NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

University Libraries
Carnegie Mellon University
Pittsburgh PA 15213-3890

CMU-CS-81-130

510.7
C28c
81-130
c.2

Systolic Algorithms for Running Order Statistics in Signal and Image Processing

Allan L. Fisher

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

July, 1981

Copyright © 1981 Allan L. Fisher

This research was supported in part by the Office of Naval Research under Contracts N00014-76-C-0370, NR 048-659 and N00014-80-C-0236, NR 044-422, in part by the National Science Foundation under Grant MCS 78-236-76 and a Graduate Fellowship, and in part by the Defense Advanced Research Projects Agency under Contract F33615-78-C-1551 (monitored by the Air Force Office of Scientific Research).

Abstract

Median smoothing, a filtering technique with wide application in digital signal and image processing, involves replacing each sample in a grid with the median of the samples within some local neighborhood. As implemented on conventional computers, this operation is extremely expensive in both computation and communication resources. This paper defines the running order statistics (ROS) problem, a generalization of median smoothing. It then summarizes some of the issues involved in the design of special purpose devices implemented with very large scale integration (VLSI) technology. Finally, it presents algorithms designed for VLSI implementation which solve the ROS problem and are efficient with respect to hardware resources, computation time, and communication bandwidth.

1. Introduction

Median smoothing [16] is a filtering operation, widely used in digital signal and image processing, which involves replacing each sample value with the median of the values found within some neighborhood of itself. In the two dimensional image processing case, this typically means taking the median of 25 to 100 numbers for each of 10^5 to 10^6 pixels. As a result, the computation and memory communication resources required to implement this operation on a conventional computer are very large.

The development of very large scale integration (VLSI) technology has made feasible the production of relatively inexpensive, highly parallel special-purpose computing engines [4, 10] for the implementation of computationally demanding operations. VLSI algorithms have been designed and some prototypes implemented for such applications as pattern matching [3], convolution in image processing [8], and relational database operations [6]. This paper presents efficient VLSI algorithms which solve the running order statistics (ROS) problem, a generalization of median smoothing.

Section 1.1 defines the running order statistics problem, and mentions some of its applications. Section 1.2 explains the principles underlying the algorithm designs presented, and Section 1.3 describes the approach used in analyzing the complexity of such algorithms. Section 2 describes a VLSI algorithm for the one dimensional signal processing case. Section 3 gives an algorithm for the two dimensional image processing case, and describes the extension of that algorithm to problems of arbitrary dimension. Finally, Section 4 reviews some of the important features of the algorithms described.

1.1. The Running Order Statistics Problem

The running order statistics problem is a generalization of the median smoothing problem. Median smoothing has been widely used in speech and image processing [1, 13], especially in the elimination of *outliers*, spurious values caused by noise or other technical error. Median filtering has several properties which are particularly useful in image processing. One is that the operation does not introduce intermediate pixel values not found in the original image, as convolution methods may, and hence preserves sharp region boundaries. Also, since the median of a group of numbers is generally insensitive to the presence of a small number of outliers, median smoothing is not subject to the problem of *artifact ringing*, the propagation of an erroneous value through a region of an image. For this reason, median smoothing is sometimes used as a preprocessing step before an image is filtered by other means.

The median smoothing problem can be generalized by considering order statistics other than the median; while the median of k numbers is the one having rank $(k + 1)/2$ (for odd k), we can consider asking for the element having some arbitrary rank r. We will say that an instance of the running order statistics problem has

dimension n if the array of numbers to be filtered has n dimensions. For the sake of simplicity, we will require that the neighborhood around each element over which statistics are taken be in the form of an n-dimensional hypercube with odd edge length centered on that element, and we will say that an instance is of *order k* if the hypercube has edge length k.

Formally, the *n-dimensional running order statistics problem of order k*, for odd k, is: Given an n-dimensional array of values $[a_{i_1, i_2, \ldots, i_n}]$ whose size in dimension d is s_d (so that $1 \leq i_d \leq s_d$), and a set of ranks R; compute, for each $r \in R$ and each index tuple $[i_1, i_2, \ldots, i_n]$ such that each subscript i_d is in the range $(k + 1)/2 \leq i_d \leq s_d - (k - 1)/2$, the element having numerical rank r among the set of elements $\{a_{j_1, j_2, \ldots, j_n}$ such that $| j_d - i_d | \leq (k - 1)/2$ for each dimension $d\}$. Note that this formulation begs the question of how to handle elements which are too close to an edge of the array to be at the center of a complete hypercube; Tukey [16] gives a number of possible solutions to this problem.

