected network of processing cells.
- The cells can be of different types and perform different functions.
- The interconnections between cells are buses which transfer parallel words.
- The operation of network is synchronous.

The *network geometry* ***G*** refers to the geometrical layout of the network. The position of each processing cell in the plane is described by its Cartesian coordinates. By choosing the grid arbitrarily small it is possible to represent these coordinates by integers. Then, the interconnection between cells can easily be described by the position of the terminal cells. These interconnections support the flow of data through the network; a link can be dedicated only to one data stream of variables or it can be used for transport of several data streams at different time instances. A simple and regular geometry is desired.

The *function* ***F*** associated to each processing cell represent the totality of arithmetic and logic expression that a cell is capable to perform. We assume that each cell consists of a small number of registers, ALU and the control logic. Several different types of processing cells may coexist in the same network; however, one design goal should be to reduce the number of cell types.

The *network timing* ***T*** specifies for each processing cell the time when the processing of function ***F*** occurs and when the data communications take place. A correct timing assures that the right data reach their destinations at the right time. The speed of data streams through the network is given by the ratio between the distance of communication link over the communication time. Networks with constant data speeds are preferable because they require a simpler control logic.

In brief, the global model of the VLSI array can be formally described by a set of 3-tuples ***(G,F,T)***. The more regular the network is the simpler these functions become. This model is quite general and is sufficient for developing a methodology for designing VLSI algorithms.

Chapter 3

MODELS FOR VLSI ARRAYS AND ALGORITHMS

In this chapter, we have studied the mathematical formulation for VLSI arrays and algorithms. This formulation is required to map the algorithms into architectures.

3.1 VLSI Array Model

The basic assumption regarding the computational resources is that it consists of mesh connected network of processing cell.

A mesh connected array processor is a tuple (J^{n-1}, P), where $J^{n-1} \subset Z^{n-1}$ is the index set of the array and $P \in Z^{(n-1)\times r}$ is a matrix of interconnection primitives. For the sake of generality, the dimension of the array is considered $(n-1)$, but in practical situation the planar layout structure is favored. The position of each processing cells in the array is described by its Cartesian coordinates. The interconnections between cells are described by the difference vectors between the coordinates of the adjacent cells. The matrix of interconnection primitives is

$$P = \begin{bmatrix} \bar{p}_1 & \bar{p}_2 & \cdots & \bar{p}_s \end{bmatrix}$$

where $\bar{p}_j$ is a column vector indicating a unique direction of a communication link.

Consider, for example, the array shown in *Fig. 2.* It is represented by the tuple (J^2, P) where,

$$J^2 = \{(j_1, j_2) : 0 \leq j_1 \leq 2, 0 \leq j_2 \leq 2\}$$

$$P = \left[\begin{array}{ccccccccc} 0 & 1 & -1 & -1 & 1 & 0 & 0 & 1 & -1 \\ 0 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 \end{array}\right] \left| \begin{array}{c} j_1 \\ j_2 \end{array} \right|$$

This array has 8-neighbor bi-directional connections and also a connection within the cell.

