Proceedings of the 1983 International Conference on Parallel Processing, pp. 214-221

SORTING, MERGING, SELECTING, AND FILTERING ON TREE AND PYRAMID MACHINES

(Preliminary version)

Quentin F. Stout
Mathematical Sciences
State University of New York
Binghamton, NY 13901 USA

ABSTRACT: We develop some fundamental algorithms for d-dimensional pyramid computers, where a 1-dimensional pyramid is typically called a tree and a 2-dimensional pyramid is what is commonly meant by a pyramid computer. We give optimal $\theta(n/\lg(n))$ algorithms for sorting and merging n items stored in a tree, and show that for higher dimensional pyramids well-known algorithms are optimal. We give a selection algorithm, suitable for pyramids of all dimensions, which runs in less than $n^{**}\varepsilon$ time for any $\varepsilon > 0$. We also consider median filtering of a noisy digitized picture, giving an optimal algorithm whose running time is $\theta(D)$ when using a $D \times D$ window.

1. INTRODUCTION

Sorting, merging, and selection (i.e., finding the k'th item) are fundamental problems, and for almost any machine architecture efficient algorithms have been devised for them. However, for certain tree and pyramid computers we show that when the data is already in the base of the computer the "obvious" algorithms are not always optimal, and we present algorithms superior to any previously published. Some authors [2,3,8,11] have presented optimal sorting algorithms for tree machines in which the data passes through the root, but we are interested in situations where the data is already present and must be rearranged. Such situations arise when a different key is now being used to determine the ordering, or when data can be entered directly into the base. This latter possibility has been raised for pyramid machines devoted to image processing, for which the speed-ups introduced here may be of use. If the data must pass through the apex then all algorithms are at best linear in the number of items, and [2,3,8,11] showed that linear algorithms exist. However, in our situation one can obtain sublinear algorithms for sorting and merging, where the speed-up depends upon the dimension of the pyramid (defined below). For selection we show that the possible speed-up is even more impressive, giving an algorithm which finds, on a pyramid of any dimension, the k'th item among n in $o(n^{**}\varepsilon)$ time for any $\varepsilon > 0$.

Our interest in pyramid machines stems from our belief, and that of many others, that their geometric basis makes them a natural parallel architecture to consider for various database [2,3,11] and image processing [4,5,6,9,12,13,15,16,17,19] applications, applications in which significant parallelism is possible. Pyramids are more attractive than meshes because they have the potential of logarithmic algorithms. Furthermore, the regularity of a pyramid structure makes it feasible to construct such machines with a large number of processing elements. Several tree and pyramid machines are in various stages of design [2,3,8,11,15], and we expect that more advanced machines, with a very large number of processing elements, will be constructed.

Our results are primarily of theoretical use, in that the algorithms are somewhat complicated, but they do at least show that certain problems have solutions which are asymptotically better than was previously thought. Furthermore, our algorithms tend to use the pyramid structure in ways that are different from previously published algorithms, and perhaps such techniques will prove useful in other problems.

Throughout we will use $\lg$ to denote log base 2, and θ, Ω, O, and o to denote "order exactly", "order at least", "order at most", and "order strictly smaller than", respectively. Optimal will always mean optimal to within a multiplicative constant.

Partially supported by NSF Grant Number MCS-8301019.

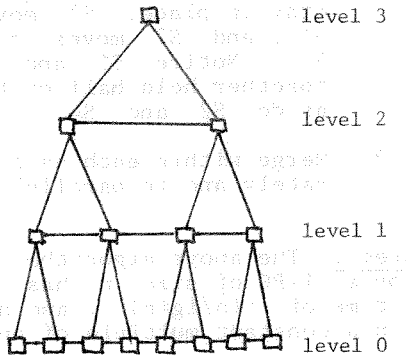
0190-3918/83/0000/0214$01.00 © 1983 IEEE

2. MACHINE MODELS

Several different models have been proposed which might naturally be called "tree" or "pyramid" machines [2,3,4,5,6,8,9,11,15,17,19]. The machines considered here can be viewed as layers of mesh-connected machines, with connections between the layers. To define a pyramid of arbitrary dimension, we first define a mesh-connected computer of arbitrary dimension. Throughout we will use d to indicate the dimension. A d-dimensional mesh-connected computer of size n**d (a d-MCC of size n**d) consists of n**d copies of a single processing element (PE), arranged in a d-dimensional cubic lattice. PEs have indices of the form (I1,...Id), where $0 \le Ii \le n-1$ for $1 \le i \le d$. PEs (I1,...,Id) and (J1,...,Jd) are neighbors if and only if $\sum_{k=1}^{d} |Ik-Jk| = 1$. Each PE (except those on sides) has 2*d neighbors and has a unit-time communication link to each of them.

