Fast Median Filtering Algorithms for Mesh Computers

Steven L. Tanimoto

Technical Report 95-03-05
March, 1995

Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195

Fast Median Filtering Algorithms for Mesh Computers

Steven L. Tanimoto
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, WA 98195

March 22, 1995

Abstract

Two fast algorithms for median filtering of images using parallel computers having 2-D mesh interconnections are given. Both algorithms assume that an $n \times n$ image is loaded onto the mesh with one processing element per pixel. One algorithm performs median filtering over $d \times d$ neighborhoods in $O(d^2)$ time and works with pixel values in an arbitrarily large range. This algorithm, while theoretically suboptimal, achieves a lower constant than a previously published asymptotically-optimal algorithm and is simpler to program. The second algorithm assumes that the range of pixel values is limited and relatively small, and it accomplishes median filtering in $O(d)$ time.

Keywords: mesh architecture, median filtering, parallel algorithm, internal scanning, overlapping scans, image processing, parallel processing, SIMD.

1 Introduction

1.1 Image Noise Elimination

Images obtained from sensors such as video cameras often contain objectionable amounts of image noise that distorts the input image. The noise results

directly from the physical process of transduction of the image into electrical signals that takes place in the sensor. Since this noise can be modeled as independent at each pixel and following a Gaussian distribution, most of the ill effects can be reduced using low-pass filtering.

Linear filtering using convolution or moving average methods is often used. However, it has the undesirable effect of smearing the step edges that occur between objects and the image background. On the other hand, the method of "median filtering" manages to reduce the noise while avoiding most of the edge-smearing (e.g., Huang[1]). With median filtering, each pixel value is mapped into the median of the pixel values in a neighborhood around the pixel. In practice, the neighborhood is almost always square and of odd width. In a typical example, one might use the 5×5 neighborhood, in which each median would be selected from a population of 25 values.

1.2 Previous Work

On a conventional, single-processor computer, median filtering requires at least $O(\log d)$ time per element (on average), and an algorithm, and an algorithm using a tree-updating procedure has been presented that requires $O(\log^2 d)$ time per element on average (Gil and Werman[2]). If a parallel computer is used, faster algorithms are possible. Using the theoretically ideal, fully interconnected CREW PRAM (Concurrent-Read, Exclusive-Write Parallel Random -Access Machine) it is possible to perform $d \times d$ median filtering on an $n \times n$ image in $O(\log^2 d)$ time using $O(n^2 \log^2 d)$ processors (Ranka and Sahni[3]). However, this model of computation is not often appropriate for real-world image-processing situations.

Two-dimensional mesh-connected parallel computers are growing in importance for high-speed image processing. Commercial examples of these systems include the MasPar MP-1, the Connection Machine CM-2 (though these systems also have routers for more general but slower interconnectivity). A number of special chips and systems have also been developed to support mesh architectures.

The problem of efficiently performing the median-filtering operation on an image using a mesh computer is an interesting one. Several alternative methods can be used. In the following, we assume that the medians are to be computed over $d \times d$ neighborhoods. If d is odd, then each neighborhood is centered around the pixel in question. Otherwise, the neighborhood is only

approximately centered, with the true center slightly to the east and south of the pixel in question. Towards the end of his paper on pyramid median filtering, Stout presented an interesting and, at $O(d)$, a theoretically time-optimal algorithm for this problem[4]. However, that algorithm requires a complicated program, and the constant is high due to a need to multiplex each processing element to simulate several logical ones. Therefore, there remains interest in finding fast mesh median filtering algorithms that can be implemented easily and which do not have large constants associated with them.

One approach to efficient and simple algorithms is to limit the pixel depth. An algorithm is given by Miller and Stout[5] for binary images (one bit per pixel) requiring two steps, the first taking $O(n)$ time to assign to each PE the number of black pixels to its left in its row, and the second taking $O(d)$ time to compute a kind of discrete line integral of this value around the perimeter of each pixel's neighborhood. The second algorithm presented below also takes advantage of a limited-pixel-depth assumption.

