## Correspondence

# From Determinacy to Systaltic Arrays

D. P. O'LEARY AND G. W. STEWART

*Abstract*—**In this paper we extend a model of Karp and Miller for parallel computation. We show that the extended model is deterministic, in the sense that under different scheduling regimes each process in the computation consumes the same input and generates the same output. Moreover, if the computation halts, the final state is independent of scheduling. The model is applied to the generation of precedence graphs, from which lower time bounds may be deduced, and to the synchronization of systolic arrays by local rather than global control.**

*Index Terms*—**Data flow algorithms, determinacy, parallel computation, systaltic arrays, systolic arrays.**

## I. Introduction

The purpose of this paper is to describe and investigate the properties of a model of parallel computation in which independent processes coordinate their actions by message passing. The model is an extension of one proposed by Karp and Miller [3] in 1966. In it, computations take place at the nodes of a directed graph, whose arcs represent lines of communication.

Graph theoretic models of computation are by no means new (see [1] for a bibliography that includes some of the more important papers). The majority of these papers begin with a control flow paradigm of computation, i.e., a sequential program or a flowchart, and extract an acyclic precedence graph of the unrolled computation which can be used in code optimization or to extract parallelism. Others, under the generic term of data flow, begin at, or very near a precedence graph.

Recently, there have appeared a number of parallel computers whose processors communicate by message passing, for example, the cosmic cube [9], the ZMOB [8], and the commercial hypercubes produced by the INTEL and NCUBE corporations. Programs for these systems are supported by small operating systems resident on each processor that coordinate the delivery of messages between processes (e.g., the DOMINO system [5], [6] or the HARMONY system [2]). One purpose of this paper is to give such systems a firm theoretical base, and the natural starting point of such a development is communicating sequential processes, rather than sequential programs. Our approach is to construct a formal mathematical model which closely corresponds to an implementation of a message passing system and then to establish its properties with full mathematical rigor.

Since the formal model requires a great deal of notation for its expression, it is appropriate to begin with an informal description. The computations in our model are done by independent *computational nodes* which cycle between requesting data from certain nodes, computing, and sending data to certain other nodes. More precisely, the nodes lie at the vertices of a directed graph, called a *computational network,* whose arcs represent lines of communication. Each time a node sends data to another node, the data are placed in a queue on the arc between the two nodes. When a node has requested data from other nodes it is blocked from further execution until the data it has requested arrive at the appropriate input queues. We shall say that a node in a data flow computation is *ready* if it has enough data in its queues to compute, i.e., if all its requests are satisfied. At any one time it is possible for several nodes in a computation to be ready, and it is natural to ask if the order in which ready nodes compute makes any difference to the overall computation. This is the question of *determinacy*, and we shall answer it by proving that whatever the order of computation, each node sees the same input and generates the same output. A related question is what is the state of the network when none of the nodes are ready, i.e., when the network halts. We shall show that however the computations proceed, the network (if it halts) halts in the same state.

Our model is very close to a subset of DOMINO, a message passing system which has been developed at the University of Maryland, and in fact this work was undertaken to establish that the system was deterministic—something we were unable to convince ourselves of by informal reasoning. We have already mentioned that the model is an extension of one proposed by Karp and Miller, the chief difference being that in their model the amount of output a node can produce on firing is fixed at the beginning of a computation. Unfortunately, their proofs, which are based on counting arguments, do not go through for the extended model.

In the next section we shall introduce the model. We apologize in advance for the amount of mathematical apparatus, but it seems to be necessary if the proofs are not to degenerate into mere hand-waving. In Section III we establish the determinacy and halting properties of the algorithm. We also prove an extension theorem which is used in the next section to construct a precedence graph for a computation in the model. This graph can be used in an obvious way to determine lower bounds on the computation time of the algorithm. Finally, in Section V we apply our theory to a device, called a diastolic control, for the local synchronization of systolic arrays.

Throughout this note, $\boldsymbol{D}$ will denote a space of data which can be transmitted between nodes. A queue will be an ordered $t$ tuple $(d_1, d_2, \cdots, d_t)$ of elements of $\boldsymbol{D}$, with $d_1$ regarded as beginning the queue. If $Q = (d_1, d_2, \cdots, d_t)$ and $\bar{Q} = (\bar{d}_1, \bar{d}_2, \cdots, \bar{d}_{\bar{t}})$ are queues, then

1) $\#(Q) = t$ is the length of $Q$;

2) head $(Q, k) = (d_1, d_2, \cdots, d_{\min\{k,t\}})$ is the queue consisting of the first $k$ elements of $Q$, or $Q$ itself if $k \geq t$;

3) tail $(Q, k) = (d_{k+1}, \cdots, d_t)$ is the queue remaining after the first $k$ elements are removed, or the empty queue if $k \geq t$;

4) $Q|\bar{Q} = (d_1, \cdots, d_t, \bar{d}_1, \cdots, \bar{d}_{\bar{t}})$ is the queue that results from appending $\bar{Q}$ to $Q$.

