A One-Way Linear Array Algorithm for the Median Filter

Matthias F.M. Stallmann

Center for Communications and Signal Processing
Computer Science Department
North Carolina State University

CCSP-TR-88/4

January 1988

Abstract

Linear array algorithms with one-way dataflow have many practical advantages, as well as being of interest as a theoretical model of computation. This paper presents a one-way linear array algorithm for the median filter (a two way algorithm is given by Fisher [4]). A variation of the algorithm is shown to have optimal area. Positive and negative results about the existence of one-way algorithms for more general filters are presented.

1 Introduction

Algorithms for linear arrays of processors have been studied extensively from both a practical and a theoretical point of view. Of particular interest are algorithms in which the data moves exclusively in one direction, called *one-way algorithms*. One-way algorithms have the following practical advantages [10].

- Faulty processors can be bypassed without affecting algorithm correctness.
- Successive problem instances can be pipelined to achieve optimum processor utilization.
- If not enough processors are available, the same result can still be obtained by recycling output data.
- One-way algorithms are usually simpler and easier to verify than their two-way counterparts.

From a theoretical point of view, the relative power of two-way versus one-way computation is a difficult open question (see [2]).

The model of computation is a linear array consisting of some number, say n, of processors (*cells*), numbered from 1 to n. During a single *time unit* cell i (for $i = 2, \ldots, n-1$) reads input from its neighbors (cells $i-1$ and $i+1$), performs a fixed number of arithmetic operations, and generates output for each neighbor (to be read during the next time unit). Cells 1 and n are special cases in that cell 1 reads input from an external host instead of its left neighbor and cell n produces output for the external host instead of its right neighbor. In a *one-way algorithm* cell i reads data only from its left neighbor, cell $i-1$ (or the external host), and produces output only for its right neighbor, cell $i+1$ (or the host). Several variations exist with respect to the memory capacity of each cell. If the local memory of each cell is fixed, the model is called a *linear iterative array* (see, e.g. [6] for a discussion of language recognition problems on this model). Unless otherwise specified, we assume that each cell has a fixed number of registers, with each register capable of storing a number in the range $i = 1, \ldots, n$. This assumption simplifies the description of the algorithms (methods for avoiding the $\Omega(\log n)$ bits per cell implicit in the assumption are pointed out whenever possible). We also assume that data in each cell is initialized to 0 (algorithms that require different cells to be initialized to different values can easily be

transformed, using n extra time units, into ones where all data is initialized to the same value [8,11]).

Previous results concerning two-way versus one-way algorithms are as follows. In each case the two way algorithm uses n cells and takes m time units.

- A two-way algorithm can be transformed into a one-way algorithm using $O(m)$ cells and taking $O(m)$ time units (note that $m \geq n$ in the model). [1]
- A two-way algorithm can be transformed into a one-way algorithm taking $O(m)$ time units on a *circular* array of n cells (see [3,8,11]).

Still open is whether the existence of a two-way algorithm implies the existence of a one-way linear array algorithm using only $O(n)$ cells when $m \gg n$. It is interesting to consider computations where m is large simply because the amount of input and output is large. This is the case with signal processing filters.

Definition. An *f-filter* of *order* n is a computation that transforms an infinite stream of inputs $x_0, x_1, \ldots$ into an infinite stream of outputs $y_0, y_1, \ldots$, where

$$y_i = f(x_i, \ldots, x_{i-n+1}, y_{i-1}, \ldots, y_{i-n+1})$$

(assume $x_i = y_i = 0$ or some other suitable fixed value when $i < 0$). A *finite f-filter* is an f-filter in which f does not depend on $y_{i-1}, \ldots, y_{i-n+1}$.

For example, a median filter and an FIR filter are finite f-filters (where f is the k-th largest of $x_i, \ldots, x_{i-n+1}$ or a weighted sum of $x_i, \ldots, x_{i-n+1}$, respectively), while an IIR filter is an f-filter which is not finite (in this case f is a weighted sum of $x_i, \ldots, x_{i-n+1}$ and $y_{i-1}, \ldots, y_{i-n+1}$).

