Parallel Computing 6 (1988) 165–184
North-Holland

Parallel solution of linear systems with striped sparse matrices *

Rami MELHEM

Department of Mathematics and Statistics, University of Pittsburgh, Pittsburgh, PA 15260, U.S.A.

Received November 1986

Abstract. The multiplication of a vector by a matrix and the solution of triangular linear systems are the most demanding operations in the majority of iterative techniques for the solution of linear systems. Data-driven VLSI networks which perform these two operations, efficiently, for certain sparse matrices are introduced. In order to avoid computations that involve zero operands, the non-zero elements in a sparse matrix are organized in the form of non-overlapping stripes, and only the elements within the stripe structure of the matrix are manipulated. Detailed analysis of the networks proves that both operations may be completed in n global cycles with minimal communication overhead, where n is the order of the linear system. The number of cells in each network as well as the communication overhead, are determined by the stripe structure of the matrix. Different stripe structures for the class of sparse matrices generated in Finite Element Analysis are examined in a separate paper.

Keywords. Linear algebra, sparse linear systems, VLSI networks, stripe structure of a sparse matrix.

1. Introduction

Iterative solvers for large sparse linear systems of equations are, once again, becoming more and more competitive with direct solvers [7,8]. In this paper, we consider the two basic operations that constitute the bulk of the work in most iterative methods: namely, the multiplication of a vector by a matrix, and the solution of triangular linear systems. The computations involved in these two operations are quite regular and thus, naturally amenable to efficient implementation on regular VLSI networks such as systolic [10] and data-driven (sometimes called self-timed) [11] arrays.

Large systems that appear in practice are usually sparse, and hence, their solution on regular VLSI networks would seem to be inefficient. More specifically, if $\zeta\%$ of the elements in the coefficient matrix are zeroes, then $\zeta\%$ of the resources in the network are wasted in trivial operations that involve zero operands, and the corresponding data communication.

In order to avoid this waste and to retain the advantage of having fast specialized cells and efficient local communication, it is suggested in [14] to consider regular data-driven networks that are designed for operations on dense matrices, and then to add to each cell in the networks the capability of recognizing and skipping trivial operations. A detailed study of the resulting networks shows large speedups for highly sparse matrices.

In this paper, we present a different approach for using regular data-driven networks in sparse matrix manipulation. It is also based on performing only non-trivial operations, but is

* This work is, in part, supported under ONR contract N00014-85-K-0339.

primarily aimed at reducing the number of computational cells in the network, rather than increasing its speed.

In order to be more specific, consider, for example, the multiplication of a vector by an $n \times n$ banded matrix A. This operation may be performed in $2n + b$ cycles on a data-driven (or systolic) network [10] that uses b cells, where b is the bandwidth of A. If A is sparse within the band, then the approach of [14] uses the same number of cells b, but takes advantage of the sparsity of A to reduce the multiplication time considerably. On the other hand, the approach presented here takes advantage of the sparsity of A to reduce the number of cells to a number π, which is, usually, much smaller than b.

The reduction of the number of computational cells is based on the assumption that the non-zero elements of the matrix are located in a few stripes which are almost parallel to the diagonal of the matrix. A specific cell is then assigned to perform the operations associated with the elements of a particular stripe. Clearly, the crucial issue is to choose a stripe structure that minimizes, or even eliminates, data conflict.

The network that we introduce here is especially suitable for the type of matrices that arise in finite element analysis. More specifically, the bandwidth depends on the order of the matrix (usually, $b = O(\sqrt{n})$ and $O(n^{2/3})$, for 2-D and 3-D problems, respectively), while the number of stripes π is bounded by a small number which depends on the maximum number of elements that may share a particular node. In other words, in contrast to the approach in [14], the size of the network is independent of the size of the problem.

At this point, we should mention that the algorithm which is applied in this paper for the manipulation of sparse matrices is a generalization of a technique suggested by Madsen, Rodrigue and Karush [12] for matrix multiplication by diagonals on vector computers. This latter technique, however, applies only if the non-zero elements of the matrix are located in a few off-diagonals. We should also mention that other approaches have been suggested for the parallel solution of general sparse linear systems. These include the application of content-addressable VLSI networks [21], and data-flow architectures [17,18] to minimize conflicts in data access, the use of networks with interconnections that reflect the underlying graph structure of the matrix [1,2], and the use of multiprocessors with general interconnections [3,4].

We start in Section 2 by defining the stripe structure of a general sparse matrix. In Section 3, we describe a VLSI network that utilizes the stripe structure in the parallel multiplication of a vector by a matrix. Then, we introduce, in Section 4, the property of non-overlapping stripes and we show that, if the input matrix has this property, then the multiplication does terminate in n global cycles, where a global cycle includes some communication activities, and a multiply/add operation. In Section 5, we estimate the communication overhead in each global cycle, and finally, in Section 6, we modify the matrix/vector multiplication network and obtain a network for the solution of triangular linear systems.

2. Stripe structures of sparse matrices

We define a stripe structure of a sparse matrix to be a set of stripes that are almost parallel to the diagonal of the matrix, and that contain all its non-zero elements. More specifically, given an $n \times n$ matrix A, with lower and upper bandwidths b_1 and b_2, respectively, we augment the set $T = \{(i, j): 1 \leqslant i, j \leqslant n\}$ of positions of A with the two triangles $T_1 = \{(i, j): i = 1, \ldots, b_1, j = i - b_1, \ldots, 0\}$ and $T_2 = \{(i, j): i = n - b_2 + 1, \ldots, n, j = n + 1, \ldots, i + b_2\}$, and assume that $a_{i,j} = 0$ for $(i, j) \in T_1 \cup T_2$. This expands the set of allowable positions of A to include the band $\{(i, j): 1 \leqslant i \leqslant n, i - b_1 \leqslant j \leqslant i + b_2\}$. Now we may define the following:

Definition 2.1. Let $I_n = \{1, \ldots, n\}$. A stripe S of the matrix A is a set of positions $S = \{(i, \sigma(i)): i \in I \subseteq I_n\}$, where σ is an increasing function; that is, if $i < j$ and $(i, \sigma(i)), (j, \sigma(j)) \in S$, then $\sigma(i) < \sigma(j)$. If S contains one entry for each row of A, that is $S = \{(i, \sigma(i)): i \in I_n\}$, then S is called a complete stripe.

Definition 2.2. Two stripes $S_1 = \{(i, \sigma_1(i))\}$ and $S_2 = \{(i, \sigma_2(i))\}$ are ordered by the relation $S_1 < S_2$ (S_1 is less than S_2) if for any i in the domain of σ_1, and j in the domain of σ_2,

$$i \leqslant j \quad \text{implies that} \quad \sigma_1(i) < \sigma_2(j).$$

Note that if S_1 and S_2 are complete stripes, then $S_1 < S_2$ if $\sigma_1(i) < \sigma_2(i)$ for $i = 1, \ldots, n$.

Definition 2.3. A stripe structure of a matrix A is a sequence of π stripes $S_1 < \cdots < S_\pi$ such that $a_{i,j} = 0$ if $(i, j) \notin S_1 \cup \cdots \cup S_\pi$ (see Fig. 1(a), where '·' and 'x' indicate a zero and a non-zero element, respectively, and each element included in a stripe is enclosed in a circle).

A special class of stripe structures is the class of structures with parallel stripes, in the sense that each stripe S_k has the form $\{(i, i + s_k): i \in I \subseteq I_n\}$ for some constant s_k. For example, consider the discretization of a partial differential equation on a rectangular grid. If the nodes of the grid are numbered regularly, and a five-point star approximation is used to discretize the differential equation, then the resulting coefficient matrix may be covered by five parallel complete stripes (see Fig. 1(b)). Similarly, if finite element analysis is used with 3-, 4-, 6- or 9-node Lagrangian elements, then the resulting stiffness matrix may be covered by 7, 9, 19 or 25 parallel complete stripes, respectively. Note that in order to obtain complete stripes in the examples of Fig. 1(b), we include in each stripe a few positions (i, j) for which $a_{i,j} = 0$.