We shall say that two queues $Q$ and $\bar{Q}$ are *comparable* if head $(Q, t) =$ head $(\bar{Q}, t)$ for $t = \min\{\#(Q), \#(\bar{Q})\}$.

## II. The Model

Formally we shall define a *computational network* as a directed graph with queues associated with each arc and a state transition function associated with each vertex. Specifically, the graph of the computational network consists of

1) A set $\boldsymbol{N} = \{N_1, N_2, \cdots, N_n\}$ of vertices called *nodes*.

2) A set of directed arcs $\boldsymbol{A} \subset \{(M, N): M, N \in \boldsymbol{N}, M \neq N\}$. (The reason why arcs of the form $(N, N)$ are not permitted will emerge later when we discuss states and their transition functions.)

3) A set of queues $\boldsymbol{Q} = \{Q(A): A \in \boldsymbol{A}\}$ associated with the arcs of the network.

In the sequel we shall need some notation for the arcs entering and leaving a node $N$. We shall denote the $l(N)$ nodes at the beginning of

Manuscript received September 11, 1985; revised August 27, 1986. This work was supported in part by the Air Force Office of Scientific Research under Grant AFOSR-82-0078.

The authors are with the Department of Computer Science, University of Maryland, College Park, MD 20742.

IEEE Log Number 8714094.

the entering arcs by $I_{N,1}, I_{N,2}, \cdots, I_{N,l(N)}$ and call the corresponding queues $Q_{N,i}^{\text{IN}} \equiv Q(I_{N,i}, N)$ the *input queues* for the node. The set of input queues for node $N$ will be denoted by $Q_N^{\text{IN}}$. Similarly, the destination nodes at the end of the arcs leaving $N$ will be denoted by $O_{N,1}, O_{N,2}, \cdots, O_{N,m(N)}$ and the corresponding queues will be called *output queues* and written $Q_{N,j}^{\text{OUT}}$. The set of output queues of $N$ will be denoted by $Q_N^{\text{OUT}}$.

We now turn to the definition of the state of a node and its transition function. A state $s$ of a node $N$ is a triplet $(Q_N^{\text{IN}}, T, g)$ consisting of the following items:

1) the set of input queues $Q_N^{\text{IN}}$

2) a vector $T = (t_1, t_2, \cdots, t_{l(N)})$ of nonnegative integers called *thresholds*

3) an *output function*

$$g: D^{t_1} \times \cdots \times D^{t_{l(N)}} \to D^{w_1} \times \cdots \times D^{w_{m(N)}}$$

where the $w_i$ are integers. These integers may depend on the thresholds and the contents of the input queues, and may be different for different states of node $N$.

We shall say that a node $N$ in state $s = (Q_N^{\text{IN}}, T, g)$ is *ready* (or alternatively $s$ is a ready state of $N$) if $\#(Q_{N,i}^{\text{IN}}) \geq t_i$ $(i = 1, \cdots, l(N))$. Informally, a node is ready if it has enough data on its input queues to compute.

With each node $N$ is associated a *state transition function* $f_N$, mapping states into states, which has the following properties. Let $s = (Q_N^{\text{IN}}, T, g)$ be a state of $N$.

1) If $s$ is not a ready state of $N$, then $f_N(s) = s$.

2) If $s$ is a ready state of $N$, then $f_N(s) = (\bar{Q}_N^{\text{IN}}, \bar{T}, \bar{g})$, where the new threshold vector $\bar{T}$ and output function $\bar{g}$ are defined by $f$ and the new input queues satisfy

$$\bar{Q}_{N,i}^{\text{IN}} = \text{tail}\,(Q_{N,i}^{\text{IN}}, t_i) \qquad (i = 1, 2, \cdots, l(N)). \tag{2.1}$$

Informally, the transition function corresponds to a single computation by the node. Data, in the quantity specified by $T$, are removed from the input queues and are used to generate output which goes to other nodes and, as we shall see, has the side effect of altering their states.

One reason for not allowing a node to communicate directly with itself is to preserve the simplicity of formula (2.1) for $\bar{Q}_N^{\text{IN}}$, which would otherwise have to be modified to treat the queue on the arc from $N$ to itself as a special case. This is not an essential restriction on our model of computation, since, as we shall see in the next section, nodes can be added to cycle output data back into node $N$.

## III. Execution Sequences, Determinacy, and Halting

Having defined how a node computes, we must now look at the behavior of the network as a whole. First consider a network in which each node $N$ has a state $s_N$. We shall define the *state of the network* to be the $n$ tuple $s = (s_{N_1}, s_{N_2}, \cdots, s_{N_n})$. At this point we cannot define a state transition function for the network, since we must first specify an order in which ready nodes are to be executed. This leads us to define a *scheduling sequence* as a sequencee $E = \langle E_1, E_2, \cdots \rangle$ of nodes from the network. An *execution sequence* is an ordered pair $(E, s(0))$, where $E$ is a scheduling sequence and $s(0) = (s_{N_1}(0), s_{N_2}(0), \cdots, s_{N_n}(0))$ is an (initial) state of the network. Given an execution sequence $(E, s(0))$ we define the state of the network at time $\tau$ inductively as follows.

