C23781
5.1.97

37

M. Tech. (Computer Science) Dissertation Series

SEMI-AUTOMATIC PARALLELISATION OF ITERATIVE ALGORITHMS

A dissertation submitted in partial fulfilment of the requirements for the **M.Tech. (Computer Science)** degree of Indian Statistical Institute

By
Manas Ranjan Jagadev

Under The Supervision of
Prof. Bhabani Prasad Sinha

Advanced Computing and Microelectronics Unit
INDIAN STATISTICAL INSTITUTE
203, Barrackpore Trunk Road
Calcutta-700035

July 1997

Certificate of Approval

This is to certify that the thesis entitled *Semi-Automatic Parallelization of Iterative Algorithms* submitted by *Manas Ranjan Jagadev*, in partial fulfilment 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.

Date : August 4, 1997.

[illegible]

(Supervisor)

Professor and Head
Electronics Unit
Indian Statistical Institute
203, Barrackpore Trunk Road
Calcutta - 7000 5

Acknowledgement

My sincerest gratitude goes to *Prof. Bhabani Prasad Sinha* for his guidance, advice, enthusiasm and support throughout the course of this dissertation.

I would also like to take this opportunity to thank *Prof. Bhargab B. Bhattacharya, Dr. Subhash Nandy, Prof. Aditya Bagchi* , *Dr. Probal Sengupta, Dr. Ritabari Roychoudhuri, Dr. Nabanita Das, Dr. Sushmita Sur Kolay, Dr. Bhabatosh Chanda* and *Dr. Shraboni Mukhopadhaya* for the excellent courses they have offered in M.Tech. which helped me in this work.

Thanks to *Mr. Sounak Mishra* and *Mr. Subhamoy Maitra* who encouraged the research by giving proper technical advice whenever required.

Mr. Venkatesh Rama Murthy , *Mr. Debashis Sarkar, Mr. Arunabho Ghosh* , *Mr. Manoj Barua, Mr. Nirmalya Barat, Mr. Rajeev Verma, Mr. Dinesh Pandey* and *Mr. Harekrishna Patel* played very crucial role in contributing numerous suggestions throughout the project. I thank them all and also my other classmates who shared those moments of joy and frustration and made my two years stay at ISI enjoyable.

ABSTRACT

Finding equivalent parallel algorithms for computation intensive sequential algorithms has gained a lot of importance with the availability of economic multiprocessors in the last decade. In this work, an effort has been made to get this work done with as much automation as possible. A program has been developed which simplifies the work of parallelization for the parallel algorithm developer.

Contents

Chapter 1

INTRODUCTION

1.1 Need for Higher Computation Speed

It is a real life fact that there is no limit of our expectations and when there is a promise of availability this expectation increases by leaps and bounds. This is the fact in case of computer technology. Thanks to the continuous development of hardware technologies, todays computers are much more faster and smaller in size as well as less in cost compared to those of the last decade. There has been a revolutionary development in VLSI technology which offers much faster CPU's and silicon memories (RAM, ROM and Cache). These two being the backbone of present day electronic computers, today we have much faster computers at an affordable price. It is a fact that the number of operations that a computer can perform per unit time has roughly doubled in every two years for the past 40 years. But the need for the speed of computation has developed at a much faster rate and today there are a lot of genuine applications whose need for computational speed far exceeds the limit that the best technology can provide. A simple example may be the weather forecasting applications, which need, such a vast amount of data analysis for good forecasting before 24 hours itself that a super computer like CRAY XMP-14 is required. But it is not uncommon for us expecting to get a forecast for the next season.

1.2 Need for Parallel Computation

As described above the need for computational speed far exceeds the availability. Also it is forecasted that the trend of improvement in electronic hardware will soon come to an end as the ultimate limiting factor is the speed of electron in any medium. Unless there are significant breakthroughs in present day research on technologies like Josephson junction (based on super conductors), the power dissipation problem, as well as that of stray capacitance and stray inductance put a limit on compaction and hence on the decrement in length of the communicating wires and thus on the speed of devices.

There is an obvious way to get around this problem, that is to use parallelism. The idea is, to use multiple devices for the same task, so that sub tasks can be performed simultaneously, and thus reduce the total time requirement significantly. It is more viable today because of decrement of production costs of mass used devices due to both improvement of VLSI technology and mass scale of production. In fact, now it is possible to assemble a parallel machine with a few tens to several thousands of processors Thus we have a strong need as well as scope for parallel computation.

1.3 Parallel Computation - Past, Present and Future

A parallel computing system consists of parallel hardware and parallel software to run on it. As both are interdependent their development has been interdependent as well. Previously due to technological limitations we had loosely coupled systems in which a number of computers were networked to work together . The memory model in those cases were, "distributed memory model", in which each processor has some memory local to it. In this model when a processor needs to access a location in the memory belonging to a remote processor, it will have to do so through static interconnection networks. Such systems are named as *Multi-Computers.* In that case communication was to be done between one computer to other and as such communication was slow. So the software developed for those systems involved partitioning the task at much higher level so that communication requirements were less. But this was a serious limitation and parallelism in lower level couldn't be exploited.

Due to technological advances (mainly in VLSI field) in the last decade, a number of processors (say 1000 to 10000) could be produced and integrated on a single chip. In this case the communication is between processor to processor using shared memory which is inside the same chip and so it is much faster. These systems are named as *Multi-Processors.* This has opened up the area of low level parallelization. But developing software for a multiprocessor system is closely involved with the architecture of parallel hardware, and so software development is more difficult, as ideally an algorithm should be architecture independent. The wide variety of machine organizations makes it more difficult as we need portability of the software as well.

Several strategies are being developed to improve this situation. One approach is based on machine dependent parallel programming notations, which take the form of new programming languages as is available with todays parallel computers. But this adds to the problem of non-portability and consumes a lot of manhours in learning languages which are complicated architecture dependent and as such complicated. The fear of those parallel computers being replaced by better computers in near future results in software engineers avoiding these languages which, they apprehend, may soon become redundant . As a result many of todays parallel computers are idle due to lack of software.

The second approach is to use problem solving environments that generate efficient parallel programs from high level specifications. 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. To our surprise work done in this area is either in its preliminary stage or is hidden in patent regime so that the work done in this case may be taken as very basic in nature.

We have started our work in this area by developing a *parallel algorithm generator* which takes a high level sequential iterative algorithm as input and generates an equivalent pipelined version of the same which can be implemented in 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 will be possible to execute a parallel version of the given algorithm on it. In this development we have used some of the algorithms used in ADVIS software developed at University of South California as mentioned in Moldovan[4]. Our software is better than ADVIS in terms of input requirements as well as that of degree of automation.

