

Purdue University  
**Purdue e-Pubs**

---

Department of Computer Science Technical  
Reports

Department of Computer Science

---

1985

## A Study of Data Interlock in VLSI Computational Networks for Sparse Matrix Multiplication

Rami G. Melhem

Report Number:  
85-505

---

Melhem, Rami G., "A Study of Data Interlock in VLSI Computational Networks for Sparse Matrix Multiplication" (1985). *Department of Computer Science Technical Reports*. Paper 426.  
<https://docs.lib.purdue.edu/cstech/426>

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.  
Please contact [epubs@purdue.edu](mailto:epubs@purdue.edu) for additional information.

*A STUDY OF  
DATA INTERLOCK IN VLSI COMPUTATIONAL NETWORKS  
FOR SPARSE MATRIX MULTIPLICATION*

Rami G. Melhem

Department of Computer Science  
Purdue University  
West Lafayette, IN 47907

**ABSTRACT**

The general question addressed in this study is: Are regular VLSI networks suitable for sparse matrix computations?. More specifically, we consider a special purpose self-timed network that is designed for certain specific dense matrix computation. We add to each cell in the network the capability of recognizing and skipping operations that involve zero operands, and then ask how efficient this resulting network is for sparse matrix computation?.

In order to answer this question, it is necessary to study the effect of data interlock on the performance of self-timed networks. For this, the class of pseudo systolic networks is introduced as a hybrid class between systolic and self-timed networks. Networks in this class are easy to analyze, and provide a means for the study of the worst case performance of self-timed networks. The well known concept of computation fronts is also generalized to include irregular flow of data, and a technique based on the propagation of such computation fronts is suggested for the estimation of the processing time and the communication time of pseudo systolic networks.

## 1. Introduction

The problem of solving large linear systems of equations on various types of parallel architectures has been receiving considerable attention. In fact, both direct and iterative parallel solution schemes have been studied for dense matrices ( e.g. [6,16] ) as well as for certain banded (e.g. [2,11,14] ) and sparse matrices (e.g. [3,4,20] ). In this paper, we focus our attention on iterative solutions of large sparse systems.

Usually, large sparse matrices that appear in practical applications correspond to large graphs with specified local connectivity (as for example in finite element analysis [24] ). A matrix of this type may be efficiently manipulated by means of a network of processes interconnected in such a way as to match the underlying graph [1,12]. However, given a specific interconnection, It is usually difficult, and sometimes impossible, to map the nodes of the specific graph into the cells of the network [5].

On the other hand, it is clearly inefficient to use the more general systolic architecture [15] for sparse matrix manipulation. In particular, assuming that  $\zeta$  is the percentage of zero elements in a matrix  $A$ , then  $\zeta\%$  of the resources of a systolic network may be wasted during the manipulation of  $A$  due to operations that involve zeroes and the associated data communication.

One way of reducing this waste of resources may be to replace the clocked synchronization in the network by a self-timed (data driven) scheme [21], and to add to each cell in the network the capability of recognizing and skipping trivial operations. The result is a shorter average operational cycle for each cell. However, data interlock may prevent any gain in the speed of the entire network. More precisely, a cell that skips an operation may be unable to start the next operation immediately if some of its input data are locked (temporarily) in neighboring cells.

In the following sections, we consider an operation that is fundamental in iterative solution schemes for sparse linear systems, namely the multiplication of a sparse matrix by a vector. More specifically, we assume that data driven networks are used for the multiplication and that each cell performs only those operations that involve non zero operands. We then evaluate such networks by studying the effect of data interlock on their efficiency and speed.

However, the asynchronous nature of data driven networks, together with the irregularity of the structures of sparse matrices, make the temporal analysis of data flow very difficult, if not impossible. For this reason, we define in Section 2 the class of Pseudo Systolic networks. Namely a hypothetical class of synchronous networks where the operation of each cell depends on the input data. These networks are slower than self-timed networks, however, they are easier to analyze and provide a tool for the establishment of upper bounds on execution times of data driven networks.

In Section 3, we generalize the well known concept of computation fronts [23] to include irregular data propagation. We also describe a technique for the estimation of the execution time of pseudo systolic networks. Our purpose is to show that the inefficiency caused by the application of systolic networks to sparse matrices may be restored if data driven synchronization is used. In other words, we may have both the efficiency of sparse matrix manipulation techniques and the speed of local communications and specialized cells. This argument is backed up by the experimental results that we present in Section 4.

## 2. Pseudo Systolic Networks.

Both systolic and self-timed computational networks may be defined [18] as networks in which each cell repeats the execution of a specific cycle. Namely 1) read the data from its input links, 2) perform a specific computation, and 3) write the results on its output links. The mechanism that initiates the cycles in the

various cells is different in the two types of networks. More specifically, systolic networks employ a global clock to initiate the cycles of all the cells in the network simultaneously, while the cycles of each cell in self-timed networks are initiated independently by the availability of input data.

Conceptually, an internal communication link  $l_{k,q}$  directed from a cell  $k$  to a cell  $q$  is a buffer that has a certain capacity of, say,  $b$  data items. Only cell  $k$  may write on the buffer (if it is not full), and only cell  $q$  may read from the buffer (if it is not empty). Here, we will refer to this buffer as either "the buffer at the input port of cell  $q$ " or "the buffer at the output port of cell  $k$ ". In practical implementations, however, the buffer may be distributed between the two ports.

In addition to internal communication links, we may have network input/output links that connect cells in the network to a host system. In order to isolate the effect of data interlock among the cells of the network from any delay that may be caused by slow communication with the host, we assume that data are provided on the network input links as soon as they are needed, and consumed from network output links as soon as they are produced. In other words, the buffers on the network input links are never empty and those on the network output links are never full.

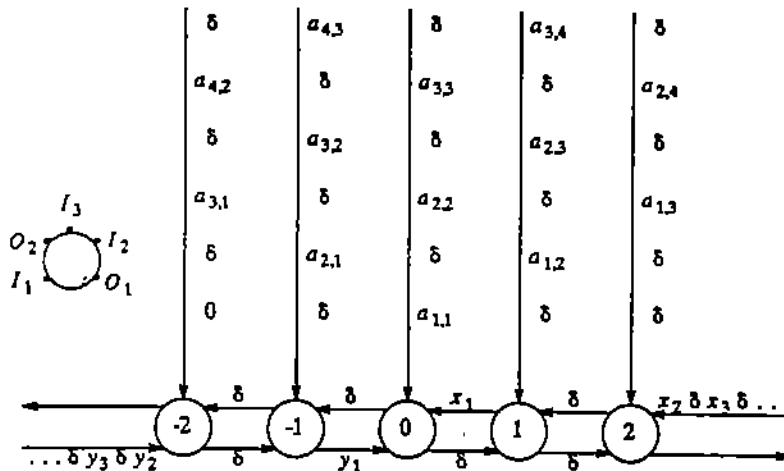


Figure 1 - The network  $MV_1$  during the third cycle

For example, consider the network  $MV_1$  shown in Fig 1 [14]. This network may be used for the computation of the product vector  $y$  of an  $n \times n$  banded matrix  $A = \{a_{i,j}\}$  by an  $n$ -dimensional vector  $x$ . For simplicity, we assume that the number of upper diagonals of  $A$  is equal to the number of lower diagonals, namely  $B_h$ , and hence, the band-width of  $A$  is  $B = 2B_h + 1$ . In Fig 1, we let  $B_h = 2$  and label the cells by integers in the range  $[-B_h, B_h]$ . Any cell in the network has three input ports and two output ports, namely  $I_1, I_2, I_3, O_1$  and  $O_2$ , respectively. Its operation may be described by one of the following algorithms, depending on the type of synchronization used. (Here,  $O \leftarrow \alpha$  means that  $\alpha$  is written on the output port  $O$  and  $[I]$  denotes the data item on the input port  $I$ . When this item is read, it is removed from the input buffer).