Definition. A *fixed-throughput* algorithm for an f-filter of order n is a linear array algorithm that, for some constant c, receives one x_i every c time units and outputs one y_i every c time units. Moreover, y_i is produced after a delay of at most $O(n)$ time units after x_i is given as input.

Two-way fixed-throughput algorithms are known for the IIR filter [1] and the median filter [4] (these algorithms assume that cell 1 does both input and output but can easily be modified for our model), and a one-way fixed-throughput algorithm is known for the FIR filter [9]. Section 2 gives a one-way fixed-throughput algorithm on a linear array for the median filter. A modification of the algorithm is shown to have optimal area. Section 3 discusses the possibility of one-way fixed-throughput algorithms on linear arrays for other f-filters, giving both a positive and a negative result.

[1] A transformation requiring $\Omega(\log n)$ bits per cell is given in [8]. Transformations using a fixed number of bits are possible. For example, one can transform from a two-way iterative to two way cellular array and then to a one-way iterative array. The first transformation is easy, while the second is reported in [8] and independently in [3].

```
for i = 1 to input length do
    read i-th input, x_i
    global_data[i] := F_0(x_i)
    for j = 1 to number of cells do
        /* data[j] = data currently in cell j */
        global_data[i] := F(global_data[i], data[j])
        data[j] := G(global_data[i], data[j])
    end do
    y_i := F*(global data[i])
    y_i is the i-th output
end do
```

Figure 1: Generic form for a one-way array algorithm

2 The Median Filter Algorithm

It is not difficult to show that any one-way linear array algorithm can be expressed as a doubly nested loop having the form shown in Figure 1. If F, F_0, F^*, and G can be computed in constant time, any algorithm having this form can be converted into a one-way linear array algorithm. Iteration i, j of the inner loop is performed by cell j during time unit $i + j$ (note that the computation in iteration i, j of the inner loop depends only on the results of the computations in iterations $i, j-1$ and $i-1, j$). The computation of F_0 and F^* are simply a matter of encoding the input and decoding the output, respectively. This can be done either by the host or by cells 1 and n.

A similar observation about the relationship between linear array and sequential algorithms is made by Ibarra, Palis, and Kim [7]. The idea of mapping iteration i, j into cell j at time $i + j$ is a special case of more general algorithmic transformations (see, e.g. [5]).

The two-way median filter algorithm maintains the set $\{x_i, \ldots, x_{i-n+1}\}$ in sorted order with cell k holding the k-th largest element. Let $w_1, \ldots, w_n$ be the sorted sequence of $\{x_{i-1}, \ldots, x_{i-n}\}$ before x_i is read, and let $w_l = x_{i-n}$ (note that, before x_0 is read, $w_i = 0$ for $i = 1, \ldots, n$). The new sequence of w's ($w'_1, \ldots, w'_n = \{x_i, \ldots, x_{i-n+1}\}$ in sorted order) is generated by deleting w_l and inserting x_i into the proper position among the remaining w's. Let h be the smallest index (other than l) for which $x_i \geq w_h$, i.e. x_i must be inserted to the left of w_h to generate the new sequence. If $l > h$, x_i can be inserted while $w_h, \ldots, w_{l-1}$ is shifted to the right. But if $l < h$, $w_{l+1}, \ldots, w_{h-1}$ must be shifted to the left; this is where Fisher's algorithm requires two-way dataflow.

The need for two way dataflow is avoided in our algorithm because the sequence $w_1, \ldots, w_n$ is allowed to shift one position to the right after each new input. While the sequence shifts right, unoccupied cells to the left of w_1 can be used to maintain a backup copy

of the most recent inputs, also in sorted order. The algorithm uses $2n$ cells and is described in detail in Figure 2. At the beginning of iteration i of the outer loop cells 1 through $n+d$ (where $d = i \bmod n$) contain $v_1, \ldots, v_d, w_1, \ldots, w_n$, respectively, where $v_1, \ldots, v_d$ is the backup copy, $x_{i-1}, \ldots, x_{i-d}$ in sorted order, and $w_1, \ldots, w_n$ is $x_{i-1}, \ldots, x_{i-n}$ in sorted order. The i-th iteration inserts x_i among the v's and among the w's, replacing $w_l = x_{i-n}$ (note that x_{i-n} does not appear among the v's). When $d = n-1$ the insertion of x_i leaves $v_1, \ldots, v_d \equiv w_1, \ldots, w_n$; both are $\{x_i, \ldots, x_{i-n+1}\}$ in sorted order. Thus, the v's can now take over the role of the w's. After every n iterations the backup copy becomes the working sequence of w's.