A third approach is to use machine learning, which is being developed in AI field. If this approach becomes successful then it will be possible to feed available sequential algorithms and the corresponding parallel algorithms for different architectures to the intelligent computers. The machine will then study the exact pattern of development and will try to find parallel algorithms for new sequential algorithms using the knowledge already acquired by it. But given the fact that "machine learning" is still in its infant stage this approach can be considered as the approach of the future.

In the field of hardware also it is expected that in future it will be possible to develop exotic architectures economically. But today it is possible to manufacture multiprocessor chips with simple systolic array architectures only using the best available technology, keeping chips cost effective. Hence we will concentrate on systolic array architecture for our work and will expect that it may be extended to other architectures in future.

1.4 Systolic Arrays

The two major problems in VLSI technology are that of limitation of number of I/O ports and communication wire length. The first problem is due to fact that the chips have limited perimeter, and the second one is due to increase in delay with long communicating wires. Systolic array architectures have been proposed by Kung [6],[7] and others as a possible solution to these VLSI problems. In the systolic concept, VLSI devices consist of arrays of interconnecting processing cells with a high degree of modularity. Each processor operates on a string of data that flow regularly through the network. Neglecting the I/O problem the throughput of such computational structures can be considered to be proportional to the number of cells.

Chapter 2

MAPPING OF ALGORITHMS

The basic structural features of an algorithm are dictated by the data and control dependencies. Data dependence represents the precedence relations of memory references whereas control dependence represents the precedence relation due to control structure of the algorithm. These two dependencies determine the execution ordering of the program in order to compute the problem correctly. The absence of dependencies indicates the possibility of simultaneous computations. In this work, we concentrate on algorithms for VLSI systolic arrays, and hence we focus on data dependencies at the variable level. The other levels in which data dependencies can be studied are:

1. blocks of computations level
2. statement (or expression) level
3. bit level

To explain data dependence , let us take a simple example
example 1: Consider the single loop

```
          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 as shown below

```
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 ;
```

We can make the following observations :

1. The output variable of the instance x(2) of statement x, is an input variable of the instance y(3) of statement y, and the value computed by x(2) is actually used by y(3). This pattern is repeated many times. In general, the value computed by the instance x(i) of a is used by the instance y(j) of y, whenever i and j are two values of index variable I such that j - i = 1. This is known as **flow dependence**. In this case it is uniform as there is a constant (dependence) distance, namely 1, such that the instance y(i+1) is always dependent on the instance x(i) whenever i and i+1 are values of I. So it is **static flow dependence**.

2. Similarly it can be found out that z(5) is dependent on y(3) due to generated used pair of B[8]. But in this case dependence value is I-1. As it is dependent on I it is not uniform. This type of dependencies are also known as **dynamic flow dependence**. As data dependence analysis in this case is run time dependent on the value of I, this problem is difficult to tackle during compilation. Hence this type of dependence analysis is not done in present work.

3. The output variable of z(3) is also an input variable of y(2), but the value of A[1] used by y(2) is the one that existed before the program segment started, and not the value computed by the z(3). This makes the instance z(3) anti dependent on the instance y(2), and the statement z **anti dependent** on the statement y. So in general whenever we have a -ve distance for a used generated pair of some indexed variable we detect anti dependence .

4. For value of i between 2 and 200, the instance x(i) of a and the instance z(i+2) of z both compute a value of A[i], such that the value computed by z(i+2) is stored after the value computed by x(i). This type of dependence is known as **output dependence**. We say that instance z(i+2) of z is output dependent on instance x(i) of x. In this case this output dependence is uniform.

5. The fourth kind of data dependence is caused by a pair of input variables; it is called as **input dependence.** As for example x(4) as well as z(5) use the same input variable B[8]. Input dependence is a useful concept in some contexts (e.g., memory management). But it is not much of importance in our case.

By data dependence we will mean any one of the three particular types of dependencies: **flow dependence**, **anti dependence**, and **output dependence.**But in our case we consider only the first two types of dependencies.

2.1 Index Sets and Data Dependencies

Let us define flow dependence for a nested for loop.