**ALG1 : A systolic cycle**

1) Wait for phase 1 of the global clock ;

$$\alpha = [I_1] ; \beta = [I_2] ; \gamma = [I_3]$$

2)  $\rho = \alpha + \beta * \gamma$

3) Wait for phase 2 of the global clock

4)  $O_2 \leftarrow \beta ; O_1 \leftarrow \rho.$

**ALG2 : A self-timed cycles**

1) Wait until the buffers at  $I_1, I_2$  and  $I_3$  are not empty ;

$$\alpha = [I_1] ; \beta = [I_2] ; \gamma = [I_3]$$

2)  $\rho = \alpha + \beta * \gamma$

3) Wait until the buffers at  $O_1$  and  $O_2$  are not full

4)  $O_2 \leftarrow \beta ; O_1 \leftarrow \rho.$

In Figure 1, we show the sequence of elements that should be applied on each

input link of the network. Namely, the elements of the  $k^{th}$  off diagonal of  $A$ ,  $-B_h \leq k \leq B_h$ , are applied to port  $I_3$  of cell  $k$ , the elements of  $x$  are applied to port  $I_2$  of cell  $B_h$ , and the elements of the result vector  $y$  (initialized to zeroes) are applied to port  $I_1$  of cell  $-B_h$ . For the systolic operation, successive elements are applied at consecutive time units, while for the self-timed operation timing is not crucial as long as the order of the elements is preserved. As a result, in systolic operation, it is usually necessary to pad the sequence with don't care elements whose values are irrelevant to the computation. These don't care elements are denoted by  $\delta$  and may be removed in the self-timed network [18].

The elements  $y_1, \dots, y_n$  of the product vector may be computed in  $2n-1$  systolic or self timed cycles. Adding to this  $2B_h+1$  cycles that are needed to fill in the initial data and flush the results, we conclude that the total execution time of the network is the time for the completion of  $2(B_h + n)$  cycles. More precisely, we have

$$T_{sys}(MV_1) = 2 (B_h + n) ( \tau_{c,sys} + \tau_m )$$

and

$$T_{self}(MV_1) = 2 (B_h + n) ( \tau_{c,self} + \tau_m )$$

Where  $\tau_m$  is the time for a floating point multiply/add operation, and  $\tau_{c,sys}$  and  $\tau_{c,self}$  are defined as the time needed for communication during the execution of one systolic or self timed cycle, respectively. This includes the time for reading data from the buffers associated with the input ports and for writing data on the buffers associated with the output ports, plus the time for the transmission of signals and any synchronization overhead (clock propagation or shake hand protocol). Note that the execution time of one cycle of ALG2 is equal to  $\tau_{c,self} + \tau_m$  only if the input buffers are not empty when step 1 is reached and the output buffers are not full when step 3 is reached.

Assuming that  $\zeta\%$  of the elements in the band of the matrix  $A$  are zeroes, then it is clear that  $\zeta\%$  of the resources in either the systolic or the self timed versions of  $MV_1$  are wasted in the execution of trivial operations in step 2 of ALG1 and ALG2. In order to reduce this waste, we may attempt to skip the floating point operation whenever  $[I_3] = 0$ . More specifically, we may replace step 2 in ALG1 and ALG2 by

2) IF  $(\gamma = 0)$  THEN 2.1)  $\rho = \alpha$   
ELSE 2.2)  $\rho = \alpha + \beta * \gamma$

An execution of a cycle that goes through step 2.1 is called a trivial execution of the cycle, otherwise the execution is called non trivial. In the case of systolic networks, the time for completing either a trivial or a non-trivial execution is the same, namely, the period of the global clock.

On the other hand, trivial executions of self timed cycles may, or may not, be shorter than non trivial executions, depending on the time spent in steps 1 and 3 of the cycle. Hence, the total execution time of the self timed network, denoted in this case by  $T_{self/skip}(MV_1)$ , depends primarily on the effect of data conflict on the execution of the individual cells.

How much, if at all, do we gain by skipping trivial operations in self timed networks?, or stated differently, how much of the  $\zeta\%$  inefficiency in  $T_{self}(MV_1)$  may be restored in  $T_{self/skip}(MV_1)$ ? The precise answer to this question necessitates the construction of a mathematical model for the estimation of  $T_{self/skip}$ , which is very complex due to the asynchronous concurrency that exists between the cells of the network. Alternatively, we may apply a worst case analysis to obtain an upper bound on  $T_{self/skip}$  and then use this bound to estimate safely the speed-up ratio  $T_{self} / T_{self/skip}$ .



In order to pursue the second alternative, we consider a new hypothetical type of networks that are both data driven and synchronous, namely Pseudo Systolic networks. Each cell in such a network repeats the execution of the following algorithm:

**ALG3 : Pseudo Systolic Cycle**

- 1) Wait until the buffers at  $I_1, I_2$  and  $I_3$  are not empty ;  
 $\alpha = [I_1] ; \beta = [I_2] ; \gamma = [I_3]$
- 2) IF ( $\gamma \neq 0$ ) THEN
  - 2.1) Wait until the buffers at  $O_1$  and  $O_2$  are not full
  - 2.2)  $O_1 \leftarrow \alpha ; O_2 \leftarrow \beta$
  - 2.3) Go To step 1.
- 3) Wait for a synchronization signal
- 4)  $\rho = \alpha + \beta * \gamma$
- 5) Wait until the buffers at  $O_1$  and  $O_2$  are not full,
- 6)  $O_2 \leftarrow \beta ; O_1 \leftarrow \rho$ .

In other words, trivial operations are first skipped until a non trivial operand is found in  $[I_3]$ , then, the multiplication is performed. The role of the synchronization in step 3 will be discussed later. Note, however, that if this synchronization is removed, then the execution of the network will be identical to the execution of the self-timed network. The only difference is that trivial executions of successive self timed cycles are executed in a single pseudo systolic cycle.

Because a slow down in the execution of any cell in the self-timed network cannot speed up the execution of the entire network, we may conclude that pseudo systolic networks cannot be faster than self-timed networks. Hence,  $T_{self/skip}(MV_1)$  is bounded from above by  $T_{pseudo}(MV_1)$ ; the execution time of the pseudo systolic version of  $MV_1$ .



connected to a controller that detects the termination of the communication phase and issues a synchronization signal. At that instant, all the cells that are blocked at step 3 perform the multiplication (step 4), simultaneously, while the other cells remain idle. This is the processing phase. A communication phase followed by a processing phase is called a global cycle of the network.

### 3. Efficiency and speed of pseudo systolic networks

#### 3.1. Consistency of data flow.

In order to estimate the execution time of pseudo systolic networks, we first formalize two conditions that are necessary for the consistency of any stream of data  $z_1, z_2, \dots$  flowing through a series of linearly connected cells  $c_1, c_2, \dots$  (see Figure 2). These conditions ensure that the order of data is preserved and that the capacity of the communication lines is not violated. More precisely, if at any instant,  $z_i$  is at cell  $c_k$  and  $z_j$  is at cell  $c_q$ , with  $k > q$ , then

C1)  $i < j$ ,

C2)  $(j - i) \leq b (k - q)$ , where  $b$  is the number of data items that can be buffered between any two consecutive cells.

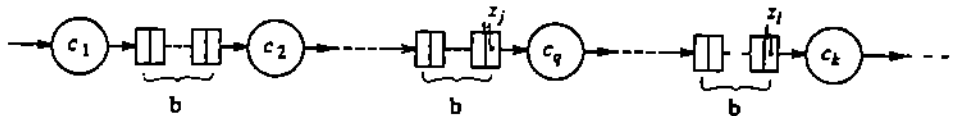


Figure 2 - Data flow through linearly connected cells.

#### 3.2. Computation fronts

In this section, we will introduce the concept of computation fronts using the pseudo systolic version of the network  $MV_1$  as an example. It should be clear, however, that the same concept may be applied to any systolic or pseudo systolic network. It may also be applied to self timed networks in which data communication

and actual processing take place in separate phases of execution (see for e.g. [16]).

Given a specific sparse matrix  $A$ , we assume that  $MV_1$  completes the multiplication  $y=Ax$  in  $N_1$  global cycles, and we define the function  $\alpha : [-B_h, B_h] \times [1, N_1] \rightarrow A$  such that  $\alpha(k, i)$  is the element of  $A$  that appears on port  $I_3$  of cell  $k$  at the beginning of the processing phase of the  $i^{th}$  global cycle. Here,  $\alpha$  is a total function by the assumption that the elements of  $A$  are supplied to the network as soon as they are needed.

Although a data item is always available on  $I_3$  of any cell during a specific processing phase, only those cells that receive the corresponding elements of the vectors  $x$  and  $y$  on  $I_1$  and  $I_2$ , respectively, perform a floating point operation, while the other cells remain idle. We let  $M_i$  be the subset of cells that are not idle during the  $i^{th}$  processing phase and we define the  $i^{th}$  computation front as the set of elements of  $A$  that are operated upon during this phase. More precisely, we define

$$CF_i = \{\alpha(k, i) \mid k \in M_i\}$$

Note that the members of  $CF_i$ , for any  $i$ , are non zero elements of  $A$ .

