Report Number: WUCS-91-19

1991-01-01

Rank Order Filters and Priority Queues

Anne Kaldewaij and Jan Tijmen Udding

A derivation of a parallel algorithm for rank order filtering is presented. Both derivation and result differ from earlier designs: the derivations are less complicated and the result allows a number of different implementations. The same derivation is used to design a collection of priority queues. Both filters and priority queues are highly efficient: they have constant response time and small latency.

Rank Order Filters and Priority Queues

Anne Kaldewaij
Jan Tijmen Udding

WUCS-91-19

January 1991

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Rank Order Filters and Priority Queues

Anne Kaldewaij [*]

Jan Tijmen Udding

Department of Computer Science
Washington University
Campus Box 1045
St Louis, MO 63130

Abstract

A derivation of a parallel algorithm for rank order filtering is presented. Both derivation and result differ from earlier designs: the derivations are less complicated and the result allows a number of different implementations. The same derivation is used to design a collection of priority queues. Both filters and priority queues are highly efficient: they have constant response time and small latency.

1 Introduction

Many computations require a stream of input values to be transformed into a stream of output values. Sometimes each output value depends on the entire stream of input values thus far, for instance when computing the mean of all values received thus far. More often, each output value is determined by only a fraction of the input values. So-called window computations are an example of such computations. The output value to be produced is determined by the last N input values, for some fixed natural number N.

Quite often, these problems can be solved using a network of many, though very simple, identical processing cells arranged in a regular pattern. These cells communicate with each other and the network's environment by exchanging messages, much like processes in CSP [1]. Communications between cells are possible only if they are

[*]On leave of absence, current address: Eindhoven University of Technology, Department of Mathematics and Computing Science, P.O. Box 513, 5600 MB Eindhoven, The Netherlands

neighbors. Implementing a computation on such a network boils down to defining the task to be performed by each cell and determining the interconnection pattern of the cells. Computations like these are often called systolic [3].

A useful technique to determine the task that each cell has to perform is to find a recurrence relation between a particular value in the output stream and previous values in the output stream and the input stream. Usually, such a recurrence relation suggests how to distribute the computation over the cells and how these cells should be interconnected. What eventually remains is the choice of the order in which a cell sends values to and receives values from its neighboring cells. The choice made is crucial for the performance of the network. A single erroneously ordered communication for a cell may cause the entire network to come to a halt.

We use sequence functions, cf. [5, 6], to analyze the speed of the computation as a result of a particular choice of communication behavior. A sequence function assigns a time slot to each communication over a channel, in accordance with the computations of the two processes involved. This gives an upper bound for the number of slots that is needed for a particular communication to occur, provided that the communication behavior of a cell is choice-free and independent of the values communicated. We say that a network has *constant response time* if we can find a sequence function such that the number of time slots between two successive external communications is bounded by a constant. The *latency* of an output of a network is the number of time slots between the occurrence of that output and the last of the input values determining the value of that output.

In the sequel we use these techniques to derive an efficient rank order filter. The derivation is much simpler than in [2] and the result quite different. It allows a variety of implementations. We start out by deriving the recurrence relation mentioned above. This immediately suggests the specification of a cell and the connection pattern of the network. What remains is the task to determine the communication behavior of a cell. We explore a number of choices and we show how each choice leads to an actual network of cells and how that choice affects the operation and performance of the network. As it turns out, some of these networks may equally well be used to implement priority queues, enjoying the same efficiency characteristics. We begin with a formal problem description.

2 Rank Order Filters

Filters are mechanisms that transform streams of values into other streams of values. The output stream is a function of the input stream. The idea of rank order filtering is to move a window of fixed size, say N, over the input stream and to output for each position of the window the value that would occupy some fixed position, say K, in the window if the values in the window were ordered ascendingly.

Let sequence $a(j : j \geq 0)$ be the input stream and let sequence $b(j : j \geq 0)$ be the output stream. For $0 \leq p \leq q$, we denote window $a(j : p \leq j < q)$ by $a[p..q)$ and we denote the bag (multiset) of values in $a[p..q)$ by $\tilde{a}[p..q)$. The value that would occupy the K-th position if the values in $\tilde{a}[p..q)$ were sorted is denoted by $\tilde{a}[p..q).K$. This value is also referred to as the K-th value in the bag.