```
FOR $I^1 := l^1$ TO $u^1$ STEP $d^1$ DO
    FOR $I^2 := u^2$ DOWNTO $l^2$ STEP $d^2$ DO
        .
          .
            .
              FOR $I^m := l^m$ TO $u^m$ STEP $d^m$ DO
              BEGIN
                  $S_1(\bar{I})$ ;
                  .
                  .
                  .
                  $S_n(\bar{I})$ ;
              END
```

where, l^j, u^j, are integer valued expressions involving the integer valued index set I^1, ... , I^{m-1} and,$\bar{I} = (I^1, I^2, \ldots I^{m-1})$, and S_1, $S_2, \ldots S_n$ are assignment statements of the form "$X := E$" where X is a variable and E is an expression of some input variables. Let $\bar{I}_x$ represent an instance of $\bar{I}$

Let X and Y be two variables using the index sets $f(\bar{I})$ and $g(\bar{I})$. 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 variable $X(f(\bar{I}))$ if,

a. Entries in vector $(f(\bar{I}) - g(\bar{I}))$ are divisible by steps of corresponding for loops and let after such division and multiplication by -1 in case of DOWN TO type for loop the index set $\bar{I}$ be transformed to $\bar{i}$.

b. $\bar{i}_1 < \bar{i}_2$ (Here "<" means less than in lexicographic sense)

c. $f(\bar{i}_1) = g(\bar{i}_2)$

d. $X(f(\bar{I}_1))$ is an input variable in statement $S(\bar{I}_2)$

   The vector $\bar{d} = \bar{i}_2 - \bar{i}_1$ is called the flow dependence vector. It is convenient to represent all dependencies D as a matrix $D = [d_1\, d_2 \ldots d_n]$

For our purpose it is also required to store all anti dependencies for which the above conditions remain same except the second condition which changes to $\bar{i}_1 > \bar{i}_2$. We multiply the whole vector by -1 such that it becomes positive in lexicographic sense.

2.2 Algorithm Model

In order to map algorithms into VLSI array processors, we need suitable transformation in index set keeping the equivalence of algorithms intact. For this purpose let us define an algorithm model.

An algorithm A over an algebraic structure S is a 5 tuple $A = (J^n, C, D, X, Y)$ where:

J^n, index set, is a finite index set of A , J^n is a subset of set of n-tuple of +ve integers.

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

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

X, is the set of input variables for A.

Y, is the set of output variables for A.

2.3 Execution Ordering

For completeness of description of an algorithm, along with algorithm model we need to define execution ordering defined by,

1. the specification of a partial lexicographic ordering O on J^n (called execution ordering) such that for all $(\bar{d}, v, \bar{j}) \in D$ we have $\bar{d}$ greater than zero in lexicographic sense.

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

2.4 Algorithm Equivalence

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

1. Algorithm $\hat{A}$ is input-output equivalent to A; $A \equiv \hat{A}$.

2. Index set of $\hat{A}$ is the transformed index set of A; $\hat{J}_n = T(J_n)$ where T is a bijection and a monotonically increasing function.

3. Any operation of A corresponds to an identical operation in $\hat{A}$ and vice **versa**.
4. Dependencies of $\hat{A}$ are the transformed dependencies of A, written $\hat{D} = T(D)$.

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 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. Let us find out how a transformation T can be selected such that the transformed algorithm can be mapped into a VLSI array. To understand VLSI communication requirement let us define VLSI array model.

2.5 VLSI array model

It is assumed that the computational resource consists of a mesh connected network of processing cells.

A mesh connected array processor is a tuple (j^{n-1}, P) where j^{n-1} is a subset of $n-1$ tuple of non -ve integers, is the index set of the array and P is an element of set of (n-1 $*$ r) dimensional matrices of integers , is a matrix of interconnection primitives.

For sake of generality, we consider that VLSI arrays are (n-1) dimensional. The position of each processing cell in the array is described by its Cartesian coordinates. The interconnections between cells are described by the difference vectors between the coordinates of adjacent cells. The matrix of interconnection primitives is :

$$P = [\bar{p}_1, \bar{p}_2 \ldots, \bar{p}_s]$$

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

Consider, for example, the array shown in FIG. 1. ; Its model is described as (J_2, P) where,

$$J_2 = \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)\}$$

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

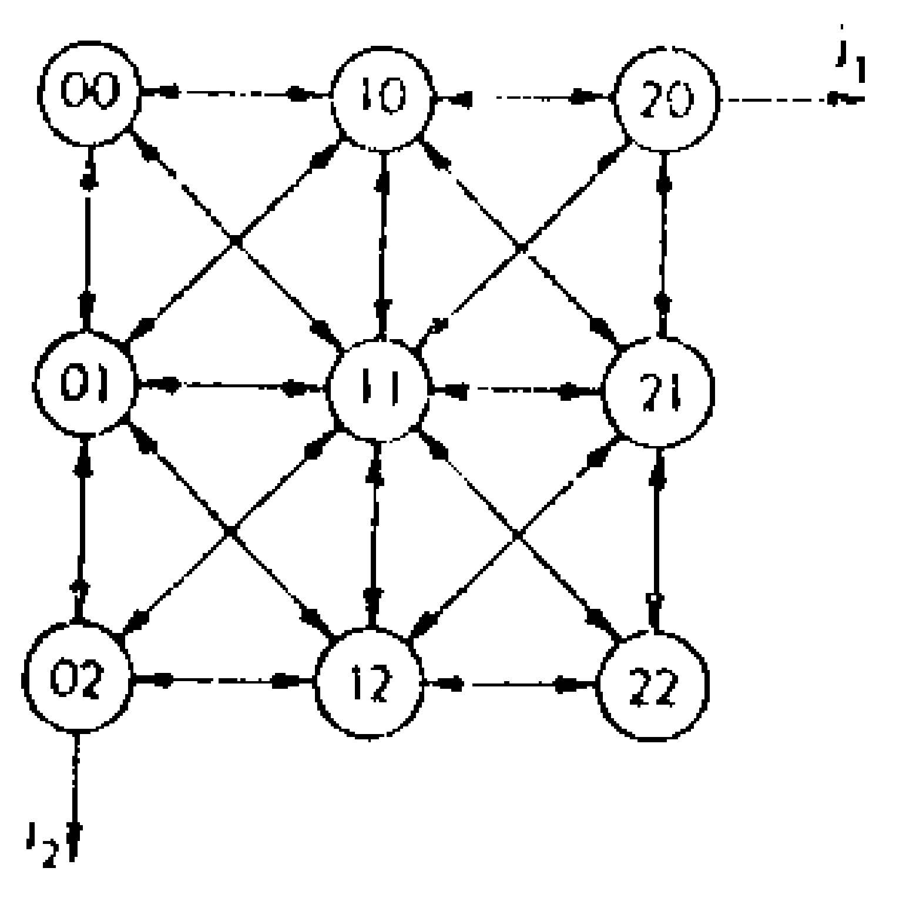

FIG.1. A square array with 8 neighbour connections.

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}.

2.6 Mapping Algorithms into VLSI Arrays

A transformation which transforms an algorithm A into an algorithm $\hat{A}$ is defined as

$$T = \begin{bmatrix} \Pi \\ S \end{bmatrix} \text{ where,}$$

Π is a m dimensional row matrix of integers, (m being the level of nesting of FOR loops in the Algorithm) which maps an index tuple J^m to $\hat{J}^1$, an integer (considered as time) ,

S is a $(n-1 \times m)$ dimensional matrix of integers, which maps an index tuple J^m to $\hat{J}^{n-1}$ a $n-1$ tuple of integers ($\hat{J}^{n-1}$ is considered as the Cartesian coordinate representing position of the processor).

Thus the set of algorithm dependencies D (i.e $D = \{d_1, d_2, \ldots, d_k\}$, each d_i being a $(m \times 1)$ column vector) is transformed into $\hat{D} = T * D$. The mapping Π is selected such that the transformed data dependence matrix $\hat{D}$ has positive entries in the first row. This ensures a valid execution ordering, and can be written as follows:

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

By doing the above we get the advantage that we can regard correctly the first coordinate of the transformed algorithm $\hat{j}_0$ as the time coordinate. Thus, a computation indexed by $\bar{j}_i$ (a m tuple of integers) in the original algorithm will be processed at the time :

$$\hat{\bar{j}}_i = \Pi \bar{j}_i,$$

in the transformed algorithm.

So, the total running time of the new algorithm is usually $t = \max_i \hat{\bar{j}}_i - \min_i \hat{\bar{j}}_i + 1$. This assumes a unitary time increment. In general, the time increment is given by the smallest transformed dependence, i.e. minimum $\Pi * \bar{d}_i$. Thus the execution time of the parallel algorithm is given by the ratio

$$t = \left\lceil \frac{\max_l \Pi(\bar{j}_l - \bar{j}_{l'}) + 1}{\min_i \Pi \bar{d}_i} \right\rceil$$

for $\bar{j}_l, \bar{j}_{l'}$, any two valid index sets and $\bar{d}_i$ any dependency vector .

Here it may be noted that $max_l \Pi(\bar{j}_l - \bar{j}_{l'})$, is same as $\|\Pi\| \|\bar{j}_{max} - \bar{j}_{min}\|$, where $\bar{j}_{max}$ and $\bar{j}_{min}$ may be considered as maximum and minimum index tuples respectively, being divided by steps of respective for loops.

Let's take an example to explain $\prod$ transformation and running time of transformed algorithm.

Example : (matrix multiplication *(pipelined version)*)

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

Here *nesting level* $m = 3$;

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

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

If we choose $\prod$ as (1,1,1) then,

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

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

Hence total execution time required for the transformed algorithm is,

$$t = \lceil \frac{\max_{l} \Pi(\bar{j}_l - \bar{j}_{l'}) + 1}{\min_i \Pi d_i} \rceil$$

$$= \lceil \frac{(6-3)+1}{1} \rceil = 4 \; ;$$

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

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

where the matrix P is $(m-1 \times (3^{m-1}-1))$ dimensional, m being the maximum level of nesting in the algorithm.

Each column in P contains a $m-1$ tuple of 1, -1 or 0 but at least one entry non zero, which represents a direction of communication possible in systolic array architecture. Thus P represents all direction of communication possible.

K is a $((3^{m-1}-1)*n)$ dimensional matrix of non -ve integers, n being the number of dependency vectors listed in D. Each column represents the utilization of connectivities (listed in P) being utilized for a particular dependency.

The constraints for K matrix $= [k_{ji}]$ is such that

$$k_{ji} \geq 0 \qquad \ldots (1)$$
$$\textstyle\sum_j k_{ji} \leq \Pi \bar{d}_i. \qquad \ldots (2)$$

These constraints mean that, there can't be -ve utilization of connectivities (as it is meaningless) and the use of connectivities should be done within the time taken for present computation, as there will be need of input output communications as soon as the next computation begins.

Let us take the above example to explain S vector and resulting communication pattern obtained.

In this case,

$$J_2 = \{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)\}$$

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

One possible S may be,

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

For this S we get,

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

$$\text{i.e. } K = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

We can check that this satisfies the constraints for K and so it is a **valid transformation**. The architecture with this P and K is as shown in fig. 2.

Here it may be noted that most often many S transformations can be found out, for the best possible Π, and 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 best S matrix. In case of big set of choices designers intuition may be used, and this is the point why we call this transformation **semi automatic** . Further, the possibility of finding not a single S matrix for a given Π matrix may not be ruled out. In that case we have 2 choices

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

The above is an optimization problem in integer domain and requires further discussion.

This problem may be formulated as below :

Find a $T(m * m)$ matrix whose first row is considered as $\Pi(1 * m)$ matrix and rest as $S(m - 1 * m)$ matrix such that,

$$t = \lceil \frac{\|\Pi\| \bar{j}}{\min_i \Pi \bar{d}_i} \rceil$$

is minimized.

$\bar{j}$ is a $(m * 1)$ dimensional matrix of +ve integers is given and,

$\bar{d}_i(m * 1) \in D, \qquad \forall i, 1 \leq i \leq n$, are given, each $\bar{d}_i$ being a m tuple of integers, not all zero, and first entry in each such tuple is non -ve.

The constraints are,

$$\Pi \bar{d}_i > 0 \qquad \forall i, 1 \leq i \leq n. \qquad \ldots (1)$$

The solution to the diophantine equation $S * D = P * K$ exists, $\ldots (2)$

where:

P, a $(m - 1 * (3^{m-1} - 1))$ dimensional matrix, whose entries are 1, 0 or -1, is given,

$K = [k_{ji}]$, a $((3^{m-1} - 1) * n)$ dimensional matrix, whose elements are non -ve integers, satisfying the constraints,

$$k_{ji} \geq 0, \qquad \forall i, j, 1 \leq i \leq (3^{m-1} - 1), 1 \leq j \leq n, \qquad \ldots (3)$$

and

$$\sum_j k_{ji} \leq \Pi \bar{d}_i, \qquad \forall \mathrm{i}, 1 \leq \mathrm{i} \leq \mathrm{n}. \qquad \ldots (4)$$

For the above problem following algorithm was proposed by D. I. Moldovan[4].

Step 1 : Heuristically, find a transformation Π, such that $\Pi\bar{d_i}$ is greater than zero for all d_i's and,

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

is minimized.

Step 2: Generate all possible k matrices $K = [k_{ji}]$ where each k_{ji} is an integer, and k satisfies the following conditions :

1) $k_{ji} >= 0$ and
2) $\sum_j k_{ji} \leq \Pi\bar{d}_i$

Step 3: Find all possible transformations S (n-1 * n) whose elements are integers and which satisfy the following conditions:

1) diophantine equation SD = PK can be solved for S and
2) matrix transformation

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

is non singular.

Following the above algorithm, we may obtain some valid transformations T. If no S can be found to satisfy all the above conditions, then either we compromise the fast execution time by selecting another Π in step 1, or we compromise the locality of data communication by selecting another set of primitives P.

Step 4: From all the possible transformations select the one that minimizes time.

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

Chapter 3

IMPLEMENTATION

3.1 Algorithms

3.1.1 main