Computation fronts may be constructed directly from the structure of  $A$  by applying the conditions of Section 3.1. In order to be more specific, we note that successive inputs to port  $I_3$  of cell  $k$  are the elements of the  $k^{th}$  off diagonal of  $A$ . In other words, we may define the function  $d : [-B_h, B_h] \times [1, N_1] \rightarrow [1, \pi]$  such that  $\alpha(k, i) = a_{d(k, i), d(k, i)+k}$ . Hence, at the beginning of the  $i^{th}$  processing phase the data items at  $I_1$ ,  $I_2$  and  $I_3$  of any cell  $k \in M_i$  are  $x_{d(k, i)+k}$ ,  $y_{d(k, i)}$  and  $a_{d(k, i), d(k, i)+k}$ , respectively. Similarly, the corresponding data at any other cell  $q \in M_i$  are  $x_{d(q, i)+q}$ ,  $y_{d(q, i)}$  and  $a_{d(q, i), d(q, i)+q}$ , respectively. assuming that  $k > q$  and applying the conditions for the consistency of data flow on the  $y$  and  $x$  data streams, respectively, we get

$$0 < d(q, i) - d(k, i) \leq b(k - q) \quad (1)$$

$$0 < d(k, j) + k - d(q, j) - q \leq b(k - q) \quad (2)$$

Now, consider the two axes  $I$  and  $J$  shown in Figure 3. The line joining any two elements  $a_{d(k, j), d(k, j)+k}$  and  $a_{d(q, j), d(q, j)+q}$  in  $A$  has a slope  $s$  on the  $J$  axis given by

$$s = \frac{d(k, j) - d(q, j)}{d(k, j)+k - d(q, j)-q} \quad (3)$$

If these elements are in the same computation front  $CF_i$ , then  $s$  should satisfy (1) and (2). For this, straight forward manipulation gives

$$-\infty < s < 0 \quad (4)$$

That is, if the elements of a specific computation front are joined by a piece-wise linear curve (see Figure 3), then, its line segments should have a slope in the range  $(90^\circ, 180^\circ)$ .

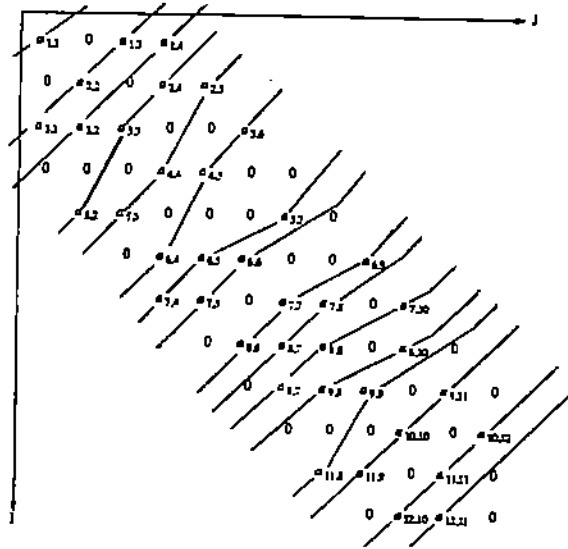


Figure 3 - Construction of computation fronts

With this result, we may now construct successive computation fronts graphically by starting at the top left corner of  $A$  and joining the non zero elements of  $A$  by piecewise linear curves that satisfy (4). This construction is shown in Figure 3 where it is found that 16 fronts are required to complete the computation for the given specific structure of  $A$ . That is  $N_1 = 16$ .

On the other hand, if the operations involving zeroes are not skipped, then  $2n - 1 = 23$  computation fronts are needed for the given matrix (each having a slope =  $135^\circ$ ). However, 33 elements of the 72 entries in the band of the given matrix are zeroes, that is  $\frac{33}{72} = 46\%$  of the computation is wasted in trivial operations. In terms of computation fronts, this is equivalent to  $23 * 46\% \approx 10$  wasted fronts. Clearly, by skipping trivial operations, we reduced the number of fronts by 7, that is we restored about 70% of the wasted resources. Of course, data conflict is the reason that prevents the complete restoration of the wasted resources.

It is obvious that the number of computation fronts is determined by both the number of zeroes in  $A$  and the distribution of these zeroes. In particular, consider a matrix with non zero diagonal elements. The condition that computation fronts cannot be parallel to the diagonal of the matrix implies that its non zero elements cannot be covered with less than  $n$  computation fronts. That is the maximum speed up factor that may be obtained by skipping trivial operations is  $\frac{2n-1}{n} \approx 2$ , irrespective of the number of zeroes in the matrix. Given that most of the matrices that result from practical applications have non zero diagonal elements, it seems necessary to reorganize the network such that computation fronts parallel to the diagonal of the matrix are allowed.

### 3.3. A Network for Matrices with non zero diagonals.

Consider the network  $MV_2$  shown in Figure 4. It is a linear network composed of  $B$  cells labeled by the integers  $1, \dots, B$ . Each cell has two input ports and two output ports, namely  $I_1, I_2, O_1$  and  $O_2$ , respectively, and is equipped with a counter 'Ct' and an accumulator 'Acc'. Assuming systolic (or self timed) synchronization, the cycle of each cell may be described by the following algorithm:

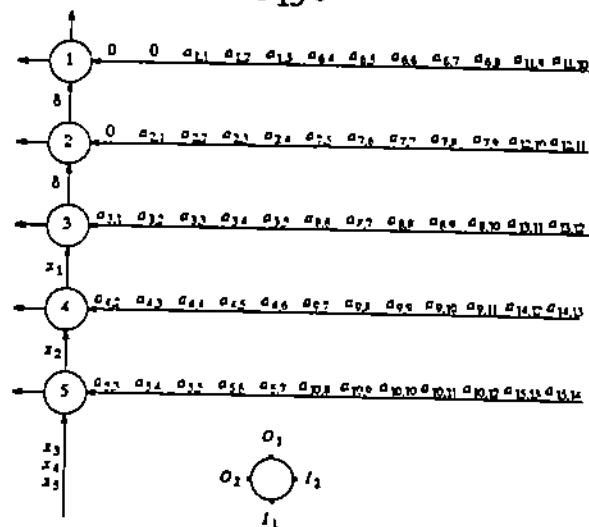


Figure 4 - A snap shot of  $MV_2$  after two cycles ( $B = 5$ )

**ALG5**

/\* Initially,  $Ct = B_h + 1$ . This allows data to fill-in during the first  $B_h$  cycles \*/

1) Wait for phase 1 of the clock (or until the buffers at  $I_1$  and  $I_2$  are not empty);  $\alpha = [I_1]$ ;  $\beta = [I_2]$

2)  $\rho = [Acc] + \alpha * \beta$

3) Wait for phase 2 of the clock (or until the buffer at  $O_1$  is not full)

4) If ( $Ct = B$ ) THEN  $O_1 \leftarrow \alpha$ ;  $O_2 \leftarrow \rho$ ;  $[Acc] = 0$ ;  $Ct = 1$

ELSE  $O_1 \leftarrow \alpha$ ;  $Acc \leftarrow \rho$ ;  $Ct = Ct + 1$

The elements of the vector  $x$  are applied to port  $I_1$  of cell  $B$ , and the elements in the rows  $k, k+B, k+2B, \dots$ , of the matrix  $A$  are concatenated and applied to port  $I_2$  of cell  $k$  (see Fig 4). More precisely, we may define a new  $B \times n$  matrix,  $A^*$ , such that for  $i=1, \dots, B$  and  $j=1, \dots, n$ , we have  $a_{i,j}^* = a_{i+rB,j}$ , where  $r = (j-i+B_h) \div B$ . That is the band of the matrix  $A$  is sliced every  $B$  rows, and all the slices are concatenated into the matrix  $A^*$ . With this, the inputs to port  $I_2$  of cell  $k$  are simply the elements of row  $k$  in the matrix  $A^*$ .

Given this input, it may be easily seen (or formally verified using the models in [17,19] or [7]) that the elements  $y_1, \dots, y_B$  of the result vector are produced on port

$O_2$  of cells  $1, \dots, B$ , respectively, at the end of cycle  $B_h + B$ . The elements  $y_{B+1}, \dots, y_{2B}$  are produced on the same ports at the end of cycle  $B_h + 2B$ , and in general,  $y_{rB+1}, \dots, y_{rB+B}$ ,  $r=0,1,\dots$  are produced at the end of cycle  $B_h + (r+1)B$ . That is, the computation terminates after  $B_h + \beta B$  cycles, where  $\beta = ((n-1) \div B) + 1$ . This result applies to both systolic and self timed synchronization and hence the time for the completion of the computation in either case is

$$T_{sys}(MV_2) = (B_h + \beta B) (\tau_m + \tau_{c,sys}) \quad (5.a)$$

or

$$T_{self}(MV_2) = (B_h + \beta B) (\tau_m + \tau_{c,self}) \quad (5.b)$$

As for the case of  $MV_1$ , the execution time of the self timed version of  $MV_2$  may be reduced if we replace step 2 in ALG5 by a conditional statement that skips trivial operations. The execution time of the resulting computation, namely  $T_{self/skip}(MV_2)$ , is bounded by the execution time of the corresponding pseudo systolic computation, namely  $T_{pseudo}(MV_2)$ .