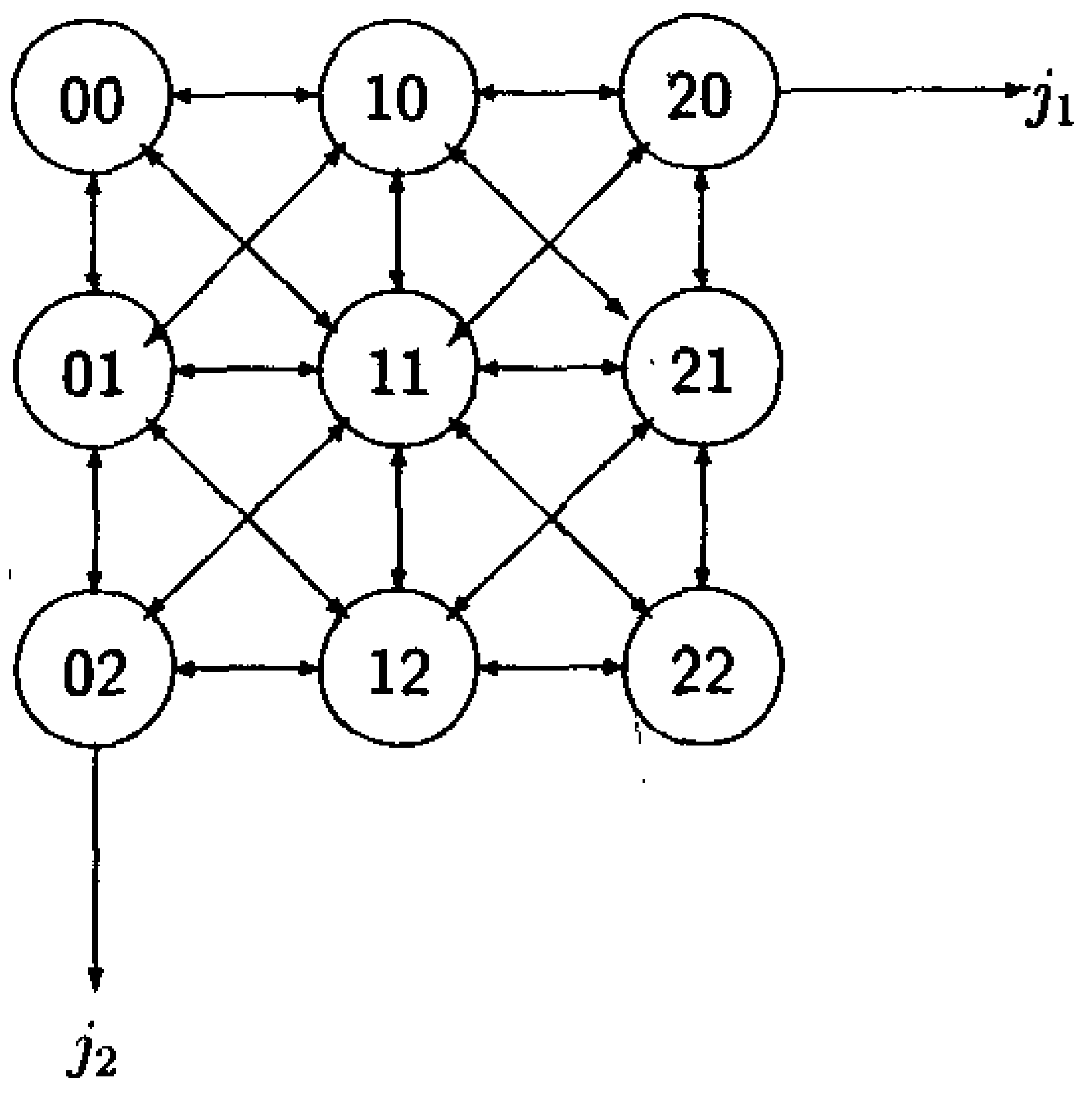

Fig. 2 *A square array with 8-neighbor connections.*

The structural details of the cells and the timings are derived from algorithms which are mapped into such arrays, For simplicity, it is considered that all cells are identical. If an algorithm requires an array with several different types of cells, then the model can be easily modified to describe the function of each cell in index set J^{n-1}.

3.2 Algorithm Model

Here, the study is done primarily on those algorithms which have nested loops. In order to map algorithms into VLSI array processors, it is convenient to define an algorithm model. The prime characteristic of an algorithm model is *data dependency, index set, the computation performed at each index point,* and *the input-output variables.* In the following articles we describe the algebraic structure of the algorithm model.

3.2.1 Data Dependency

The basic structural features of an algorithm are dictated by data and control dependencies. Data dependence represents the precedence relations of memory references whereas control dependence refers to the precedence relations due to control structure of the algorithm. The two dependencies determine the execution ordering of the algorithm in order to compute the problem correctly. The absence of dependencies indicates the possibility of simultaneous computations. These dependencies can be studied in several distinct levels : *blocks of computational level, statement (or expression) level, variable level,* and even *bit level.* Since, the main focus here is on algorithms for VLSI systolic arrays, the data dependencies at variable level, which is the lowest possible level before bit level, is studied.

Here is a simple example to explain the data dependencies.
Example : 1

```
        FOR i := 2 TO 200 STEP 1 DO
        BEGIN
X.         a[i]      := b[2*i] + c[i];
Y.         b[2*i+2] := a[i-1] + c[i-1];
Z.         a[i-2]    := b[i+3] +1;
        END
```

The first four iterations of the loop are shown in the following table :

Statement	Expression
X_2	a[2] := b[4] + c[2];
Y_2	b[6] := a[1] + c[1];
Z_2	a[0] := b[5] + 1;
X_3	a[3] := b[6] + c[3];
Y_3	b[8] := a[2] + c[2];
Z_3	a[1] := b[6] + 1;
X_4	a[4] := b[8] + c[4];
Y_4	b[10] := a[3] + c[3];
Z_4	a[2] := b[7] + 1;
X_5	a[5] := b[10] + c[5];
Y_5	b[12] := a[4] + c[4];
Z_5	a[3] := b[8] + 1;

The following observations are made regarding the data dependencies within the loop.

- In a single instance of the iteration, all the statements X, Y, Z are independent of one another, since no variable appears within the statements more than once.
- The output variable of instance X_2 is the input variable of the instance Y_3 and the computed by X_2 is used by Y_3. This is repeated many times during execution of the loop. In general, the result at Y_i depends on the result at X_{i-1}. This kind of dependence is called **flow dependence** or **true dependence**. Moreover, the nature of dependency is constant throughout the execution of the loop; so it is **static flow dependence**.
- Another flow dependence is observed between the instances Y_3 and Z_5 where the variable b[8], used by Z_5 is actually generated at Y_3. But here the dependency is not uniform; it changes during execution of the loop. So, it is called **dynamic flow dependence**.

- The generated variable a[1] of instance Z_3 is an used variable of instance Y_2. But the instance Y_2 is actually executed before the start of Z_3 and the value of a[1] used by Y_2 is the earlier stored value. So there is an **anti-dependence** of the statement Z on the statement Y.
- When the execution of a statement depends on the input variables, the data dependency then called **input dependence**.
- There is another type of data dependency, called **output dependence**, in which two different statements write on the same memory location at different time instants. For example, if the variable a[i] is considered, it can be noticed that it is modified by Z_{i+2} after X_i has written on it. Here the output dependence is static in nature. In some applications, it may be non-uniform also.