```
PROCEDURE main ( fname.dat )
(* fname.dat is the input filename *)

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 innermost
                DO LOOP ;

            find out $\prod$ transformation for minimum time ;

            find out $S$ transformation for best geometry and interconnection
                taking suggestions from user ;

        END { else }

END { main }
```

3.1.2 pipeline

```
        PROCEDURE pipeline ( table, fname.dat )
(*
        table is the table of information about all variables and for loops,
              made during syntax analysis; fname.dat is the input filename.

        output :  the pipelined version of the input in fname.con  and implicit
                  storing of the dependencies of the pipelined new variables .
*)

BEGIN

        WHILE ( ( b_var = find_broadcasted_variable() ) $\neq$ NULL )

        BEGIN

            find all occurrences of b_var with same index set

            FOR all index variables not used by b_var DO

            BEGIN

                FOR all generated instances of b_var DO

                    include the index variables missing ;

                FOR all used instances of b_var DO

                BEGIN

                    FOR each missing index variable i_var DO

                    BEGIN

                        IF the for loop corresponding to i_var is DOWNTO type
                            THEN

                                include "i_var + STEP" ;

                                (* STEP is the step of the for loop *)

                            ELSE
                    include "$i\_var$ - $STEP$" ;
                END { for }
            END { for }
        END { for }
        IF b_var is used at least once THEN
            do initialization for each missing index ;
            (* i.e. initialize for the first instance of pipelining *)
        add dependency due to b_var in dependency vector list.
    END { while }
END { pipeline }
```

3.1.3 find_data_dependence

```
        PROCEDURE find_data_dependence ( )
( *
        finds data dependency for variables which are not broadcasted but are
                embedded in innermost for loop

        input :  details about indexed variables generated during syntax analysis

        output : adds dependencies as well as anti dependencies in
                 dependency vector list
* )
BEGIN
        WHILE( ( d_var = find_candidate_variable) $\neq$ NULL )
        (*
            a candidate variable has both used as well as generated occurrences
            and has not been pipelined already
        *)
```

```
BEGIN

    FOR all used occurrences of $d\_var, d\_var_i$ DO

    BEGIN

        FOR all generated occurrences of $d\_var_i$ starting from
            first occurrence DO