Computation fronts for pseudo systolic executions of  $MV_2$  may be constructed by applying the conditions C1 and C2 of Section 3.1 on the  $x$  data stream. More precisely, we let  $N_2$  be the number of global cycles needed for the completion of the computation for a specific matrix  $A$  and we define the function  $g : [1,B] \times [1,N_2] \rightarrow [1,n]$  such that for any  $1 \leq k \leq B$ , the element  $a_{k,g(k,j)}^*$  of  $A^*$  is at port  $I_2$  of cell  $k$  at the beginning of the processing phase of the  $i^{th}$  global cycle. If  $M_i$  is the subset of cells that are not idle during this phase, then we may define the  $i^{th}$  computation front by

$$CF_i = \{a_{k,g(k,j)}^* \mid k \in M_i\}$$

Noting that for any cell  $k \in M_i$ ,  $x_{g(k,j)}$  is at port  $I_1$  at the beginning of the  $i^{th}$  processing phase, we may apply the conditions C1 and C2 to conclude that if  $a_{k,g(k,j)}^*$  and  $a_{q,g(q,j)}^*$  are in  $CF_i$ , and  $q < k$ , then



$$0 < g(k,i) - g(q,i) \leq b (k - q) \quad (6)$$

Given the two axes shown in Figure 5, we may use (6) to prove that the slope  $s$  (on the  $J$  axis) of the line joining any two elements in the same computation front should satisfy

$$\frac{1}{b} \leq s < \infty \quad (7)$$

where  $b$  is the buffer capacity of any communication line in the network.

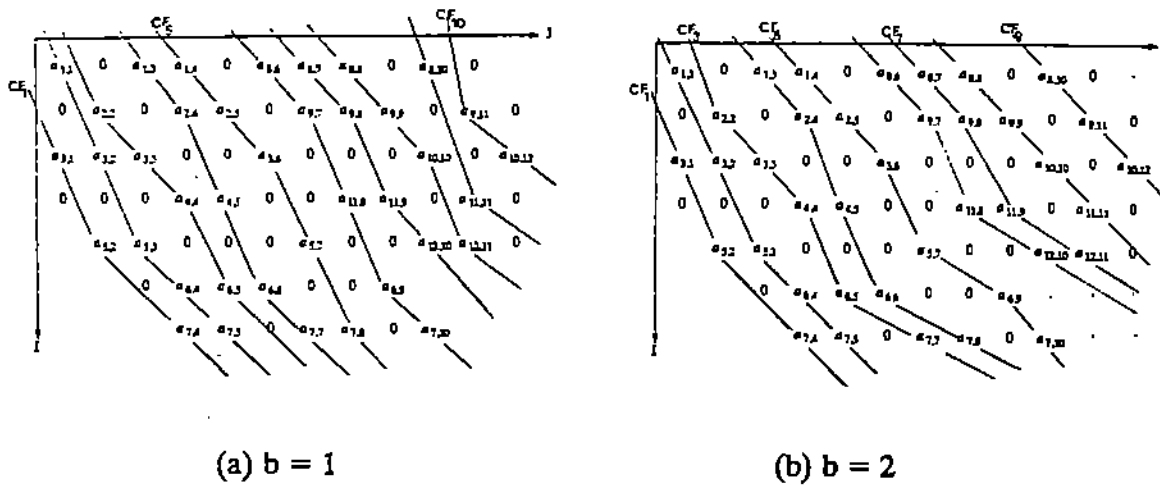


Figure 5 - Computation fronts for  $MV_2$

Condition (7) may be used to construct computation fronts for any sparse matrix. The result of this construction for the same matrix used in Section 3.2 is shown in Figures 5a and 5b, where we assumed, respectively, that  $b = 1$  and  $b = 2$ . From (7), the slope of a computation front is restricted to the range  $[45^\circ, 90^\circ)$  if  $b = 1$ , and  $[\tan^{-1}\frac{1}{2}, 90^\circ)$  if  $b = 2$ . Clearly, 10 and 9 fronts are needed, respectively, to complete the computation.

The number of computation fronts may be used for the comparison of the speed of different computations provided that the time for physically moving the data in the network is negligible with respect to the time for floating point operations, that is  $\tau_{c, self} \ll \tau_m$ . This assumption may be justified in the case of local

dedicated interconnections, especially if the synchronization and the communication protocols are implemented in hardware. On the other hand, if these protocols are implemented in software, or common communication channels are used, then the value of  $\tau_{c\_self}$  may be relatively large and hence the time for the communication phases of pseudo systolic networks should not be ignored.

### 3.4. Communication time in MV<sub>2</sub>

The synchronization of pseudo systolic networks is such that all possible communications take place during the communication phases of the global cycles. More specifically, during the  $i^{th}$  communication phase, each cell transmits from  $I_1$  to  $O_1$  successive elements of the vector  $x$  corresponding to zero elements of  $A$ . This process continues until either an element of  $x$  corresponding to a non zero element of  $A$  is received on  $I_1$ , or no more data items become available at  $I_1$ .

In order to estimate the time of the communication phases, we define for each global cycle  $i$  the  $x$ -stream profile,  $xP_i$ , to indicate the content of the buffers on the communication links transmitting the elements of the vector  $x$ . Future inputs to the network are included in  $xP_i$  by assuming that they are stored in an arbitrarily long buffer associated with the input link of cell  $B$ . More precisely, we define the function  $xP_i : [1, B] \times \{1, 2, \dots\} \rightarrow \{x_j ; j=1, \dots, n\}$  such that  $xP_i(k, q)$  is the content of location  $q$  in the buffer associated with the input port  $I_1$  of cell  $k$  at the beginning of the  $i^{th}$  processing phase. If this buffer location is empty, then  $xP_i(k, q)$  is undefined (denoted by  $\epsilon$ ). Note that the domain of the second argument to  $xP_i$  is taken to be the set of positive integers rather than  $[1, b]$ , where  $b$  is the capacity of each buffer. This is consistent with the assumption that the input buffer to cell  $B$  may be arbitrarily long.

We also define the inverse function  $xP_i^{-1} : \{x_j ; j=1, \dots, n\} \rightarrow [1, B] \times \{1, 2, \dots\}$  such that  $xP_i^{-1}(x_j)$  is the location of  $x_j$  in the  $i^{th}$  profile. More precisely

$$xP_i^{-1}(x_j) = \begin{cases} (k, q) & \text{if } xP_i(k, q) = x_j \\ \uparrow & \text{if } x_j \notin \text{range of } xP_i \end{cases}$$

Given a certain buffer location  $(k, q)$ , the following predicate tests whether this location is occupied or not at the beginning of the  $i^{\text{th}}$  processing phase

$$occ_i(k, q) = \begin{cases} 0 & \text{if } xP_i(k, q) = \uparrow \\ 1 & \text{otherwise} \end{cases}$$

In order to construct the profile  $xP_i$  from the  $i^{\text{th}}$  computation front, we assume that  $M_i = \{c_1, c_2, \dots, c_m\}$  with  $c_1 < c_2 < \dots < c_m$ . For each cell  $c_a \in M_i$ , we know that  $x_{g(c_a, j)}$  is at port  $I_1$  of  $c_a$  during the  $i^{\text{th}}$  processing phase. Hence, we may set  $xP_i(c_a, 1) = x_{g(c_a, j)}$ . Moreover, given any two cells  $c_a$  and  $c_{a+1}$ ,  $1 \leq a \leq m-1$ , the elements  $x_{g(c_a, j)}, \dots, x_{g(c_{a+1}, j)-1}$  of the vector  $x$  should occupy consecutive buffer locations on the communication lines between  $c_a$  and  $c_{a+1}$  starting at cell  $c_a$ . Finally, the elements  $x_{g(c_m, j)}, x_{g(c_m, j)+1}, \dots$  should occupy consecutive buffer locations on the communication lines following cell  $c_m$ . More precisely,  $xP_i$  may be computed as follows:

FOR  $a=1, \dots, m$  DO

1) IF  $(a \neq m)$  THEN  $d = g(c_{a+1}, j) - g(c_a, j) - 1$   
 ELSE  $d = n - g(c_m, j)$