Therefore, we can formally state the problem of rank order filtering as follows. For N and K integers such that $0 \leq K < N$, we want to design a parallel program, with input channel a and output channel b, that has input/output-relation

$$b.i = \tilde{a}[i{-}N..i).K$$

for $i \geq N$. In order to facilitate the computation of the successive b-values we generalize this relation by replacing constant K by variable k, $0 \leq k < N$. That is, for $i \geq N$ we consider the input/output-relation

$$b.k.i = \tilde{a}[i{-}N..i).k$$

The first step in our derivation of the algorithm is to find recurrence relations that enable us to express $b.k.i$ in a-values and other b-values. This permits us to compute b-values successively, given that we have a starting point. We still have the freedom to choose the first N b-values any way we like. By defining $a.i = \infty$ for $i < 0$, all values $b.k.0$, $0 \leq k < N$, become ∞, which gives an easy starting point for our recurrence relations.

¿From the input/output-relation we conclude that $b.k.i$ and $b.k.(i{+}1)$ are determined by $\tilde{a}[i{-}N..i)$ and $\tilde{a}[i{+}1{-}N..i{+}1)$ respectively. Comparing these two bags, we see that they differ in $a.i$, which is not in the former, and $a.(i{-}N)$, which is not in the latter. Since we also want to address priority queues later on, we analyze this situation in the next paragraphs in a somewhat more general setting.

Given a bag B of N values and integer k, $0 \leq k < N$, we want to know how $B.k$, the k-th value of the bag, changes by adding one element to and deleting one element from B. It is clear that if both values are on the same side of $B.k$, i.e. they are both smaller or both larger than $B.k$, then $B.k$ does not change. The other cases are less clear and we proceed by first stating a few obvious properties involving single insertions and deletions. Then we address the remaining cases in Theorem 1. Bag union is denoted by $\oplus$ and bag difference by $\ominus$.

Removing an element from the bag that is at least the $(k{+}1)$-th value does not change the k-th value:

Property 1

For integer y we have $\quad y \geq B.(k{+}1) \;\Rightarrow\; (B \ominus \{y\}).k = B.k.$

□

Note that this property holds for $k = N{-}1$ as well, provided that $B.N$ is defined as ∞.

If an element is added that is at most $B.k$ then the new $(k{+}1)$-th value equals $B.k$. Moreover, the new k-th value is the larger of the value added and $B.(k{-}1)$. This also holds for $k = 0$, provided that we define $B.(-1)$ to be $-\infty$:

Property 2

For integer x we have

$$x \le B.k \;\Rightarrow\; (B \oplus \{x\}).(k{+}1) = B.k \;\wedge\; (B \oplus \{x\}).k = x \,\mathbf{max}\, B.(k-1)$$

□

Using these properties we prove the following theorem.

Theorem 1

For integers x and y, such that $y \in B$, we have

$$x \le B.k \le y \;\Rightarrow\; (B \oplus \{x\} \ominus \{y\}).k = x \,\mathbf{max}\, B.(k{-}1)$$

$$y \le B.k \le x \;\Rightarrow\; (B \oplus \{x\} \ominus \{y\}).k = x \,\mathbf{min}\, B.(k{+}1)$$

Proof

For the first part we derive

$$\begin{array}{ll}
 & x \le B.k \le y \\
\Rightarrow & \quad \{\,\text{Property } 2\,\} \\
 & (B \oplus \{x\}).(k{+}1) \le y \;\wedge\; (B \oplus \{x\}).k = x \,\mathbf{max}\, B.(k-1) \\
\Rightarrow & \quad \{\,\text{Property } 1\,\} \\
 & (B \oplus \{x\} \ominus \{y\}).k = (B \oplus \{x\}).k \;\wedge\; (B \oplus \{x\}).k = x \,\mathbf{max}\, B.(k-1) \\
\Rightarrow & \quad \{\,\text{calculus}\,\} \\
 & (B \oplus \{x\} \ominus \{y\}).k = x \,\mathbf{max}\, B.(k-1)
\end{array}$$

The second part of the theorem follows from the first part by symmetry. This is due to the fact that, ordering the elements of B ascendingly, the k-th value from the left is the $(N{-}k)$-th value from the right in $B \oplus \{x\} \ominus \{y\}$, since $y \in B$.
□

Note that a value x to be inserted and a value y to be removed are either on one side of $B.k$ or they straddle $B.k$ as in Theorem 1.