1) If $E_\tau$ is not ready at time $\tau - 1$ (i.e., if $s_{E_\tau}(\tau - 1)$ is not a ready state of $E_\tau$), then $s(\tau) = s(\tau - 1)$.

2) If $E_\tau$ is ready at time $\tau - 1$, then set $s_{E_\tau}(\tau) = f_{E_\tau}[s_{E_\tau}(\tau - 1)]$, and alter the states of the output nodes of $E_\tau$ by setting

$$Q_{E_\tau,j}^{\text{OUT}}(\tau) = Q_{E_\tau,j}^{\text{OUT}}(\tau - 1)|g_j \qquad (j = 1, \cdots, m(E_\tau)),$$

where $g_j$ is the $j$th component of the output function of $s_{E_\tau}(\tau - 1)$ evaluated at $[\text{head}\,(Q_{E_\tau,1}^{\text{IN}}(\tau - 1), t_1), \cdots, \text{head}\,(Q_{E_\tau,l(E_\tau)}^{\text{IN}}(\tau - 1), t_{l(k)})]$.

In other words this definition says that just before time $\tau$, node $E_\tau$, if it is ready, changes state and places its output on the queues of its output arcs. In this case we shall say that $E_\tau$ *fires* at time $\tau$ producing state $s(\tau)$.

Before we state and prove the determinacy theorem, let us pause to consider what scheduling and execution sequences mean in a parallel processing environment. Imagine that the nodes of a computational network have been divided into groups and each group has been assigned to a processor. On each processor the resident operating system will schedule the members of its group for execution, thus creating a local scheduling sequence. If we take the local scheduling sequences and merge them according to the order in which the nodes were scheduled in real time (ignoring ties, which are of probability zero), we obtain a scheduling sequence and a corresponding execution sequence for the computation as a whole.

A change in scheduling algorithms or a change in processor speeds will change the execution sequence. We should like to know that the computation is in some sense unaffected by these changes. In terms of our model, we should like to show that two execution sequences with the same initial state produce essentially the same states in the nodes of the network. However, there are two complications. First, we cannot simply compare the states of a node when they have appeared the same number of times in two execution sequences, since the node may have fired more often in one execution sequence than in the other. The second complication is that the input queues of a node need not be the same for the two execution sequences, even when the node has fired the same number of times.

To circumvent the first complication, we shall say that an execution sequence is *busy* if all its nodes are ready at their respective times. If $E_\tau$ is the $p$th occurrence of a node in a busy execution sequence, then the node will fire for the $p$th time at time $\tau$. Observe that any execution sequence can be converted to an equivalent busy execution sequence by deleting all nodes that do not fire. Thus, there is no loss of generality in confining ourselves to busy execution sequences.

The second complication is handled by the notion of comparability. We say that two states $s_k$ and $\bar{s}_k$ of a node are *comparable* if their thresholds and output functions are the same and if their input queues are comparable. Since the initial parts of the input queues of two comparable states of a node are the same, if the node fires from either state it will produce the same output and the input queues will remain comparable.

We are now in a position to state and prove the *determinacy theorem.*

*Theorem 3.1:* Let $(E, s(0))$ and $(\bar{E}, s(0))$ be busy execution sequences with the same initial state. Suppose that $E_\tau$ is the $p$th occurrence of a node $N$ in $E$ and $\bar{E}_{\bar{\tau}}$ is the $p$th occurrence of $N$ in $\bar{E}$. Then $s_N(\tau)$ and $\bar{s}_N(\bar{\tau})$ are comparable.

*Proof:* The proof is by contradiction. Let $E_\tau = N$ be the first member of $E$ for which the theorem fails, and let $\bar{E}_{\bar{\tau}}$ be the corresponding occurrence of $N$ in $\bar{E}$. Let $\tau_1, \cdots, \tau_p = \tau$ and $\bar{\tau}_1, \cdots, \bar{\tau}_p = \bar{\tau}$ be the times at which $N$ fires in $(E, s(0))$ and $(\bar{E}, s(0))$. Since $s_N(\tau_{p-1})$ and $\bar{s}_N(\bar{\tau}_{p-1})$ are comparable, $s_N(\tau - 1)$ and $\bar{s}_N(\bar{\tau} - 1)$ have the same thresholds and output functions. Thus, the result will be established if we can show that $s_N(\tau - 1)$ and $\bar{s}_N(\bar{\tau} - 1)$ have comparable input queues.