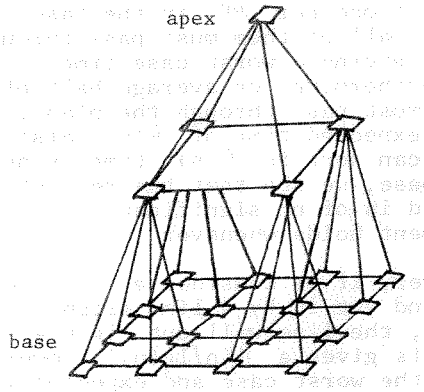
A d-dimensional pyramid computer of size n**d (a d-PC of size n**d) consists of a d-MCC of size n**d, a d-MCC of size (n/2)**d,..., and a d-MCC of size 1, along with additional communication links specified below. (Note that a d-MCC of size n**d has exactly n**d PEs while a d-PC of size n**d has between n**d and 2*n**d PEs.) The d-MCC of size n**d is called the base of the pyramid, and the d-MCC of size 1 is the apex. The d-MCC of size (n/2**k)**d is at level k, e.g., the base is level 0 and the apex is level lg(n). A PE at position (I1,...,Id) on level k is connected to all 2**d of its sons, which are those processors on level k-1 which are at a position of the form (J1,...,Jd), where each Ji is either 2*Ii or 1+2*Ii. Thus, except for processors along the sides, each PE is connected to 2**d + 1 + 2*d others, namely its 2**d sons on the level below, its parent at the level above, and 2*d neighbors at the same level. Figure 1a shows a 1-PC, typically called a tree machine, and Figure 1b shows a 2-PC, which is often what is meant by a pyramid machine. To date we know of no serious interest in pyramids of higher dimensions, but since our algorithms naturally extend to higher dimensions we have done so. We also note that there is beginning to be some interest in 3-dimensional digitizations [7], and 3-PCs are a natural architecture for such problems.

We assume that the PEs have a fixed number of registers, each of which holds a word of length $\theta(\lg(n))$, and all operations take unit time. All communication links are bidirectional, and any PE can send and receive a word along any or all links simultaneously, all taking unit time. We also assume each processor holds its level and its coordinates within that level. If this is not the case, it can easily be performed in $\theta(\lg(n))$ time. (Note: all analyses of time assume d is fixed and consider time as a function of n. While some authors [10] also consider d as a parameter, this is made difficult by the fact that a PE in a d-PC is fundamentally different from one in a (d+1)-PC). Earlier pyramid models [4,12,13,19] assumed that the PEs were copies of a finite state automation which was designed independent of n, in which case a single PE could not store the level it was on nor its position within that level. Using finite state automata also drastically changes the nature of



A 1-PC of size 8

a)



A 2-PC of size 16

b)

Figure 1

sorting, merging, and selection since there can only be a fixed number of keys, independent of n.

3. SORTING AND MERGING

[2,3,8,11] considered the problem of using machines similar to a 1-PC to perform selection, insertion, deletion, and retrieval operations. Requests arrive at the apex at unit time intervals and are performed on-line, although there is a logarithmic delay between the time a request arrives and its answer appears. This approach sorts n items in $\theta(n)$ time, and is optimal if all items are required to pass through the apex. However, we are interested in situations where the data is already in the pyramid, stored one item per PE in the base. For example, in a d-PC of size $n^{**}d$ we can sort $n^{**}d$ items in $\theta(n)$ time by ignoring everything except the base and using the d-MCC sorting algorithm in [18]. This compares quite favorably with the $\Omega(n^{**}d)$ time required if all items pass through the apex, and it is natural to ask if we can do even better by utilizing the rest of the pyramid. We prove that further speed-up is possible if and only if $d=1$.

Suppose we have a 2-PC of size $n^{**}2$ sitting "naturally" in 3-dimensional space, by which we mean the layers are all square grids parallel to each other, with each parent above the middle of its four offspring. If we consider a plane cutting perpendicular to both the base and one of its edges, and passing just to one side of the apex, we see that the plane will cut n wires at level 0, $n/2$ wires at level $1,\ldots,2$ wires at level $\lg(n)-1$, and 2 wires connecting the apex to level $\lg(n)-1$, cutting a total of $2n$ wires. To sort $n^{**}2$ items stored one per PE in the base it may be that all of them must pass through the plane, giving a worst case time of $\Omega(n)$. Furthermore, on average half of the items must pass through the plane, giving an expected time of $\Omega(n)$ also. Since one can sort in $\theta(n)$ time using only the base, we see that the rest of the pyramid is of no significant use. This argument holds whenever $d \geq 2$.