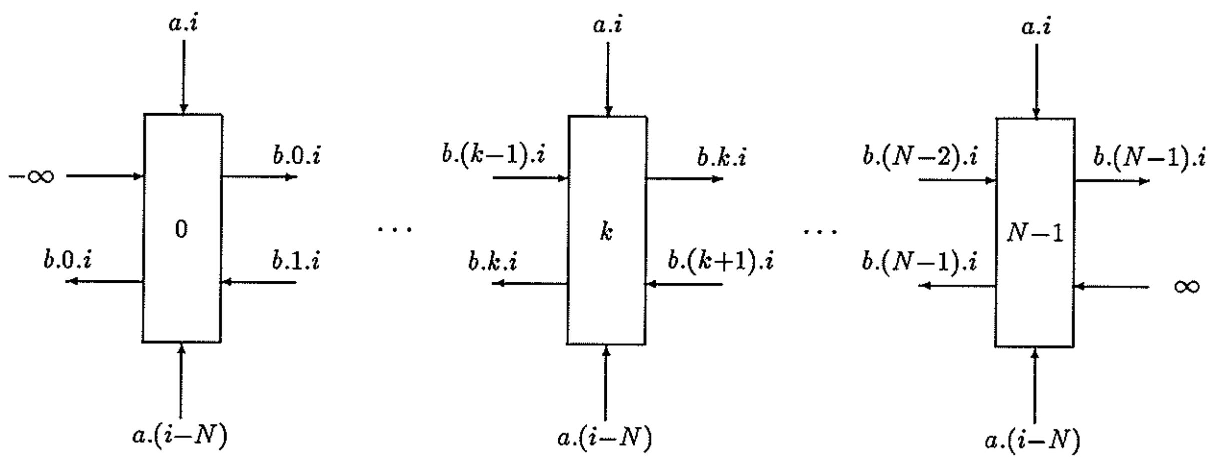

Figure 1: A linear network of cells

Returning to the rank order filter, going from $\tilde{a}[i-N..i)$ to $\tilde{a}[i+1-N..i+1)$, the value removed is $a.(i-N)$ and the value added is $a.i$. With the above analysis we then find the following recurrence relations for $b.k.i$ ($i \geq 0$):

$$b.k.0 = \infty$$

$$b.k.(i+1) = \begin{cases} a.i \max b.(k-1).i & \text{if } a.i \leq b.k.i \leq a.(i-N) \\ a.i \min b.(k+1).i & \text{if } a.(i-N) \leq b.k.i \leq a.i \\ b.k.i & \text{otherwise} \end{cases}$$

where $b.(-1).i = -\infty$ and $b.N.i = \infty$.

¿From these relations we conclude that a linear network of N cells is appropriate, cf. Figure 1. The k-th cell computes $b.k$ and needs b-values only from its immediate neighbors. In the next section we show some implementations of such a network.

3 Implementations

We consider implementations that consist of a network of N identical cells, numbered from 0 up to and not including N. Cell k ($0 \leq k < N$) generates the values $b.k.i$ in order of increasing i.

To compute $b.k.(i+1)$, cell k needs the values $a.i$, $a.(i-N)$, $b.(k-1).i$, and $b.(k+1).i$. For the time being, we assume that cell k has input channels a and d to communicate the values $a.i$ and $a.(i-N)$ respectively. That is, the i-th value along channel a is $a.i$

and the i-th value along d is $a.(i-N)$. (Sequence $a.i$ is provided by the environment and a program to produce the sequence $a.(i-N)$ is derived in the next section.) Between cells k and $k-1$ a channel is needed to communicate $b.(k-1)$ and a channel to communicate $b.k$. For the sake of convenience, we denote, for fixed k, the input channel to communicate $b.(k-1)$ by c and the output channel to communicate $b.k$ by e. Similarly, we denote the channels to communicate $b.k$ and $b.(k+1)$ with cell $k+1$ by $\overline{c}$ and $\overline{e}$ respectively (cf. Figure 2). Putting the equalities together as

$$c.i = b.(k-1).i \ , \ \overline{e}.i = b.(k+1).i \ , \ e.i = \overline{c}.i = b.k.i \ , \ d.i = a.(i-N),$$

we can rewrite the recurrence relations for cell k as follows

$$e.0 = \overline{c}.0 = \infty$$

$$e.(i+1) = \overline{c}.(i+1) = \begin{cases} a.i \max c.i & \text{if } a.i \leq e.i \leq d.i \\ a.i \min \overline{e}.i & \text{if } d.i \leq e.i \leq a.i \\ e.i & \text{otherwise} \end{cases}$$