Consider the input node $I_{N,i}$. Let $q$ be the number of occurrences of $I_{N,i}$ in $E$ before time $\tau$ in the execution sequence $(E, s(0))$; let $g_1, \cdots, g_q$ be the outputs from $I_{N,i}$ to $N$; and let $Q_{N,i}^{\text{IN}}(\tau)$ be the status of the queue $Q_{N,i}^{\text{IN}}$ at time $\tau$. Let the same quantities with bars over them refer to the execution sequence $(\bar{E}, s(0))$.

By the determinacy of the execution sequence $(E, s(0))$ before time $\tau$, if $r = \min\{q, \bar{q}\}$ then $g_j = \bar{g}_j$ $(j = 1, \cdots, r)$. Moreover, the $i$th thresholds of $N$ satisfy $t_{N,i}(\tau_j) = \bar{t}_{N,i}(\bar{\tau}_j)$ $(j = 1, \cdots, p - 1)$. Hence, by direct calculation of what comes in and what goes out, both $Q_{N,i}^{\text{IN}}(\tau - 1)$ and $\bar{Q}_{N,i}^{\text{IN}}(\bar{\tau} - 1)$ begin with

$$\text{tail}\,[Q_{N,i}^{\text{IN}}(0)|g_1|g_2|\cdots|g_r, t_{N,i}(\tau_1) + \cdots + t_{N,i}(\tau_{p-1})]. \tag{3.1}$$

Moreover, depending on whether $r = q$ or $r = \bar{q}$, either $Q_{N,i}^{\text{IN}}(\tau - 1)$

or $\bar{Q}^{\mathrm{IN}}_{N,i}(\bar{\tau}-1)$ is equal to (3.1). Hence, $Q^{\mathrm{IN}}_{N,i}(\tau-1)$ and $\bar{Q}^{\mathrm{IN}}_{N,i}(\bar{\tau}-1)$ are comparable, which establishes the theorem.

Because Theorem 3.1 concerns ongoing calculations, it makes no assertions about equality of all states at a given time, or even about their comparability. However, we should hope that if the calculation stops, the confusion introduced by different execution sequences will sort itself out. To make the notion of stopping precise, we shall say that an execution sequence $(E, s(0))$ *halts* at time $\tau$ if all the nodes in the network are unready at time $\tau$, and $\tau$ is the smallest integer for which this is true. Note that if a busy execution sequence halts at time $\tau$, the corresponding scheduling sequence terminates with $E_\tau$.

Before we treat the states of the network, we give the following useful lemma. Let us agree to call two busy execution sequences *equipollent* if they have the property that a node firing $p$ times in one fires $p$ times in the other.

*Lemma 3.2:* If $(E, s(0))$ and $(\bar{E}, s(0))$ are equipollent of length $\tau$, then $s(\tau) = \bar{s}(\tau)$.

*Proof:* This follows from determinacy and a direct computation of the input queues in the spirit of (3.1).

We are now ready to prove the following *halting theorem*.

*Theorem 3.3:* Let $(E, s(0))$ and $(\bar{E}, s(0))$ be busy execution sequences that halt. Then $(E, s(0))$ and $(\bar{E}, s(0))$ are equipollent. Hence, they both halt in the same state.

*Proof:* We begin by showing that a node can fire no more times in $(\bar{E}, s(0))$ than in $(E, s(0))$. Let the node $N$ fire $p_N$ times in $(E, s(0))$. If $N$ fires more than $p_N$ times in $(\bar{E}, s(0))$, let $\tau_N$ denote the time of its $(p_N + 1)$th firing.

If there are no nodes $N$ that fire more than $p_N$ times in $(\bar{E}, s(0))$, then we are through. Otherwise, let $N$ be the node for which $\tau_N$ is smallest. Let us compare $\bar{s}_N(\tau_N - 1)$ with $s_N(\tau)$, i.e., the states of $N$ just before its $(p_N + 1)$th firing in $(\bar{E}, s(0))$ and at the end of $(E, s(0))$. By determinacy the two states have the same thresholds and the same output functions. Moreover, the input queues in the state $\bar{s}_N(\tau_N - 1)$ cannot be longer than those in $s_N(\tau)$, for the number of firings of each node $M \neq N$ in $(E, s(0))$ is greater than or equal to the number of firings in $(\bar{E}, s(0))$ up to time $\tau_N - 1$, and by determinacy each firing places the same amounts of data on the input queues. Further, $N$ has fired an equal number of times in both execution sequences up to this time and, by determinacy, has removed an equal amount of data from its input queues. It follows that, since $s_N(\tau)$ is not a ready state of $N$, $\bar{s}_N(\tau_N - 1)$ is not a ready state of $N$. Hence, $N$ cannot fire in $(\bar{E}, s(0))$ at time $\tau_N$—a contradiction.