        BEGIN

            **subtract** indices used in generated instance from that of
                used instance and store the resultant tuple in result ;

            **multiply** the entries in result by -1 for DOWNTO type for loops ;

            **divide** the entries in result by respective *STEPs* of *for loops* ;

            if there is remainder for some entries in the above division

                **continue** ; (* i.e. don't consider this generated instance *)

            IF the first non-zero entry in result is -ve THEN

                **store** the result as *anti dependency* ;

            ELSE

                IF all the entries in result are equal to ZERO

                BEGIN

                    IF the generated variable appears *in a line*
                        before the used variable THEN

                        **store** zero for *dependency* due to $d\_var_i$ ;

                    ELSE

                        **continue** ;

                        (* i.e. don't consider this generated instance *)

                END { *if* }
                ELSE
                    IF the result is lexicographically less than *dependency*
                    till yet obtained due to $d\_var_i$ THEN

                        **store** it for ***dependency*** due to $d\_var_i$ ;

                END {*for* }

                IF ***dependency*** due to $d\_var_i$ is not zero THEN

                    **add** *dependency* due to $d\_var_i$ in ***dependency*** list;

            END {*for* }

        END {*while* }

    END { *find_data_dependency* }
```

3.1.4 find_PI_transform

```
PROCEDURE find_PI_transform()
(*
    input :   inputs are numerous and through global table data structure .

    output :  it trys to find out best of the $\prod$ transforms for certain
              heuristic ; in case it fails to find any $\prod$ transform it reports
              failure.
*)

    BEGIN

        FOR counter := 1, TO counter := maximum_level_of_nesting, DO

        BEGIN

            find all $\prod$'s whose sum of absolute values is equal to counter ;

            FOR all the $\prod$'s which satisfy the constraint of +ve time, $\prod_i$, DO

                IF $\prod_i$ gives less total execution time than the best $\prod$
                    available yet THEN
                **store** $\prod_i$ as the *best* $\prod$ *available* ;
          END
          IF a *valid best* $\prod$ *is available* THEN
               **report** it ;
          ELSE
               **report** failure ;
     END { *find_PI_transform* }
```

3.1.5 find_S_transform

```
PROCEDURE find_S_transform()
(*
     input :   inputs are numerous and through global *table* data structure.

     output :  the user is given the option to choose a good S matrix until
               he wants to try. If a valid S transform is found for users input
               success is reported. Else if he chooses to have all valid S
               transforms then all valid S transforms existing satisfying the
               time constraint are found out. The task of choosing good S among
               these is left on the user.

     note :    this leaving the task of choosing best S among many valid S
               transforms makes this software SEMIAUTOMATIC.
*)
BEGIN
     WHILE ( *not end of user's choices* ) DO
     BEGIN
          **read** user's input ;
          **test validity** of the user's $S$ ;
```

```
        IF user's $S$ *is valid* THEN
        BEGIN
            **report** success;
            **report** the final architecture and interconnection pattern;
        END { *if* }
    END { *while* }
    IF the user wants *all valid S* transforms THEN
    BEGIN
        **ask** for the dimension of the *systolic array* to be used;
        FOR all *communication pattern's* possible with the given dimensional
            *systolic array* DO
        BEGIN
            **find** the $SD$ satisfying the communication pattern ;
            (* $SD$ is the resultant geometry after transform *)
            **check for validity** of corresponding $S$;
            IF it is valid THEN
            BEGIN
                **add** $S$ as well as the resultant architecture to the
                    existing list of valid $S$ transforms ;
            END { *if* }
        END { *for all communications ...* }
    END { *if the user wants ...* }
END { *find_S_transform* }
```

3.2 Data Structures

Various data structures used for implementation of the algorithms are as described below.

[**ind_st**] This is a linked list with a string of length 20 in its data field. This is used to store the index variables used in the algorithm.

[**ind_str2**] This is a similar data structure as above. But it has a character pointer in its data field (aliasing). This is used to list the indices used with each indexed variable.

[**ind_str3**] This is a linked list storing pointer to ind_str2 data structure. It is used to store for_loop indices so that it can be used during pipelining broadcasted variables.

[**for_type**] This is used to store the data about each for_loop. It is a linked list having fields to store index variable, line no., flag to know whether DOWN TO is used, *from_string* storing the *from expression*, *to_string* storing the *to expression* , *end_line_no* storing the line number where the statement of the for loop ends and *step* storing step by which increment or decrement is done.

[**var_type**] This is used to store all use/generate information about one indexed variable. It is a triply linked list linking,

1. all indexed variables with same name,
2. indexed variable to its left,
3. indexed variable to its right,

(left right mean top to bottom in left to right manner)

[**updata**] This is a linked list used in pipelining. In each node it stores the pointer to original variable as well as the new set of indices to be used.

[**ind_type**] This is used to store make a table of all indexed variables. It is linked list storing in each node, a variable name and root of the linked list storing all uses of that variable.

[**int_str**] This is a linked list to store dependency vectors. It stores a dependency vector as well as anti dependency flag in each node.

[**depend_type**] This is a structure storing information to link dependency vectors and the variables in which it is found. In each node it stores the root of dependency vector list, the root of the list of variables due to which these dependencies are found (in matching order), and pointer to the innermost for loop in (nested) set of for loops.

[**var_list_type**] This is a linked list storing a link to var_type data structure as its data. It is used to create lists of used and generated variables of same variable name. It is useful in finding data dependency.

[**addline**] This is a circular linked list with each node containing a string of 134 characters representing the line already present in the file and pointer to a linked list of strings representing the lines to be added below the original line. This is required for typical string manipulation operations in a FILE to pipeline the broadcasted variables.

[**int_list**] This is a linked list to store parametric part of generalized solution for a variable as a result of solving Diophantine equations. Each node contains parameter number as well as its coefficient. It is used in soln_type data structure as explained below.

[**soln_type**] This is a linked list storing the parametric solutions of variables, used to return solutions of Diophantine equations. In each node it has constant (an int) as well as root of linked list representing the parametric part.

SAMPLE INPUT OUTPUT

We take a simple example to show how the program works. Let the following algorithm be written in a file say **inp1.dat**.