The analysis of data dependencies in high-level language programs must be done for the purpose of detecting concurrency of operations. Here, three types of data dependencies are considered – *flow dependence, anti dependence* and *output dependence.*

3.2.2 Index Set and Data Dependence Vector

To define index set following nested *for* loop is considered.

```
BEGIN
    FOR $I^1$ := $l^1$ TO $u^1$ STEP $p^1$ DO
    BEGIN
        FOR $I^2$ := $l^2$ TO $u^2$ STEP $p^2$ DO
        BEGIN
            ⋱
                FOR $I^n$ := $l^n$ TO $u^n$ STEP $p^n$ DO
                BEGIN
                    $S_1(\bar{I})$ ;
                    $S_2(\bar{I})$ ;
                    ⋮
                    $S_N(\bar{I})$ ;
                END;

        END;
    END;
END;
```

where, l^j, u^j, and p^j are integer-valued linear expressions involving integer-valued index sets I^1, I^2, …, I^{j-1} and $\bar{I} = (I^1, I^2, \ldots, I^n)$. The assignment statements $S_1, S_2, \ldots, S_N$ are of the form $X := E$ where X is a variable and E is an expression of some input variables.

Let $\mathcal{N}$ denotes the set of all integers and $\mathcal{N}^n$ denotes the set of all n-tuples of integers. The index set $\mathcal{L}^n(\bar{I})$ of the above nested loop is defined as,

$$\mathcal{L}^n(\bar{I}) = \{(I^1, I^2, \ldots, I^n) \ : \ l^1 \leq I^1 \leq u^1,\ l^2 \leq I^2 \leq u^2, \ldots,\ l^n \leq I^n \leq u^n\} \ \subset \ \mathcal{N}^n$$

When the above loop is executed, the elements of $\mathcal{L}^n$ are get ordered in lexicographic ordering.

Now, let X and Y are two indexed variables using index sets $f(\bar{I})$ and $g(\bar{I})$, i.e., we write them as $X(f(\bar{I}))$ and $Y(g(\bar{I}))$ where f and g are two integer-valued function defined on the set $\mathcal{L}^n$. Variables X and Y are generated in statements $S_i(\bar{I}_1)$ and $S_j(\bar{I}_2)$ respectively.

Variable $Y(g(\bar{I}))$ is said to be *flow dependent* on the variable $X(f(\bar{I}))$ if,

1. $\bar{I}_1 \ < \ \bar{I}_2$ (here, " $<$ " means "*less than*" in lexicographical sense),
2. $f(\bar{I}_1) \ = \ g(\bar{I}_2)$,
3. $X(f(\bar{I}))$ is an input variable in the statement $S_j(\bar{I}_2)$,
4. Entries in the vector $(f(\bar{I}) - g(\bar{I}))$ are divisible by the steps of the corresponding *for* loop.

The vector $\bar{d} = \bar{I}_2 - \bar{I}_1$ is called the data dependence vector. An algorithm has number of such data dependence vectors. In general, the data dependence vectors are functions of the elements of the set $\mathcal{L}^n$, i.e., $\bar{d} = \bar{d}(\bar{I})$. However, depending on applications, these data dependency vectors may be constant also.

3.2.3 Definition of Algorithm Model

In order to map the algorithms into VLSI array processors, we need suitable transformation in index set keeping the motive of the algorithm intact. For this purpose, we define an algorithm as follows :
An algorithm $\mathcal{A}$ over an algebraic structure $\mathcal{S}$ is is a 5-tuple $\mathcal{A}= (J^n, C, D, X, Y)$ where,

1. J^n is a finite index set of $\mathcal{A}$, $J^n \subset \mathcal{N}^n$;

2. C, set of computations, is a set of triples $(\bar{j}, v, t)$ where $\bar{j} \in J^n$, v is a variable and t is a term built from operations of $\mathcal{S}$ and variables ranging over $\mathcal{S}$. The variable v is called generated at $\bar{j}$, and any variable appearing in t is called used variable;

3. D, set of dependencies, is a set of triples $(\bar{j}, v, \bar{d})$, where $\bar{j} \in J^n$, the instance of index set at which the variable v is used and $\bar{d}$, an element of the set of n-tuple of integers with at least one non-zero entry, is the data dependence vector;

4. X is the set of input variables for $\mathcal{A}$;

5. Y is the set of input variables for $\mathcal{A}$;

3.2.4 Execution Ordering

For completeness of the description of an algorithm through the algorithm model, it is necessary to define the execution ordering of the algorithm. The execution ordering of an algorithm $\mathcal{A}= (J^n, C, D, X, Y)$ is defined as,

1. the specification of a partial lexicographic ordering Θ on J^n (called execution ordering) such that for all $(\bar{d}, v, \bar{j}) \in D$, we have $\bar{d}\ \Theta\ 0$ (i.e., $\bar{d}$ larger than 0 in the sense of Θ);

2. the execution rule : until all computations in C have been performed, execute $(\bar{j}^0, v, t)$ for all $\bar{j}^0\ \Theta\ \bar{j}$ for which $(\bar{j}, v, t)$ have terminated.