A program for cell k corresponding to these relations is easily derived. We use the programming notation of [6]. As in CSP, assigning an incoming value on channel a to program variable x is denoted by $a?x$. Sending a value y on channel b is denoted by $b!y$. Ordering between statements is expressed by a semicolon, and repetition is denoted by Kleene's star. The comma is used to express that two statements are executed concurrently. The comma has a higher priority than the semicolon. In the program we use local variables x, y, z, v, and w. We have as loop invariant $z = e.i$. This invariant can easily be checked to hold in the following program for cell k, by introducing a ghost variable i, initialized at 0, that is increased by 1 at the end of each step of the repetition. This makes it also clear that we have the following equalities for x, y, v, and w in the selection statement:

$$x = a.i \ , \ y = d.i \ , \ v = c.i \ , \ w = \overline{e}.i$$

The program reads

```
|[  var $x, y, z, v, w$ : integer;
    $z := \infty$;
    ($a?x$ , $d?y$ , $c?v$ , $e!z$ , $\overline{c}!z$ , $\overline{e}?w$;
     if  $x \leq z \leq y$  →  $z := x \max v$
     []  $y \leq z \leq x$  →  $z := x \min w$
     []  otherwise    →  skip
     fi
    )*
]|
```

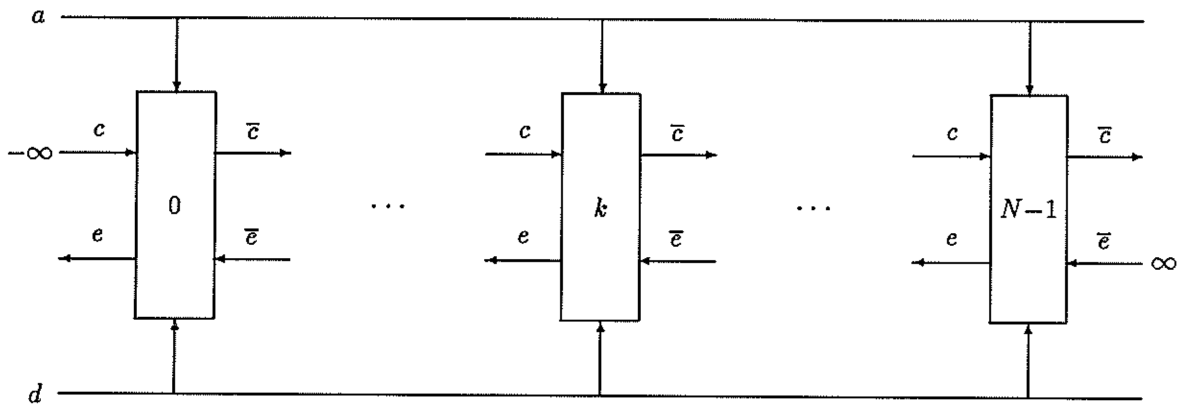

Figure 2: the values of a and d are broadcast

The communication behavior of this cell is $(a, d, c, e, \overline{c}, \overline{e})^*$. With respect to c and e this behavior is $(c, e)^*$ and with respect to $\overline{c}$ and $\overline{e}$, we have the same behavior: $(\overline{c}, \overline{e})^*$, hence, (cf.[6]) with respect to these channels there is no danger of deadlock.

So far we have said nothing about communication channels a and d. We have a number of choices each leading to a different implementation. As a first solution we use the network of Figure 2. Channels a and d are directly connected to all cells. At step i all cells receive $a.i$ and $d.i$. Then all cells compute the values $b.k.i$ concurrently, after which the next values of a and d can be supplied.