1.3 The Mesh Architecture

The algorithms presented here use the following model of a computer. (This model architecture is referred to here as a "two-dimensional mesh numeric machine" or 2DMNM.) There is an array of $n \times n$ processing elements (PEs). Each PE except those in row 0, row $n-1$, column 0 and column $n-1$ is connected to four neighbors. That is, PE_{ij} is connected to $\mathrm{PE}_{i-1,j}$, $\mathrm{PE}_{i+1,j}$, $\mathrm{PE}_{i,j-1}$, and $\mathrm{PE}_{i,j+1}$. There is a central control processor which interprets program instructions. The control processor is essentially a standard Von Neumann architecture with its own memory and computing ability, but it is also connected to the array of processing elements. Some instructions are broadcast to the PEs, whereas the others are executed directly on the control processor. Those broadcast to the PEs are performed immediately by all PEs (except those in a disabled state). The architecture is thus a single-instruction-stream/multiple-data-stream distributed memory structure. Each PE has an accumulator register (the "A" register) and a local memory. The register and memory are presumed to be large enough to hold standard integers and floating-point numbers. The PEs communicate with each other whenever a RECEIVE instruction is broadcast to them. For example, the RECEIVE NORTH instruction causes each PE to read into its

own A register the value of the A register of its northern neighbor. Each PE on the northern border of the array (i.e., in row 0) receives a 0, since it has no north neighbor. Each PE is is presumed capable of conventional arithmetic and logical operations. For example ADD X causes the memory location referred to by X to be added to the contents of the A register with the result stored in the A register. Since the 2DMNM has a single instruction stream and multiple data streams, it is classified as an SIMD architecture under Flynn's taxonomy.

In addition to the broadcasting of instructions, the control processor communicates with the PE array by sensing whether all of the A registers are simultaneously zero or not. A special instruction, JALLZ L causes the controller to jump to location L if all A registers contain zero. This capability is sometimes called "Global OR."

2 Sorting and Finding Medians

Before describing the two new algorithms of this paper, some related algorithms are given to provide a context for the two. Note that any median filtering program needs a policy on how to handle neighborhoods at the borders of the array. In this paper it is assumed that the 2DMNM model's assumption that north, east, west or south communication brings in zeros from outside the array leads to acceptable results. Thus no special-case processing for borders is provided in any of the algorithms discussed here.

2.1 Use of Sorting

One approach to median filtering requires sorting the pixel values within every $d \times d$ neighborhood. If we wish to use an $O(n)$ mesh sorting method, this can be done as follows. First we partition the mesh into a nonoverlapping set of $d \times d$ submeshes. There is thus an array of submeshes that is $m \times m$ on a side, where $m = \lfloor n/d \rfloor$. Two example partitions are shown in Fig. **??**. In $O(d)$ time, we independently sort the m^2 groups of d^2 values, using one of the well-known mesh-sorting techniques (for example that of Thompson and Kung[6]). If we are using the snaking row-major order and an odd value of d, then the median value ends up at the pixel where it belongs. Otherwise, the median can be routed to where it belongs in $O(d)$ time.

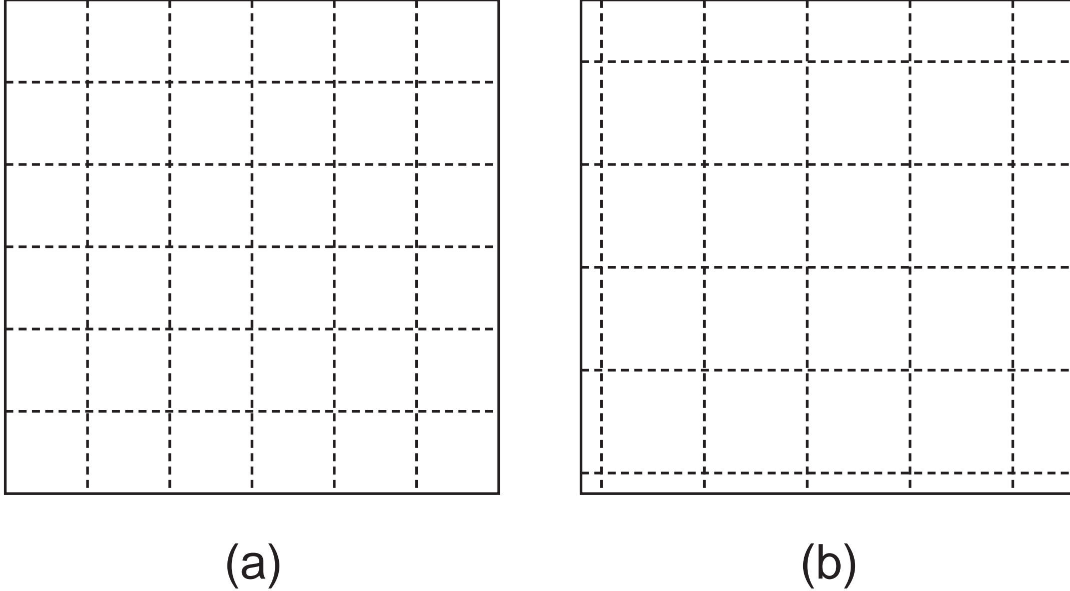


Figure 1: (a) Partition of an $n \times n$ mesh into an $m \times m$ array of $d \times d$ submeshes, with $n = 24$, $m = 6$, and $d = 8$, and (b) partition when d is not a divisor of n and partition is shifted to permit other pixels to be centers of submeshes (here, $n = 24$, $m = 4$, and $d = 5$).