Some minor points that need to be clarified are as follows.

- The current value of d can be computed by maintaining a mod n counter in cell 1.
- To detect when the data in cell j is x_{i-n} each data item is given an age attribute. As *global_data*$[i]$ moves across cell j, the age of *data*$[j]$ is incremented. If the age reaches n, the item is known to be x_{i-n}.

The algorithm of Figure 2 computes a median filter of order n in area $O(n(\log n + s))$, where s is the number of bits in each input data item. This can be reduced to $O(ns)$, as follows. The only two uses of mod n counters are to indentify cells d, $d+k+1$, and $d+n$ and to maintain ages. The first use can be avoided if cells 1, $k+1$, and n are given mod n counters, used to regenerate moving flags that identify cells d, $d+k+1$ and $d+n$, respectively. The second use can be avoided if the i-th input is the pair x_i, x_{i-n}. This can be accomplished by duplicating the input and sending one copy through a length n FIFO queue, with area $O(ns)$ (Fisher assumes the existence of such a queue when calculating the area of the two way algorithm [4]). Note also that when s is $o(\log n)$ we can compute the median filter using 2^s cells. The j-th cell counts the number of elements of $\{x_i, \ldots, x_{i-n+1}\}$ that are less than or equal to j. Area is $O(2^s \log n)$, but this approach requires the FIFO queue preprocessor mentioned above for $O(ns)$ total area. The following theorem shows that we cannot do better.

Theorem 1 *Any order n median filter handling data items s bits long requires area $\Omega(ns)$.*

Proof. There are $2^{ns/2}$ distinct possible sequences of the first $n/2$ inputs. Suppose the bits stored after $x_{n/2}$ enter the array fail to distinguish between two of these sequences, say $a_1, \ldots, a_{n/2}$ and $b_1, \ldots, b_{n/2}$. Then there exists j such that the multiset $\{a_j, \ldots, a_{n/2}\}$ is not identical to the multiset $\{b_j, \ldots, b_{n/2}\}$. If $k = n/2$ it is easy to choose $x_{n/2+1}, \ldots, x_{n+j-1}$ so that the k-th largest of $\{a_j, \ldots, a_{n/2}, x_{n/2+1}, \ldots, x_{n+j-1}\}$ is not the same as the k-th largest of $\{b_j, \ldots, b_{n/2}, x_{n/2+1}, \ldots, x_{n+j-1}\}$. Such a choice would result in two different values of y_{n+j-1}. Thus, the bits stored in the array must distinguish all possible sequences of the first $n/2$ inputs, which requires at least $\log_2(2^{ns/2}) = ns/2$ bits. □

for $i = 0$ **to** ∞ **do**
    /* note: here *global* $data[i]$ includes new, $carry$, $result$, and d */
    $new := carry := x_i$; $d := i \bmod n$; $result := \emptyset$

    /* current sequence of data is $v_1, \ldots, v_d, w_1, \ldots, w_n$;
       w_h and w_l are as defined in the text */
    for $j = 1$ **to** d **do**
        /* $data[j] = v_j$, $carry = v_{j-1}$ (x_i if $x_i < v_{j-1}$ or $j = 1$) */
        if $carry \geq data[j]$ **then** swap $carry, data[j]$ **endif**
    end do