In order to analyze the speed of this network we use a sequence function, cf. [6]. Sequence functions assign a time slot to each communication over a channel in accordance with the communication behaviors of the two processes involved. We subscript the channels of cell k with k. Note that each channel between two cells has two names: one with and one without an overbar. A possible sequence function is:

$$\sigma.a_k.i = \sigma.d_k.i = \sigma.c_k.i = \sigma.\overline{c}_k.i = \sigma.e_k.i = \sigma.\overline{e}_k.i = i$$

Value $b.K.i$, which we had to compute, is output by cell K along channel e_K, for which we have

$$\sigma.e_K.i = i$$

It is clear that this sequence function respects the communication behavior of each cell. Moreover, it is well-defined, since $\sigma.e_k.i = \sigma.\overline{e}_{k+1}.i$ and $\sigma.c_k.i = \sigma.\overline{c}_{k+1}.i$. This solution has constant response time and latency 1, since $\sigma.e_K.(i+1) - \sigma.a_K.i = 1$. Note that after step i, $b.k.i = \tilde{a}[i-N..i).k$, for $0 \leq k < N$, i.e. this network can be used to obtain the sorted version of the last N values received along channel a.

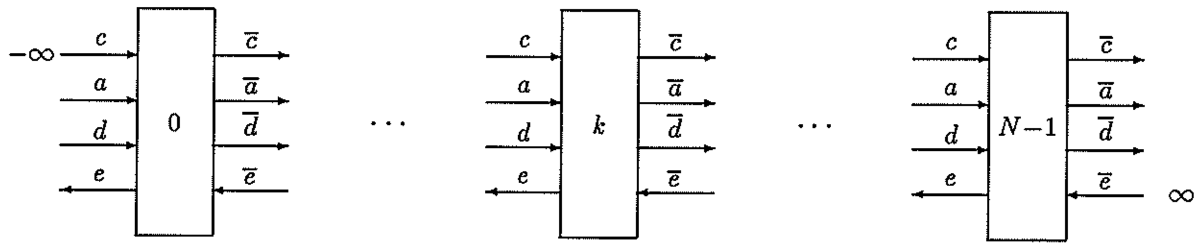

Figure 3: the values of a and d are supplied at cell 0

A quite different solution is obtained if the values of a and d are not broadcast to each cell, but supplied only to one cell, for instance cell 0. Cell k, $0 < k < N$ will then get its a and d values from cell $k-1$. More formally, we define sequences $a.k$ and $d.k$ for $0 \leq k < N$ by $a.k.i = a.i$ and $d.k.i = d.i$. We denote for fixed k channels $a.(k+1)$ and $d.(k+1)$ by $\overline{a}$ and $\overline{d}$ respectively and $a.k$ and $d.k$ by a and d respectively (cf. Figure 3). In addition to the earlier computation a cell now also has the obligation to send the a and d values on to the next cell. This yields the following program.

```
|[  var $x, y, z, v, w$ : integer;
    $z := \infty$ ;
    $(a?x\,, d?y\,, c?v\,, e!z\,;\ \overline{a}!x\,, \overline{d}!y\,, \overline{c}!z\,, \overline{e}?w\,;$
     if  $x \leq z \leq y$  →  $z := x \max v$
     []  $y \leq z \leq x$  →  $z := x \min w$
     []  otherwise       →  skip
     fi
    $)^*$
]|
```

The communication behavior of this cell is $(a, d, c, e\ ;\ \overline{a}, \overline{d}, \overline{c}, \overline{e})^*$. Since a, d, c, and e occur concurrently, as do $\overline{a}$, $\overline{d}$, $\overline{c}$, and $\overline{e}$, it suffices to present a sequence function for a and $\overline{a}$ only. As before we subscript the channels of cell k with k. A possible sequence function is:

$$\sigma.a_k.i = 2i + k \quad \sigma.\overline{a}_k.i = 2i + k + 1 \quad (i \geq 0)$$

It is clear that this sequence function respects the communication behavior of each cell. Moreover, it is well defined, since $\sigma.\overline{a}_k.i = \sigma.a_{k+1}.i$. The latency of this solution