Section 2 describes a VLSI algorithm for the one dimensional ROS problem. The structure described is based on the same idea as Leiserson's systolic priority queue [9], and is presented here mainly in order to lay the groundwork for the discussion of the second algorithm. The second algorithm, described in Section 3, solves the two dimensional ROS problem, and may be extended to handle problems of arbitrary dimension.

1.2. Systolic Algorithms

The chief performance advantage offered by VLSI technology is the availability of massive parallelism, achieved by the harnessing together of many processing units. The exploitation of this potential requires more than the creation of the raw processing power to solve a problem. It is also necessary to provide for data transfer between individual processing units and between the processing units and mass storage. The need for mass storage persists even in the face of advances in miniaturization; technological forecasts [11, 12] make it clear that it will not be possible to have a number of processors comparable to the number of data items used by an application at any time in the foreseeable future. Thus, the communication architecture of a system is a dominating factor in its performance.

Figure 1 schematizes a common computer system structure, the "von Neumann" architecture, in which a processing unit, **P**, receives data and instructions from a memory unit, **M**, and returns the results of its computations to the memory. This architecture has the disadvantage that, for most computations, the operation rate achievable is limited not only by the speed of the processor but also by the bandwidth of the processor-memory communications link. This limitation is commonly referred to as the "von Neumann bottleneck".

One solution to this problem is the concept of *systolic arrays* [5, 7]. A systolic array is a collection of

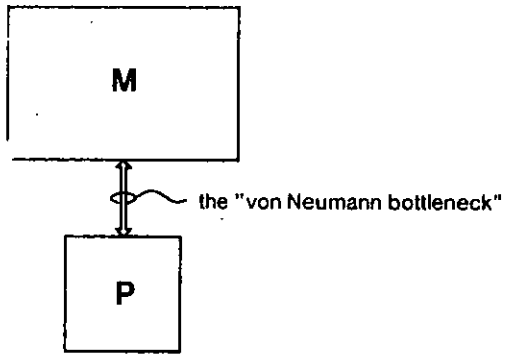

Figure 1: Standard von Neumann system architecture

relatively simple processing units, either all of the same type or a mixture of a few different types, which are connected by a simple communications network and operate in parallel[1]. The performance advantage of a systolic array architecture, as illustrated in Figure 2, is that it uses each datum retrieved from memory many times without having to store and retrieve intermediate results, thus potentially allowing speedups relative to memory bandwidth which are proportional to the number of processors used.

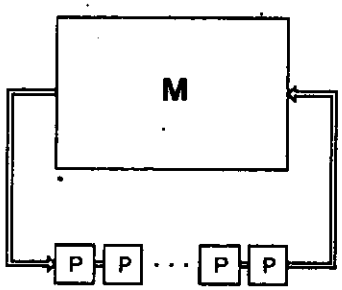

Figure 2: Systolic system architecture

In order for fabrication of a systolic system to be reasonable in practice, the communication structure which connects the processors must be simple and regular. In particular, a linear pipeline structure, as illustrated in Figure 2, has several important features. First, it requires memory bandwidth which is independent of the size of the array, as contrasted with a two dimensional structure. Second, a large pipeline can be constructed simply by concatenation of smaller pipelines. Finally, since the interface of a linear pipeline to the outside

[1]The examples in this paper will be discussed in terms of fully synchronous operations for the sake of simplicity, but they could be broken up into self-timed segments [14] communicating by some protocol while maintaining the same asymptotic performance achieved synchronously.

world is of bounded size, increases in integrated circuit density can be exploited while retaining constant chip pinout by laying out pipeline segments on chips in zigzag fashion.

1.3. Complexity Measures for VLSI

In order to obtain size and performance measurements for the machines proposed here, we require a model of VLSI technology which is amenable to asymptotic complexity analysis. Thompson [15], among others, proposes such a model. The analyses carried out here rest on a simplified version of Thompson's model. The pertinent features of the model are as follows:

- Logic gates of constant fan-in and fan-out require constant area and switch in constant time.
- A constant-width wire of any length can be driven in constant time by drivers which can be implemented in area proportional to the wire's length. In particular, this means that wires occupy area proportional to their length.