    /* $data[d+1] = w_1$, $carry = v_d$ (x_i if $x_i < v_d$ or $d = 0$) */
    swap $carry, data[d+1]$

    for $j = d + 2$ **to** $d + n$ **do**
        /* $data[j] = w_{j-d}$,
           $carry = w_{j-d-1}$ ($\emptyset$ if $j - d > l$),
           $new = x_i$ if $j - d \leq h$,
                  w_{j-d-2} if $h < j - d \leq l$,
                  w_{j-d-1} if $j - d > l$ and $j - d > h$
           $result = k$-th largest of $\{x_i, \ldots, x_{i-n+1}\}$ if $j - d > k + 1$ */

        /* $carry$ is used to shift $w_1, \ldots, w_{l-1}$ one cell to the right */
        if $carry = x_{i-n}$ **then** $data[j] := \emptyset$ **endif**
        if $carry \neq \emptyset$ **then** swap $carry, data[j]$ **endif**

        /* new is used to shift $w_h, \ldots, w_n$ one unit to the right
           (and to insert x_i before w_h);
           note: if $h < l$, $w_h, \ldots, w_{l-1}$ get shifted twice. */
        if $new \geq data[j]$ **then** swap $new, data[j]$ **endif**

        /* $data[j] = w'_{j-d-1}$ at this point */
        if $j = d + k + 1$ **then** $result := data[j]$ **endif**
    end do

    /* $carry = \emptyset$, $new = w_n$ (x_i if $x_i < w_n$) */
    $data[d + n + 1] := new$
end do

Figure 2: One-way median filter algorithm

3 Generalizations to Other Filters

It is not hard to see that the median filter algorithm generalizes to other finite f-filters that are easily computed when the set $\{x_i, \ldots, x_{i-n+1}\}$ is presented in sorted order. An example is $y_i = \sum_{j=1}^{n} a_j w_j$, i.e. a weighted sum of the elements in sorted order (the median filter is the special case where $a_k = 1$ and all other a_j's are 0). A more precise generalization is given by the following result, which can be proved using a modification of the median filter algorithm.

Theorem 2 *A finite f-filter of order n can be computed in fixed-throughput on a linear array if $f(x_i, \ldots, x_{i-n+1}) = g(w_1, \ldots, w_n, a_1, \ldots, a_n)$, where $w_1, \ldots, w_n$ is $\{x_i, \ldots, x_{i-n+1}\}$ sorted by any key that can be computed in constant time, $a_1, \ldots, a_n$ are constants, and g is real-time computable on a RAM with constant space when the j-th input is the pair (a_j, w_j).*

Note that under the conditions of the theorem, the j-th step in the computation of $y_i = f(x_i, \ldots, x_{i-n+1})$ can be simulated in cell $d + j + 1$ (where $d = i \bmod n$). The only additional modification is that the sequence $a_1, \ldots, a_n$ must be stored in cells $1, \ldots, n$. A copy of the sequence shifts right during every iteration of the outer loop and is regenerated in cell 1 every n-th iteration. It is not known whether this result can be extended to all f-filters that have two-way fixed-throughput algorithms.

Infinite f-filters are more difficult to implement on one-way linear arrays. Two-way dataflow appears to be necessary to ensure that all of $y_{i-1}, \ldots, y_{i-n+1}$ are encountered during the computation of y_i (as in the two-way IIR filter algorithm [1]). The following theorem applies specifically to IIR filters, given by the formula $y_i = \sum_{j=0}^{n-1} a_j x_{i-j} + \sum_{j=1}^{n-1} b_j y_{i-j}$.

Theorem 3 *Any order n IIR filter algorithm in which the computation of y_i depends explicitly on previously computed intermediate results $b_j y_{i-j}$ (for $j = 1, \ldots, n-1$) takes time $\Omega(mn)$ to produce m outputs on a one-way linear array with $O(n)$ cells*

Proof. The conditions of the theorem give a partial order $\prec$ on the intermediate computations ($A \prec B$ if B depends explicitly on the previous computation of A). In a one-way linear algorithm $A \prec B$ means that A cannot be computed later than B or in a cell to the right of where B is computed. Since $y_0 \prec y_1 \prec \ldots$ and there are $O(n)$ cells, there exist i_0 and c_0 such that for all $i \geq i_0$, y_i is computed in cell $c_0 n$.