Since a single $m \times m$ partition only gives rise to m^2 medians, it is necessary to shift the partitioning and repeat the sorting d^2 times, so that every pixel can receive the median over its neighborhood. The time required for this algorithm is thus $O(d^3)$.

A simpler, non-mesh sorting algorithm actually yields better performance. For this scheme, first we route the d^2 values of each pixel's neighborhood to the pixel, storing them in d^2 local memory locations. Next we sort these values in parallel at every PE. A reasonably efficient means of doing this is to emulate a bitonic sorting network (at each PE). The ordering of all compare-exchange operations here is independent of the data values, and so it works well on the SIMD 2DMNM. The time required for this sorting is proportional to the number of compare-exchange modules in the sorting network. For sorting d^2 values, we need a network with $O(d^2 \log^2 d)$ modules. After sorting, the medians are then immediately accessible in local memory. The time complexity for this approach is $O(d^2) + O(d^2 \log^2 d) = O(d^2 \log^2 d)$.

2.2 Finding a Median on a 2DMNM-Σ

Let us now consider some methods for computing medians that avoid sorting.

Given a 2DMNM with a special controller register called SIGMA which is automatically updated at all times to contain the sum of all the A register contents, we can find the median value over an image X using a binary search process. The 2DMNM-Σ is a 2DMNM which contains a register SIGMA whose contents are maintained as

$$\Sigma = \sum_{i,j=0}^{n-1} A[i,j].$$

The pixel values for the image X are assumed to be integers in the range from 0 to MAXVAL. In practice, MAXVAL is very often 255. To compute the median, the following algorithm may be used.

```
const MAXVAL 255;
plural int x;
int test, incr, i;

test := 0;  incr := MAXVAL / 2;
for i := 0 to log2 MAXVAL do
  begin
    A := 0;
    if x > test then A := 1;
    if Sigma > N/2 then test := test + incr;
    incr := incr / 2;
  end
output test
```

On a 2DMNM (without the special Sigma register), it is possible to simulate a 2DMNM-Σ by using a reduction scan in $O(n)$ time. Thus, we can find the global median on a 2DMNM in $O(n \log \mathrm{MAXVAL})$ time. Taking MAXVAL as a constant, this reduces to $O(n)$ time.

Now in order to find many medians over small neighborhoods, the mesh can be partitioned (as above) into an $m \times m$ array of submeshes, each of size $d \times d$. By running the m^2 submeshes in parallel, we can find m^2 medians in parallel with this approach in $O(d)$ time. (Note that the values of `test`

must be broadcast within each submesh using a scan, but this is an $O(d)$ operation, and it does not increase the asymptotic complexity.)

If we do the obvious thing and repeat this median-finding d^2 times, we obtain the median-filtered image in $O(d^3)$ time. This is as slow as the first method that used mesh sorting. We can reduce the time by carefully "overlapping" the scans required to find the counts of pixels in each neighborhood above the test value.

3 The Internal Scanning Algorithm

Before describing the full median-filtering algorithm based on "internal scanning," let us consider a simpler algorithm that computes, for each pixel, the median in a $1 \times d$ window around it.

3.1 One-Dimensional Median Filtering

The *horizontal median filtering problem* is to compute, for each pixel $f(i,j)$ the median of the values $f(i, j+k)$ where $k = 0, 1, \ldots, d-1$.

This problem is solved by the following algorithm. The image data is shifted horizontally to $2d-1$ different positions, so that each PE can compare its own value to each of the others within the d neighborhoods that it contributes to. The PE creates two binary vectors of length $2d-1$ in its own local memory as follows:

- The PE places the value 1 in $G[q]$ if the qth value seen is greater than the home value; otherwise it places 0 in $G[q]$.
- The PE places the value 1 in $L[q]$ if the qth value seen is less than the home value; otherwise it places 0 in $L[q]$.

Next, each PE scans through its list G creating a new list of length d. The rth element $NG[r]$ of this list stores the number of values greater than the home value within the rth d-element neighborhood. The PE then scans list L to create list NL similarly. Next, each PE scans the lists NG and NL creating a new binary-valued list M of d elements. It places a 1 in $M[r]$ if $NG[r] < d/2$ and $NL[r] \leq d/2$. This means that the home value is the median of the rth neighborhood to which it belongs.

Finally, the medians are communicated to their destinations in a loop.

- For r = 1 to d do begin

  If M[r] = 1 then set A register to home value (i.e., LOAD X).