Matrices with parallel complete stripes may be efficiently stored diagonal by diagonal [12]. Moreover, by using a data structure similar to the one used in [19], the diagonal storage scheme may be easily extended to matrices with general stripes. More specifically, given a matrix A with π stripes, we may store the elements of the kth stripe in the kth column of an $n \times \pi$ rectangular array E_A, such that, for $i = 1, \ldots, n$ and $k = 1, \ldots, \pi$,

$$E_A(i, k) = \begin{cases} a_{i,\sigma_k(i)} & \text{if } (i, \sigma_k(i)) \in S_k, \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

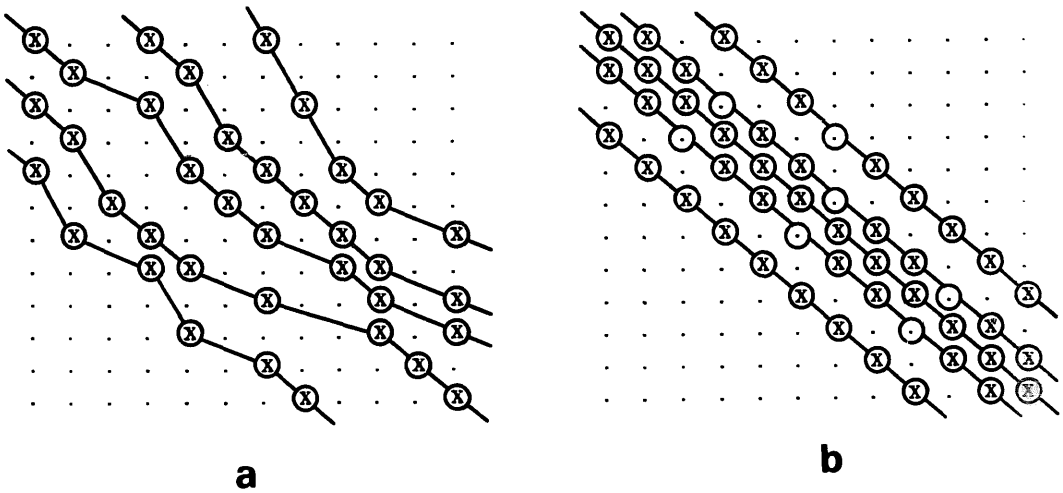

Fig. 1. Examples of stripe structures. (a) With 5 stripes. (b) With 5 complete, parallel, stripes.

In addition to E_A, another $n \times \pi$ integer array P_A is needed to store the values of $\sigma_k(i)$. In other words,

$$P_A(i, k) = \begin{cases} \sigma_k(i) & \text{if } (i, \sigma_k(i)) \in S_k, \\ -b_1 & \text{otherwise.} \end{cases} \tag{2}$$

Note that $-b_1$ is not a valid column number in $T \cup T_1 \cup T_2$. Note also that a more efficient scheme for storing incomplete stripes of A might be obtained if we compact every column k of E_A and P_A by storing only the entries corresponding to the elements of S_k, and then keep the compacted columns of E_A and P_A in two linear arrays. Of course, an additional array is needed in this case in order to keep track of the row number of each element in E_A.

Clearly, many stripe structures may be constructed for a given sparse matrix. Among the different structures for any banded matrix is the structure obtained by considering b parallel stripes, where $b = b_1 + b_2 + 1$ is the bandwidth. However, in order to take full advantage of the above stripe storage scheme, we should determine the stripe structure of the matrix that minimizes the number of stripes (see the appendix).

The efficiency of the stripe storage scheme for any $n \times n$ matrix A striped with π stripes is given by the ratio $r_A/n\pi$, where r_A is the number of non-zero elements in A. Although this ratio may be low for general sparse matrices, it is shown in [16] that the stripe scheme is very efficient for the type of matrices resulting from the discretization of partial differential equations. Moreover, the stripe scheme has an important advantage over other sparse schemes [9], namely, it is a regular scheme that may be exploited efficiently in parallel processing.

3. Multiplication of a striped matrix by a vector

In this section, an asynchronous computational network is suggested for the multiplication of a vector by a sparse matrix, assuming that a stripe structure of the matrix is given. The condition that stripes should be strictly increasing is used to prove that the computation may not reach a dead-lock state and that the network will indeed compute the matrix/vector product.

The actual performance of the network is very hard to estimate due to the asynchronous nature of the computation. For this reason, a worst-case analysis technique is used in Section 3.3 to predict the worst possible performance of the network for any specific input. The basic idea in the analysis technique is to assume a hypothetical mode of execution in which communication and computation take place in different phases. With this, a wave front model is used to express the progress of execution and to estimate, safely, the execution time.

3.1. A VLSI data-driven network

The systolic network given in [10] for the multiplication of a vector x by a banded matrix A uses b cells and completes the computation of the product vector $y = Ax$ in $2n + b$ cycles, where n and b are the order and bandwidth, respectively, of A. In this section, we modify this network such that if A has π stripes, $\pi \ll b$, then the vector y may be computed using a network of only π computational cells. In order to be consistent with our future notation, we denote the stripes of A by S_k, $k = -\pi_1, \ldots, \pi_2$, where $\pi_1 + \pi_2 + 1 = \pi$.

In Fig. 2, we show the modified network which we call from now on MAT/VEC. Each cell in MAT/VEC has five input ports, namely I_r, $r = 1, \ldots, 5$, and two output ports, namely O_1 and O_2. The elements of the vector x are fed to the network from port I_1 of cell π_2 and the elements of the result-vector y, initialized to zero, are fed from port I_2 of cell $-\pi_1$. Successive

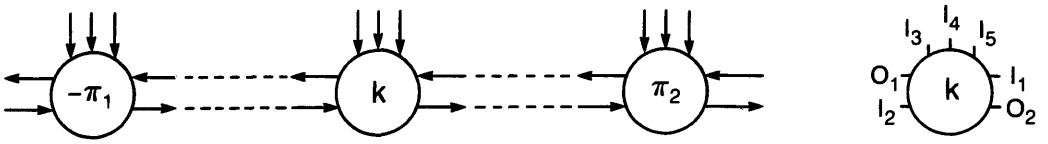

Fig. 2. A network (MAT/VEC) for striped matrix/vector multiplication.

non-zero elements in a particular stripe S_k are supplied on port I_3 of cell k in increasing row order. Along with each element $a_{i,\sigma_k(i)} \in S_k$ supplied on I_3 of cell k, the values of i and $\sigma_k(i)$ are supplied on the input ports I_5 and I_4, respectively. Note that the row index supplied on I_5 may be eliminated if the stripes are complete or if the elements of column k of E_A, defined by (1), rather than the elements of S_k, are supplied to cell k. In this case an internal counter may be used to keep track of the value of i.

Each communication link $l_{q,k}$ directed from cell q to cell k in MAT/VEC is regarded as a queue. Only cell q may write on this queue and only cell k may read (and delete) its front element. For simplicity, we will assume, for now, that the queues have unlimited capacity. Then, we will derive in Section 5 the actual size of the queues needed for proper operation. Note that, in practice, any communication link is just a connector, and hence, any queues associated with the link should be physically located in its source or destination cells (or distributed among the two).

In order to provide the flexibility needed to deal with sparse structures, we assume that each computational cell contains two counters CX and CY to keep track of the indices of the data received on I_1 and I_2, respectively. We also assume that the network is data driven, that is the cycle of each computational cell is controlled by the availability of the input data. Finally, in order to study the effect of internal data conflicts on the operation of the network, we assume that external data, that is data on ports I_3, I_4 and I_5, as well as port I_1 of cell π_2 and port I_2 of cell $-\pi_1$, are always available when needed. With this, the operation of each cell k may be described by the following cycle which is executed repeatedly by the cell: ($[I_r]$ denotes the content of I_r, and $O_r \leftarrow Rx$ means that the value stored in the internal register Rx is written on port O_r).

Cycle 1: / * Initially, CX = CY = 0 * /
Step 1. $Ra = [I_3]$; $Rj = [I_4]$; $Ri = [I_5]$
Step 2. Do Steps 2.1 and 2.2 in parallel
2.1. wait until data is available on I_1;
$Rx = [I_1]$; CX = CX + 1;
If CX < Rj Then {$O_1 \leftarrow Rx$; Go To 2.1}
Else JOIN Step 2.2.
2.2. wait until data is available on I_2;
$Ry = [I_2]$; CY = CY + 1;
If CY < Ri Then {$O_2 \leftarrow Ry$; Go To 2.2}
Else JOIN Step 2.1.
Step 3. $Ry = Ry + Ra * Rx$
Step 4. $O_1 \leftarrow Rx$; $O_2 \leftarrow Ry$.