Now consider (for any $i \geq i_0 + n$) all intermediate results of the form $b_j y_{i-k}$, where $1 \leq j \leq n-1$ and $j \leq k \leq n$. There are $n(n+1)/2 - 1 \geq n^2/2$ (assuming $n \geq 2$) such results and for each of them we have $y_{i-n} \prec b_j y_{i-k} \prec y_i$. Since each cell can compute at most C (for some constant C) intermediate results during a single time unit and all of these results must be computed in cell $c_0 n$, at least $n^2/2C$ time units must elapse between the computation of y_{i-n} and that of y_i. When $m \geq i_0 + n$, the total number of time units before y_m is produced is at least $(m/n)(n^2/2C)$. □

Note that the above argument specifically rules out any fixed-throughput *topological simulation* (as defined by Culik and Yu [3], a topological simulation must preserve the partial order of intermediate results computed) of the two-way IIR filter algorithm on a one-way linear array (to be fixed-throughput an algorithm must produce m outputs in time $O(m+n)$). Note also that the result can be generalized to other types of algorithms and other f-filters: if there are $p(n)$ intermediate results, each lying between y_{i-n} and y_i with respect to $\prec$, then the total time is $\Omega(m(p(n)/n))$. This rules out, for the purposes of one-way real-time implementation, any algorithm where the number of intermediate results computed between y_{i-n} and y_i is non-linear in n. Of course a non-linear lower bound on the number of such intermediate results would be extremely difficult to prove.

References

[1] Peter R. Cappello and Kenneth Steiglitz. Digital signal processing applications of systolic algorithms. In H.T. Kung, B. Sproull, and G. Steele, editors, *Proceedings of CMU Conference on VLSI Systems and Computations*, pages 245–254, Computer Science Press, 1981.

[2] Jik H. Chang, Oscar H. Ibarra, and Anastasios Vergis. On the power of one-way communication. In *Proceedings 27th Annual IEEE Symposium on Foundations of Computer Science*, pages 455–464, 1986.

[3] K. Culik II and S. Yu. Translation of systolic algorithms between systems of different topology. In *Proceedings of the International Conference on Parallel Processing*, pages 756–763, 1985.

[4] Allan L. Fisher. Systolic algorithms for running order statistics in signal and image processing. In H.T. Kung, B. Sproull, and G. Steele, editors, *Proceedings of CMU Conference on VLSI Systems and Computations*, pages 265–272, Computer Science Press, 1981.

[5] J.A.B. Fortes and D.I. Moldovan. Parallelism detection and transformation techniques useful for VLSI algorithms. *Journal of Parallel and Distributed Computing*, 2:277–301, 1985.

[6] Oscar H. Ibarra and Tao Jiang. On one-way cellular arrays. *SIAM Journal on Computing*, 16(6):1135–1154, December 1987.

[7] Oscar H. Ibarra, Michael A. Palis, and Sam M. Kim. Designing systolic algorithms using sequential machines. In *Proceedings 25th Annual IEEE Symposium on Foundations of Computer Science*, pages 46–55, 1984.

[8] Anwer Z. Kotob. *Transforming Computations with Bi-directional Data Flow into Ones with Uni-directional Data Flow on Linear Systolic Arrays.* Master's thesis, North Carolina State University, 1987.

[9] H.T. Kung, Lawrence M. Ruane, and David W.L. Yen. A two-level pipelined systolic array for covolutions. In H.T. Kung, B. Sproull, and G. Steele, editors, *Proceedings of CMU Conference on VLSI Systems and Computations*, pages 255–264, Computer Science Press, 1981.

[10] Carla D. Savage and Matthias F.M. Stallmann. Fault-tolerance and decomposability issues in one-dimensional array architectures. In preparation.

[11] Carla D. Savage, Matthias F.M. Stallmann, and Anwer Z. Kotob. *Simulation of Two-Way Computations on Arrays with One-Way Data Flow.* Technical Report CCSP-TR-87/6, North Carolina State University Center for Communications and Signal Processing, 1987.