  Shift A registers' contents West (i.e., RECEIVE EAST).

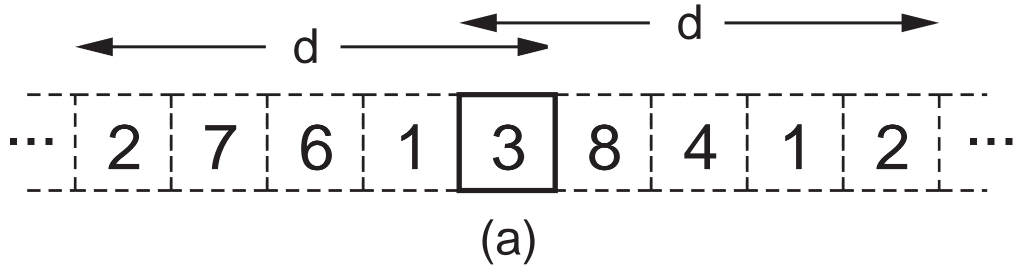
This algorithm requires $O(d)$ communication steps and $O(d)$ computation steps to perform its internal scanning. Thus it requires $O(d)$ time. Since $\Omega(d)$ time is required to propagate the influence of each data item in a window to the site where the median must end up, this algorithm is time-optimal.

3.2 An Example

Let us consider an example of 1-D median filtering with the internal scanning algorithm. Figure **??**a illustrates a sequence of numbers assigned to PEs, one per PE. Taking $d = 5$, the diagram also shows the westmost and eastmost neighborhoods to which the highlighted pixel in the center belows. The highlighted pixel has value 3. The algorithm proceeds to determine for this highlighted PE (and, respectively, for all other PEs) which of the five neighborhoods have this PE's value as the median. In Figure **??**b, the results for this PE and four of its neighbors are shown, and it can be seen that the highlighted PE has given its value to three of the neighbors.

The following charts illustrate the computation of the arrays stored internally at each PE.

value	$G[q]$	$L[q]$	Scan$(G,+)[q]$	Scan$(L,+)[q]$
2	0	1	0	1
7	1	0	1	1
6	1	0	2	1
1	0	1	2	2
3	0	0	2	2
8	1	0	3	2
4	1	0	4	2
2	0	1	4	4



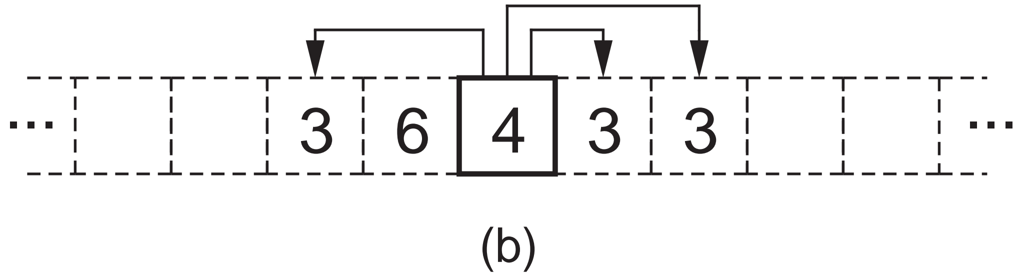


Figure 2: One-dimensional median filtering with internal scanning: (a) initial data layout, and (b) final results.

r	$NG[r]$	$NL[r]$	$M[r]$
1	2	2	1
2	3	1	0
3	3	1	0
4	2	2	1
5	2	2	1

The name "internal scanning" derives from the fact that the bulk of the computation in this algorithm consists of each PE scanning through its arrays located in its local memory to determine the neighborhoods for which its pixel value is the median.

3.3 Two-Dimensional Internal Scanning

The extension of the above method to two dimensions is straightforward. Each PE$_{u,v}$ holds one pixel of the image X, namely $x_{u,v}$. The relative layout of the PEs mentioned below is illustrated in Figure **??**.