More descriptively, after a cell k receives $a_{i,\sigma_k(i)}$ (Step 1), it continues to transmit the components of x from I_1 to O_1 (Step 2.1), and the components of y from I_2 to O_2 (Step 2.2), until it finds $x_{\sigma_k(i)}$, and y_i. At this time, the inner product is computed (Step 3), and the results are written out (Step 4). The JOIN statements in Steps 2.1 and 2.2 indicate that Step 3 should not start before both Steps 2.1 and 2.2 are completed. Note that the parallel execution of Step

2.1 and 2.2 guarantees that if either the x or y data stream is blocked, the other stream may continue flowing. The parallel execution may be simulated by a busy wait loop that polls ports I_1 and I_2 for data. More specifically, we may replace Step 2 in CYCLE 1 by

Step 2 While (CX < *Rj*) or (CY < *Ri*) Do
2.1. If (CX < *Rj*) and (data is available on I_1) Then
{*Rx* = [I_1]; CX = CX + 1;
If (CX < *Rj*) Then $O_1 \leftarrow Rx$}
2.2. If (CY < *Ri*) and (data is available on I_2) Then
{*Ry* = [I_2]; CY = CY + 1;
If (CY < *Ri*) Then $O_2 \leftarrow Ry$}

Next, we show that the network described above does compute the elements y_i, $i = 1, \ldots, n$, of the product vector $y = Ax$ correctly if the matrix A has non-intersecting stripes.

3.2. *Proof of correctness*

The following properties of the input will be used:

P1. If $a_{u,\sigma_k(u)}$ and $a_{v,\sigma_k(v)}$, are the inputs to port I_3 of cell k at two consecutive cycles, $t-1$ and t, respectively, then $v > u$.
P2. From P1 and Definition 2.1, we have $\sigma_k(v) > \sigma_k(u)$.
P3. The input matrix A is striped according to Definition 2.3; that is, $S_k < S_q$ if $k < q$.

Let us first assume that the network will not reach a dead state, that is every cycle t of any cell k will terminate. From the operation of each cell (CYCLE 1), and P1, it is clear that every element y_i that is read by cell k (from I_2) during cycle t satisfies $u < i \leqslant v$. If $i = v$, then the term $[a_{i,\sigma_k(i)} x_{\sigma_k(i)}]$ is accumulated in y_i before y_i is written on O_2. On the other hand, if $u < i < v$, then y_i is copied unmodified to O_2. In this case, P1 guarantees that $(i, \sigma_k(i)) \notin S_k$, because otherwise $a_{i,\sigma_k(i)}$ should have been supplied to I_3 after $a_{u,\sigma_k(u)}$ and before $a_{v,\sigma_k(v)}$. Given that any element of A that does not belong to some stripe is equal to zero, we conclude that y_i accumulates all the non-zero terms of $\sum_{j=1}^{n} a_{i,j} x_j$ during its flow from cell $-\pi_1$ to cell π_2.

In order to complete the proof, we need to show that the network does not reach a deadlock state, where a cell k is blocking the y data stream and another cell q, $q > k$, is blocking the x stream (see Fig. 3). More specifically, a state in which

(1) cell q is waiting for some y_l that is locked behind cell k, and
(2) cell k is waiting for some $x_{\sigma_k(v)}$ that is locked behind cell q.

Assume that this deadlock state is reached, and that the data appearing on ports I_1 and I_2 of cells q and k, respectively, are x_j and y_i. Hence,

$$l \geqslant i \quad \text{and} \quad \sigma_k(v) \geqslant j. \tag{3}$$

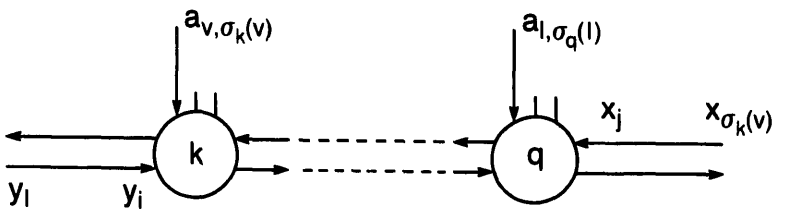

Fig. 3. A deadlock configuration.

The fact that y_i is not copied to port O_2 of cell k and x_j is not copied to port O_1 of cell q implies that $i \geqslant v$ and $j \geqslant \sigma_q(l)$. But $i > v$ may only be satisfied if the previous input on I_3 of cell k, say $a_{u,\sigma_k(u)}$, satisfies $u = i - 1$, which means that $u \geqslant v$ and contradicts P1. Similarly, we may show that $j > \sigma_q(l)$ contradicts P2. Hence

$$i = v \quad \text{and} \quad j = \sigma_q(l). \tag{4}$$

From (3) and (4), we get

$$v \leqslant l \quad \text{and} \quad \sigma_k(v) \geqslant \sigma_q(l)$$

which contradicts Definition 2.2 for $S_k < S_q$, and hence, given that $k < q$, contradicts property P3 of the input.

3.3. *Pseudo-systolic synchronization*

In order to study the behavior and estimate the execution time of the network MAT/VEC, we follow the technique suggested in [14] for the study of self-timed computations. Namely, we introduce a simpler, hypothetical, computation (called a pseudo-systolic computation) that is obtained by forcing some synchronization on the self-timed computation. The additional synchronization may only slow down execution, and hence the execution time of the pseudo-systolic computation forms an upper bound on the execution time of the self-timed computation.

A pseudo-systolic version of the self-timed computation discussed in this section may be obtained by replacing Step 3 in CYCLE 1 by the following:

Step 3. wait for a synchronization signal SYNC;
$Ry = Ry + Ra * Rx$

The purpose of the SYNC signal is the synchronization of all the cells such that the execution of the network alternates between two phases; a communication phase and a processing phase. During the communication phase, the data flows in the network until each cell is either blocked waiting for data (in Steps 2.1 or 2.2), or waiting for SYNC (in Step 3). We assume that all the cells are connected to a hypothetical controller that issues the signal SYNC after it detects the termination of the communication phase. At that instant, all the cells that are waiting in Step 3 perform the multiplication, 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. In this paper, we let N be the total number of global cycles needed to terminate the computation, and we denote by CP_t and PP_t, $t = 1, \ldots, N$, the computation phase and the processing phase, respectively, of the global cycle t.

Given that external data on I_3, I_4 and I_5 are available when needed, we may define the function $\alpha : [-\pi_1, \pi_2] \times [1, N] \rightarrow A$ such that $\alpha(k, t)$ is the element of A that is stored in the register Ra of cell k (read from port I_3) during the processing phase PP_t. Although, for any specific t, $\alpha(k, t)$ is defined for all k cells, some cells will be idle during PP_t, and hence, will not operate on $\alpha(k, t)$. Let M_t be the set of cells that are not idle during PP_t, and define the tth computation front CF_t as the set of elements of A that are operated upon during that phase. More specifically,

$$CF_t = \{\alpha(k, t) : k \in M_t\}.$$

The succession of computation fronts represents the progress in the execution of the pseudo-systolic computation. More specifically, given a certain matrix, we may connect the elements of each computation front by a piece-wise linear curve and thus obtain a visual

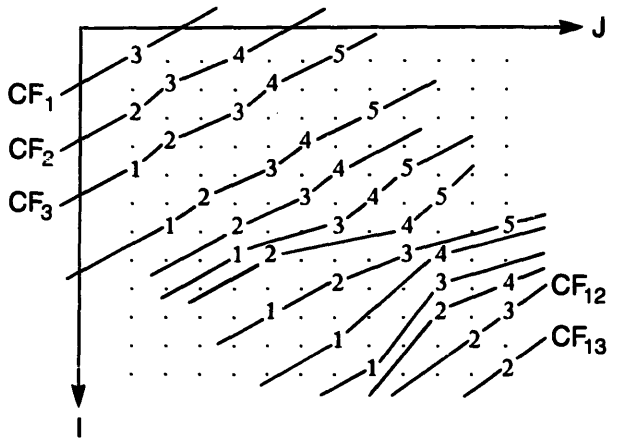

Fig. 4. Computation fronts.

picture that describes the propagation of the computation. For example, we show in Fig. 4 the different computation fronts that result from the pseudo-systolic execution of MAT/VEC on the matrix A of Fig. 1(a). For clarity, we represent each non-zero element, $a_{i,j} \in S_k$, of A by its stripe number, k, and we represent the zero elements of A by dots. Note that the concept of computation fronts is the same as that suggested in [20]. However, by allowing irregular fronts, we are able to model data-driven computations that depend on the value of the input as well as its availability.