```
(* MATRIX MULTIPLICATION *)
BEGIN
    FOR $i := 1$ TO $n$ DO
    BEGIN
        FOR $j := 1$ TO $n$ DO
        BEGIN
            $c[i,j] := 0$;
            FOR $k := 1$ TO $n$ DO
            BEGIN
                $c[i,j] := c[i,j] + a[i,k] \times b[k,j]$;
            END ;
        END ;
    END ;
END ;
```

We may use a command as follows :

```
/user/mtc9512/project> parallelize inp1.dat
```

output :

Following is the *dependency matrix* **found out. after pipelining**

variable −>		*b$3*	*a$3*	*c$3*	
index	*step*	*dependency*	*dependency*	*dependency*	*for_type*
i	1	1	0	0	*to*
j	1	0	1	0	*to*
k	1	0	0	1	*to*

line no. 3 : FOR $i := 1$ TO n DO

value of limit variable n is [100] ? 5 <−

line no. 5 : FOR $j := 1$ TO n DO

value of limit variable n is [100] ? 5 <−

line no. 8 : FOR $k := 1$ TO n DO

value of limit variable n is [100] ? 5 <−

The ∏ matrix is [1 1 1] ;

Do you want to give S matrix [y] ? <−

Dimension of systolic array [2] ? <−

S[1,1] = [0] ? 1 <− ; S[1,2] = [0] ? <− ; S[1,3] = [0] ? <− ;

S[2,1] = [0] ? <− ; S[1,2] = [0] ? 1 <− ; S[1,3] = [0] ? <− ;

showing transform and resultant dependency :

$$T = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{array}{|c|} \hline \Pi \\ \hline \\ S \\ \hline \end{array}$$

$$\hat{D} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{array}{|c|} \hline \textit{TIME} \\ \hline \\ \textit{GEOMETRY} \\ \hline \end{array}$$

Do you want to give S matrix *[y]* ? n <−

```
Do you want all solutions [n] ? y <-
Dimension of systolic array [2] ? <-
```

As a result of pipelining we found the following file **inp1.con**