2)  $k = c_a$  ;  $q = 1$

3) FOR  $l = g(c_a, j), \dots, g(c_a, j)+d$  DO

3.1)  $xP_i(k, q) = x_l$

3.2) /\* Get the next buffer location \*/

IF  $(q < b$  OR  $k = B)$  THEN  $q = q+1$

ELSE  $k = k+1$  ;  $q = 1$

Before the beginning of execution, the data profile may be defined by

$$x^{P_0}(k, q) = \begin{cases} x_q & \text{for } k = B, \text{ and } q = 1, \dots, n \\ 1 & \text{otherwise} \end{cases}$$

That is all input items are stored in the buffer of cell  $B$ . Noting that the data profile do not change during processing phases, then it becomes clear that the time for the  $i^{\text{th}}$  communication phase is the time required to change the data profile from  $x^{P_{i-1}}$  to  $x^{P_i}$ . If we denote this time by  $\Delta_i$ , then the execution time for the entire pseudo systolic computation may be expressed by

$$T_{\text{pseudo}}(MV_2) = \sum_{i=1}^{N_2} (\Delta_i + \tau_m) \quad (8)$$

assuming that  $\Delta_i(x_j)$  is the time needed to move  $x_j$  from its position in  $x^{P_{i-1}}$  to its position in  $x^{P_i}$ , then we may write

$$\Delta_i = \max \{ \Delta_i(x_j) ; x_j \in \text{range of } x^{P_i} \}$$

The mathematical formula for the computation of  $\Delta_i(x_j)$  for any  $x_j \in x^{P_i}$ , is complex and it seems that the simplest way for the evaluation of  $\Delta_i$  is the discrete simulation of the transformation from  $x^{P_{i-1}}$  to  $x^{P_i}$ . However, an upper bound may be easily obtained for  $\Delta_i(x_j)$ . For this, we let  $(k, q) = x^{P_{i-1}^{-1}}(x_j)$  and  $(k', q') = x^{P_i^{-1}}(x_j)$  be the locations of  $x_j$  in  $x^{P_i}$  and  $x^{P_{i-1}}$ , respectively. Then, the maximum number of read/write sub-cycles that have to elapse before  $x_j$  reaches the position  $(k, q)$  starting from  $(k', q')$  is bounded by  $\sigma_i(x_j)$ , where

$$\sigma_i(x_j) = (k' - k) + \sum_{u=k}^{k'-1} \sum_{v=2}^b \text{occ}_i(u, v) + \sum_{v=2}^{q'} \text{occ}_i(k', v)$$

From this and (8), we may establish the following bound

$$T_{\text{self/skip}}(MV_2) \leq T_{\text{pseudo}}(MV_2) \leq \sigma \tau_{c, \text{self}} + N_2 \tau_m \quad (9)$$

where  $\sigma = \sum_{i=1}^{N_2} \max \{ \sigma_i(x_j) ; x_j \in \text{range of } x^{P_i} \}$ .

### 3.5. Partitioning the computation by folding rows

So far, we assumed that the number of cells, say  $\lambda$ , in  $MV_2$  is equal to the band width  $B$  of the matrix  $A$ , and hence that each row of the modified matrix  $A^*$  is allocated to one cell in  $MV_2$ . If, however,  $B$  is larger than  $\lambda$ , then the rows of  $A^*$  may be partitioned into  $\lambda$  groups that are allocated to the  $\lambda$  cells of  $MV_2$ . More specifically, if  $B = r\lambda$ , for some  $r$ , then every consecutive  $r$  rows of  $A^*$ , namely rows  $(rk - i)$ , for some  $k$ ,  $1 \leq k \leq B$  and  $i=0, \dots, r-1$ , may be allocated to cell  $k$  in  $MV_2$ . Whenever an element  $x_j$  is received by that cell, it is multiplied by the corresponding  $r$  elements  $a_{rk-i, j}^*$ ,  $i=0, \dots, r-1$  in the allocated rows before it is passed to the next cell. For this mode of operation, each cell should be equipped with  $r$  accumulators to store the partial results corresponding to the  $r$  rows. Note that if  $\lambda$  does not divide  $B$  exactly, then  $r = ((B - 1) \div \lambda) + 1$  rows are allocated to each cell except the last cell that is allocated the last  $B - r(\lambda-1)$  rows. In the remainder of this paper, we will call  $r$  the "degree of folding".

Systolic, self-timed and pseudo systolic cycles for the cells of  $MV_2$  may be easily written for a general degree of folding  $r$ . However, we will only be concerned here with the effect of such folding on the efficiency of pseudo systolic networks when operations involving zeroes are skipped.

Assume, as before, that at most one floating point operation may be executed in each computational cells in a global cycle, and define the  $i^{\text{th}}$  computation front  $CF_i$  as the set of elements of  $A^*$  that are operated upon during the processing phase of the  $i^{\text{th}}$  global pseudo systolic cycle. Then, any two elements  $a_{rk-j, u}^*$  and  $a_{rq-p, v}^*$ ,  $1 \leq k, q \leq \lambda$ ,  $0 \leq j, p \leq r-1$ , in the same computation front should reside in two different cells, that is  $k \neq q$ . Moreover, the application of conditions C1 and C2 of Section 3.1 shows that if the slope  $s$  of the line joining these two elements is defined by

$$s = \frac{k - q}{u - v}$$

then  $s$  should satisfy the same condition (7). Given this, consecutive wave fronts may be constructed and the number of global cycles may be estimated. Moreover, the communication time  $\Delta_i$  for each global cycle  $i$  may be estimated using the same concept of data profile discussed in the last section.

#### 4. Numerical experiments

In order to test the effect of data interlock on the pseudo systolic version of the network  $MV_2$ , we wrote a program that constructs the computation fronts and the data profiles for any given matrix  $A$ , assuming a specific number of buffers  $b$  and degree of folding  $r$ .

Besides the number of global cycles  $N_2$  and the total number of communication subcycles  $\sigma$ , the program also computes the utilization of the network  $\mu$  defined by

$$\mu = \frac{1}{N_2} \sum_{i=1}^{N_2} \frac{M_i}{\lambda}$$

where  $\lambda$  is the number of cells in the network and  $\frac{M_i}{\lambda}$  is the percentage of cells that are not idle during the processing phase of the  $i^{\text{th}}$  global cycles.

The value of  $\mu$  may be a good measure for the efficiency of the pseudo systolic network for sparse matrix computation. It also gives a lower bound on the efficiency of the self timed network in which trivial operations are skipped. However, in order to measure the gain obtained by skipping trivial operations, we may compute the relative speed up  $\pi$  defined by

$$\pi = \frac{T_{self}(MV_2)}{T_{self/skip}(MV_2)}$$

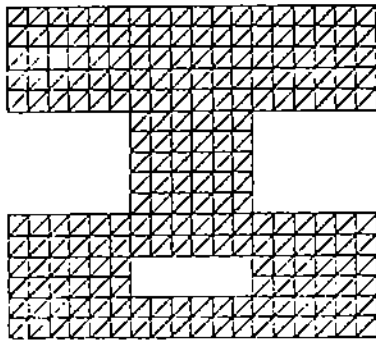
From (5) and (9) we get

$$\pi \geq \frac{T_{self}(MV_2)}{T_{pseudo}(MV_2)} \geq \frac{N_s (\tau_m + \tau_{c, self})}{N_2 \tau_m + \sigma \tau_{c, self}} = \frac{\pi_m \pi_c}{\pi_m \rho_c + \pi_c (1 - \rho_c)} \quad (10)$$

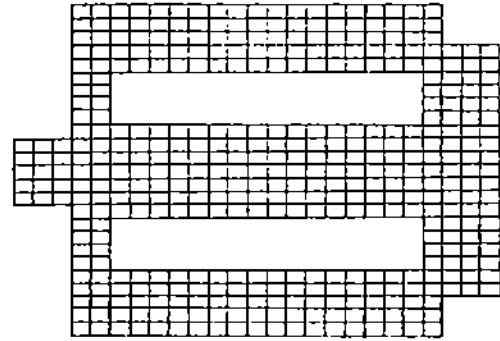
where  $\rho_c = \frac{\tau_{c, self}}{\tau_m + \tau_{c, self}}$  is the relative cost of communication in a read/compute/write cycle,  $N_s = r (B_h + \beta B)$  is the number of cycles in the systolic (or self timed) computation,  $r$  is the degree of folding and  $\pi_m = \frac{N_s}{N_2}$  and  $\pi_c = \frac{N_s}{\sigma}$  are the costs of actual processing and communication, respectively, in the self timed network relative to the corresponding costs in the pseudo systolic network.

We analyzed the performance of the network  $MV_2$  for many specific sparse matrices that result from the application of the finite element analysis to boundary value problems. However, due to space limitations, we report here the results of the analysis for only five of these matrices.