A similar argument shows that a node fires no more times in $(E, s(0))$ than in $(\bar{E}, s(0))$. Hence, each node fires the same number of times in both sequences, which with Lemma 3.2 establishes the theorem.

The practical consequence of the determinacy and halting theorems is that if one can imagine an execution sequence in which a computation terminates properly, then it must work for all execution sequences. One such execution sequence may be obtained by assigning all the nodes of the network to a single processor. In other words, on systems that conform to our model, one can debug on a single processor with the assurance that the program will run when it is distributed on a network of processors.

The halting theorem has the major drawback that it cannot be applied to execution sequences that do not halt. To circumvent this difficulty we are going to show how two execution sequences for a network can be extended to a point where they can be compared. But first we must discuss the problem of internal states of a node.

The difficulty is that we have not given our nodes a great deal of computing power. Their state transition functions have no memory, and a node is unable even to count for itself. However, we can circumvent this difficulty by embedding a computational network in a larger one which effectively provides memory for each node. With each node $N$ associate a *memory node* $M$ and arcs $(M, N)$ and $(N, M)$. Whenever $N$ fires it places a single item of data on $Q(N, M)$ and sets the threshold for $Q(M, N)$ to one. The node $M$ always has a threshold of one associated with $Q(N, M)$, and its transition function causes the data item on $Q(N, M)$ to be transferred to $Q(M, N)$. In this way a (variable) data item circulates between $N$ and $M$, passing from $N$ to $M$ when $N$ fires and back to $N$ when $M$ fires, as it must, before $N$ fires again. Note that a node can bring itself to a halt by refusing to send data to a memory node.

By adding memory nodes we can endow a node with as much memory as we require. This in effect converts our nodes to ordinary sequential computers, and in the sequel we will speak freely of programming and reprogramming nodes.

The above observations allow us to prove the following *extension theorem*. It will be used in the next section to construct precedence graphs for computations in our model.

*Theorem 3.4:* Let $(E, s(0))$ and $(\bar{E}, s(0))$ be finite busy execution sequences. For each $N \in \mathbf{N}$ let $p_N$ be the number of occurrences of the node $N$ in $(E, s(0))$ and $\bar{p}_N$ be the number of occurrences in $(\bar{E}, s(0))$. Let $q_N = \max\{p_N, \bar{p}_N\}$. Then $(E, s(0))$ and $(\bar{E}, s(0))$ can be extended to equipollent sequences $(E^*, s(0))$ and $(\bar{E}^*, s(0))$ of length $\tau = \sum_{N \in \mathbf{N}} q_N$.

*Proof:* Reprogram each node $N$ so that it halts after $q_N$ firings but otherwise behaves as before. Now if some node $N$ in one of the sequences $(E, s(0))$ or $(\bar{E}, s(0))$ has not fired $q_N$ times, then at the end of one of the sequences, some node must be ready, for otherwise both sequences halt with $N$ having fired exactly $q_N$ times in one sequence and less than $q_N$ times in the other, contradicting the halting theorem. We may, therefore, extend the sequence possessing the node that is ready by adding that node to it. Continuing in this manner, we extend both sequences until each node $N$ has fired $q_N$ times, at which point both sequences halt.

## IV. Precedence Graphs and Lower Bounds

In this section we shall show how to associate a unique *precedence graph* with an execution sequence, which gives a static picture of the entire computation. We have noted in the Introduction that precedence graphs of sequential algorithms can be used to uncover parallelism. For computational networks, the precedence graph is less useful, since parallelism is built into the very definition. Nonetheless, a precedence graph can be used to determine lower bounds on the execution time of a computation. In practice it is good to have such lower bounds to shoot at when deciding how to assign nodes to processors and how to schedule nodes on processors.

Before we begin, we pause to introduce a notational convention. The elements $E_\tau$ of a scheduling sequence are simply nodes, perhaps repeated many times in the sequence. In what follows we shall need to distinguish the occurrence of a node at time $\tau$ from an occurrence of the same node at another time $\sigma$. This can be done by working with the ordered pair $(E_\tau, \tau)$. To simplify our formulas we shall let $E^\tau$ stand for this ordered pair.

Let $(E, s(0))$ be a busy execution sequence for a computational network. The precedence graph for $(E, s(0))$ is a directed graph $P(E, s(0))$ which is constructed as follows. The vertices of the graph are the pairs $E^\tau$. The arcs are assigned as follows.

1) Draw an arc from $E^\sigma$ to $E^\tau$ if $\tau$ is the smallest integer greater than $\sigma$ such that $E_\sigma = E_\tau$.

2) For each $E^\sigma$, Let $g_1, \cdots, g_{m(E_\sigma)}$ be the outputs from the firing of $E^\sigma$. For $j = 1, 2, \cdots, m(E_\sigma)$ draw an arc between $E^\sigma$ and $E^\tau$ if $E_\tau = O_{E_\sigma, j}$ and the firing of $E^\tau$ consumes some of the message $g_j$.