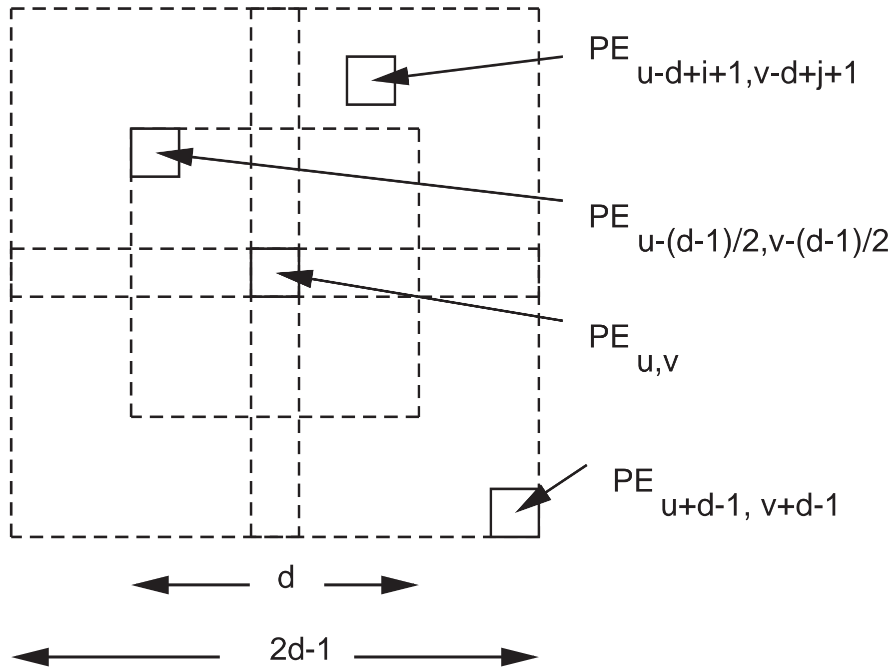


Figure 3: Relationships of various processing elements within a $(2d-1)^2$ block of neighborhoods for the internal scanning algorithm. The central pixel value $x_{u,v}$ is potentially the median for any of d^2 neighborhoods in its vicinity. The PE in the lower right serves as the agent for $x_{u,v}$. It examines all the x values within the larger $(2d-1)^2$ block and then performs "internal scanning" to to determine which PEs to send $x_{u,v}$ to.

Step 1: Each PE loads its A register with its pixel value from the image X, and then the A registers are shifted $d-1$ cells to the east and $d-1$ cells to the south. This brings each pixel value $x_{u,v}$ to PE$_{u+d-1,v+d-1}$.

Step 2: The A registers are reloaded with the original pixel values. Then the data is again shifted among A registers, but this time in a scanning

pattern over a $(2d-1)\times(2d-1)$ square to the east and south. In this manner, each $\mathrm{PE}_{u+d-1,v+d-1}$ sees each of the pixels within all the $d \times d$ neighborhoods to which the pixel $x_{u,v}$ belongs. The order of scanning is raster order. In the course of this scanning, the PE constructs two binary vectors each of length $(2d-1)^2$ in its local memory. Each may be organized as a two-dimensional array $G[s,t]$ and $L[s,t]$. When the PE looks at the st^{th} data element $x' = x_{u-d+s+1,v-d+t+1}$ in this scanning, it sets $G[s,t]=1$ if $x' > x_{u,v}$ and 0, otherwise. Similarly it sets $L[s,t]=1$ if $x' < x_{u,v}$ and 0, otherwise.

Step 3: From these each PE (e.g., $\mathrm{PE}_{u+d-1,v+d-1}$) creates a pair of integer arrays $NG[s,t]$ and $NL[s,t]$, each of dimensions $d \times d$. There are d^2 PEs in the neighborhood of $\mathrm{PE}_{u,v}$ which potentially have $x_{u,v}$ as their median, and it is up to $\mathrm{PE}_{u+d-1,v+d-1}$ to determine who they are. Array NG gives for each of these potential recipients, the number of elements in the neighborhood of $\mathrm{PE}_{u,v}$, that are greater than $x_{u,v}$, and NL gives the number that are less than $x_{u,v}$. To compute NG and NL, the $\mathrm{PE}_{u+d-1,v+d-1}$ creates temporary arrays GTEMP$[s,t]$, and LTEMP$[s,t]$ of dimensions $(2d-1)\times d$ which contain the number of elements greater than (less than) the home value $x_{u,v}$ in the tth $1\times d$ neighborhood in the sth row. Then $NG[s,t]$ is computed by a vertical summing process on GTEMP, adding the first d values in each row, and then successively adding a new value and subtracting an old one.

Step 4: Next, the $\mathrm{PE}_{u+d-1,v+d-1}$ constructs $M[s,t]$, where

$$M[s,t] = \begin{cases} 1, \text{if } NG[s,t] < d^2/2 \text{ and } NL[s,t] \le d^2/2; \\ 0, \text{ otherwise.} \end{cases}$$

It does this by making a single scan through its arrays NG and NL.

Step 5: Finally, the mesh scans (externally) in a $d \times d$ pattern again to create the median-filtered image. During this scan, which can be considered as controlled by a nested pair of loops with descending indices s and t, each PE loads its value into its A register (i.e., at the sth row, t^{th} column), whenever it finds that $M[s,t]=1$. At the end of the scan, all A registers have values, and each value is the median for a $d \times d$

neighborhood including the PE and extending $d-1$ rows to the north and $d-1$ columns to the west.