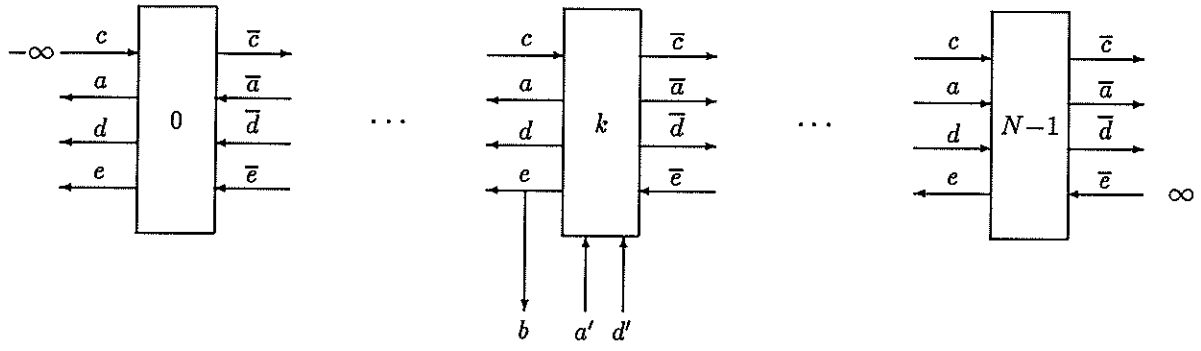

Figure 4: the values of a and d are supplied at cell K as a' and d'

is $\sigma.e_K.(i{+}1) - \sigma.a_0.i = 2(i{+}1) + K - 2i = K{+}2$ and the solution has constant response time.

The final solution that we discuss has a small latency as in the broadcast solution, but the input stream is supplied to only one cell, as in the previous solution. Rather than supplying the input stream to cell 0, we supply it to cell K, where $b.K$ is the output stream that we originally wanted to be computed. If $K \neq 0$ then the input streams a and d have to be relayed both to the left and to the right. The specifications of the other cells remain exactly the same, albeit that the input streams go from right to left in the cells to the left of K, as in a solution in which the input streams are fed into cell N. Calling the input streams a' and d' for the time being, cell K should replicate this stream both to the right, on a and d, and to the left, on $\overline{a}$ and $\overline{d}$ (cf. Figure 4). Thus, the program for cell K becomes

```
|[  var $x, y, z, v, w$ : integer;
     $z := \infty$;
     $(a'?x, d'?y\ ;\ a!x, d!y, \overline{a}!x, \overline{d}!y, c?v, \overline{c}!z, e!z, \overline{e}?w$;
      if   $x \le z \le y$   →   $z := x \max v$
      []   $y \le z \le x$   →   $z := x \min w$
      []   otherwise         →   skip
      fi
     $)^*$
]|
```

The communication behavior of this cell is

$$(a', d'\ ;\ a, d, \overline{a}, \overline{d}, c, \overline{c}, e, \overline{e})^*,$$

which nicely matches the communication behaviors of the other cells. A suitable sequence function is

$$\sigma.a'.i = 2i \quad \sigma.a_K.i = \sigma.\overline{a}_K.i = 2i + 1$$

and for $K < k < N$:

$$\sigma.a_k.i = 2i + k - K$$
$$\sigma.\overline{a}_k.i = 2i + k - K + 1$$

and for $0 \leq k < K$:

$$\sigma.\overline{a}_k.i = 2i + K - k$$
$$\sigma.a_k.i = 2i + K - k + 1$$

Hence, the latency is $\sigma.e_K.(i{+}1) - \sigma.a'.i = 2(i{+}1) + 1 - 2i = 3$. The solution has constant response time.

3.1 Generating Sequence d

So far we have assumed that the a and d streams would somehow be supplied by the environment. Since the a-stream is the only one being actually generated by the environment, we have to construct the d-stream. In this section we derive a network that generates the d-stream. That is, for $i \geq 0$, we have input/output-relation

$$d.i = a.(i{-}N)$$

where $a.i = \infty$ for $i < 0$, as before. The external behavior of the program should be such that $a.i$ and $a.(i{-}N)$ are available in the same time slot, starting with $a.0$ and $a.(-N)$.

We use the same technique as in the previous section. We begin with deriving a recurrence relation, which will suggest a network topology. Then we choose the communication behavior. Compared to the previous section, this derivation is quite simple.