Computation fronts may be constructed systematically if the conditions that governs the relation between the elements of the fronts are known. In order to derive these conditions, we first observe that if $a_{i,\sigma_k(i)}$ is in CF_t, then both y_i and $x_{\sigma_k(i)}$ should be at cell k during the processing phase PP_t. Similarly, if $a_{l,\sigma_q(l)}$ is in the same front CF_t, then both y_t and $x_{\sigma_q(l)}$ should be at cell q during PP_t. Assuming that $q > k$, then the sequential order of the x and y data streams requires that $l < i$ and $\sigma_q(l) > \sigma_k(i)$, respectively. In other words the following should be satisfied:

Consistency of data flow condition: If $a_{l,\sigma_q(l)}$ and $a_{i,\sigma_k(i)}$, $q > k$, are in the same computation front, then

$$l < i \quad \text{and} \quad \sigma_q(l) > \sigma_k(i). \tag{5a}$$

If the queues on the communication lines have infinite capacity, then any number of data items may be buffered between cells k and q. That is, there is no upper limit on the values of $(i - l)$ or $(\sigma_q(l) - \sigma_k(i))$, and hence (5a) is the only necessary condition for $a_{i,\sigma_k(i)}$ and $a_{l,\sigma_q(l)}$ to be in the same front. More descriptively, (5a) means that if s is the angle between any line segment in a computation front and the J-axis (see Fig. 4), then s should satisfy

$$-\infty < \tan(s) < 0,$$

that is

$$90° < s < 180°. \tag{5b}$$

In addition to the condition imposed on each individual front, we have to ensure that the fronts propagate in the same direction. More specifically,

Unidirectional propagation condition: If $a_{i,\sigma_k(i)} \in CF_t$ and $a_{l,\sigma_k(l)} \in CF_\tau$, $t > \tau$, then

$$i > l. \tag{6}$$

Now, given the zero pattern of any matrix A, we may construct the computation fronts as follows:

ALG1: / * Construction of the computation fronts * /

Step 1. Start from the upper left corner of A and construct CF_1 such that
- C1. It includes as many non-zero elements of A as possible,
- C2. Condition (5) is satisfied, and
- C3. All elements of A enclosed between CF_1 and the two axes are zeroes (implied by condition (6)).

Step 2. For $t = 2, 3, \ldots,$ repeat until every non-zero element of A is in some front
- 2.1. Given CF_{t-1}, construct CF_t such that
  - C1. It includes as many non-zero elements of A as possible,
  - C2. Condition (5) is satisfied, and
  - C3. All elements of A enclosed between CF_{t-1} and CF_t are zeroes (implied by condition (6)).

By the definition of the pseudo-systolic network, all possible communications are performed before the beginning of a processing phase. Moreover, every cell that receives all its operands during the communication phase, executes Step 3 of CYCLE 1 upon reception of the SYNC signal. For these reasons, we construct the computation fronts by including in each front as many elements of A as possible.

Large matrices that appear in practical applications often have non-zero diagonal elements. For this type of matrices, we may establish the following lower bound:

Proposition 3.1. *If A is an $n \times n$ matrix with non-zero diagonal elements, then at least n computation fronts are required in order to cover all the non-zero elements of A.*

Proof. Each diagonal element should be in some front, and condition (5) does not allow a single front to include more than one diagonal element. □

In order to establish an upper bound on the number of computation fronts, we consider not including in each front as many elements of A as possible. More specifically, assume that during the construction of CF_t, a particular element $a_{i,\sigma_k(i)}$ can be included in CF_t. This means that at the end of the communication phase CP_t, cell k is waiting in Step 3 of CYCLE 1. The exclusion of $a_{i,\sigma_k(i)}$ from CF_t may only result if cell k remains idle during the processing phase PP_t, say because SYNC did not reach that cell due to some transmission error. Although this error does not cause a failure of the computation, it does slow it down because cell k will stay at Step 3 of CYCLE 1 waiting for the SYNC signal of the next global cycle.

Hence, any set of computation fronts that satisfies conditions (5) and (6) corresponds to some execution of the pseudo systolic network with unreliable broadcast of SYNC. In order to reserve the term computation fronts to the sets that are constructed by ALG1 and that correspond to the execution of a reliable pseudo-systolic network, we introduce the following definition:

Definition 3.2. If condition C1 is removed from ALG1, then any set of fronts that results from the construction is called a set of contours of the matrix A.

Clearly, the number of computation fronts that cover A is less than or equal to the number of contours in any set of contours that covers A. This may be restated as follows:

Proposition 3.3. *Given a matrix A. If we may include all the non-zero elements of A in $\overline{N}$ contours that satisfy* (5) *and* (6), *then the pseudo-systolic execution of* MAT/VEC *terminates in at most $\overline{N}$ global cycles.*

4. Matrices with non-overlapping stripes

In this section, we study a class of matrices for which the upper bound on the number of computation fronts provided by Proposition 3.3 coincides with the lower bound established by Proposition 3.1. In other words, the execution of MAT/VEC for any matrix in this class is guaranteed to terminate in exactly n global cycles.

Let S_1 and S_2 be two complete stripes. By Definition 2.2. $S_1 < S_2$ if $\sigma_1(i) < \sigma_2(i)$, $i = 1, \ldots, n$. A more restrictive condition may be obtained if we require that, for every $i = 2, \ldots, n$, the intervals $\sigma_1(i) - \sigma_1(i-1)$ and $\sigma_2(i) - \sigma_2(i-1)$ do not overlap, that is,

$$\sigma_1(i) \leqslant \sigma_2(i-1), \quad i = 2, \ldots, n.$$

The following definition extends this simple condition to stripes that are not complete.

Definition 4.1. The π stripes of a matrix A are said to be non-overlapping, if for any stripe S_k, $-\pi_1 \leqslant k \leqslant \pi_2$, and any element $(i, \sigma_k(i)) \in S_k$, we have

$$\sigma_k(i) \leqslant \sigma_{k+m}(i-m) \tag{7}$$

where m is the smallest positive integer such that $(i-m, \sigma_{k+m}(i-m)) \in S_{k+m}$. If the inequality in (7) is strict, that is $<$ replaces $\leqslant$, then the stripes of A are called strictly non-overlapping.

For example, the matrix shown in Fig. 5(b) has strictly non-overlapping stripes, while the matrix shown in Fig. 5(a) has overlapping stripes. The positions where overlap occurs are indicated on the figure.

The following lemma may be easily proved by induction on (7).