Here, the ordering larger than 0 is used in lexicographic sense. Thus, if

$$\bar{d} = \bar{j} - \bar{j}^*\ \Theta\ 0$$

it means that the computations indexed by $\bar{j}^*$ must be performed before those indexed by $\bar{j}$.

3.2.5 Algorithm Equivalence

Two algorithms $\mathcal{A}= (J^n, C, D, X, Y)$ and $\hat{\mathcal{A}}= (\hat{J}^n, C, \hat{D}, X, Y)$ are said to be τ equivalent if and only if,

1. Algorithm $\hat{\mathcal{A}}$ is input-output equivalent to $\mathcal{A}$; i.e., $\mathcal{A} = \hat{\mathcal{A}}$

2. Index set of $\hat{\mathcal{A}}$ is the transformed index set of $\mathcal{A}$; $\hat{J}^n = T(J^n)$ where T is a monotonically increasing and bijective function.

3. To any operation of $\mathcal{A}$ it corresponds an identical operation in $\hat{\mathcal{A}}$ and vice versa.

4. Dependencies of $\hat{\mathcal{A}}$ are the transformed dependencies of $\mathcal{A}$, and written as, $\hat{D} = T(D)$.

Here, we are interested in transformed algorithms for which the ordering imposed by the first coordinate of the index set is an execution ordering. The motivation is that if only one coordinate of the index set preserves the correctness of the computation by maintaining an execution ordering, then the rest of index coordinates can be selected by the algorithm designer to meet some VLSI communication requirements.

Chapter 4

MAPPING OF ALGORITHM

4.1 Transformation of Algorithms into VLSI Arrays

A transformation which transform an algorithm $\mathcal{A}$ into an algorithm $\hat{\mathcal{A}}$ is defined as,

$$T \ : \ <\mathcal{L}^n, D> \ \rightarrow \ <\mathcal{L}^n_T, D_T>$$

The transformation T is partitioned into two functions as follows:

$$T = \begin{bmatrix} \Pi \\ S \end{bmatrix}$$

where, the mapping Π and S are defined as,

$$\Pi \ : \ J^n \ \rightarrow \ \hat{J}^1 \qquad and \qquad S \ : \ J^n \ \rightarrow \ \hat{J}^{n-1}$$

Here, Π is an n-dimensional row matrix of integers, and S is an $(n-1 \times n)$ dimensional matrix of integers, where n is the level of nesting. We consider only linear transformation T, i.e., $T \in Z^{n\times n}$. Thus the set of algorithm dependencies $D = d_1, d_2, \ldots, d_k$ is transformed into $\hat{D} = TD$. The mapping Π is selected such that such that the transformed data dependence matrix $\hat{D}$ has positive entries in the first row. This imposes a valid execution of ordering and can be written as,

$$\Pi \bar{d}_i > 0, \qquad for\ any\ \bar{d}_i \in D,\ 1 \leq i \leq k.$$

In this procedure the advantage gained is that, the first coordinate can be correctly regarded as the time coordinate. Thus a computation indexed by $\bar{j} \in J^n$ in the original algorithm will be processed at time $\hat{j}_0 = \Pi \bar{j}$. Moreover, the total running time of the new algorithm is usually $t = max\ \hat{j}_0 - min\ \hat{j}_0 + 1$. In general, the time increment may not be unitary; but it is given by smallest transformed dependence, i.e., minimum($\Pi \bar{d}_i$). Thus, the execution time of the parallel algorithm is given by,

$$t \ = \ \left\lceil \frac{max\ \Pi(\bar{j}^1 - \bar{j}^2) + 1}{min\ \Pi \bar{d}_i} \right\rceil$$

for any $\bar{j}^1, \bar{j}^2 \in J^n \quad and \quad \bar{d}_i \in D$.

The transformation S can then be selected such that the transformed dependencies are mapped into VLSI array modeled as, $(\hat{J}^{n-1}, P)$ to our suitability. This can be written as,

$$\hat{J}^{n-1} = SD = PK$$

where K indicates the utilization of primitive interconnections in matrix P. The matrix $K = [k_{ji}]$ is such that

$$k_{ji} \geq 0$$

$$\sum_{j} k_{ji} \leq \Pi \bar{d}_i$$

The above constraints indicates that there can not be any negative utilization of connectivities and the use of connectivities should be done within the time taken for the present computation, as there will be need of input-output communications as soon as the next computation begins.