Given these elements, we can obtain asymptotic measures of the resources required to apply an algorithm to a problem of some specified size. We will be concerned with the *area* required for the implementation of an algorithm, which translates roughly to its hardware cost, and with the *time* required to perform the algorithm.

2. Algorithm for the One Dimensional ROS Problem

This section describes a VLSI algorithm which, for the order k one dimensional running order statistics problem, yields any particular running order statistic of a vector of length m in time $O(m)$, while occupying area $O(k)$. The algorithm makes a left-to-right sweep over the input sequence, computing the required order statistic of each contiguous subsequence of length k. The hardware structure of the algorithm is a pipeline consisting of k cells, which hold the k values under consideration at each step. The idea is to keep the k elements in order, so that elements having particular ranks may be extracted from corresponding positions in the pipeline, and to update the contents of the pipeline at step n by deleting a_{n-k} and inserting a_n.

The updating is effected by passing messages from cell to cell down the pipeline; at each step, the left end of the pipeline receives a series of messages, and passes them to its right. First, a message is sent down the line which seeks out a_{n-k}, the element to be deleted from the array, and causes it to be deleted. This is followed by a message containing a_n, the new value to be inserted, which passes down the line until it reaches its appropriate position in order, at which point the value is inserted. High throughput is achieved by pipelining the processing of the messages, so that many messages are active in the pipeline at one time, each in a separate cell.

2.1. Description of the Algorithm

The algorithm consists of a linear pipeline of k cells. Each cell deals with two pieces of information, a *stored value* and a *message*. The *message*, in turn, has two components, an *action* and a *message value*. In each cycle of the machine, each cell receives a *message* from the left, updates its *stored value* (possibly consulting its right neighbor's *stored value*), and passes a *message* to the right. In essence, the algorithm works by sending, at each step, a message down the pipeline to delete the element which becomes obsolete at that step, followed by a message to add the new element. Thus at step n, the $n-k^{th}$ element is deleted, and the n^{th} is added. A deletion message travels down the pipe until its message value matches the stored value of the receiving cell, at which time that cell overwrites its stored value with that of its right neighbor, and passes a message to the right which will cause the remaining stored values in the pipe to shift left. An insertion message moves to the right until it reaches a cell whose stored value matches or exceeds the message value, at which time that cell replaces its stored value with the message value, and sends to the right a message to insert its old stored value.

The algorithm uses four action types: **delete**, which specifies an element to be deleted; **pull**, which causes a cell to perform a left shift; **insert**, which specifies a new element to be added; and **wait**, meaning "do nothing", which is necessary to provide a gap between an insertion and a deletion. The computation that each cell performs in each time step depends on *action* as follows:

delete: If *message value* does not match *stored value*, pass on the same message. If they do match, set *stored value* to the *stored value* of the cell to the right, and pass the message **[pull]**.

pull: Set *stored value* to the *stored value* of the cell to the right, and pass the message **[pull]**.

insert: If *message value* > *stored value*, pass along the same message. Otherwise, set *stored value* to *message value*, and pass the message **[insert** old *stored value***]**.

wait: Pass the message along.

Each step in the execution of the algorithm takes three machine cycles, in each of which a message is injected into the left end of the pipeline. At step n, the sequence of messages injected is as follows:

[delete a_{n-k}**]**

[insert a_n**]**

[wait]

Initially, the pipeline is primed by setting each *action* to **wait**, each *stored value* to the maximum value possible, and taking $a_{1-k}, a_{2-k}, \ldots, a_0$ to have this maximum value. During the operation of the algorithm, the rightmost cell receives a maximum value when it attempts to read its right neighbor's value. The element

having rank r among $a_{n-k+1}, \ldots, a_n$ can be read from cell r during machine cycle $3n+r-1$, while it holds the wait message issued immediately after the message **[insert a_n]**. The rank of the item to be read could be either fixed in hardware or selected in a previous set-up phase. The value itself could be either put onto an output bus or passed, cell by cell, to the right end of the pipeline as an additional message component. More than one ROS could be computed simultaneously at the cost of extra hardware and time. One way to achieve this would be to use the **count-result** mechanism discussed in Section 3.