We generalize function d by defining for $0 \leq k \leq N$ and $i \geq 0$ function $d.k$ by

$$d.k.i = a.(i - k)$$

Then $d.N$ is the function that we are looking for. For $i \geq 0$, we have $d.0.i = a.i$ and for $0 \leq k < N$:

$$d.(k{+}1).0 = a.(-k{-}1) = \infty$$

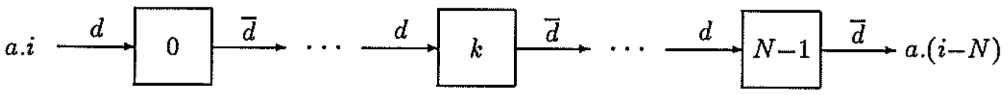

Figure 5: Generation of the sequences $a.i$ and $a.(i{-}N)$

$$\begin{array}{ll}
 & d.(k{+}1).(i{+}1) \\
= & \quad \{\text{definition of } d.k\,\} \\
 & a.(i{-}k) \\
= & \quad \{\text{definition of } d.k\,\} \\
 & d.k.i
\end{array}$$

Hence, we have the following recurrence relations

$$\begin{array}{ll}
d.0.i = a.i & \text{for } i \geq 0 \\
d.(k{+}1).0 = \infty & \text{for } 0 \leq k < N \\
d.(k{+}1).(i{+}1) = d.k.i & \text{for } 0 \leq k < N \text{ and } i \geq 0.
\end{array}$$

This gives rise to a network of N cells. Cell k ($0 \leq k < N$) has input channel $d.k$ and output channel $d.(k{+}1)$. Cell $N{-}1$ is going to produce the function $d.N$ that we were interested in (cf. Figure 5). The input for cell 0 is supplied by the environment ($d.0.i = a.i$). All cells are equal: the program for each cell reads

```
|[  var $x, w$ : integer;
      $w := \infty$;
      $(d?x\,, \overline{d}!w\,;\, w := x)^*$
]|
```

where, for cell k, d denotes the channel connecting cell k to cell $k{-}1$, i.e. $d.k$, and $\overline{d}$ denotes the channel connecting cell k to cell $k{+}1$, i.e. $d.(k{+}1)$. An invariant for cell k is $w = d.(k{+}1).i$.

The communication behavior of the cells is $(d, \overline{d})^*$, hence we have as possible sequence function $\sigma.d_k.i = \sigma.\overline{d}_k.i = i$. In particular, values $a.i$ and $a.(i{-}N)$ are available in the same time slot.

Here it becomes apparent how essential the choice of $a.i$ is for $i < 0$: by choosing a constant value, such as ∞, the first N values of the d-sequence do not depend on the a-sequence and can be generated immediately. The choice of, for instance, $a.i = a.0$ for $i < 0$ yields a more complicated solution.

4 Priority Queues

A rank order filter can be used to implement a priority queue. A priority queue is a bag upon which the following operations can be applied: add an element, remove an element, and return the minimum element. Rather than just being able to return the minimum element we implement for fixed N and K, $0 \leq K < N$, a bag B, initially empty, that can hold at most N integers, upon which the following operations can be performed.

add.x	corresponding to $B := B \oplus \{x\}$
remove.x	corresponding to $B := B \ominus \{x\}$, provided $x \in B$
return.x	corresponding to $x := B.K$, which is the K-th element of B.

If B contains fewer than K elements then return.x establishes $x = \infty$.

We use the implementations of a rank order filter of the previous section to realize such a data type. We choose one of the two solutions with small latency, i.e. either the broadcast implementation of Figure 2 or the implementation of Figure 4 in which the external channels are connected to cell K. Both admit external communication behavior

$$(a\,,d\,;b)^*$$

and input/output-relation

$$b.i = \tilde{a}[i{-}N..i).K$$

provided that $d.i = a.(i{-}N)$ for $i \geq 0$. Note that $\tilde{a}[i{-}N..i)$ can also be written as $\tilde{a}[-N..i) \ominus \tilde{d}[0..i)$.

In the derivation of the rank order filter, (cf. Theorem 1), we have made sure that the fact that $d.i$ equals the first element of the window $a[i{-}N..i)$ does not play a role. What matters is that $d.i \in \tilde{a}[i{-}N..i)$. Thus, the i/o-relation can be generalized to

$$b.i = (\tilde{a}[-N..i) \ominus \tilde{d}[0..i)).K$$

provided that $\tilde{d}[0..i) \subseteq \tilde{a}[-N..i)$ for $i \geq 0$. Note that this holds for $i = 0$.