4.1.1 Example of Transformation

Here an example is considered to explain the above transformation scheme.
Example : 2

```
BEGIN
   FOR i := 1 TO 2 DO
   BEGIN
      FOR j := 1 TO 2 DO
      BEGIN
         c[i,j,0] := 0;
         FOR k := 1 TO 2 DO
         BEGIN
            b[k,j,i] := b[k,j,i-1];
            a[i,k,j] := a[i,k,j-1];
            c[i,j,k] := c[i,j,k-1] + a[i,k,j] * b[k,j,i];
         END;
      END;
   END;
END;
```

Here, the level of nesting $n = 3$ and the index space and the data dependency vectors are as follows:

$$J^3 = \{(1,1,1),(1,1,2),(1,2,1),(1,2,2),(2,1,1),(2,1,2),(2,2,1),(2,2,2)\};$$

$$D = \{(1,0,0),(0,1,0),(0,0,1)\};$$

If we choose Π as $(1,1,1)$ then,

$$\Pi d_1 = 1,\ \Pi d_2 = 1,\ \Pi d_3 = 1;$$

$$\hat{J}^1 = \Pi J = \{3,4,4,5,4,5,5,6\};$$

Hence, total execution time required for the transformed algorithm is

$$t = \left\lceil \frac{max\ \Pi(\bar{j}^1 - \bar{j}^2) + 1}{min\ \Pi \bar{d}_i} \right\rceil = \left\lceil \frac{(6-3)+1}{1} \right\rceil = 4\ ;$$

To determine S let us consider the array model shown in *Fig. 2* i.e.,

$$J^2 = \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)\}$$

$$P = \begin{bmatrix} 0 & 1 & -1 & -1 & 1 & 0 & 0 & 1 & -1 \\ 0 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 \end{bmatrix} \begin{vmatrix} j_1 \\ j_2 \end{vmatrix}$$

One possible S may be

$$S = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

For this S we get,

$$J^2 = SD$$
$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

The utilization matrix is

$$K = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

4.1.2 Why Semi-automatic

When we are in the way to determine the transformation $T = \begin{bmatrix} \Pi \\ S \end{bmatrix}$, first the mapping Π is selected so that the parallel execution time is minimized and then the mapping S is determined by solving the diaphantine equation $SD = PK$. Each transformation leads to a different array. This flexibility gives the designer the possibility to choose between a large number of arrays with different characteristics. It also complicates the task of choosing the *best* S matrix. In case of large number of choices, the designer may choose one accodring to his intuition. That's why this method is called **Semi-Automatic**.

There is another point regarding the determination of the transformation. If for a choice of Π, no S could be found to satisfy all the conditions, then are two options:

1. Iteratively find the next best Π matrix for which at least one suitable S can be found out.
2. For the S found out within some limit of $\Pi\bar{d}_i$, obtain a new Π such that the constraint is satisfied.

This problem can be formulated as follows :
Determine the transformation matrix T of order $n \times n$ whose first row is considered as the Π-transformation which is a matrix of order $1 \times n$ the remaining part of T is the S-transformation of order $(n-1) \times n$, subject to the constraint that the time

$$t = \left\lceil \frac{max\ \Pi(\bar{j}^1 - \bar{j}^2) + 1}{min\ \Pi\bar{d}_i} \right\rceil$$

is minimized.
$\bar{j}$ is $n \times 1$ dimensional matrix of positive integers is given and,
$\bar{d}_i \in D, \ \forall i, \ 1 \le i \le N$ are given, each $\bar{d}_i$ being n-tupleof integers, not all zero, and first entry in each such tuple is non-negative.
The constraints are,

$$\Pi\bar{d}_i > 0, \ \forall i, \ 1 \le i \le N$$

Then, the transformation S is obtained by solving the diaphantine equation $SD = PK$ where,
P is a $(n-1) \times (3^{n-1} - 1)$ dimensional matrix whose entries are $1, 0, \ or \ -1$ is given,
$K = [k_{ij}]$ is a $(3^{n-1} - 1) \times n$ dimensional matrix, whose elements are non-negative integers, satisfying the constraints,

$$k_{ji} \ge 0, \ \forall i, j, \ 1 \le i \le (3^{n-1} - 1), \ 1 \le j \le n,$$

and

$$\sum_j k_{ji} \le \Pi\bar{d}_i \ \ \forall i, \ 1 \le i \le n$$

4.2 Mapping Procedure

For the above problem following algorithm was proposed by D. I. Moldovan [1], [3].

STEP-1 Heuristically, find a transformation Π, such that $\Pi\bar{d}_i > 0$ and which minimizes,

$$t = \left\lceil \frac{max\ \Pi(\bar{j}^1 - \bar{j}^2) + 1}{min\ \Pi\bar{d}_i} \right\rceil \quad for\ any\ \bar{j}_1, \bar{j}_2 \in J^n\ and\ \bar{d}_i \in D.$$

STEP-2 Generate all possible K matrices where each element k_{ij} of K are integers and they satisfy the constraints,

1. $k_{ji} \geq 0$ and
2. $\sum_j k_{ji} \leq \Pi \bar{d}_i \;\; \forall i, \; 1 \leq i \leq n$

STEP-3 Find all possible transformations S whose elements are integers and which satisfy the following conditions,

1. Diaphantine equation $SD = PK$ can be solved for S
2. The matrix transformation T is non singular.

STEP-4 From all possible transformations select the one that minimizes the time.

STEP-5 The mapping of indices to processors is as follows :
Each index point $\bar{j} \in J^n$ is processed in a processor whose i-th coordinate is $\hat{j}_i = \Pi_i \bar{j}$.