A check of the correctness of the algorithm can be carried out by checking two assertions. The first is that the abstract sequence of operations specified by the messages injected yields the desired result; that is, that the processing of a particular message, in the absence of other messages, leaves the pipeline in the intended state. The second condition that the pipelining of the operations is carried out correctly, in that each message causes each cell to perform the same computation in both the pipelined and non-pipelined cases.

In order to check the correctness of the operation sequence, consider the computation performed by an ordinary uniprocessor simulating the systolic algorithm by simulating, for each message in turn, the effects of that message as it travels the entire length of the array; that is, each message is propagated through the array before the simulation of the next message is begun. In the course of such a simulation, the element holding rank r among $a_{n-k+1}, \ldots, a_n$ is held in cell r after the propagation of the message **[insert a_n]**.

Turning to the issue of pipelining, note that only the **delete** and **pull** messages can possibly cause problems, since neither of the other message types causes a cell to refer to the values stored by any other cell. However, such a message is always immediately preceded by a **wait** message; thus the cell to the right, whose stored value will be accessed, is in the state that it would reach after the preceding **insert** message had been propagated in the uniprocessor simulation. Therefore, the results of the pipelined computation are the same as those of the serial computation.

Complexity analysis of a VLSI algorithm is concerned with two measures: the area required to implement the algorithm, and the time that it takes to perform a computation. In this case, assuming that the precision of the numbers to be processed is fixed, so that each comprises a constant number of bits, the area required by each cell is a constant, independent of k. Thus the area required for the entire algorithm is proportional to k, since it consists of k constant-size cells. The time required to process a sequence of numbers is equal to the product of the number of cycles required to pass the sequence through the machine and the time required to perform a machine cycle. A sequence of length m requires $3\times(m+k)$ machine cycles, and cycle time is constant, regardless of the value of k. Thus, assuming that $m>>k$, a sequence of m numbers can be processed in time $O(m)$.

2.2. An Example

Figure 3 shows five stages in the operation of the algorithm for order five, when presented with input containing the subsequence

. . . , 8, 5, 4, 6, 2, 9, 1, 3, . . .

beginning just after the message **[insert 1]** has been injected into the left end of the pipeline.

The behavior caused by **delete** messages is demonstrated by the message **[delete 5]** which is in the second cell from the left in the top snapshot; the message is passed to the right until it reaches a cell whose stored value is 5 (the third cell, in this case), at which point the 5 is overwritten with the value to the right. A **pull** message is then sent to the right; this causes the remaining elements in the pipeline to shift to the left. At the right end of the pipeline, a maximum value is shifted in as "filler", to be shifted out later.

Two cases in the example summarize the effect of **insert** messages. The message **[insert 1]** which is in the leftmost cell in the top snapshot causes that cell to set its stored value to 1, and to pass its previous stored value of 2 to the right to be inserted. This effect ripples down the pipeline, essentially causing the elements stored to shift to the right. When an **insert** message reaches the rightmost cell of the pipeline, as the message **[insert 9]** does in the second snapshot, the value held in the message replaces the maximum value left in that cell by the previous **delete** or **pull** message.

3. Algorithm for the Two Dimensional ROS Problem

This section presents an algorithm for the two dimensional running order statistic problem which may be extended to handle ROS problems of arbitrary dimension. In the two dimensional case, the algorithm yields a set of s order statistics of order k for a matrix with m elements in time $O(ms\log\log k)$, while occupying area $O(k^2\log k)$. Like the algorithm presented for the one dimensional problem, this algorithm is based on a linear array of cells, down which messages are passed to maintain an ordered sequence of values.

As in the algorithm of Section 2, many messages are processed simultaneously, in separate cells. The algorithm for the two dimensional problem has an additional level of parallelism, however, in that it operates on data belonging to k squares of size $k\times k$ simultaneously. Essentially, the algorithm sweeps a rectangular window of $2k-1$ rows and k columns across the array, and at each step produces order statistics for the k overlapping $k\times k$ squares contained in such a rectangle.

Since the algorithm works on more than one square at a time, each array value is tagged with a row number, ranging from 1 to $2k-1$, in order to make it possible to calculate to which squares it belongs. Also, since the mixing of values from different squares makes it impossible to compute results just by reading the

message action	insert	delete	wait	insert	delete
message value	1	5		9	8
stored value	2	4	5	6	8

message action	wait	insert	delete	wait	insert
message value		2	5		9
stored value	1	4	5	6	max

message action	delete	wait	insert	pull	wait
message value	4		4		
stored value	1	2	6	6	9

message action	insert	delete	wait	insert	pull
message value	3	4		6	
stored value	1	2	4	9	9

message action	wait	insert	delete	wait	insert
message value		3	4		9
stored value	1	2	4	6	max

Figure 3: Example of the one dimensional algorithm

contents of particular cells at particular times, order statistics are gathered by special messages which count the number of elements in a given square up to a specified rank, then pass the value having that rank to the end of the pipeline as the result.