The properties of the precedence graph are described in the following theorem.

*Theorem 4.1:* The precedence graph of a busy execution sequence $(E, s(0))$ is acyclic. If $(\bar{E}, s(0))$ is an equipollent sequence, then $P(E, s(0))$ and $P(\bar{E}, s(0))$ are isomorphic. If there is a path in $P(E, s(0))$, from $E^\sigma$ to $E^\tau$, and $E^\sigma(E^\tau)$ represents the $p$th ($q$th) occurrence of $E_\sigma(E_\tau)$ in the execution sequence, then in any execution sequence beginning with $s(0)$ the $p$th occurrence of $E_\sigma$ precedes the $q$th occurrence of $E_\tau$.

*Proof:* The fact that $P(E, s(0))$ is acyclic follows from the fact that if there is an arc from $E^\sigma$ to $E^\tau$ then $\sigma < \tau$.

The isomorphism is defined as follows. If $E^\tau$ represents the $p$th occurrence of $E_\tau$ in $(E, s(0))$, then $E^\tau$ is mapped onto $\bar{E}^{\bar{\tau}}$ where $\bar{E}^{\bar{\tau}}$ represents the $p$th occurrence of $\bar{E}_{\bar{\tau}}$ in $(\bar{E}, s(0))$. The determinacy theorem assures us that if two vertices are joined in $P(E, s(0))$, then they will be joined in $P(\bar{E}, s(0))$.

To establish the last statement, truncate the sequence $(E, s(0))$ so that it ends with $E^{\tau}$. Let $(\bar{E}, s(0))$ be another sequence and truncate it so that it ends with the $q$th occurrence of $E_{\tau}$. Use the extension theorem to extend the truncated sequences to equipollent sequences. Since the precedence graphs of equipollent sequences are isomorphic and since $E^{\sigma}$ precedes $E^{\tau}$ in the first graph, the $p$th occurrence of $E_{\sigma}$ must precede the $q$th occurrence of $E_{\tau}$ in the second graph.

The theorem justifies calling $P(E, s(0))$ the precedence graph of the execution sequence. Not only is it the same for all execution sequences with the same initial state, but it contains the basic information about what can and cannot be implemented in parallel. If two vertices lie on a path in the precedence graph, the execution of one must precede that of the other; if they do not, then their executions are independent.

The precedence graph can be used to determine a lower bound on the execution time of a network. The technique has been exhaustively treated in the operations research literature under the rubric critical path method, and accordingly we shall confine ourselves to a sketch of the basic construction.

Assume that an execution time $d_{\tau}$ has been assigned to each vertex $E^{\tau}$ of the precedence graph. We define the duration $D_{\tau}$ of vertex $E^{\tau}$ as the maximum over all paths terminating at $E_{\tau}$ of the sum of the durations of vertices on the path. It can then be shown that the shortest time in which a network can be executed is the largest duration in the precedence graph. Moreover, on a system in which each node is assigned to a separate processor, this bound can be attained, provided communication is instantaneous.

In highly structured problems, the precedence graph and the associated durations can frequently be determined by inspection. For example, the authors [6] have used this approach to determine lower bounds on the execution time of a network for computing the Cholesky decomposition.

## V. Systaltic Arrays

In this section the model developed in the previous sections will be used to show that a feedback technique for synchronizing systolic arrays will work properly (for a survey of systolic arrays see [4]). To set the stage, consider the simple systolic array, shown in Fig. 1, for computing an inner product of two vectors. The components of the vectors are generated in the processors labeled $x_{i-1}$ and $y_{i-1}$. They are passed onto the processor labeled $p = x_i * y_i$ where their product is formed. The product then goes to a fourth processor, labeled $s = s + p$, where the inner product is accumulated.

As they are usually conceived, systolic arrays require global control. For example, in the inner product array the data generators must release $x_{i-1}$ and $y_{i-1}$ at exactly the time the product processor releases $p = x_i * y_i$. Although this kind of global control is easy enough to effect in a toy array like the one in Fig. 1, it is more difficult in the two-dimensional arrays that are being devised to solve problems in numerical linear algebra.

A partial solution to this problem comes from the observation that a systolic array can be modeled as a computational network in an obvious way. It then follows from determinacy and the halting theorem that global control is not required for the array to compute the correct results. We could instead allow each processor to fire when its data become available. However, this raises the possibility of a quickly firing node sending data to another, slower node at a rate faster than the slower node can consume it. Even providing the slower node with a queue will not save this situation, since the amount of excess data is potentially unbounded.

The problem of data accumulation can be solved by placing what will be called *diastolic control elements* between the processors. Fig. 2 shows such an element, labeled $D$, between two processors labeled $M$ and $N$. It is assumed that $M$ originates data which $N$ needs for its computation. The element consists of an intelligent buffer, data lines from $M$ to $D$ and from $D$ to $N$, and control lines from $N$ to $D$ and from $D$ to $M$ [labeled $(N, D)$ and $(D, M)$]. A signal on $(D, M)$

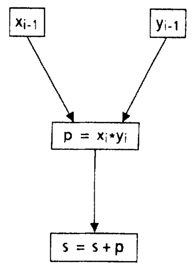

Fig. 1. Systolic array for the inner product.