Lemma 4.2. *The π stripes of A are non-overlapping if and only if for any k, $-\pi_1 \leqslant k \leqslant \pi_2$, and integers i and m, such that $(i, \sigma_k(i)) \in S_k$, and $(i-m, \sigma_{k+m}(i-m)) \in S_{k+m}$, equation (7) is satisfied.*

The property of strictly non-overlapping stripes guarantees that if both $a_{i,\sigma_k(i)}$ and y_i are at cell k during a specific global cycle t, then $x_{\sigma_k(i)}$ may not be locked behind another cell $k+m$, $m > 0$, and hence should arrive at cell k during the same global cycle t. In other words, the computation is not delayed due to internal data conflicts. This is formalized by the following

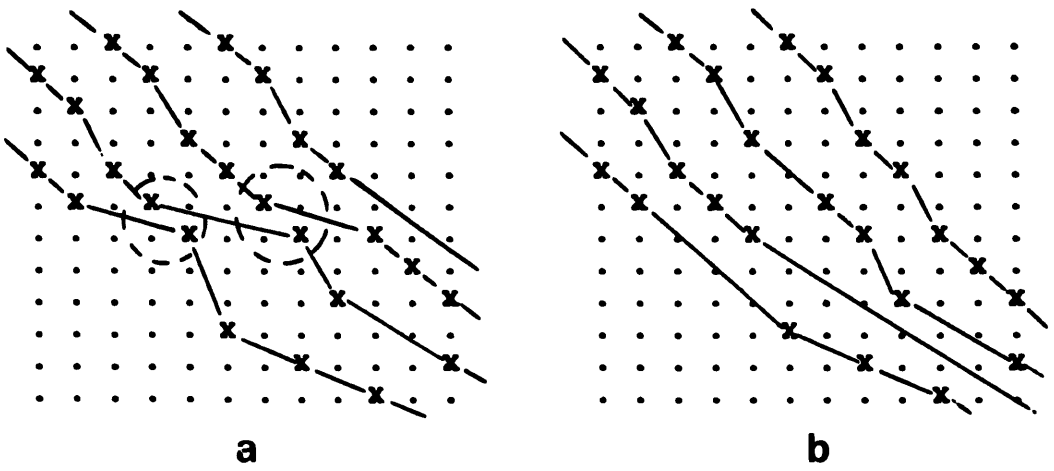

Fig. 5. (a) A matrix with overlapping stripes. (b) The same matrix with strictly non-overlapping stripes.

proposition:

Proposition 4.3. *Let A be a matrix with non-zero diagonal elements. If A is striped such that all its diagonal elements are covered by one stripe and all its stripes are strictly non-overlapping, then the pseudo-systolic computation of* MAT/VEC, *with input matrix A, terminates in exactly n global cycles.*

Proof. Let $\pi = \pi_1 + \pi_2 + 1$ be the stripe count of A, and denote the stripes of A by S_k, $k = -\pi_1, \ldots, 0, \ldots, \pi_2$, where $S_k < S_{k+1}$ and S_0 is the stripe that contains all the diagonals. Construct for each $r = 1, \ldots, n$ a contour C_r that includes $a_{r,r}$ and has a slope of one stripe per row. That is C_r includes for each $k = -\pi_1, \ldots, \pi_2$, the element (if any) of stripe k which is at row $r-k$. More specifically,

$$C_r = \left\{ a_{r-k,\sigma_k(r-k)} : -\pi_1 < k < \pi_2 \text{ and } (r-k, \sigma_k(r-k)) \in S_k \right\}. \tag{8}$$

For any specific $a_{i,j} \neq 0$, there exists a unique k such that $a_{i,j} \in S_k$, that is, $\sigma_k(i) = j$. Hence, there exists exactly one contour that includes $a_{i,j}$, namely C_{i+k}. In other words, the contours C_r, $r = 1, \ldots, n$ cover all the non-zero elements of A. Moreover, if $a_{v-k,\sigma_k(v-k)} \in C_v$ and $a_{u-k,\sigma_k(u-k)} \in C_u$ are in the same stripe S_k, then $v-k > u-k$ implies that $v > u$, that is, the condition of unidirectional propagation, namely equation (6), is satisfied.

It remains to prove that the consistency of the data flow condition, namely (5), is satisfied. Let $(i, \sigma_k(i))$ and $(l, \sigma_q(l))$ be any two elements in C_r. If $q > k$, then from the definition of C_r, $i = r-k$ and $l = r-q$, and hence $l < i$; however, the stripes of A are strictly non-overlapping, and thus, by using $m = q-k$ in (7), we obtain

$$\sigma_k(i) < \sigma_{k+q-k}(i-(q-k)) = \sigma_q(r-k-q+k) = \sigma_q(l)$$

which satisfies (5), and completes the proof that all the non-zero elements of A may be covered by n contours that satisfy (5) and (6). The result of the proposition, then, follows directly from Propositions 3.1 and 3.3. □

Proposition 4.3 applies only to matrices with strictly non-overlapping stripes. A similar result may be obtained for non-overlapping stripes, if we weaken the condition on computation fronts (contours) such that the line segments of a specific front (contour) may be vertical. That is, $\sigma_q(l)$ and $\sigma_k(i)$ in (5a) may be equal, which means that a component of the x data stream may be at two different cells k and q during the same processing phase. This may be achieved if each cell in MAT/VEC writes, immediately, on O_1 the value of x that it reads from I_1. More precisely, the operation of each cell (CYCLE 1) should be modified to the following:

CYCLE 2: / * initially, CX = CY = 0 * /
Step 1. $Ra = [I_3]$; $Rj = [I_4]$; $Ri = [I_5]$
Step 2. Do Steps 2.1 and 2.2 in parallel
2.1. wait until data is available on I_1;
$Rx = [I_1]$; $O_1 \leftarrow Rx$; CX = CX + 1;
If CX < Rj Then Go To 2.1
Else JOIN Step 2.2.
2.2. wait until data is available on I_2;
$Ry = [I_2]$; CY = CY + 1;
If CY < Ri Then {$O_2 \leftarrow Ry$; Go To 2.2}
Else JOIN Step 2.1.
Step 3. $Ry = Ry + Ra * Rx$
Step 4. $O_2 \leftarrow Ry$.

The proof of the following proposition is very similar to that of Proposition 4.3.

Proposition 4.4. *Let A be a matrix with non-zero diagonal elements, that is striped such that all its diagonal elements are covered by one stripe and all its stripes are non-overlapping. If each cell executes* CYCLE 2, *then the pseudo-systolic computation of* MAT/VEC, *with input A, terminates in n global cycles.*

Given a sparse matrix A, the advantage of constructing a non-overlapping stripe structure for A is clear. However, assuming that π is the minimum number of stripes that may cover A, and π_n is the number of non-overlapping stripes that may cover A, then, usually, $\pi_n > \pi$. Hence, a trade-off should be considered between

(1) using a network with π_n cells that terminates execution in n global cycles, or

(2) using a network with π cells which requires an execution time larger than n global cycles.

The cost, in terms of execution time, of incorporating the non-overlap condition (7) into an algorithm which constructs the stripe structure of general matrices (see the appendix) is usually high. However, for some type of matrices, non-overlapping stripes may be obtained for very low additional cost. For example, for the class of finite element stiffness matrices considered in [16], non-overlapping stripes may be obtained by renumbering the nodes of the grid from which the matrix is generated.

5. The communication issue

Besides the number of global cycles, the execution time of MAT/VEC, is determined by the time needed for the completion of each global cycle. This clearly depends on the communication activities which take place during the communication phase of the cycle. In this section, the data distribution in the network at any given cycle is modeled by a mathematical function. The changes in the functions associated with different cycles are used to estimate the amount of communication in the network.

By the definition of pseudo-systolic networks, no communication takes place during the processing phases of global cycles. Hence, during a specific processing phase PP_t, data assumes a static profile. In other words, a function may be defined for each data stream to specify the data items at each computational cell during PP_t.

For example, consider the x data stream in MAT/VEC and assume that the register Rx is set initially to an arbitrary value, say x_0 (the value of x_0 is irrelevant to the computation). The x-stream profile at the processing phase PP_t may then be specified by a function $xP_t: [-\pi_1, \pi_2] \rightarrow [0, n]$ such that $xP_t(k) = j$, where the register Rx in cell k contains x_j during PP_t. The y-stream profile at PP_t may be specified by a similar function $yP_t: [-\pi_1, \pi_2] \rightarrow [0, n]$. Note that the registers Rx and Ry always contain some values, and hence xP_t and yP_t are defined for every cell k.