However, if we consider a 1-PC of size n and cut it by a line slightly off-center, the line will cut $\lg(n)+1$ wires. This gives a $\Omega(n/\lg(n))$ lower bound for the worst case and expected case sorting times, a bound which is unattainable using the base alone.

Theorem 1 The algorithm outlined below for sorting on a 1-PC of size n has a worst case and expected case time of $\theta(n/\lg(n))$, and is thus within a constant multiple of optimal.

Our algorithm is a simple merge sort, utilizing the fact that the two sons of the apex can be viewed as the apexes of disjoint subpyramids.

> To sort on a 1-PC of size n, $n \geq 1$, sort each half (separately and in parallel), and merge.

If $S(n)$ denotes the worst case sorting time, we have

$S(1) = 0$

$S(n) = S(n/2) + M(n) \qquad n > 1$

where $M(n)$ is the worst case time to merge two runs in a 1-PC of size n. We see that

$$S(n) = \sum_{k=0}^{\lg(n)} M(n/2^{**}k)$$

In Theorem 2 below we show that $M(n) = \theta(n/\lg(n))$, which proves Theorem 1.

MERGING

Suppose we have a run R1 of items in processors $0..k$ of the base, and a run R2 of items in processors $(k+1)..(n-1)$. We merge these into a simple run as follows:

1. Find the median of all the items (since n is even we will just use the $n/2$'th item).
2. There are 4 subruns to consider: those items in R1 less than or equal to the median (called subrun S1), those items in R1 greater than the median (S2), those items in R2 less than or equal to the median (S3), and those items in R2 greater than the median (S4). S1 and S4 stay in place, S3 moves behind S1, and S2 moves in front of S4. Notice S1 and S3 together hold half of the items, as do S2 and S4.
3. Merge within each half (separately and in parallel).

Theorem 2 The above algorithm for merging on a 1-PC of size n has a worst case time of $\theta(n/\lg(n))$, and hence is within a constant multiple of optimal.

Proof: As for sorting, the worst case time must be $\Omega(n/\lg(n))$. To show that this is attained we analyze the time spent in each step. Step 1 can be done by a simple binary search. First the median of R1 is sent up to the apex and

back down to the base, at which time each processor whose item is less than or equal to this sends up a 1 . These are summed, after which it is known if the value is too high or low. Then the median of R2 is sent up, then an appropriate quartile point of R1 , then a quartile point of R2 , etc. There are at most $\theta(\lg(n))$ probes, each taking $\theta(\lg(n))$ time, for a total of $\theta(\lg(n)^{**}2)$ time for step 1.

It is step 2 which takes most of the time. It can be reduced to two applications of the following problem: how fast can a run be moved to its destination? To do this rapidly we divide the base into $M = \lfloor \lg(n)/2 \rfloor$ blocks. PEs $0..\lfloor n/M \rfloor - 1$ are in block 1, $\lfloor n/M \rfloor .. 2*\lfloor n/M \rfloor - 1$ are in block 2, etc. We define a sequence of processors $G1,\ldots,GM$ by taking G1 to be the apex and G(i+1) as Gi's right son. For each i we construct a path Pi connecting Gi to the leftmost processor in block i , with the property that all paths are disjoint. The paths are constructed recursively, with P1 being the leftmost edge of the pyramid. For $i > 1$, Pi is construced from the base upwards, at each processor going upwards if it does not run into P(i-1) , and otherwise going right. (See Figure 2.) It can be shown that the longest path has length $o(n/\lg(n))$, and in $o(n/\lg(n))$ time each processor can decide which, if any, path it is on and which of the processors it is connected to are on the same path.

We notice that when a run needs to move, for any block i containing some items in the run there is a j such that either all of those items are to be moved to block j , or else some go to block j and the others go to block j+1 . We call block j the goal of block i , and move all the run's items from block i to block j . If some of these belong to block j+1 then we move them along the base from block j to block j+1 . We construct a path Qi from the leftmost PE in block i to the leftmost PE in block j as follows: let $k=\min(i,j)$. Qi follows Pi up to level k , then moves sideways to meet Pj , and then goes down Pj . Since any block is the goal of at most two blocks, it is easy to see that any procesor is in at most 3 of the Q paths. An item starting in block i moves left until it reaches the start of Qi , then follows Qi , and then moves right in block j and also in block j+1 if that is its destination. The total distance is $O(n/\lg(n))$, and since no procesor is in more than 3 paths the data movement can be arranged so that the worst case time for step 2 is $\theta(n/\lg(n))$.