Step 6: Finally, the median values are brought to the centers of their neighborhoods by performing $(d-1)/2$ RECEIVE EAST instructions and the same number of RECEIVE SOUTH instructions.

The time required by this algorithm is the sum of the times of each of the six steps. The first step requires $O(d)$ time, and each subsequent step requires $O(d^2)$ time. Therefore, the overall time required for this algorithm is $O(d^2)$.

Looking at the computation in more detail gives us an idea of the constants involved. In the first step, $2d-2$ communication operations are performed. In the second step, each PE looks at $4d^2-4d+1$ values requiring $4d^2-4d$ communication operations, and it makes two comparisons for each value. In the third step, the computation of GTEMP and LTEMP each use $(3d-3)(2d-1)$ or $6d^2-9d+4$ addition and subtractions. The computation of NG and NL use $d(3d-3)$ or $3d^2-3d$ additions and subtractions. Step four makes two comparisons for each of d^2 values. Step five performs d^2-1 communication operations and d^2 comparisons, and Step six uses $d-1$ communication operations.

The totals are $5d^2-d-5$ communication steps, $11d^2-8d+2$ comparisons, and $18d^2-24d+8$ additions/subtractions. A small number of LOAD, STORE and control instructions are also needed, but these do not contribute significantly to the overall time. The additions and subtractions as well as the step-five comparisons are on values in the range from 0 to d^2, which could allow efficient bit-serial implementations on mesh implementations with narrow word size, such as the MasPar MP-1 (which has 4-bit wide ALUs). The comparisons of step 2 are on pixel values, typically limited to 8 bits each, also permitting fast execution on narrow-word machines. With this algorithm, median filtering with a 5×5 neighborhood can thus be expected to take in the vicinity of 1000 machine instructions to complete on a 2DMNM, assuming one PE per pixel. A parallel mesh that achieves one million 2DMNM instructions per second can therefore be expected to require approximately a millisecond to complete the 5×5 median-filtering operation.

4 The Overlapped Scanning Algorithm

4.1 Overlapped Reduction Scans

Suppose we have a uniform test value of, say, 27. Each PE tests whether its pixel value is above 27. If true, it stores a 1 in a variable called Above. If false, it stores a zero there. For each $d \times d$ block in the image, we need to compute the sum of the Above values over the block. This is easily accomplished in $O(d)$ time by a set of overlapping reduction scans. To compute these reductions, with the 2DMNM, we proceed as follows. Each PE sets its Count variable to zero. Now we repeat d times to get row sums: read the Count variable of the west neighbor PE and add Above to it. Next each PE sets its ColCount variable to zero, and we repeat d times to get column sums of the row subtotals: read the ColCount value of the North neighbor PE and add Count to it. At the completion of this loop, each PE's ColCount variable contains the number of PEs (in the $d \times d$ block of which the PE is the southeast corner) whose pixel value is above 27. This procedure is described in a more program-like notation later.

4.2 Adaptations to the 2DMNM

In order for the overlapping to work out successfully, it is necessary that each PE in the whole mesh sets its Above value using the same test value as the others. This make it difficult, in the search for the medians, to employ the binary search strategy (used in our earlier algorithm for the 2DMNM-Σ), and leads to a linear search through the set of possible pixel values. With the assumption that this set is not too large, and that its size may be considered a constant, we end up with an $O(d)$ time median filtering algorithm.

Were we to try to continue with binary search for the median within each d by d block, we would find that each PE would have to compare its test value against $d \times d$ separate values in each iteration of the binary search, and this would suggest that a minimum of d^2 time would be required to get the pixel values to the PEs.

What saves us is the assumption that the number of bits per pixel in the image is a constant. This is equivalent to the assumption that the number of possible pixel values is a constant. This permits us to perform a separate set of overlapping count-reduction operations for each possible pixel value in a

total of $O(d)$ time, with the actual selection of the median done either during the main loop or after all these reductions are complete. The result is that median filtering of an $n \times n$ image on an $n \times n$ with a $d \times d$ neighborhood can be accomplished in $O(d)$ time. The following algorithm summarizes this approach. (Note that output pixels near the borders get somewhat low values due to the assumption that off-image neighbors have value zero; in practice, some extra work to handle border pixels differently may be desirable). The nature of overlapping scans is illustrated in Figure **??**, and is detailed in the pseudocode algorithm description below.