First, we consider the regular grid that covers the domain shown in Figure 6a. We assume that each one of the 432 triangular elements of the grid has three nodes located at its corners and, then, we generate the stiffness matrix by assembling these elements. Given that the bandwidth and the profile of the resulting  $270 \times 270$  matrix depend on the method used to number the nodes of the grid, we generate two matrices  $A_1$  and  $A_2$  from two different numbering schemes. More specifically,  $A_1$  is obtained by numbering the nodes row-wise in a regular way, and  $A_2$  is obtained by applying the Cuthill-McKee scheme [8] starting from the node at the upper left corner of the grid. The bandwidths of  $A_1$  and  $A_2$  are 39 and 79, respectively, and only 16% of the elements in the band of  $A_1$  and 7% of the elements in the band of  $A_2$  are non zero elements. In other words, the time for the multiplication of  $A_1$  and  $A_2$  by a vector on self timed computational arrays may be improved by up to  $\pi_{m, max} = \frac{100}{16} \approx 6.36$  and  $\frac{100}{7} \approx 14.93$ , respectively, if trivial operations are skipped.



(a) The grid used to generate  $A_1$  and  $A_2$



(b) The grid used to generate  $A_5$ .

Figure 6

In the second example, we consider a three dimensional  $7 \times 7 \times 7$  grid that covers a cube with 343 elements, each having 8 nodes located at its corners. Again, a regular plane-wise/row-wise numbering is used to obtain a matrix  $A_3$  and a Cuthill-McKee scheme is used to obtain another matrix  $A_4$ . Both matrices are of order 512 and their bandwidths are 147 and 341, respectively. The percentages of non zero elements in the bands of  $A_3$  and  $A_4$  are 12% and 5%, respectively, that is  $\pi_{m,max} \approx 8.12$  and 21.76, respectively.

Finally, we consider the domain shown in Figure 6b. We cover this domain by a grid that contains 402 quadrilateral, 9-node elements (Lagrange elements), and we use a regular, column-wise, numbering to label the 1780 nodes in the grid. The resulting matrix  $A_5$  is of order 1780. Its bandwidth is 209, with only 7% non zero elements in the band. That is  $\pi_{m,max} \approx 14.88$ .

The results of the analysis are shown in Tables 1,2 and 3, where the values of the utilization  $\mu$ , the number of computation fronts  $N_2$ , the speed up in processing time  $\pi_m$  and the slow down in communication time  $\frac{1}{\pi_c}$  are reported for different degrees of folding  $r$  and number of buffers  $b$ . Following are some comments on the results:



|     |     | Results for $A_1$<br>$B=39, \zeta=0.16, \pi_{n,max}=6.36$ |       |       |         |                   | Results for $A_2$<br>$B=79, \zeta=0.07, \pi_{n,max}=14.93$ |       |       |         |                   |
|-----|-----|---|-------|-------|---------|-------------------|--|-------|-------|---------|-------------------|
| $b$ | $r$ | $\lambda$   | $\mu$ | $N_2$ | $\pi_m$ | $\frac{1}{\pi_c}$ | $\lambda$  | $\mu$ | $N_2$ | $\pi_m$ | $\frac{1}{\pi_c}$ |
| 1   | 1   | 39  | .715  | 60    | 4.867   | 1.455             | 79   | .399  | 59    | 6.017   | 1.901             |
| 2   | 1   | 39  | .740  | 58    | 5.034   | 1.630             | 79   | .543  | 39    | 9.103   | 1.544             |
| 3   | 1   | 39  | .795  | 54    | 5.407   | 1.682             | 79   | .543  | 39    | 9.103   | 1.592             |
| 4   | 1   | 39  | .825  | 52    | 5.615   | 1.747             | 79   | .543  | 39    | 9.103   | 1.696             |
| 1   | 2   | 20  | .266  | 315   | 1.854   | 7.017             | 40   | .142  | 295   | 2.407   | 6.361             |
| 2   | 2   | 20  | .709  | 118   | 4.949   | 1.579             | 40   | .414  | 101   | 7.030   | 1.910             |
| 3   | 2   | 20  | .709  | 118   | 4.949   | 1.733             | 40   | .606  | 69    | 10.29   | 1.468             |
| 4   | 2   | 20  | .775  | 108   | 5.407   | 1.760             | 40   | .663  | 63    | 11.27   | 1.451             |
| 5   | 2   | 20  | .782  | 107   | 5.458   | 1.860             | 40   | .663  | 63    | 11.27   | 1.485             |
| 3   | 4   | 10  | .487  | 344   | 3.395   | 2.442             | 20   | .320  | 261   | 5.441   | 1.882             |
| 4   | 4   | 10  | .722  | 232   | 5.034   | 1.548             | 20   | .557  | 150   | 9.467   | 1.425             |
| 5   | 4   | 10  | .741  | 226   | 5.168   | 1.716             | 20   | .674  | 124   | 11.45   | 1.211             |
| 6   | 4   | 10  | .764  | 219   | 5.333   | 1.644             | 20   | .697  | 120   | 11.83   | 1.366             |
| 7   | 4   | 10  | .782  | 214   | 5.458   | 1.705             | 20   | .697  | 120   | 11.83   | 1.417             |
| 5   | 6   | 7   | .616  | 388   | 4.515   | 1.435             | 14   | .454  | 263   | 8.099   | 1.231             |
| 6   | 6   | 7   | .712  | 336   | 5.214   | 1.500             | 14   | .597  | 200   | 10.65   | 1.425             |
| 7   | 6   | 7   | .725  | 330   | 5.309   | 1.620             | 14   | .675  | 177   | 12.03   | 1.344             |
| 8   | 6   | 7   | .747  | 320   | 5.475   | 1.592             | 14   | .694  | 172   | 12.38   | 1.361             |
| 9   | 6   | 7   | .774  | 309   | 5.670   | 1.517             | 14   | .703  | 170   | 12.53   | 1.420             |
| 7   | 8   | 5   | .759  | 441   | 5.297   | 1.476             | 10   | .559  | 299   | 9.498   | 1.166             |
| 8   | 8   | 5   | .805  | 416   | 5.615   | 1.784             | 10   | .663  | 252   | 11.27   | 1.307             |
| 9   | 8   | 5   | .813  | 412   | 5.670   | 1.846             | 10   | .721  | 232   | 12.24   | 1.338             |
| 10  | 8   | 5   | .823  | 407   | 5.740   | 1.866             | 10   | .743  | 225   | 12.62   | 1.285             |
| 11  | 8   | 5   | .835  | 401   | 5.825   | 1.825             | 10   | .757  | 221   | 12.85   | 1.358             |
| 9   | 10  | 4   | .825  | 507   | 5.759   | 1.479             | 8  | .624  | 335   | 10.60   | 1.149             |
| 10  | 10  | 4   | .854  | 490   | 5.959   | 1.651             | 8  | .697  | 300   | 11.83   | 1.265             |
| 11  | 10  | 4   | .861  | 486   | 6.008   | 1.651             | 8  | .733  | 285   | 12.46   | 1.296             |
| 12  | 10  | 4   | .868  | 482   | 6.058   | 1.740             | 8  | .763  | 274   | 12.96   | 1.358             |
| 13  | 10  | 4   | .876  | 478   | 6.109   | 1.842             | 8  | .777  | 269   | 13.20   | 1.473             |