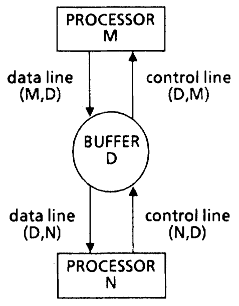

Fig. 2. Diastolic control element.

means that the buffer is empty. Although the processor $M$ can compute as much as is necessary, $M$ must wait for a signal from the control line to send information to $N$, at which point it transmits the information not directly to $N$ but to the buffer $D$. Likewise, a signal on $(N, D)$ means that $N$ is ready to receive data. If $D$ has data, it must wait for a signal on $(N, D)$ to transmit it, after which it puts a signal on $(D, M)$ to indicate that the buffer is empty. On receiving the data, $N$ drops the signal on $(N, D)$ until it is ready to process more data.

This method of control lends itself to a suggestive nomenclature. Systolic arrays were named from an analogy with a beating heart; as the heart pumps blood, so systolic arrays pump data. However, the systole is only the contractive phase of the heart's beat. The expansive phase is called the diastole, and the adjective systaltic refers to the heartbeat as a whole. Diastolic control elements are so called because they force processors to rest in sort of a diastole after they have received data and performed their computations. To continue the analogy, a systolic array with diastolic control will be called a *systaltic array*.

We claim that any systolic network can be converted to a systaltic network that runs without global control. Physically the conversion amounts to placing diastolic control elements between processors as described above. The processors must also be modified to send and accept the control signals associated with the diastolic control elements and to fire whenever the data they need are available.

To prove that the modification works, we must establish two things. First, that the resulting systaltic network computes the same results as the original systolic network. Second, that there is no buildup of data. The model of parallel computations developed previously will now be used to show that both are true.

The first step is to cast the notion of systolic network in terms of computational networks and execution sequences. It turns out that the essence lies not so much in the network as in the way the nodes are executed. Hence, a busy execution sequence $(E, s(0))$ will be called *systolic* if there are integers $0 = b_0 < b_1 < b_2 \cdots$ such that

1) There is no path in the precedence graph between any two elements of $B_i = \{E^{b_{i-1}+1}, \cdots, E^{b_i}\}$.

2) Elements in $B_i$ have connections in the precedence graph only *from* elements in $B_{i-1}$ and only *to* elements in $B_{i+1}$.

The first property in this definition implies that all the elements of the set $B_i$ can be executed in parallel (call this execution the $i$th beat). Moreover, no output they generate can be consumed during the $i$th beat. The second property assures that all data generated in a beat will be consumed in the next beat. These two properties characterize the systolic arrays in the literature.[1]

The next step in the verification that the systaltic network works is to insert nodes into the systolic network that correspond to the diastolic control elements. To simplify the discussion, it will be convenient to assume that all messages in the systolic execution sequence are of length one. This is no essential restriction; since the data space $D$ has not been specified, a data item on a queue could be anything from a single bit to a complicated message.

Let the arc $(M, N)$ belong to the network. Add a diastolic control node $D$ by deleting the arc $(M, N)$ and adding data arcs $(M, D)$ and $(D, N)$ and control arcs $(N, D)$ and $(D, M)$. The states of $D$ are contrived so that $D$ can fire only when there is a data item in $Q(M, D)$ and one in $Q(N, D)$. When $D$ fires, it places the item in $Q(M, D)$ on $Q(D, N)$ and places an item on $Q(D, M)$ to signal that the buffer is empty.

The states of $M$ and $N$ must also be modified. If $M$ is in a state whose firing would send a message along the arc $(M, N)$, then set the threshold of the arc$(D, M)$ to one, so that $M$ can fire only when the buffer in $D$ is ready to receive an item. If $N$ is in a state such that its *next* firing would consume an item from $M$,[2] let its firing place an item on $Q(N, D)$, so that $D$ can pass the item on.

The systolic execution sequence for the array must now be modified to account for the diastolic control nodes. For the initial state, let $M$ and $N$ be any pair of nodes with a diastolic control node $D$ between them. Let $Q(D, N)$ take the contents of the old $Q(M, N)$ in the initial state, and initialize $Q(M, D)$, $Q(D, M)$, and $Q(N, D)$ so that they are empty. Now augment the execution sequence so that after each beat all diastolic control nodes that are ready fire.

It is evident that the systaltic network is equivalent to the original systolic network. The only difference between the execution of the two networks is that the systaltic network transfers the data generated in a beat to the diastolic control nodes. These then fire in a diastole, transferring the data to where it would originally have gone. It now follows from the halting theorem that the systaltic network can execute in any order and still produce the same results as the systolic array. In particular, processors can fire whenever their data are ready.