3.1. Description of the Algorithm

As before, the structure on which the algorithm is based is a pipeline. For the two dimensional case, the pipeline consists of $2k^2-k$ cells, each divided into two sections, a *stored* section and a *message* section. As before, in each machine cycle each cell in the pipeline examines a message from its left neighbor, updates its stored data (possibly consulting its right neighbor's stored data), and sends a message to its right neighbor to be processed by it during the next cycle.

Each cell's *stored* data has two components: a *row*, which tells from which of the $2k-1$ rows of data currently being considered the value was drawn; and a *value*, representing one of the a_{ij}. A *message*, on the other hand, has four components: an *action*, specifying the computation to be performed by a cell; a *count*, which is used in gathering statistics; and a *row* and a *value*, as in the stored data.

The algorithm makes use of six different actions: **delete**, **pull**, **insert**, and **wait**, all of which operate similarly to their counterparts in the one dimensional algorithm; **count**, which is sent through the pipeline to find the element within a $k \times k$ square which has a given rank; and **result**, which is used to pass the result of a counting operation through to the end of the pipeline. The computations specified by each *action* are as follows:

delete: If *message row* and *message value* match *stored row* and *stored value*, then set *stored row* and *value* to those of the cell to the right, and pass the message **[pull]**. Otherwise, pass the message along unchanged.

pull: Set *stored row* and *value* to those of the cell to the right, and pass along the message **[pull]**.

insert: If *message value* > *stored value*, pass the message along unchanged. Otherwise, set *stored row* and *value* to those of the *message*, and pass the message **[insert** old *stored row*, old *stored value***]**.

wait: Pass the message along.

count: If *message row* $-k <$ *stored row* $\leq$ *message row* and *message count* > 1, then pass the message **[count** *message count* -1, *message row***]**. If *stored row* is in this range and *message count* $= 1$, then pass the message **[result** *stored value***]**.

result: Pass the message along.

When applied to a matrix with r rows and c columns, the algorithm makes r/k passes across the columns of

the matrix, reading rows $(n-1)\times k+1$ through $(n+1)\times k-1$ and yielding order statistics for the squares whose first rows are numbered $(n-1)\times k+1$ through $n\times k$ during the n^{th} pass. In other words, it reads rows in the intervals $1\,..\,2k-1$, $k+1\,..\,3k-1$, ..., calculating order statistics for the squares whose first rows fall in the intervals $1\,..\,k$, $k+1\,..\,2k$, and so forth. Each pass across the columns takes c steps, in each of which $2k-1$ numbers are deleted from the queue, $2k-1$ are added, and $s\times k$ **count** messages are injected in order to find s order statistics in each of the k squares being scanned. As in the one dimensional algorithm, the queue is initialized with maximum values before each pass is begun.

A check of the correctness of the algorithm can be performed in the style of Section 2.1. Again, consideration of a uniprocessor simulation makes it apparent that the sequence of operations applied compute the correct results, and the demonstration of the correctness of the algorithm's pipelining is identical.

The complexity of the algorithm, though, is more complicated than that of the algorithm for the one dimensional case, because each cell must handle numbers ranging up to $O(k^2)$. In particular, each cell must be capable of performing comparison and subtraction operations on these numbers. By encoding the numbers in 2's-complement binary notation, both of these operations can be expressed in terms of addition and testing for zero value of $O(\log k)$-bit numbers. Brent and Kung [2] describe a general adder design which yields b-bit adders requiring area $O(b)$ and time $O(\log b)$. Substituting $\log k$ for b, the necessary additions can be performed in area $O(\log k)$ and time $O(\log\log k)$. Testing for zero value can be performed with a binary tree of OR gates in area $O(\log k)$ and time $O(\log\log k)$. Thus, since each of $2k^2-k$ cells requires area proportional to $\log k$, the entire linear array requires area $O(k^2\log k)$. The time to process m numbers, computing s order statistics for each square, can be calculated as $O(ms)$ machine cycles multiplied by a cycle time proportional to $\log\log k$, yielding the result $O(ms\log\log k)$.