```
(* MATRIX MULTIPLICATION *)
BEGIN
    $i := 1;$
        FOR $j := 1$ TO $n$ DO
            FOR $k := 1$ TO $n$ DO
                $b\$3[k, j, i-1] := b[k, j];$
    FOR $i := 1$ TO $n$ DO
    BEGIN
        $j := 1$ ;
        FOR $k := 1$ TO $n$ DO
            $a\$3[i, k, j-1] := a[i, k]$ ;
        FOR $j := 1$ TO $n$ DO
        BEGIN
            $c[i, j] := 0;$
            $k := 1;$
            $c\$3[i, j, k-1] := c[i, j];$
            FOR $k := 1$ TO $n$ DO
            BEGIN
                $b\$3[k, j, i] := b\$3[k, j, i-1];$
                $a\$3[i, k, j] := a\$3[i, k, j-1];$
            $c\$3[i,j,k] := c\$3[i,j,k-1] + a\$3[i,k,j] \times b\$3[k,j,i]$;
         END ;
         $c[i,j] := c\$3[i,j,n]$;
      END ;
   END ;
END ;
```

Also a file named *inp1.trn* was created giving all valid transforms. It was found out that we have 576 valid transformations in this case.

The architecture and communication pattern corresponding to the transform due to user's S, found as above, is shown below.

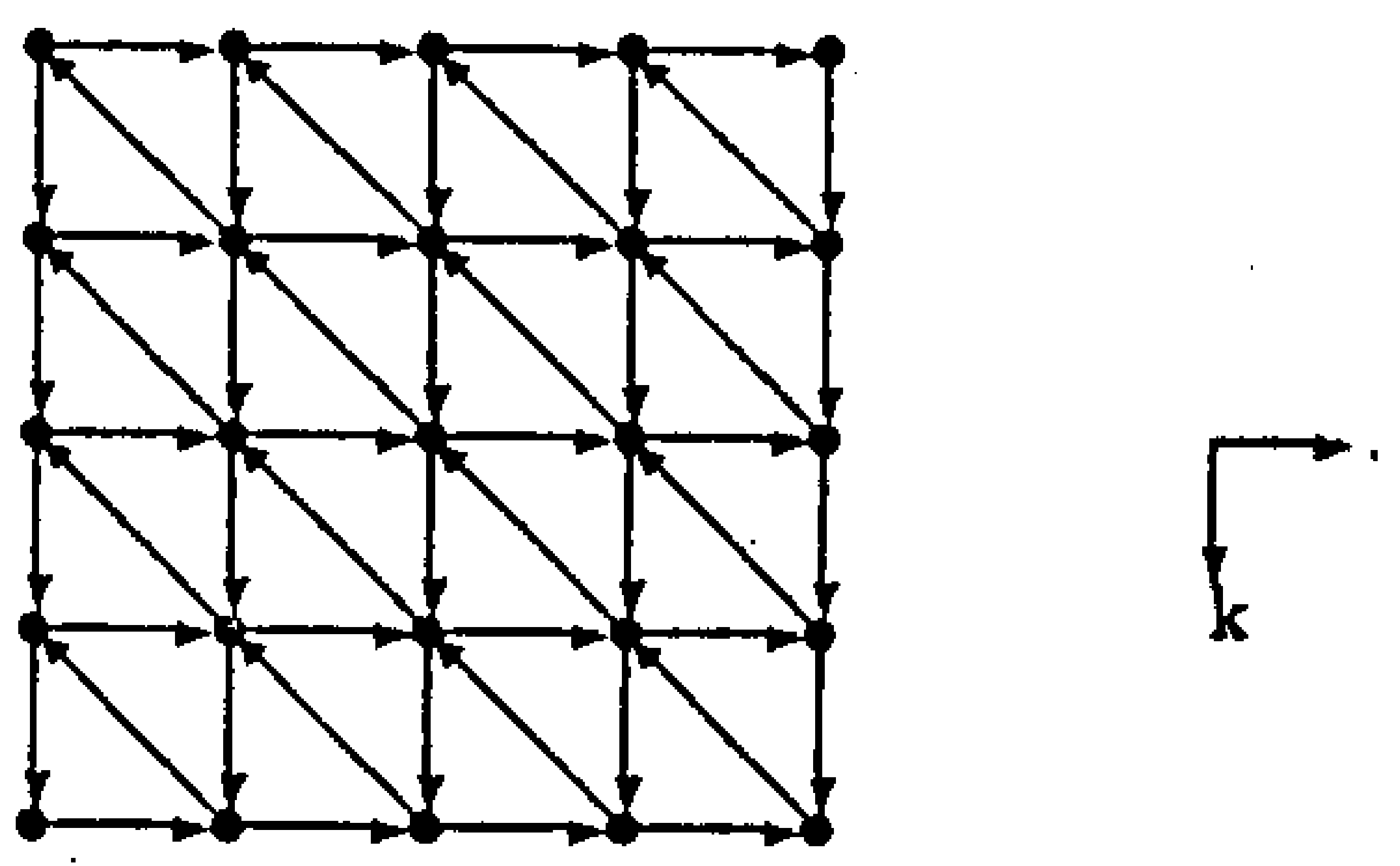

Fig. 2. Systolic Architecture for matrix multiplication.

CONCLUSIONS

We have implemented and enhanced the algorithm, developed by Moldovan[4], to convert sequential algorithms into its equivalent parallel form. The results obtained from the software, are same as results obtained by applying various other techniques for mapping algorithms into systolic array architecture manually.

We have found out that the algorithms for finding optimum transforms are exponential in time, but are still applicable in practical cases where number of iterations may be high, but level of embeddings and number of dependencies inside the innermost loop should be limited. But for theoretical, general cases the algorithm would be much time consuming.

we have also found out that the main problem in fully automating the process, is that of *finding a goodness criteria*, to compare different systolic architectures with different communication patterns. In absence of any human intuition, (i.e when our program is run in fully automatic mode) a large number of valid S transformations (S transform determines the resultant processor geometry and interconnection pattern) are found. Choosing the best architecture (manually) among this big set of possible architectures is tedious.

In a similar implementation, in a software named ADVIS, developed at the University of Southern California, as mentioned in Moldovan[4], the input requirements were algorithm index set, dependence matrix, and the matrix of permissible interconnections. Finding these in a big algorithm is time consuming for a user. In our software we have overcome this problem, as we need only the high level algorithm as input. This has been made possible by writing a **syntax analyzer** as well as finding correct algorithms for *pipelining* and for *determining data dependency.*

In course of implementation, we have generalized the algorithm for *pipelining broadcasted variables*, and the algorithm for *finding data dependency*, for a general case of *do loops* allowing *non unit steps* and *-ve steps as well* .

SCOPE OF FURTHER WORK

Finding a criteria to compare two systolic architectures can be taken as the immediate next step in this course of work.

Finding transforms for other regular architectures, besides systolic array architectures also remains open. Although we have done some work on it, it is yet to be completed.

In this work we are finding the induced redundant ordering on iterations by dependency analysis and trying to remove it by suitable transformations. But the task to find and remove induced dependency by operators, by operation analysis remains to be done. It is obvious that without this we may not reach for those good parallel algorithms like, *matrix multiplication in cube architecture*, by our automated method. So this task also remains to be done.

We have concentrated on static dependency only. Finding out values of dynamic data dependencies characterized by variables in dependency values, at run time is another related problem. It is important as many real life algorithms contain static as well as dynamic dependencies.

Last, but not the least, we will have to allow control structures like *IF THEN* and *GO TO* , before this semi automatic parallelization method can be said to be robust. At present the user has to rewrite the high level algorithm converting *IF THEN*'s to equivalent *DO loops* and avoiding *GO TO*'s before feeding it to our parallelizers. But this is tedious and needs to be automated.

APPENDIX

Diophantine Equations

An equation or a system of equations in one or more unknowns, which is to be solved in an integer domain (bounded or unbounded) is called a Diophantine equation or system of Diophantine equations. A simple example is

$$2x + 3y = 10;$$

Theorem : Integer solutions of the equation $a_1x_1 + a_2x_2 + \ldots + a_nx_n = a$, where the a_i's are integers, $a_1a_2, \ldots, a_n \neq 0$, exist if and only if gcd of $a_1, a_2, \ldots, a_n$ divides a .

This condition is obviously necessary and hence without loss of generality we may assume that the equation is already divided by the gcd of coefficients in left of the equation, such that gcd of $a_1, a_2, \ldots, a_n$ is 1.

Case 1: Now suppose that $n = 2$.

$$\text{i.e., } a_1x_1 + a_2x_2 = a, \qquad \ldots(1)$$

where we may suppose that, gcd of a_1, a_2 is 1, $a_1 > 0, a_2 > 0$ on writing $-x_2$ for x_2 if need be.

The solution can be given by following algorithm which is equivalent to Euclidian algorithm of finding gcd of a_1 and a_2.

1. Choose the greater of a_1 and a_2 (say a_2). We may write,

$$a_2 = q_1a_1 + r_1, \qquad \ldots(2)$$

where $0 < r_1 < a_1$, gcd of r_1, a_1 is 1 and q_1 is an integer.

Then substituting (2) in (1) we get,

$$a_1x_1 + q_1a_1x_2 + r_1x_2 = a, \qquad \ldots(3)$$

substituting $x_{11} = x_1 + q_1x_2$ we get,

$$a_1x_{11} + r_1x_2 = a, \qquad \ldots(4)$$

where $x_1 = x_{11} - q_1 x_2$. $\quad \dots(5)$

Next we may 2write, $a_1 = q_2 r_1 + r_2$, $\quad \dots(6)$

where $0 < r_2 < r_1$, gcd of r_2, r_1 is 1 and q_2 is an integer.

Substituting (6) in (4) we get,

$$q_2 r_1 x_{11} + r_2 x_{11} + r_1 x_2 = a, \qquad \dots(7)$$

substituting $x_{21} = x_2 + q_2 x_{11}$ we get,

$$r_2 x_{21} + r_2 x_{11} = a, \qquad \dots(8)$$

where $x_2 = x_{21} - q_2 x_{11}$ $\quad \dots(9)$

Continuing this process, we have a decreasing set of positive integers $r_1, r_2, \dots, r_n$ until we come to a stage where,

$$r_n x_{1i} + r_{n-1} x_{2j} = a, \text{ where } r_n = 1.$$

Then $x_{1i}, x_{2j}, x_{1,i-1}, x_{2,j-1}, x_{1,i-2}, x_{2,j-2}, \dots, x_2, x_1$ are successively given in terms of x_{2j}, x_{2j} can be taken as a parameter for the general solution.