Chapter 5

IMPLEMENTATION

5.1 Algorithm

5.1.1 Main

```
Input  : The sequential program file — inp.dat
Output : The parallel version of the program and the time and processors'
           interconnections.
Procedure :
BEGIN
   Check syntax of input;
   IF syntax is wrong THEN
      Stop;
   ELSE
      BEGIN
         Pipeline all broadcasted variables;
         Find out data dependencies for all variables embedded in inner
            most for loop;
         Find out $\Pi$ transformation for minimum time;
         Find out $S$ transformation for best geometry and interconnection
            talking suggestions from the user;
      END;
END;
```

5.1.2 Pipeline

```
Input  : The table of information about all variables and the for loop.
Output : The pipelined version of the program in file — inp.out.
Procedure :
BEGIN
   brod_var a variable that is to be broadcasted.
   DO
```

```
      BEGIN
         Find all occurrences of brod_var with same index set.
         FOR all index variables not used by brod_var DO
         BEGIN
            FOR all generated instances of brod_var DO
               include the missing index variable;
            FOR all used instance of brod_var DO
            BEGIN
               FOR each missing index variable ind_var DO
               BEGIN
                  IF step negative THEN
                     include ind_var + step;
                  ELSE
                     include ind_var - step;
               END;
            END;
         END;
         IF brod_var is used at least once THEN
            Do initialization for each missing index;
         Add dependency due to brod_var in dependency vector list;
      END DO UNTIL no more broadcast variables can be found;
   END;
```

5.1.3 Data Dependence

Input : The table of information about all variables generated.
Output : Determine the dependency vector list.
Procedure :

```
BEGIN
data_var a variable that is not pipelined yet.
WHILE data_var is found
   BEGIN
      FOR all used occurrence of data_var, $data\_var_i$ DO
      BEGIN
         FOR all generated occurrences of $data\_var_i$ starting from first occurrence DO
         BEGIN
            substract indices used in generated instance from that of used
            and store the resultant tuple in the result;
            multiply the entries in result by $-1$ if step is negative;
            divide the entries in result by the value of step;
            IF the first non-zero entry is negative THEN
               store the result as anti-dependency;
            ELSE
               IF all entries in the result are zero THEN
```

```
                BEGIN
                  IF the generated variable appears in a line before
                  the used variable THEN
                    store zero for dependency due to $data\_var_i$;
                  ELSE
                    continue;
                END;
                ELSE
                  IF the result is lexicographically less than dependency till yet
                  obtained due to $data\_var_i$ THEN
                    store it for dependency due to $data\_var_i$;
            END;
            IF dependency due to $data\_var_i$ is not zero THEN
              Add dependency due to $data\_var_i$ in dependency list;
          END;
        END;
      END;
```

5.1.4 Π-Transform

Input : The table of information about data dependency.
Output : It finds out Π transformation as best as possible.
Procedure :

```
BEGIN
  FOR cnt:= 1, TO cnt:=maximum_nested_for_loop, DO
  BEGIN
    find all Π whose sum of absolute values is equal to cnt;
    FOR all the Π which satisfy the constraint of +ve time $\Pi_i$, DO
      IF $\Pi_i$ gives less total execution time than the best Π available yet THEN
        strore $\Pi_i$ as the best Π available;
  END;
  IF valid best Π is available THEN
    report it;
  ELSE
    report failure;
END;
```

5.1.5 *S*-Transform

Input : The table of information about data dependency and input processor interconnection.
Output : It finds out Π transformation as best as possible.

```
Procedure :
BEGIN
  WHILE user's are giving choices DO
      BEGIN
        read user's input;
        test validity of user's chioce;
        IF user's $S$ is valid THEN
        BEGIN
          report success;
          report the final architecture of the network;
        END;
      END;
  IF the user wants all valid $S$ transformations THEN
  BEGIN
    ask for the dimension of the systolic array;
    FOR all possible communication patterns DO
    BEGIN
      find the $SD$ satisfying the communication pattern;
      check for validity of the corresponding $S$;
      IF the $S$ is valid THEN
      BEGIN
        add $S$ as well as the resultant architecture the existing
        list of valid $S$ transformations;
      END;
    END;
  END;
END;
```

SAMPLE RESULT

Input File

LU decomposition of a matrix

```
BEGIN
   FOR k := 0 TO n-1 DO
   BEGIN
      u[k,k] := 1 / a[k,k];
      FOR j:= k+1 TO n-1 DO
      BEGIN
         u[k,j] := a[k,j];
      END;
      FOR i:= k+1 TO n-1 DO
      BEGIN
         l[i,k] := a[i,k] * u[k,k];
      END;
      FOR i:= k+1 TO n-1 DO
      BEGIN
         FOR j:= k+1 TO n-1 DO
         BEGIN
            a[i,j] := a[i,j] - l[i,k] * u[k,j];
         END;
      END;
   END;
END;
```