If $k \in M_t$, that is cell k is not idle during PP_t, then $xP_t(k)$ and $yP_t(k)$ may be determined from the computation front CF_t. More specifically, if $k \in M_t$, then $a_{l,\sigma_k(l)} \in CF_t$ for some l, and hence, y_l and $x_{\sigma_k(l)}$ are at cell k during PP_t; that is,

$$a_{l,\sigma_k(l)} \in CF_t \quad \Rightarrow \quad yP_t(k) = l \text{ and } xP_t(k) = \sigma_k(l). \tag{9}$$

We call the values of $xP_t(k)$ and $yP_t(k)$, for $k \in M_t$, the knots of the profiles. On the other hand, if $k \notin M_t$, then the values of $xP_t(k)$ and $yP_t(k)$ may not be determined from a simple

formula. However, from the specification of the operation of each cell (CYCLE 1), it is clear that the following properties should be satisfied for $k = -\pi_1, \ldots, \pi_2 - 1$ and any t:

$$xP_t(k)\begin{cases} < xP_t(k+1) & \text{if } k+1 \in M_t, \\ \leqslant xP_t(k+1) & \text{if } k+1 \notin M_t, \end{cases} \tag{10a}$$

$$yP_t(k)\begin{cases} > yP_t(k+1) & \text{if } k \in M_t, \\ \geqslant yP_t(k+1) & \text{if } k \notin M_t. \end{cases} \tag{10b}$$

Note that is is possible that $xP_t(k) = xP_t(k+1)$ if $k+1 \notin M_t$. More specifically, when cell $k+1$ is waiting for a new input, its register Rx keeps the old value of x that has been written on O_1 during CP_t. If this value is also read by cell k during CP_t, then the registers Rx in both cells k and $k+1$ contain the same value during PP_t.

Also, the elements of each stream arrive at a specific cell in order. That is, the following is satisfied for $k = -\pi_1, \ldots, \pi_2$ and any t:

$$xP_t(k)\begin{cases} < xP_{t+1}(k) & \text{if } k \in M_t, \\ \leqslant xP_{t+1}(k) & \text{if } k \notin M_t, \end{cases} \tag{11a}$$

$$yP_t(k)\begin{cases} < yP_{t+1}(k) & \text{if } k \in M_t, \\ \leqslant yP_{t+1}(k) & \text{if } k \notin M_t. \end{cases} \tag{11b}$$

Again, the equalities may hold because a register does retain its value if it is not overwritten by a new one.

Equations (10) and (11) force on data profiles the same conditions that equations (5) and (6) force on computation fronts. In fact, it is straightforward to derive (5) and (6) from (10) and (11). Moreover, given some computation fronts which satisfy (5) and (6), that is, which simulate an execution of MAT/VEC, there should exist some functions that satisfy (9), (10) and (11) and correspond to the data profiles during execution. However, the mathematical construction of these functions is complex and involves the solution of a set of simultaneous inequalities, namely (10) and (11). For this reason, we seek further restrictions on the computation fronts and data profiles, by limiting the communication capabilities of the network.

5.1. Communication links with limited buffer capacity

A communication link directed from a cell k to cell $k+1$ may be regarded as a queue. Only cell k may append to this queue and only cell $k+1$ may access (and delete) its front element. So far, we have assumed that the communication queues (buffers) in MAT/VEC have infinite capacity, that is, $d_x = d_y = \infty$, where d_x and d_y are the capacities of individual queues on the x-stream and y-stream, respectively. With this assumption, we were able to derive the conditions (5) and (6) which enable us to construct the computation front for any given matrix. Clearly, any limitation on d_x and d_y represents some additional restrictions that should be taken into account during that construction.

More specifically, and without going into the implementation details of the communication protocols, if x_u and x_v are at cells k and $k+1$, respectively, during the processing phase PP_t, then $v-u$ elements of the x-stream should be buffered between the two cells, requiring a queue capacity of at least that size. That is

$$xP_t(k+1) - xP_t(k) \leqslant d_x. \tag{12a}$$

Similarly, for the y-stream

$$yP_t(k) - yP_t(k+1) \leqslant d_y. \tag{12b}$$

Equations (12) may be translated into restrictions on computation fronts. More precisely, we may derive the following from (9) and (12):

Buffer capacity condition: If $a_{l,\sigma_q(l)}$ and $a_{i,\sigma_k(i)}$, $q > k$, are in the same computation front CF_t, then

$$\sigma_q(l) - \sigma_k(i) \leqslant (q-k)d_x \quad \text{and} \quad i - l \leqslant (q-k)d_y. \tag{13a,b}$$

The buffer capacity condition is weaker than conditions (12) because it restricts the collective capacity of the links between cells q and k, rather that the capacities of the individual links.

If conditions (13) are added to ALG1 of Section 3, then computation fronts that satisfy (5), (6) and (13) may be constructed for any given matrix. However, because (13) is weaker than (12), the constructed fronts represent the execution of MAT/VEC only if it is possible to find a corresponding data profile (with knots specified by (9)) that satisfy (10), (11) and (12). Although this technique of constructing the computation fronts and then checking that they represent the actual execution of the network may, in general, fails, it can be used to show that the results of Proposition 3.3 and 4.3 are independent of the size of the queues on the y-stream links.

More specifically, consider the minimum value of d_y, namely $d_y = 1$, and keep $d_x = \infty$. It is easy to check that the contours C_r, $r = 1, \ldots, n$, used in the proof of Proposition 4.3 do satisfy (13b) with $d_y = 1$. Moreover, let

$$yP_r(k) = r - k, \quad r = 1, \ldots, n, \; k = -\pi_1, \ldots, \pi_2. \tag{14}$$

The knots of this profile correspond to C_r (as specified by (9)). Also, (14) satisfies (10b), (11b) and (12b) with $d_y = 1$. Hence, the n contours C_r given by (8) correspond to some execution of MAT/VEC with $d_y = 1$.

In other words, if $d_y = 1$ and $d_x = \infty$ in MAT/VEC, then the execution of the network for any $n \times n$ matrix with non-zero diagonal elements does terminate in n global cycles. Although this is a good result, it is preferable to replace the condition $d_x = \infty$ by one of the form $d_x \geqslant d_{\min}$. In order to derive the lower bound $d_{\min}$. We should construct an x-stream profile that satisfies (9), (10a) and (11a), and then from (12a) get

$$d_{\min} = \max_{r,k} \{ xP_r(k+1) - xP_r(k) \mid r = 1, \ldots, n, \; k = -\pi_1 + 1, \ldots, \pi_2 \}. \tag{15}$$

For general striped matrices, the construction of such an x-stream profile seems difficult. However, for certain types of stripe matrices the construction is straightforward (see [16] for examples).

In order to give further meaning to the bound (15), it is useful to consider matrices with complete, non-overlapping, stripes. In this case, it is easy to see that the contours given by (8) are the actual computation fronts, that is

$$CF_t = \{ a_{t-k,\sigma_k(t-k)} \mid k = -\pi_1, \ldots, \pi_2 \}. \tag{16}$$

From (9), the corresponding x-stream profile is given by

$$xP_t(k) = \sigma_k(t-k) \tag{17}$$

which by the very properties of the stripes satisfies (10a) and (11b). Now, from (15)

$$\begin{aligned} d_{\min} &= \max_{k,t} \{ \sigma_{k+1}(t-k-1) - \sigma_k(t-k) \} \\ &< \max_{k,t} \{ \sigma_{k+1}(t-k) - \sigma_k(t-k) \} \\ &= \text{the maximum separation between the stripes of the matrix.} \end{aligned}$$

In other words, the separation between the stripes determines the minimum size of the queues needed on the x-stream communication links.

Finally, we note that, with finite queue capacity, the communication protocol should not allow a cell to write on a queue that is full. More specifically, with $d_y = 1$, the statement $O_2 \leftarrow Ry$ in CYCLE 1, should be interpreted as "wait until the queue is not full, then write the content of Ry".

5.2. Communication time in MAT/VEC

Let τ_m be the time required by a cell in MAT/VEC to complete a floating-point operation (Step 3 of CYCLE 1), and let τ_c be the time required to move a data item from the input port of some cell to that of the next cell. For example, a data item, say x_j, may be moved from port I_1 of cell k to port I_1 of cell $k-1$ in time τ_c. This includes the time for the execution of Step 2.1 of CYCLE 1 and the associated protocols, as well as the time required for the signals to travel on the communication lines.