The algorithm may be extended to handle problems of any dimension n by using a $(2k-1)^{n-1}k$ cell pipeline, and sweeping the array with a hyperwindow which has width k in the direction of the sweep and width $2k-1$ in every other direction. At each step in the sweep, the algorithm would read $(2k-1)^{n-1}$ values and produce k^{n-1} sets of order statistics. Each value in the pipeline would be accompanied by $n-1$ numbers indicating to which hypercubes, among the k^{n-1} contained in the window, the value belongs.

3.2. An Example

Figure 4 demonstrates the operation of the algorithm for order 3 (that is, taking order statistics over each 3×3 square), when presented with input containing the block

action	delete	count	count	count	insert	delete	wait	insert	pull	wait	insert	delete	wait	insert	pull
count		5	4	2											
row	1	5	4	3	5	5		4			1	3		2	
value	33				32	31		36			35	43		41	
row	2	1	3	3	5	2	5	1	1	4	3	4	5	3	3
value	19	21	23	26	28	30	31	33	33	34	37	38	39	43	43

action	insert	delete	count	count	count	insert	delete	wait	insert	pull	wait	insert	delete	wait	insert
count			5	3	1										
row	1	1	5	4	3	5	5		4			3	3		3
value	22	33				32	31		36			37	43		43
row	2	1	3	3	5	2	5	1	4	4	1	4	5	2	
value	19	21	23	26	28	30	31	33	34	34	35	38	39	41	max

action	wait	insert	delete	count	count	count	insert	pull	wait	insert	pull	wait	insert	delete	wait
count				4	2	1									
row		1	1	5	4	3	5			4			4	3	
value		22	33				32			36			38	43	
row	2	1	3	3	5	2	1	1	4	1	1	3	5	2	3
value	19	21	23	26	28	30	33	33	34	35	35	37	39	41	43

action	delete	wait	insert	delete	count	count	result	insert	pull	wait	insert	pull	wait	insert	delete
count					3	2									
row	2		1	1	5	4	3	1			4			5	3
value	41		22	33			30	33			36			39	43
row	2	1	3	3	5	2	5	4	4	1	3	3	4	2	3
value	19	21	23	26	28	30	32	34	34	35	37	37	38	41	43

action	insert	delete	wait	insert	delete	count	count	result	insert	pull	wait	insert	pull	wait	insert
count						2	1								
row	2	2		3	1	5	4	3	4			3			2
value	27	41		23	33			30	34			37			41
row	2	1	1	3	5	2	5	1	1	1	4	4	4	5	
value	19	21	22	26	28	30	32	33	35	35	36	38	38	39	max

Figure 4: Example of the two dimensional algorithm

17 33 21 35 22
22 41 19 30 27
43 37 26 23 29
18 34 38 36 40
31 39 28 32 42.

The algorithm moves from left to right across the data, and at each column position reads the new data and deletes the old data from top to bottom. In this example, the only order statistic considered is the median (thus each **count** message has its *count* field initially set to 5). The figure illustrates five machine cycles, beginning just after the message **[delete** *row* 1 *value* 33**]** has been injected.

The handling of the **insert**, **delete**, **wait**, and **pull** messages is essentially the same as before. The actions specified by the **count** and **result** messages are illustrated by the progress of the **count** message which appears in the fourth cell from the left of the first snapshot. The stored value of 26, drawn from row 3, falls within the range (rows 1 to 4) of the **count** message. Thus, cell 5 of the second snapshot shows the same message with its count reduced by one. The stored value in this cell is from row 5, out of the range of this message, so the same message is passed unchanged to cell 6, as seen in the third snapshot. The value stored in cell 6 is drawn from row 2, within the range of the message; since the count is down to 1, this value is passed out in a **result** message, seen in cell 7 of the fourth snapshot.