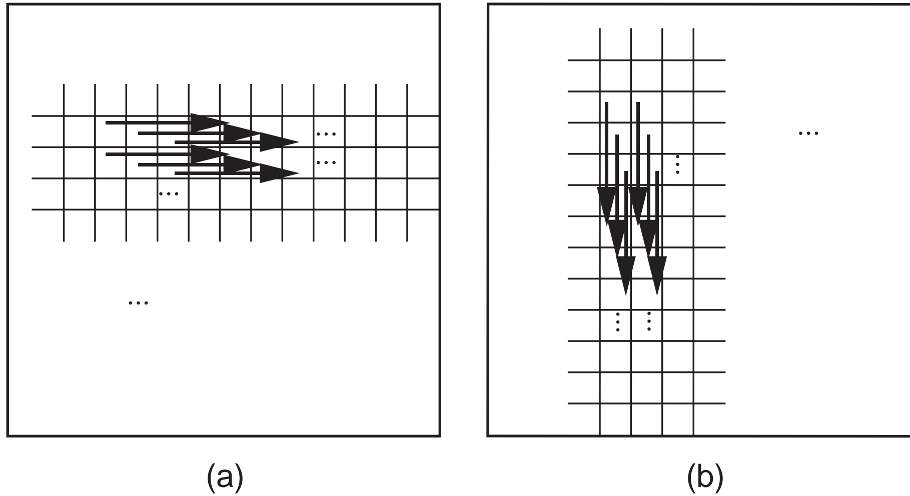


Figure 4: Computation of overlapping scans over $d \times d$ blocks: (a) horizontal phase, which computes $\texttt{hcount}[i,j] = \sum_{l=0}^{d-1} \texttt{above}[i, j-l]$, and (b) vertical phase, which completes the computation of $\texttt{count}[i,j] = \sum_{k,l=0}^{d-1} \texttt{above}[i-k, j-l]$.

```
/* Compute the median-filtered version of input image x,
   using d by d blocks.  Assume x is stored one pixel per PE.
   Output is put in "median".  */
Procedure MedianFilter(x, d, median);
  plural int x, median, above, count;
  plural boolean known;
  int d;
```

```
  known := false;
  For i := 0 to MAXVAL do
    begin
      above := 0;
      If x > i, then set above := 1;
      OverlappingBlockCount(above, d, count);
      If not known and count < d*d / 2, then
        begin
         median := i;
         known := true;
        end;
    end
  /* Now put medians back in centers of d by d blocks... */
  ShiftImage(median, -d / 2, d / 2);
  Return;

Procedure OverlappingBlockCount(above, d, count);
 /* produce counts of the "above" variable over each d by d block,
    leaving the count at the lower right corner of each block */
  plural int rowcount;

  rowcount := 0;
  For i := 0 to d
    rowcount := above + RECEIVE(WEST, rowcount);
  For i := 0 to d
    count := rowcount + RECEIVE(NORTH, count);
  Return;

Procedure ShiftImage(x, dx, dy);
  plural int x;
  int dx, dy, i;
  for i := 0 to abs(dx) do
    if dx > 0 then x = RECEIVE(WEST, x)
    else if dx < 0 then x = RECEIVE(EAST, x);
  for i := 0 to abs(dy) do
    if dy > 0 then x = RECEIVE(SOUTH, x)
```

```
  else if dy < 0 then x = RECEIVE(NORTH, x);
Return
```

The principal result for this section is that median filtering on a mesh (particularly a 2DMNM), using $d \times d$ neighborhoods, even with a simple algorithm in comparison to that of Stout[4] takes only $O(d)$ time. However, if the pixel depth is large, then the constant for this algorithm may be high enough to make another algorithm or a shortcut method attractive for some applications.

5 Operation Counts

In order to present a more practical comparison of these algorithms than is afforded by the simple use of big-O notation, a table of operation counts is provided. These values are not derived from an actual implementation, but they are justified estimates of the numbers of instructions that would be executed on a sensibly programmed mesh computer as represented by the 2DMNM model. The table is shown in Figure ??. The following paragraphs explain the basis for these counts.

	d=3				d=5				d=7			
Algorithm	1-bit	2-bit	4-bit	8-bit	1-bit	2-bit	4-bit	8-bit	1-bit	2-bit	4-bit	8-bit
Internal Sorting	346				2529				~6000			
Miller & Stout	2571	N/A			2577	N/A			2583	N/A		
Internal Scanning	602				1869				3826			
Overlapping Scans	41	73	265	4105	55	95	395	5155	69	117	405	6165

Figure 5: Table of justified estimates of the numbers of instruction executions needed by different mesh algorithms for median filtering.