In order to estimate the execution time of any specific global cycle, we assume that cell k executes Step 2.1 $\xi_t(k)$ times and Step 2.2 $\eta_t(k)$ times during the communication phase CP_t of the tth global cycle. Clearly, $\xi_t(k)$ and $\eta_t(k)$ may be estimated from the data profiles as follows:

$$\xi_t(k) = xP_t(k) - xP_{t-1}(k), \qquad \eta_t(k) = yP_t(k) - yP_{t-1}(k). \tag{18a,b}$$

For $d_y = 1$ and $d_x \geqslant d_{\min}$, the y-stream profile is given by (14), from which we find that $\eta_t(k) = 1$ for any k and t. Also, it is easy to see that if the stripes are strictly non-overlapping, then data is always available on port I_1 of any cell k, and hence, the $\xi_t(k)$ executions of Step 2.1 at cell k during CP_t are not delayed. However, each cell in MAT/VEC has to wait until all the other cells complete their communication activities. Hence, the duration of the communication phase CP_t is determined by the maximum of $\xi_t(k)$ for $k = -\pi_1, \ldots, \pi_2$. That is by

$$\xi_t = \max\{\xi_1(k) \mid k = -\pi_1, \ldots, \pi_2\}. \tag{19}$$

With this, the execution time of MAT/VEC is given by

$$T = \sum_{t=1}^{N} (\tau_m + \xi_t \tau_c) = N\tau_m + \tau_c \sum_{t=1}^{N} \xi_t. \tag{20}$$

The values of ξ_t, $t = 1, \ldots, N$, depend on the specific stripe structure of the input matrix. However, without knowing the specific stripe structure of the matrix, we may only bound the execution time of MAT/VEC by

$$T < N(\tau_m + \xi_{\max}\tau_c) \quad \text{where } \xi_{\max} = \max\{\xi_t \mid t = 1, \ldots, N\}.$$

As we did in the last section, we may give further meaning to $\xi_{\max}$ by considering matrices with complete stripes. For this type of matrix, the x-stream profile is given by (17), from which we obtain

$$\xi_{\max} = \max_{k,t}\{\sigma_k(t-k) - \sigma_k(t-k-1)\} = \max_{k,i}\{\sigma_k(i) - \sigma_k(i-1)\}$$
$$= \text{the maximum slope of any stripe in the matrix.}$$

Note that equation (20) estimates the execution time of MAT/VEC assuming the hypothetical pseudo-systolic synchronization. In actual execution, however, the synchronization of MAT/VEC should be data driven (no wait in Step 3 of CYCLE 1), and hence faster than the pseudo-systolic execution. In other words, the value of T given by (20) is an upper bound for the actual execution time of MAT/VEC.

6. The iterative solution of sparse linear systems

In this section, we consider one of the most efficient iterative techniques for the solution of linear systems of the form $Ax = z$, namely the preconditioned conjugate gradient method (PCCG). This method applies conjugate gradient iterations to the system $M^{-1}Ax = M^{-1}z$, where the preconditioner matrix M is a suitable approximation of A^{-1}. In many preconditioning techniques, such as SSOR [1] and incomplete factorizations with no fill-ins [13], the matrix M may be expressed as the product of a unit lower triangular matrix L, a diagonal matrix D, and a unit upper triangular matrix U, that is $M = LDU$. The property that makes these preconditioners attractive is that the matrices L and U have the same zero structure as the lower and upper parts of the matrix A, respectively.

The bulk of the work in each iteration of the PCCG method is the multiplication of a vector by the matrix A, and the solution of two triangular linear systems $Ly = u$ and $Uz = v$.

It was shown in the previous sections that, if a suitable stripe structure is found for the matrix A, then the multiplication of any vector by A may be efficiently executed in parallel on the linear network MAT/VEC. Here, we show that, with very simple modification. MAT/VEC may also be used for the solution of any unit lower triangular system $Ly = u$. The unit upper triangular system $Uz = v$ may also be viewed as a unit lower triangular system $U^{T}z' = v'$, where z' and v' are obtained by reversing the elements of z and v, respectively. That is, if the order of U is n, then $z'_i = z_{n-i+1}$ and $v'_i = v_{n-i+1}$.

Assume, as before, that A has $\pi_1 + \pi_2 + 1$ stripes $S_{-\pi_1} < \cdots < S_{\pi_2}$, where S_0 is a complete stripe that contains all the diagonal elements. Hence, L has $\pi_1 + 1$ stripes that coincide with those in the lower triangular part of A. Namely, $S_{-\pi_1}, \ldots, S_0$.

Let MAT/VEC, with $d_y = 1$ and $d_x \geq d_{\min}$, be applied to the multiplication of L by any vector (Fig. 6). In this case, the inputs on ports I_4 and I_5 of cell 0 are not needed because all the elements in the complete stripe $S_0 = \{(i, i) \mid i = 1, \ldots, n\}$ are supplied, in order, to port I_3. For the same reason, the counters CX and CY of CYCLE 1 are not needed in cell 0. In other words, the cycle of cell 0 may be described by

CYCLE 3:
Step 1. $Ra = [I_3]$
Step 2. wait until data is available on I_1 and I_2;
$Rx = [I_1]$; $Ry = [I_2]$
Step 3. $Ry = Ry + Ra * Rx$
Step 4. $O_1 \leftarrow Rx$; $O_2 \leftarrow Ry$.

The same network of Fig. 6 may be used for the solution of the triangular system $Ly = u$ if the elements of the vector u, instead of the elements of S_0, are supplied to I_3 of cell 0, and the operation of this particular cell is described by the following cycle (instead of CYCLE 3):

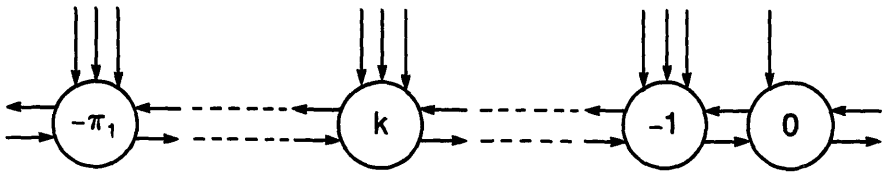

Fig. 6. A modified version of MAT/VEC.

CYCLE 4:
Step 1. $Ra = [I_3]$
Step 2. wait until data is available on I_2
$Ry = [I_2]$;
Step 3. $Ry = Ra - Ry$
Step 4. $O_1 \leftarrow Ry$; $O_2 \leftarrow Ry$.

We call the resulting network TRIANG. Note that the input port I_1 of cell 0 in TRIANG is not used, and hence no data need be supplied on it. Also, the elements of the result vector y are produced on the output port O_2.

The computation fronts for the pseudo-systolic execution of TRIANG may be obtained by applying our earlier rules to a matrix L^* which is obtained by replacing the diagonal elements in L by the elements of the vector u. But the zero structure of L^* is identical to that of L, which, in turn, is identical to the lower triangle of A. Hence, the stripe structure of A may be used to estimate the execution time of TRIANG. More specifically, if the stripes of A are strictly non-overlapping, then the pseudo-systolic execution of TRIANG terminates in n global cycles. The capacities of the buffers, as well as the communication time, in TRIANG also may be estimated from the stripe structure of A.

The networks MAT/VEC and TRIANG may be used as two high speed peripheral devices for a host computer which executes the PCCG iterations. In fact, only MAT/VEC is needed if a mechanism is available for switching the function of cell 0 between CYCLE 3 and CYCLE 4.

In addition to the above matrix operations, each PCCG iteration requires the computation of two inner products which involve the vectors computed by MAT/VEC and TRIANG. These inner products may be computed by a separate cell which is attached to the network and receives the elements of the output vectors as soon as they are produced. With this arrangement, an inner product which involves the output vector of the network may be available just one cycle after the network completes its computation.

Finally, we would like to mention that the algorithms presented in this paper have been modified to suit globally synchronized systolic arrays like, for example, the WARP systolic machine [5]. The details of the modifications are reported in [22].

7. Conclusion

We introduced the concept of striping a sparse matrix, which is, namely, the inclusion of the non-zero elements of the matrix in a structure which is regular enough to allow for efficient parallel manipulation. Although the concept is general, we only discussed its application to the design of regular VLSI networks for sparse matrices.

The operation of each cell in the networks presented in this paper are data dependent, as well as data-driven. This makes the application of formal analysis models (e.g. [6,15]) extremely difficult. For this reason, the additional simplification of pseudo-systolic synchronization was assumed, which allowed for the establishment of upper bounds on the performance of the networks. The actual performance of the data-driven networks may only be estimated by a detailed simulation of the computations.