It also easy to see that there can be no data accumulation in the systaltic network; in fact, there can be at most one item on a queue at any time in any execution sequence. For consider two nodes $M$ and $N$ with a diastolic control node $D$ between them. Since $M$ can send at most one item along the arc $(M, D)$ and since $D$ consumes one item from $Q(M, D)$ when it fires, the only way $Q(M, D)$ can have more than one item is for $M$ to fire twice, sending an item to $D$ each time, while $D$ fires not at all. But after the first firing of $M$, $Q(D, M)$ is empty and remains so until $D$ fires, which prevents $M$ from sending another item to $D$—a contradiction. A similar argument shows that the queue $Q(D, M)$ can contain no more than one item of data.

Thus, we have shown how any systolic array can be replaced by a systaltic array that requires no global synchronization for its correct functioning. There are two comments to be made about this construction.

First, synchronization by diastolic control elements is not confined to communication within networks; they can also be used to connect systaltic networks that would ordinarily run at different speeds. This has the important consequence that systaltic arrays can be designed without detailed consideration of timing in the larger environment in which they will operate.

Second, although the diastolic control element is a conceptual entity that could be implemented physically and placed between two processors, its functions can also be dividied between two existing processors. Indeed, some of the parts may become identical with registers used by the processors for other purposes, and the control may undergo simplification. For example, in an acyclic network the buffer $D$ can be identified with an input register of the target processor $N$, and $N$ can send back the control message to $M$ whenever it has finished using the registers. It is the same diastolic control element, but the hardware implementing the element has become inextricably mixed with the hardware of $N$.

[1] It is interesting to note that in what follows we may replace the second property by the requirement that at no time does any input queue have more than one element. This allows data to skip beats.

[2] Note that this can be determined from the state transition function evaluated at the current state.

## References

[1] J. C. Brown, "Formulation and programming of parallel computations: A unified approach,", in *Proc. 1978 Int. Conf. Parallel Processing,* D. Degroot, Ed. Washington, DC: IEEE Computer Society Press, 1985, pp. 624-631.

[2] W. M. Gentleman, "Using the harmony operating system," Nat. Res. Council of Canada Tech. Rep. ERB-966, 1985.

[3] R. Karp and R. Miller, "Properties of a model for parallel computations: Determinacy, termination, queuing," *SIAM J. Appl. Math.,* vol. 14, pp. 1390-1411, 1966.

[4] H. T. Kung, "Why systolic architectures?" *IEEE Computer.,* vol. 15, pp. 37-46, 1982.

[5] D. P. O'Leary and G. W. Stewart, "Data-flow algorithms for parallel matrix computations," *Commun. ACM,* vol. 28, pp. 840-853, 1985.

[6] D. P. O'Leary and G. W. Stewart, "Assignment and scheduling in parallel matrix factorization," *Linear Algebra Appl.,* vol. 77, pp. 275-299, 1985.

[7] D. P. O'Leary, G. W. Stewart, and Robert van de Geijn, "DOMINO: A message passing environment for parallel computation," Univ. Maryland Comput. Sci. Tech. Rep. TR-1648, 1986.

[8] C. Rieger, "ZMOB: Hardware from a user's viewpoint," in *Proc. IEEE Comput. Soc., Conf. Pattern Recognition Image Processing,* 1981, pp. 399-408.

[9] C. L. Seitz, "The cosmic cube," *Commun. ACM,* vol. 28, pp. 22-33, 1985.

# Decoding of DBEC–TBED Reed–Solomon Codes

ROBERT H. DENG AND DANIEL J. COSTELLO, JR.

*Abstract*—**A problem in designing semiconductor memories is to provide some measure of error control without requiring excessive coding overhead or decoding time. In LSI and VLSI technology, memories are often organized on a multiple bit (or byte) per chip basis. For example, some 256K bit DRAM's are organized in 32K × 8 bit-bytes. Byte-oriented codes such as Reed–Solomon (RS) codes can provide efficient low overhead error control for such memories. However, the standard iterative algorithm for decoding RS codes is too slow for these applications.**

**In this correspondence we present a special decoding technique for double-byte-error-correcting (DBEC), triple-byte-error-detecting (TBED) RS codes which is capable of high-speed operation. This technique is designed to find the error locations and the error values directly from the syndrome without having to use the iterative algorithm to find the error locator polynomial.**

*Index Terms*—**Byte error correction and detection, byte-organized memory systems, error control coding, Reed–Solomon codes, VLSI memory systems.**

Manuscript received July 16, 1984; revised September 9, 1986. This work was supported by NASA under Grant NAG 2-202.

The authors are with the Department of Electrical and Computer Engineering, University of Notre Dame, Notre Dame, IN 46556.

IEEE Log Number 8714096.