The internal sorting algorithm provides a baseline of comparison for the other algorithms. The counts for the internal sorting algorithm are derived from the following assumptions. The data from the neighborhood of each

pixel is collected by its PE using a spiral scan that requires $2d^2 - 1$ instructions. In the case of $d = 3$, nine values are collected and this instruction count here is 17. Next the values are sorted by simulating a bitonic sorting network. For $d = 3$ this network is for eight inputs, and the nineth input is inserted with an extra stage of comparison and exchange operations. Each compare/exchange operation requires 10 instructions on the 2DMNM, including setting up the comparison, conditionally disabling the PE, doing the exchange via the A registers and temporary memory locations, and resetting the PE. For $d = 3$, the bitonic sorting network with the extra insertion stage involves 32 compare/exchange operations. For $d = 5$, there are 25 values to be sorted and a simple sorting network requires approximately 248 compare/exchange operations. The total count for $d = 5$ is estimated at $10 \times 248 + 49 = 2529$. When $d = 7$, the sorting network must be considerably larger, and 6000 instructions is a conservative estimate of the instruction count.

The algorithm of Miller and Stout[5] works only on binary images. It requires that row scans be performed across the entire image; using 5 instructions per column on a $n = 512$ image, 2560 instruction executions are needed to complete the scans. Then the algorithm computes "line integrals" around the neighborhoods, and this requires $3d + 2$ instruction executions. For $d = 3, 5, 7$ these counts are 11, 17, and 23, respectively. The total counts are 2571, 2577, and 2583, respectively.

The operation counts for the internal scanning algorithm can be determined by counting elementary operations in each of the six steps and adding them up. Step 1 requires $2d - 1$ operations, step 2 needs $14(2d - 1)^2$ operations, step 3 takes $18d^2 - 12d + 2$ operations, step 4 uses $8d^2$ operations, step 5 consumes $5d^2$ operations, and step 6 adds $d - 1$ operations. The totals are

$$\begin{array}{lrcr} d = 3: & 5 + 350 + 128 + 72 + 45 + 2 & = & 602 \\ d = 5: & 9 + 1139 + 392 + 200 + 125 + 4 & = & 1869 \\ d = 7: & 13 + 2366 + 800 + 396 + 245 + 6 & = & 3826 \end{array}$$

The overlapping scans algorithm requires 2^b iterations, where b is the number of bits in each pixel. Each iteration involves a call to `OverlappingBlockCount` requiring d operations and 10 operations to test whether or not the PE has the median. The counts are given in the table.

6 Shortcuts

Median filtering with larger neighborhoods becomes computationally more expensive. However, it is possible in practice to get results nearly as good as those for median filtering without expending as much time.

One good way to reduce the time required for median filtering is to do something analogous to the kernel decomposition that is often done for convolution. For example, instead of computing medians over the 25 values in a 5×5 neighborhood, one may compute row medians and column medians in two, serial steps, each dealing with only five values. The results are not generally identical to to the 5×5 median operation, but usually good enough. Two five-element sorting operations require far fewer Compare-Exchange operations than does one 25-element sorting operation. Less data movement is involved as well.

7 Discussion

There are a number of interesting questions that remain open regarding median filtering on mesh computers. One is whether efficient mesh methods for *weighted* median filtering can be developed[7,2]. Another is whether some combination of neighborhood decomposition and mesh communication could be used to reduce the $O(d^2)$ time of the internal scanning algorithm without complicating the algorithm unduly by necessitating tree manipulations as used by Ranka and Sahni[3] and Gil and Werman[2], or multiple sort steps as used by Stout[4].

8 Acknowledgements

The author thanks the anonymous referee who suggested the inclusion of the table of operation counts and several clarifications in the narrative. Some of this research is based on insights gained working under NSF Grant IRI-8605889 and support from the NASA/CESDIS program.

9 References

1. Huang, T. S. 1981. *Two-Dimensional Signal Processing II: Transforms and Median Filters.* Berlin: Springer-Verlag.

2. Gil, J., and Werman, M. 1993. Computing 2-D min, median, and max filters. *IEEE Trans. Pattern Analysis and Machine Intelligence*, Vol. 15, No. 5, pp.504-507.

3. Ranka, S., and Sahni, S. 1991. Efficient serial and parallel algorithms for median filtering. *IEEE Trans. Signal Processing*, Vol. 39, No. 6, pp.1462-1466.

4. Stout, Q. F. 1983. Sorting, merging, selecting, and filtering on tree and pyramid machines. *Proceedings of the 1983 International Conference on Parallel Processing*, pp.214-221.

5. Miller, R., and Stout, Q. F. *Parallel Algorithms for Regular Architectures.* MIT Press: To appear.

6. Thompson, C. D., and Kung, H.-T. 1977. Sorting on a mesh-connected parallel computer. *Communications of the A. C. M.*, Vol. 20, pp.263-271.

7. Brownrigg, D. R. 1984. The weighted median filter. *Communications of the Assoc. for Computing Machinery*, Vol. 27, No. 8, pp.204-208.