It is proved that for an $n \times n$ matrix with π non-overlapping stripes, ($\pi \ll n$) and minimum separation $d_{\min}$ between stripes, the multiplication of a vector by the matrix may be performed in n global cycles, using a linear network of π cells connected by links that can buffer $d_{\min}$ data items. The same result also applies to the solution of triangular linear systems.

The task of finding a stripe structure for a sparse matrix may be accomplished in many different ways, including the algorithm given in the appendix. However, for some types of matrices, a stripe structure may be naturally determined from the underlying problem. For example, stiffness matrices arising in finite element analysis are usually generated from finite grids, and the stripe structure of a stiffness matrix is directly specified by the neighboring relation between the nodes of the corresponding grid. For more details we refer to [16].

Acknowledgment

I would like to thank Werner Rheinboldt for his many comments that helped in the formalization of the concept of striped matrices.

Appendix

An algorithm (in the form of a FORTRAN subroutine) is presented for the determination of a stripe structure for any given matrix A. The structure is specified by a matrix PA in which each column k specifies the positions of the elements in stripe S_k (see equation (2)). More precisely, if PA(i, k) = j, then $a_{i,j} \in S_k$, and if PA(i, k) = 0, then there are nc elements in row i that belong to S_k. Note that, for simplicity, we replaced $-b_1$ in (2) by 0.

The linear array 'Last' should be set, in the calling program, such that Last(i), $1 \leqslant i \leqslant n$, contains the number of non-zero elements in row i of the matrix A. Also, the integer 'nstrip' should be set to the maximum of Last(i), i = 1,...,n. The column numbers of the 'Last(i)' elements in row i of A are initially stored in the first 'Last(i)' locations of row i of PA (see Fig. A.1). The subroutine, then, transforms this input form of PA into the one that specifies a stripe structure of A. The number of stripes is also returned in the variable 'nstrip'. Following is the algorithm:

```
      subroutine stripes(PA,last,n,nstrip)
      integer PA(n,1), last(n)
c
      j = 0
 10     j = j + 1
        do 1000 i = 2,n
c
c          * * Find the previous element in the current stripe * *
 200       m = 0
```

x x . . x . .	1 2 5 0	0 1 2 5
. x . x . x .	2 4 6 0	0 2 4 6
x . x	1 3 0 0	1 3 0 0
. . x x x . x	3 4 5 7	3 4 5 7
. . . . x . x	5 7 0 0	0 5 7 0
. . . x . x .	4 6 0 0	4 6 0 0
. . . . . x x	6 7 0 0	6 7 0 0
a	b	c

Fig. A.1. Inputs and outputs of subroutine stripes. (a) The matrix A. (b) PA at input. (c) PA at output.

```
 300      m = m + 1
          if (i − m .LE. 0) go to 1000
          if (PA(i − m,j) .EQ. 0) go to 300
c         * * If stripe is not strictly increasing, then shift * *
c         * * row i − m by one position starting at column j * *
          if (PA(i − m,j) .GE. PA(i,j) then
            call shift(PA,last,n,nstrip,i − m,j)
            go to 300
          endif
c
 1000 continue
c
      if (j .LT. nstrip) go to 10
c
      return
      end
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
      subroutine shift(P,last,n,nstrip,i,j)
      integer P(n,1),last(n)
c
      last(i) = last(i) + 1
      jt = last(i)
      if (jt .LT. nstrip) nstrip = nstrip + 1
 10     P(i,jt) = P(i,jt − 1)
        jt = jt − 1
        if (jt .GT. j) go to 10
      P(i,j) = 0
      return
      end
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

The above algorithm have been used to construct the stripe structures for many sparse matrices generated from finite element analysis (see e.g. [16]). It is a greedy algorithm in the sense that, after the construction of the stripes $S_1, \ldots, S_{k-1}$, the next stripe S_k is constructed by including in it as many elements of A as possible. Given that including an element of A in S_{k+1} rather than S_k may not reduce the total number of stripes, then it is clear that the greedy scheme produces a stripe structure with the minimum number of stripes.

Finally, we note that it is possible to modify the above algorithm such that the resulting stripes are non-overlapping. However, it seems impossible to enforce the condition of strictly non-overlapping stripes. In fact, the existence of a stripe structure with strictly non-overlapping stripes is not always guaranteed. The matrix of Fig. 7(a) is an example for which such structure does not exist.

References

[1] L. Adams, Iterative algorithms for large sparse linear systems on parallel computers, Ph.D. Thesis, University of Virginia, 1982.

[2] L. Adams and R. Voigt, Design, development and use of the finite element machine, ICASE Report-172250, NASA-Langley Research Center, 1983.

[3] H. Amano, T. Boku, T. Kudoh and H. Aiso, A new version of the sparse matrix solving machine, *Proc. 12th International Symposium on Computer Architecture* (1985) 100–107.

[4] C. Arnold, M. Parr and M. Dewe, An efficient parallel algorithm for the solution of large sparse matrix equations, *IEEE Trans. Comput.* **32** (1983) 265–272.

[5] E. Arnould, H.T. Kung, O. Menzilcioglu and K. Sarocky, A systolic array computer, *Proc. 1985 IEEE Conference on Acoustics, Speech and Signal Processing* (1985) 232–235.

[6] M. Chen, Space-time algorithms: semantics and methodology, Ph.D. Thesis, California Institute of Technology, 1983.

[7] P. Concus, G. Golub and D. O'Leary, A generalized conjugate gradient method for the numerical solution of elliptic PDE's, in: J. Bunch and D. Rose, eds., *Sparse Matrix Computations* (Academic Press, New York, 1975).

[8] D. Evans, On preconditioned iterative methods for partial differential equations, in: D.J. Evans, ed., *Preconditioning Methods, Theory and Applications* (Gordon & Breach, London, 1982).

[9] A. George and J. Liu, *Computer Solutions of Large Sparse Positive Definite Systems,* Prentice-Hall Series in Computational Mathematics (Prentice-Hall, Englewood Cliffs, NJ, 1981).

[10] H.T. Kung and C.E. Leiserson, Systolic arrays for VLSI, in: C. Mead and L. Conway, eds., *Introduction to VLSI Systems* (Addison-Wesley, Reading, MA, 1980).

[11] S.Y. Kung, K.S. Arun, R.J. Gal-Ezer and D.B Rao, Wavefront array processor: Language, architecture and applications, *IEEE Trans. Comput.* **31** (1982) 1056–1066.

[12] N. Madsen, G. Rodrigue and J. Karush, Matrix multiplication by diagonals on a vector/parallel processor, *Inform. Process. Lett.* **5** (2) (1976) 41–45.

[13] T. Manteuffel, An incomplete factorization technique for positive definite linear systems, *Math. Comput.* **34** (150) (1980) 673–697.

[14] R. Melhem, A study of data interlock in computational networks for sparse matrix multiplication, *IEEE Trans. Comput.* **36** (9) (1987) 1101–1107.

[15] R. Melhem and W. Rheinboldt, A mathematical model for the verification of systolic networks, *SIAM J. Comput.* **13**(3) (1984) 341–365.

[16] R. Melhem, A study of the stripe structure of finite element stiffness matrices, *SIAM J. Numer. Anal.* **24** (6) (1987).

[17] D. Reed and M. Patrick, Iterative solution of large, sparse linear systems on a static data flow architecture: Performance studies, *IEEE Trans. Comput.* **36** (1985) 874–880.

[18] D. Reed and M. Patrick, Parallel, iterative solution of sparse linear systems: Models and architectures, *Parallel Comput.* **2** (1985) 45–67.

[19] J. Rice and R. Boisvert, in: *Solving Elliptic Problems with Ellpack* (Springer, Berlin, 1984).

[20] V. Weiser and A. Davis, A wavefront notation tool for VLSI array design, in: H.T. Kung, B. Sproull and G. Steele, eds., *VLSI Systems and Computations* (Computer Science Press, Rockville, MD, 1981).

[21] O. Wing, A content addressable systolic array for sparse matrix computation, *J. Parallel Distributed Comput.* **2** (1985) 170–181.

[22] R. Melhem, Iterative solution of sparse linear systems on systolic arrays, Proc. 1987 International Conference on Parallel Processing (1987) 560–563.