Table 1 - Results of the analysis for  $A_1$  and  $A_2$

|     |     | Results for $A_3$<br>$B=147, \zeta=0.12, \pi_{n,max}=8.12$ |       |       |         |                   | Results for $A_4$<br>$B=341, \zeta=0.05, \pi_{n,max}=21.76$ |       |       |         |                   |
|-----|-----|--|-------|-------|---------|-------------------|---|-------|-------|---------|-------------------|
| $b$ | $r$ | $\lambda$  | $\mu$ | $N_2$ | $\pi_m$ | $\frac{1}{\pi_c}$ | $\lambda$   | $\mu$ | $N_2$ | $\pi_m$ | $\frac{1}{\pi_c}$ |
| 1   | 1   | 147  | .690  | 105   | 6.295   | 1.251             | 341   | .157  | 200   | 4.260   | 6.276             |
| 2   | 1   | 147  | .690  | 105   | 6.295   | 1.271             | 341   | .245  | 128   | 6.656   | 2.924             |
| 3   | 1   | 147  | .690  | 105   | 6.295   | 1.352             | 341   | .255  | 123   | 6.927   | 3.036             |
| 4   | 1   | 147  | .690  | 105   | 6.295   | 1.352             | 341   | .255  | 123   | 6.927   | 3.258             |
| 1   | 2   | 74   | .234  | 614   | 2.153   | 13.67             | 171   | .098  | 636   | 2.679   | 11.56             |
| 2   | 2   | 74   | .685  | 210   | 6.295   | 1.475             | 171   | .195  | 321   | 5.308   | 5.717             |
| 3   | 2   | 74   | .685  | 210   | 6.295   | 1.575             | 171   | .322  | 194   | 8.784   | 2.559             |
| 4   | 2   | 74   | .685  | 210   | 6.295   | 1.634             | 171   | .347  | 180   | 9.467   | 2.581             |
| 5   | 2   | 74   | .685  | 210   | 6.295   | 1.667             | 171   | .351  | 178   | 9.573   | 2.461             |
| 3   | 4   | 37   | .412  | 698   | 3.788   | 4.799             | 86  | .184  | 675   | 5.049   | 4.450             |
| 4   | 4   | 37   | .685  | 420   | 6.295   | 1.865             | 86  | .255  | 487   | 6.998   | 3.539             |
| 5   | 4   | 37   | .707  | 407   | 6.496   | 1.534             | 86  | .336  | 370   | 9.211   | 2.345             |
| 6   | 4   | 37   | .707  | 407   | 6.496   | 1.611             | 86  | .378  | 329   | 10.36   | 2.350             |
| 7   | 4   | 37   | .711  | 405   | 6.528   | 1.605             | 86  | .398  | 312   | 10.92   | 2.581             |
| 7   | 8   | 19   | .609  | 920   | 5.748   | 2.224             | 43  | .311  | 798   | 8.541   | 2.209             |
| 8   | 8   | 19   | .732  | 766   | 6.903   | 1.239             | 43  | .349  | 713   | 9.560   | 2.072             |
| 9   | 8   | 19   | .732  | 766   | 6.903   | 1.530             | 43  | .382  | 650   | 10.49   | 1.934             |
| 10  | 8   | 19   | .732  | 766   | 6.903   | 1.579             | 43  | .423  | 587   | 11.61   | 1.907             |
| 11  | 8   | 19   | .732  | 766   | 6.903   | 1.573             | 43  | .452  | 550   | 12.39   | 1.951             |
| 14  | 15  | 10   | .713  | 1494  | 6.637   | 2.042             | 23  | .396  | 1173  | 10.89   | 1.892             |
| 15  | 15  | 10   | .752  | 1416  | 7.002   | 2.159             | 23  | .418  | 1111  | 11.50   | 1.865             |
| 16  | 15  | 10   | .759  | 1403  | 7.067   | 2.372             | 23  | .438  | 1061  | 12.05   | 1.817             |
| 17  | 15  | 10   | .759  | 1402  | 7.072   | 2.337             | 23  | .460  | 1011  | 12.64   | 1.918             |
| 18  | 15  | 10   | .759  | 1402  | 7.072   | 2.442             | 23  | .483  | 962   | 13.28   | 1.904             |

Table 2 - Results of the analysis for  $A_3$  and  $A_4$

1)  $\pi_m > 1$  and  $\pi_c < 1$ , for any  $r$  and  $\lambda$ . That is, by skipping trivial operations, the time for arithmetic computation decreases and the time for data communication increases. The actual speed up ratio depends on the relative values of  $\tau_m$  and  $\tau_{c, self}$  as given by equation (10). For example, if  $MV_2$  is emulated on a system like the Pringle [13], where  $\tau_m \approx \tau_{c, self}$  (all data communication share one pipelined communication channel), then large speed ups should not be expected. On the other hand, if dedicated links are used for data transmission between neighboring cells, then  $\tau_m \gg \tau_c$  and large speed ups may be obtained. Example of this type of machines are the wave front machine [16] and the CHiP system [22].

2) It is very inefficient to use  $b < r$ . The reason for this inefficiency may be clarified by an example: Consider a diagonal matrix  $diag(a_{1,1}, \dots, a_{8,8})$  and a network with  $\lambda = 4$  cells, that is  $r = 2$ . If  $b=2$ , then it is clear that the matrix may be processed in two global cycles. More specifically, the computation fronts are  $CF_1 = \{a_{1,1}, a_{3,3}, a_{5,5}, a_{7,7}\}$  and  $CF_2 = \{a_{2,2}, a_{4,4}, a_{6,6}, a_{8,8}\}$ . On the other hand, if  $b=1$  and  $x_1$  is at cell 1 during a specific cycle, then  $x_3$  may not be at cell 2 during the same cycle because there are no buffers to store  $x_2$  between cells 1 and 2. Due to this type of data interlock, five cycles are needed to complete the computation and the corresponding fronts are:  $\{a_{1,1}\}$ ,  $\{a_{2,2}, a_{3,3}\}$ ,  $\{a_{4,4}, a_{5,5}\}$ ,  $\{a_{6,6}, a_{7,7}\}$  and  $\{a_{8,8}\}$ .

3) Given a specific  $r$ , any increase in  $b$  (up to a certain limit) results in a larger  $\mu$ , that is a better performance: In Figure 7a, we plot the values of  $\mu$  versus  $b$  for  $A_5$  and  $r=8$ . It may be seen that, for  $b \geq r$ , the best improvement in performance occurs when  $b$  is changed from  $r$  to  $r+1$ . Hence, if we consider the "performance improvement per additional buffer" as an optimality criteria, then  $b = r+1$  gives the optimal performance. This curve is typical in all the examples that we studied, with the exception that, in some cases, the optimal performance is obtained at

$b = r + 2$  instead of  $b = r + 1$ .

4) Better performance is obtained at higher degrees of folding. That is to say, partitioning of the computation improves the performance. This is illustrated in Figure 7b where we fix  $b = r + 1$  and we plot  $\mu$  and  $\frac{\pi_m}{\pi_{m,max}}$  versus  $r$ , for  $A_5$ . Note that both  $\mu$  and  $\frac{\pi_m}{\pi_{m,max}}$  approach unity as  $r$  approaches  $B$  ( $\lambda = 1$ ). Note also that  $\mu$  is not monotonically increasing. For example,  $\mu = 0.653$  and  $0.635$  at  $r = 11$  and  $12$ , respectively. The reason for this is obvious; For  $r = 11$ , nineteen cells are used and each cell operates on the elements of exactly 11 rows. On the other hand, for  $r = 12$ , eighteen cells are used with the last cell operating on only 5 rows. In other words, the utilization of the last cell may not exceed  $\frac{5}{12}$ , which reduces the average utilization of the network.

|     |     | results for $A_5$                      |       |       |         |                   |
|-----|-----|--|-------|-------|---------|-------------------|
|     |     | $B=209, \zeta=0.07, \pi_{m,max}=14.88$ |       |       |         |                   |
| $b$ | $r$ | $\lambda$                              | $\mu$ | $N_2$ | $\pi_m$ | $\frac{1}{\pi_c}$ |
| 1   | 1   | 209                                    | .445  | 284   | 6.989   | 1.786             |
| 2   | 1   | 209                                    | .504  | 251   | 7.908   | 1.811             |
| 3   | 1   | 209                                    | .514  | 246   | 8.069   | 1.940             |
| 1   | 2   | 105                                    | .126  | 2004  | 1.981   | 15.87             |
| 2   | 2   | 105                                    | .506  | 497   | 7.988   | 3.243             |
| 3   | 2   | 105                                    | .596  | 422   | 9.408   | 1.992             |
| 4   | 2   | 105                                    | .626  | 402   | 9.876   | 2.063             |
| 3   | 4   | 53                                     | .232  | 2153  | 3.688   | 6.817             |
| 4   | 4   | 53                                     | .529  | 943   | 8.420   | 3.601             |
| 5   | 4   | 53                                     | .584  | 853   | 9.308   | 2.095             |
| 6   | 4   | 53                                     | .615  | 811   | 9.790   | 1.954             |
| 7   | 8   | 27                                     | .442  | 2215  | 7.169   | 2.892             |
| 8   | 8   | 27                                     | .577  | 1695  | 9.369   | 3.077             |
| 9   | 8   | 27                                     | .617  | 1586  | 10.01   | 2.325             |
| 10  | 8   | 27                                     | .623  | 1571  | 10.11   | 2.345             |
| 13  | 14  | 15                                     | .611  | 2881  | 9.646   | 2.687             |
| 14  | 14  | 15                                     | .654  | 2692  | 10.32   | 2.911             |
| 15  | 14  | 15                                     | .672  | 2621  | 10.60   | 2.624             |
| 16  | 14  | 15                                     | .677  | 2600  | 10.68   | 2.472             |
| 19* | 20  | 11                                     | .656  | 3659  | 10.85   | 3.075             |
| 20  | 20  | 11                                     | .666  | 3605  | 11.01   | 2.975             |
| 21  | 20  | 11                                     | .672  | 3575  | 11.11   | 2.894             |
| 22  | 20  | 11                                     | .673  | 3571  | 11.12   | 2.751             |