If one solution (p, q) is known, all the solutions are given by $x_1 = p + ta_2, x_2 = q - ta_1$, where t is an integer parameter.

Case 2: Now suppose $n > 2$.
i.e., $a_1 x_1 + a_2 x_2 + \dots + a_n x_n = a$ $\quad \dots(1)$

Here we choose the coefficient with minimum absolute value, say ai and rewrite the equation as,

$$(a_1 \text{mod} a_i) x_1 + (a_2 \text{mod} a_i) x_2 + \dots + a_i x_{i1} + \dots + (a_n \text{mod} a_i) x_n = a, \dots(2)$$

where,

$$x_{i1} = (a_1/a_i) x_1 + (a_2/a_i) x_2 + \dots + (a_i/a_i) x_i + \dots + a_n/a_i x_n \qquad \dots(3)$$

It may be noted that the absolute value of all coefficients are less than or equal to a_i now.

Continuing the above process, and thus decreasing the absolute value of coefficients we will end up in the following condition,

one of the coefficients say of x_{ij}, the jth substituted value of x_i, is non zero and all other coefficients are zero. (We may note that if the above theorem is satisfied this coefficient will divide a, otherwise it will not divide and there will not be any integer solution to the above diophantine equation). So we get the value of the x_{ij} as an integer.

Now back substituting this value in the substitution done before we will get a parametric solution of $x_{i,j-1}$.

Continuing this process of back substitution finally we will get the parametric solution of $x_1, x_2, x_3, ..., x_n$.

Let's take the following example to show the functioning of the above algorithm.

Example :

$$45x_1 + 35x_2 + 30x_3 = 5, \qquad \ldots(1)$$

$$\text{i.e., } 15x_1 + 5x_2 + 30x_{31} = 5,$$

$$\text{where } x_{31} = x_1 + x_2 + x_3, \ldots(2)$$

$$\text{i.e., } 0x_1 + 5x_{21} + 0x_{31} = 5,$$

$$\text{where } x_{21} = 3x_1 + x_2 + 6x_{31}, \ldots(3)$$

$$\text{i.e., } x_{21} = 1, \qquad \ldots(4)$$

Substituting (4) in (3) we get,

$$x_2 = 1 - 3x_1 - 6x_{31}, \qquad \ldots(5)$$

Substituting (5) in (2) we get,

$$x_3 = x_{31} - x_1 - 1 + 3x_1 + 6x_{31},$$

$$\text{i.e., } x_3 = 7x_{31} + 2x_1 - 1; \qquad \ldots(6)$$

Thus considering $x_1 = t_1$, and $x_{31} = t_2$ as two parameters which may take any integer value we get the general solution of the above as,

$$x_1 = t_1,$$

$$x_2 = -3t_1 - 6t_2 + 1,$$

$$x_3 = 2t_1 + 7t_2 - 1,$$

It may be noted that in our case it is expected and also is seen that the number of parameters is one less than number of variables.

Case 3: Now suppose $n >= 2$ and we have a system of equations, $a_{r1}x_1 + a_{r2}x_2 + \dots + a_{rn}x_n = a_r$, $(r = 1, 2, \dots, i), i < n$.

In this case our algorithm will be extension of last case.

i. Choose first equation and solve the it (by the algorithm described in last case). If it couldn't be solved report failure, else we have expressions for $x_1, x_2, \dots, x_n$ using at best $n - 1$ parameters.

ii. Substitute the solution for first equation in the rest of equations and reduce to a system of i - 1 equations with n-1 variables.

iii. Continue the above process in the reduced system of linear equations. In case the system of equations is solvable we will finally find a single equation with n-i+1 variables. So we can solve it using the method described in last case, and the variables can now be expressed with at best n-i parameters. Back substituting the above recursively till we reach the original variables we get the generalized solution with at best n-i parameters.

Example :

$$45x_1 + 35x_2 + 30x_3 = 5, \qquad \dots(1)$$

$$10x_1 - 3x_2 - 6x_3 = 7, \qquad \dots(2)$$

Solving the first equation as described in the last case we get

$$x_1 = t_1, x_2 = -3t_1 - 6t_2 + 1, x_3 = 2t_1 + 7t_2 - 1.$$

Substituting these in 2nd equation we get,

$$10t_1 - 3(-3t_1 - 6t_2 + 1) - 6(2t_1 + 7t_2 - 1) = 7,$$

i.e., i.e. $7t_1 - 24t_2 + 3 = 7$,

i.e., i.e. $7t_1 - 24t_2 = 4$, $\dots(3)$

We solve this using algorithm of case 2.

Substituting $t_{11} = t_1 - 3t_2$, $\dots(a)$

we get,

$$7t_{11} - 3t_2 = 4 \qquad \dots(4)$$

Substituting $t_{21} = t_2 - 2t_{11}$, $\dots(b)$

we get,

$$t_{11} - 3t_{21} = 4. \qquad \dots(5)$$

Substituting $t_{12} = t_{11} - 3t_{21}$, $\dots(c)$

we get,

$$t_{12} = 4. \qquad \dots(6)$$

Back substituting value of t_{12} in (c) we get,

$$t_{11} = 4 + 3t_{21}. \qquad \ldots(7)$$

Back substituting value of t_{11} in (b) we get,

$$t_2 = t_{21} + 2(4 + 3t_{21}),$$

i.e., $t_2 = 7t_{21} + 8. \qquad \ldots(8)$

Back substituting value of t_2 and t_{11} in (a) we get,

$$t_1 = (4 + 3t_{21}) + 3(7t_{21} + 8),$$

i.e., $t_1 = 24t_{21} + 28. \qquad \ldots(9)$

Back substituting value of t_1, t_2 and t_3 in the solution of x_1, x_2, x_3 as obtained from the i'th equation we get,

$$x_1 = t_1 = 24t_{21} + 28. \qquad \ldots(10)$$

$$x_2 = -3t_1 - 6t_2 + 1,$$

i.e., $x_2 = -3(24t_{21} + 28) - 6(7t_{21} + 8) + 1,$

i.e., $x_2 = -114t_{21} - 131. \qquad \ldots(11)$

$$x_3 = 2t_1 + 7t_2 - 1,$$

i.e., $x_3 = 2(24t_{21} + 28) + 7(7t_{21} + 8) - 1,$

i.e., $x_3 = 97t_{21} + 111. \qquad \ldots(12)$

As we have a single parameter t_{21}, writing it as a single parameter t, we can express the general solution to the system of equations taken in this example as,

$$x_1 = 24t + 28,$$

$$x_2 = -114t - 131,$$

$$x_3 = 97t + 111.$$

Bibliography

1. Manish K. Singh, *"Semi-Automatic generation of parallel programs and parallel architecture"*, M.Tech. thesis, I.S.I Calcutta, July 1995.

2. 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.

3. 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.

4. 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.

5. Utpal Banerjee *et al.*, *"Automatic Program Parallelization"* , Proceedings of the IEEE, vol. 81, no. 2, Feb. 1993, pp. 211-243.

6. H. T. Kung, *"Let's design algorithms for VLSI systems"* , Proceedings of Caltech Conf. on VLSI, Jan. 1979, pp. 65-90.

7. H. T. Kung, *"The structure of Parallel Algorithms"* , Advanced Computing, vol. 19, 1980, pp. 65-111.

8. 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.

9. B. P. Sinha *et al.*, *"A parallel algorithm to compute the shortest paths and diameter of a graph and its VLSI implementation"*, IEEE Transactions on Computers, vol. C-35, no. 11, Nov. 1986, pp. 1000-1004.

10. Chien-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.

11. L. J. Mordell, *"Diophantine Equations"* , New York: Academic Press, 1969 , pp. 30-33.

12. Kai Hwang, *"Advanced Computer Architecture"* ,McGraw-Hill Inc., 1993.

13. Kai Hwang and F.A. Briggs, *"Computer Architecture and Parallel Processing"* , McGraw-Hill Inc.,1989.

14. M. J. Quinn, *"Designing Efficient Algorithms for Parallel Computers"*, McGraw-Hill Inc., 1988.

15. Selim G. Akl, *"The Design and Analysis of Parallel Algorithms"*, Prentice Hall, 1989.