As before, we define $a.i = \infty$ for $i < 0$. Hence, $\tilde{a}[-N..0) \ominus \tilde{d}[0..0)$ is a bag containing N occurrences of ∞. The relation between bag B to be implemented and sequences a and d will be that at any moment B equals the bag $\tilde{a}[-N..i) \ominus \tilde{d}[0..i)$, apart from the occurrences of ∞ in the latter, hence, initially B represents the empty bag. The implementation of B and the operations upon it are now straightforward:

add.x	$a!x\,,d!\infty\,;b?y$
remove.x	$a!\infty\,,d!x\,;b?y$
return.x	$a!\infty\,,d!\infty\,;b?x$

Since the network has constant response time and small latency, these operations are all $O(1)$.

5 Concluding Remarks

Let us briefly skim over the various steps leading to the implementations of rank order filters and priority queues.

Originally we set out to design a parallel algorithm to implement priority queues. We quickly discovered that priority queues were a generalization of rank order filters, in the sense that an *arbitrary* element is removed from a bag of values. A first problem was the characterization of the K-th element of a bag of N elements, finding suitable notation for it, and deriving appropriate properties of it. This led to Properties 1 and 2, and Theorem 1. The properties can be proved using the following definition of $B.k$ $(0 \leq k < N)$: $B.k$ is the unique solution of the equation in x:

$$(\#i : i \in B : i < x) < k \wedge (\#i : i \in B : i \leq x) \geq k$$

where # stands for "the number of". Since the properties were obvious and since their proofs were straightforward and not providing any insight in the derivation, we omitted them. ¿From Theorem 1 the recurrence relations needed to compute the values of interest were easily obtained. The result was a nice symmetric set of relations.

The difference with, for instance, [2, 4] is that in the latter, in an early stage of the design, the decision is made to compute for each value in the window $a[i-N..i)$ its rank in the bag. This leads to a much more complicated set of recurrence relations and requires an elaborate analysis of synchronization of communications.

In the second stage (the implementations), we could simplify the derivation of the programs considerably by postponing the generation of sequence $a.(i-N)$, which we dealt with in isolation. This separation allowed us to consider more than one implementation and to use the network as a priority queue as well.

Two of the implementations of the rank order filter turned out to be suitable to be used as priority queues, viz. the ones that admit with external communication behavior $(a, d; b)^*$. Since the properties and the theorem were general enough, we could construct a priority queue from these implementations without much effort.

Reasoning about these algorithms in a non-operational way turned out to be indispensable. We would not have found these algorithms without the separation of concerns, imposed by the stepwise approach of deriving recurrence relations first, and then choosing for a particular connection pattern and communication behavior.

6 Acknowledgements

This research has been partially funded by a grant from Southwestern Bell Telephone. We are grateful to the Eindhoven Algorithm Club and the Eindhoven VLSI Club for their comments on an earlier version of this paper. Collaboration over this research was made possible by the Computer Science Department of Washington University, especially by its chairman Jerry R. Cox Jr.

References

[1] Hoare, C.A.R.
Communicating Sequential Processes
Prentice-Hall, 1985.

[2] Kaldewaij A. and Rem M.
The Derivation of systolic computations
Science of Computer Programming, 14(2-3):229-242, 1990.

[3] Kung, H.T.
Let's design algorithms for VLSI systems
In *Proc 1st Caltech Conference on VLSI* (C.L. Seitz, ed.)
California Institute of Technology, Pasadena, 1979, 65-90.

[4] Kung, S.Y.
VLSI Array Processors
Prentice-Hall, Englewood Cliffs, 1988.

[5] Rem, M.
Trace theory and systolic computations
In *PARLE Parallel Architectures and Languages Europe Vol. 1*
(J.W. de Bakker et al., eds.)
Lecture Notes in Computer Science 258, Springer-Verlag, Berlin, 1987, pp 14-33.

[6] Zwaan, G.
Parallel Computations
Doctoral Dissertation, Eindhoven University of Technology, Eindhoven, 1989.