Table 3 - The results for  $A_5$

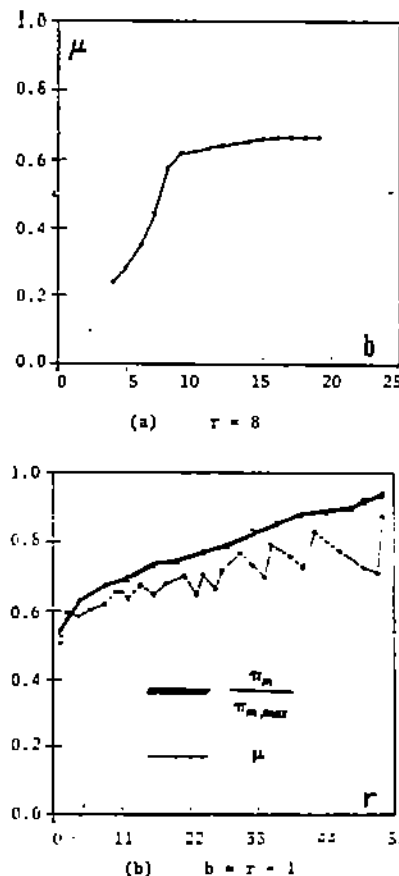


Fig 7 - Performance of  $MV_2$  for  $A_5$

5) The range of variation of  $\mu$ ,  $\pi_m$  and  $\pi_c$  with  $b$  and  $r$  depends on the method used to number the nodes of the finite element grid. More specifically, if a regular numbering is used, the efficiency of the network is relatively high, but only slight improvement in performance is obtained by increasing  $r$  and  $b$ . On the other hand, if a non regular numbering is used, as for example the Cuthill-McKee scheme, then the efficiency is relatively low for  $r = b = 1$ , but improves noticeably at higher  $b$  and  $r$ . The reason for this is that the structure of the matrix is more regular in the first case than in the second, and a well structured matrix, where the elements are clustered around few off diagonals is particularly suited for the propagation of the type of computation fronts encountered in  $MV_2$ . However, this does not leave too much room for improvement as  $\mu$  and  $\pi_m$  may not exceed their limits, namely 1 and  $\pi_{m,max}$ , respectively.

Finally, we note that the pattern of behavior described above was obtained consistently in all the other examples that we used to test  $MV_2$ . Hence, we are led to believe that this behavior is typical for the type of matrices that result in finite element analysis.

## 5. Conclusion

We suggested a technique for the estimation of lower bounds on the efficiency of self timed computational arrays. Although this technique is quite general, it was applied in this paper to specific networks for the multiplication of a sparse matrix by a vector. The propagation of the computation fronts in such networks is restricted by some conditions that are necessary for the consistency of data flow. The study of these restrictions was shown to be crucial for the choice of networks that are suitable for special types of matrices. For example, networks that do not allow computation fronts to be parallel to the diagonal of the matrix are expected to perform poorly on matrices with non zero diagonal elements.

The network presented in Section 3 for sparse matrices with non zero diagonal elements was extensively tested using many examples drawn from finite element analysis. The experimental results showed that the efficiency and utilization of the network are, in general, satisfactory. Moreover, if the size of the network is small with respect to the given matrix, and the computation is partitioned such that each cell operates on more than one row of the matrix, then the effect of data interlock is reduced, thus improving the efficiency of the network. The results also showed that the number of buffers on the communication links has a major effect on the efficiency of the network. In particular, the efficiency deteriorates severely when the number of buffers  $b$  is decreased below the degree of folding  $r$ . These results may be easily extended to the computation of the product of a matrix by more than one vector, and to the product of two matrices. The extension of the evaluation technique to more complex networks, (e.g. networks for matrix factorizations) seems possible, but requires further study.

Finally, we note that our approach for the analysis of self timed networks measures the effect of data interlock on computations without any assumption about the technology used for the implementation of the networks. More specifically, our results are independent of the parameters  $\tau_m$  and  $\tau_{c\_self}$ , that depend strongly on the architecture and technology. This type of results may not be obtained by the straight forward simulation of self timed computations.

#### References

1. L. Adams and R. Voigt, "Design, Development and Use of the Finite Element Machine," *ICASE report 172250, NASA-Langley Research Center*, Oct. 1983.
2. H. Ahmed, J. Delosme, and M. Morf, "Highly Concurrent Computing Structures for Matrix Arithmetic and Signal Processing," *Computer*, Jan. 1982.
3. H. Amano, T. Yoshida, and H. Aiso, "(SM)2 : Sparse Matrix Solving

- Machine," *Proc. of the 1983 international Conference on Parallel Processing*, 1983.
4. C. P. Arnold, M. I. Parr, and M. B. Dewe, "An Efficient Parallel Algorithm for the solution of Large Sparse Linear Matrix Equations," *IEEE Trans. on Computer*, vol. C-32, pp. 265-272, 1983.
  5. S. H. Bokhari, "On the Mapping Problem," *Proc. of the 1979 International Conference on Parallel Processing*, 1979.
  6. A. Chen and C. Wu, "Optimum Solution to Dense Linear Systems of Equations," *Proc. of the 1984 International Conference on Parallel Processing*, pp. 417-424.
  7. M. C. Chen and C. A. Mead, "Formal Specification of Concurrent Systems," *USC Workshop on VLSI and Modern Signal Processing*, Nov. 1982.
  8. E. Cuthill and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices," *Proc. of ACM national conference, New York*, pp. 157-172, 1969.
  9. S. Eisenstat, M. Gursky, M. Schultz, and A. Sherman, "Yale Sparse Matrix Package," Tec. Reports 112,114, Computer Science, Yale University, 1977.
  10. A. George and J. Liu, "Computer Solutions of Large Sparse Positive Definite Systems," *Prentice-Hall Series in Computational Math.*, 1981.
  11. L. Johnsson, "Computational Arrays for Band Matrix Equations," Tec. Report. #4287:TR:81, California Institute of Technology, 1981.
  12. H. F. Jordan, "A Multiprocessor System for Finite Element Structural Analysis," *Computer and Structures*, vol. 10, pp. 21-29, 1979.
  13. A. Kapauan, K. Wang, D. Gannon, J. Cuny, and L. Snyder, "The Pringle: An Experimental System for Parallel Algorithm and Software Testing," *Proc. of the 1984 International Conference on Parallel Processing*, pp. 1-6.

14. H. T. Kung and C. E. Leiserson, "Systolic Arrays for VLSI," in *Introduction to VLSI Systems*, ed. Mead C. and Conway L., Addison-Wesley, Reading - Mass., 1980.
15. H. T. Kung, "Why Systolic Architecture," *Computer Magazine*, pp. 37-46, Jan. 1982.
16. S. Y. Kung, K. S. Arun, R. J. Gab-ezer, and B. Rao, "Wavefront Array Processor, Language, Architecture and Applications," *IEEE Trans. on Computer*, vol. C-31, pp. 1054-1066, 1982.
17. R. G. Melhem, "Formal Analysis of a Systolic System for Finite Element Stiffness Matrices," Technical report ICMA-83-56, The University of Pittsburgh, May 1983. (Accepted for publication in the *Journal of Computer and System Sciences*)
18. R. G. Melhem, "Verification of a Class of Deadlock-Free, Self Timed, Computational Arrays," Report ICMA-84-76, The University of Pittsburgh, Sept 1984.
19. R. G. Melhem and W. C. Rheinboldt, "A Mathematical Model for the Verification of Systolic Networks," *SIAM J. on Computing.*, vol. 13, no. 3, pp. 541-565, Aug. 1984.
20. D. A. Reed and M. L. Patrick, "A Model of Asynchronous Iterative Algorithms for Solving Large, Sparse, Linear Systems," *Proc. of the 1984 International Conference on Parallel Processing*, pp. 402-409.
21. C. L. Seitz, "System Timing," in *Introduction to VLSI Systems.*, ed. C. Mead and L. Conway, Addison-Wesley, Reading - Mass., 1980.
22. L. Snyder, "Introduction to the Configurable Highly Parallel Computer," *Computer Magazine*, vol. 15, no. 1, pp. 47-56, Jan 1982.
23. U. Weiser and A. Davis, "A Wavefront Notation Tool for VLSI Array Design," in *VLSI Systems and Computations*, ed. H. T. Kung, B. Sproull and

G. Steele, pp. 226-234, Computer Science Press, 1981.

24. O. C. Zienkiewicz, *The Finite Element Method*, McGraw-Hill, 1979. Third edition.