4. Conclusion

In addition to providing cost-effective solutions to a computationally difficult family of problems, the algorithms described in this paper illustrate some important issues in the design of special-purpose parallel computing devices. First, because of their linear structure, they conserve memory bandwidth. The number of bits transferred in each cycle by the algorithm for the two dimensional problem grows as $\log k$, rather than at least k as in the case of a two dimensional array of processors. Thus a large device can run with essentially the same memory bandwidth as a small device without wasting any of its processing power. Also, each value is retrieved from memory a small constant number of times (no more than 2 if a $2k^2 - k$ cell shift register is used to keep track of values to be deleted), so the total external communication required by the algorithm is small. This feature is the result of an economy of memory reference which is central to systolic algorithms: when a value is read from memory, at least half of the computations which depend on that value are performed. This is accompanied by a dual economy of computation: no two input values are ever compared more than a small constant number of times.

Acknowledgments

Thanks are due to M. J. Foster, H. T. Kung, P. L. Lehman, and S. W. Song for helpful criticism, and to H. T. Kung for suggesting the problem.

References

[1] Andrews, H.C.
Monochrome digital image enhancement.
Applied Optics 15(2):495-503, February, 1976.

[2] Brent, R. P. and H. T. Kung.
A regular layout for parallel adders.
Technical Report CMU-CS-79-131, Carnegie-Mellon University, Computer Science Department, June, 1979.

[3] Foster, M. J. and H. T. Kung.
The design of special-purpose VLSI chips.
Computer Magazine 13(1):26-40, January, 1980.

[4] Kung, H. T.
Let's design algorithms for VLSI systems.
Technical Report CMU-CS-79-151, Carnegie-Mellon University, Computer Science Department, January, 1980.

[5] Kung, H. T.
Notes on VLSI Computation.
To be published by Cambridge University Press.

[6] Kung, H. T. and P. L. Lehman.
Systolic (VLSI) arrays for relational database operations.
In *Proceedings of ACM SIGMOD 1980 International Conference on Management of Data*, pages 105-116. Association for Computing Machinery, May, 1980.

[7] Kung, H. T. and C. E. Leiserson.
Systolic arrays (for VLSI).
In Duff, I. S. and Stewart, G. W. (editors), *Sparse Matrix Proceedings 1978*, pages 256-282. Society for Industrial and Applied Mathematics, 1979.
A slightly different version appears as Section 8.3 of Mead and Conway [10].

[8] Kung, H. T. and S. W. Song.
A systolic 2-D convolution chip.
Technical Report CMU-CS-81-110, Carnegie-Mellon University, Computer Science Department, March, 1981.
To appear in *Non-Conventional Computers and Image Processing: Algorithms and Programs*, Leonard Uhr (editor), Academic Press, 1981.

[9] Leiserson, Charles E.
Systolic priority queues.
Technical Report CMU-CS-79-115, Carnegie-Mellon University, Computer Science Department, April, 1979.

[10] Mead, C. A. and L. A. Conway.
Introduction to VLSI Systems.
Addison-Wesley, Reading, Mass., 1980.

[11] Moore, G. L.
Are we really ready for VLSI?
In C. L. Seitz (editor), *Proceedings of Conference on Very Large Scale Integration: Architecture, Design, Fabrication*, pages 3-14. California Institute of Technology, 1979.

[12] Noyce, R. N.
Hardware prospects and limitations.
In M. L. Dertouzos and J. Moses (editors), *The Computer Age: A Twenty-Year View*, pages 321-327. Institute of Electrical and Electronics Engineers, 1979.

[13] Rabiner, L. R., M. R. Sambur, and C. E. Schmidt.
Applications of a nonlinear smoothing algorithm to speech processing.
IEEE Transactions on Acoustics, Speech, and Signal Processing 23(6):552-557, December, 1975.

[14] Seitz, C. L.
System Timing.
Chapter 7 of Mead and Conway [10].

[15] Thompson, C. D.
A Complexity Theory for VLSI.
PhD thesis, Carnegie-Mellon University, Computer Science Department, August, 1980.

[16] Tukey, J. W.
Exploratory Data Analysis.
Addison-Wesley, Reading, Mass., 1977.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CMU-CS-81-130	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SYSTOLIC ALGORITHMS FOR RUNNING ORDER STATISTICS IN SIGNAL AND IMAGE PROCESSING		5. TYPE OF REPORT & PERIOD COVERED Interim 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Allan L. Fisher		8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0370
9. PERFORMING ORGANIZATION NAME AND ADDRESS Carnegie-Mellon University Computer Science Department Pittsburgh, PA. 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS		12. REPORT DATE July 1981 13. NUMBER OF PAGES 16
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Approved for public release; distribution unlimited

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)