Output File

Pipelined version

```
BEGIN
   FOR k := 0 TO n-1 DO
   BEGIN
      u[k,k] := 1 / a[k,k];
      FOR j:= k+1 TO n-1 DO
      BEGIN
         u[k,j] := a[k,j];
      END;
      FOR i:= k+1 TO n-1 DO
      BEGIN
         l[i,k] := a[i,k] * u[k,k];
      END;
      j := k+1;
      FOR i:= k+1 TO n-1 DO
      BEGIN
         l$3[i,k,j-1] := l[i,k];
      END;
      i := k+1;
      FOR j:= k+1 TO n-1 DO
      BEGIN
         u$3[k,j,i-1] := u[k,j];
      END;
      FOR i:= k+1 TO n-1 DO
      BEGIN
         FOR j:= k+1 TO n-1 DO
         BEGIN
            l$3[i,k,j] := l$3[i,k,j-1];
            u$3[k,j,i] := u$3[k,j,i-1];
            a$3[i,j,k] := a$3[i,j,k-1] - l$3[i,k,j] * u$3[k,j,i];
         END;
      END;
      FOR i:= k+1 TO n-1 DO
      BEGIN
         FOR j:= k+1 TO n-1 DO
         BEGIN
            a[i,j] := a$3[i,j,k];
         END;
      END;
   END;
END;
```

The data dependency matrix $D = \begin{bmatrix} \bar{d}_1 & \bar{d}_2 & \bar{d}_3 \end{bmatrix}$ is

$$D = \begin{array}{c|c|} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} & \begin{matrix} i \\ j \\ k \end{matrix} \\ \left| \begin{matrix} u & l & a \end{matrix} \right| & \end{array}$$

The transformation Π satisfying all the conditions is

$$\Pi = \begin{bmatrix} 1 & 1 & 1 \end{bmatrix}$$

One possible value of transformation S is

$$S = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If we choose $n = 5$, then the array architecture and the corresponding interconnections is shown in the following *Fig. 3.*

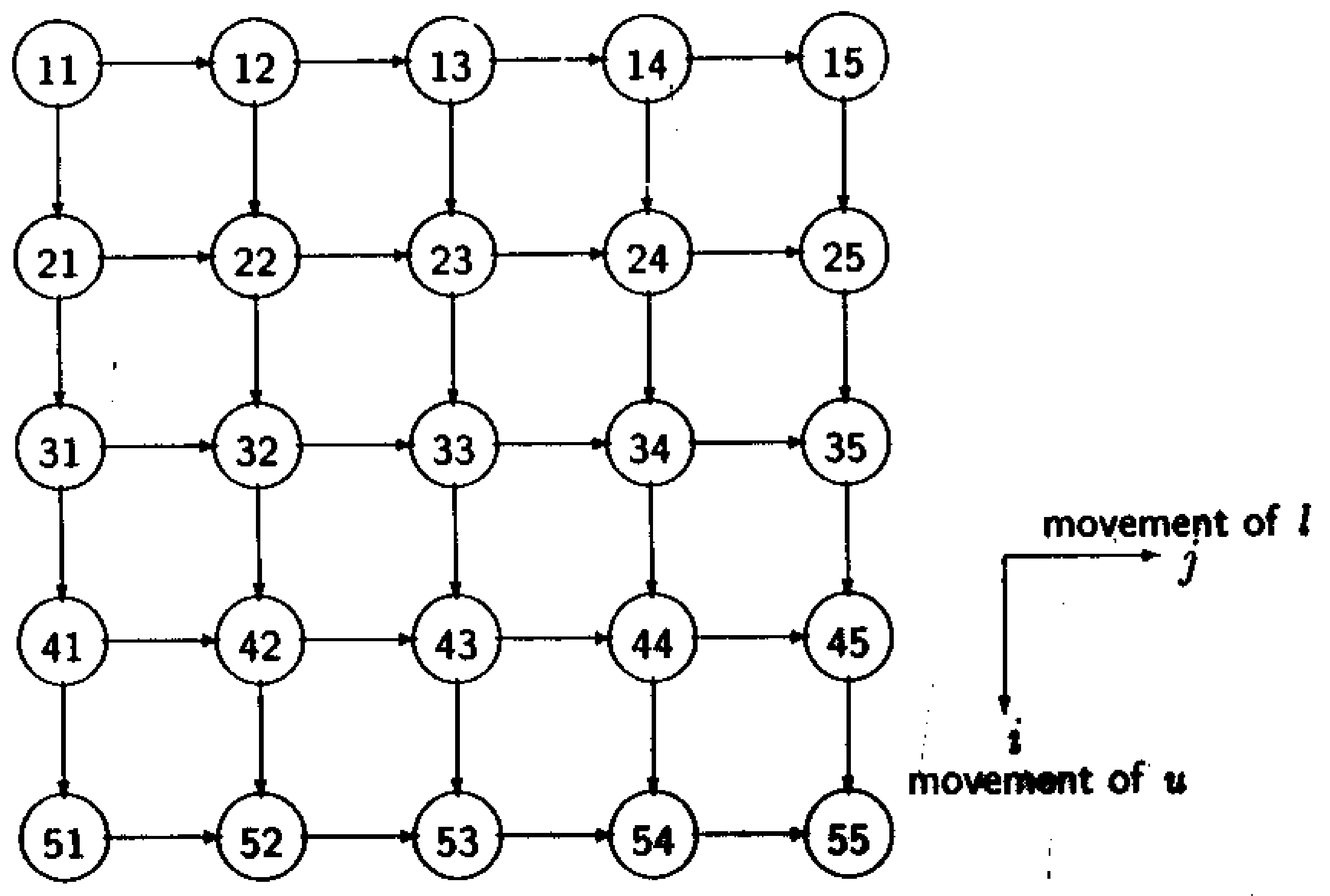

Fig. 3*Array cofiguration of LU decomposition of matrix for n=5 and above S.*

The transformed dependency matrix $\hat{D}$ is :

$$\hat{D} = \begin{bmatrix} \Pi \\ S \end{bmatrix} \times D = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The first row of $\hat{D}$ i.e., $\Pi D = [1\ 1\ 1]$ indicates the amount of time unit allowed for its respective variable to travel from the processor where it is generated to the processor where it is used. Only two inteconnection primitives are used.

Initially, all the cells are loaded with the values of elements of the matrix and after the execution of the algorithm, the lower diagonal matrix L is in the lower diagonal domain and the upper diagonal matrix U is at the upper diagonal domain.

CONCLUSION

Here the algorithm proposed by Moldovan[2], [3] was implemented which convert the sequential algorithm to parallel one and mapping of it into VLSI architecture. The sequential algorithms here considered have nested *for* loops. The algorithm stated in the preceding section parallelizes the nested *for* loop as much as possible. The algorithm for finding optimal transformation are exponential in time, but still applicable in practical cases where number of iterations are high, but level embedding and number of dependencies inside the inner most loop should be limited. In general cases, it will be highly time consuming.

The main problem in fully automated process is that of finding a goodness criteria to compare different systolic array architectures with different communication patterns. In absence of user's choice, large number of valid S transformations can be obtained, each of which represents different architectures and communication patterns. It is very difficult to get the best one from these large number of choices. Here the syntax analyzer is used to check the input program and it generates some prior information about the structure of the input program. This information is used to generate the pipelined version of the program.

There are large scope of future works in this topic. One immediate step is to determine the criteria to compare two systolic architectures. Beside this, the transformation can be found out for other type of architectures also. Here the induced ordering on iterations are tackled by dependency analysis. But the task to find and remove induced dependency by the operators, by operation analysis remains to be done. Then we can get good parallel algorithms for better architectures. Moreover, there are different other types of loops, e.g., *IF-THEN-GOTO*, can be considered to make this parallelisation method robust.

BIBLIOGRAPHY

1. **Dan I. Moldovan**, *On the design of algorithms for VLSI systolic arrays*, Proceedings of the IEEE, Vol.71, No.1, Jan.1983, pp.113-120.
2. **Dan I. Moldovan**, *On the analysis and synthesis of VLSI algorithms*, IEEE Transactions on Computers, Vol.C-31, No.11, Nov.1982, pp.1121-1126.
3. **Dan I. Moldovan** and **Jose A. B. Fortes**, *Partitioning and mapping algorithms into fixed size systolic array*, IEEE Transactions on Computers, Vol.C-35, No.1, Jan.1986, pp.1-12.
4. **Utpal Banerjee et al.**, *Automatic program parallelisation*, Proceedings of IEEE, Vol.81, No.2, Feb.1993, pp.211-243.
5. **H. T. Kung**, *Let's design algorithms for VLSI systems*, Proceedings of Caltech Conf. on VLSI, Jan.1979, pp.65-90.
6. **H. T. Kung**, *The structure of parallel algorithms*, Advanced Computing, Vol.19, 1980, pp.65-111.
7. **Manas Ranjan Jagadev**, *Semi-automatic parallelisation of iterative algorithms*, M.Tech. Dissertation Thesis, I.S.I., Calcutta, Jul. 1997.
8. **B. P. Sinha et al.**, *A parallel algorithm to compute the shortest path and diameter of a graph and its VLSI implementation*, IEEE Transactions on Computers, Vol.C-35, No.11, Nov.1986, pp.1000-1004.
9. **B. P. Sinha et al.**, *Fast parallel algorithm for binary multiplication and their implementation on systolic architectures*, IEEE Transactions on Computers, Vol.38, No.3, Mar.1989, pp.424-431.
10. **L. J. Mordell**, *Diaphantine Equations*, New York,Academic Press,1969, pp.30-33.
11. **Kai Hwang**, *Advanced Computer Architecture*, McGraw-Hill Inc.,1989.
12. **Kai Hwang** and **F. A. Briggs**, *Computer Architecture and Parallel Processing*, McGraw-Hill Inc.,1988.
13. **M. J. Quinn**, *Designing Efficient Algorithms for Parallel Computer*, McGraw-Hill Inc.,1988.

14. **Selim G. Akl**, *The Dcsign and Analysis of Parallel Algorithms*, Prentice Hall.,1989.

15. **Chein-Min Wang et al.**, *Efficient processor assignment algorithms and loop transformations for executing nested parallel loops on multiprocessors*, IEEE Transactions on Parallel and Distributed Systems, Vol.3, No.1, Jan.1992, pp.71-82.

16. **Dan I. Moldovan**, *Computational models for VLSI systems*, Rep. DIM-82-3,1982.