Therefore M(n) , the worst case time to merge two runs in a 1-PC of size n , satisfies the equations:

$$M(1) = 0$$
$$M(n) \le A*\lg(n)^{**}2 + B*n/\lg(n) + M(n/2) \qquad n \ge 1$$

which gives $M(n) = \theta(n/\lg(n))$. This completes the proof of Theorem 2.

4. SELECTION

In this section we consider the problem of finding the k'th smallest item. To simplify notation we will discuss only 1-dimensional pyramids (i.e., trees), but all of our results hold for pyramids of higher dimensions. We will also simplify discussion by assuming that all the items have distinct keys.

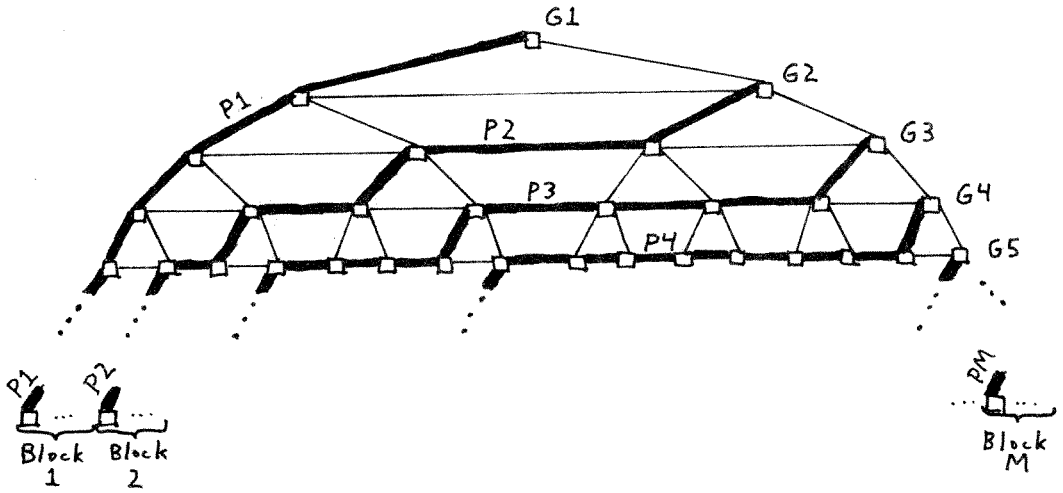


Figure 2

The easiest selection problem is k=1 , which can be solved by having each base processor pass up its item, and each higher processor pass up the smallest item received. This will take $\theta(\lg(n))$ time, and Tanimoto [16] observed that if the apex repeatedly discards the smallest item and asks the son which passed up that item to pass up another then one can find the k'th smallest item in $\theta(\lg(n)+k)$ time, which in the worst case requires $\theta(n)$ time to find the median. We can do better by sorting the base and passing up the middle item, taking $\theta(n/\lg(n))$ time. For $d > 1$ the differences are more dramatic, going from $\theta(n^{**}d)$ down to $\theta(n)$. (Tanimoto had noted this improvement for $d > 1$.) By abandoning sorting we are able to do far better, finding the k'th item in $o(n^{**}\varepsilon)$ time for any $\varepsilon > 0$.

To solve the selection problem we need to solve the weighted selection problem, in which we are given k and N pairs (vi,wi) , where vi is the value of the pair and wi is its positive integral weight. The vi are all distinct, and we want to find the vI such that

$$\sum\{wi:vi\le vI\} \ge k \text{ and } \sum\{wi:vi< vI\} < k .$$

Each of the original items in the base has a weight of 1, and intermediate calculations produce items with greater weights.

Initially each item is "active", and may later become inactive when it is known that it cannot be the answer. The algorithm is:

if n=1 then the item is the answer
else if n=2 then both items are sent to the apex, which determines the answer
else repeat
Stage 1: Each processor at level lg(n)/2 takes as its value the median of the active items beneath it, and as its weight the number of active items beneath it.

Stage 2: The apex finds the weighted median of the items found in the previous stage, call this W , and transmits it to all processors in the base.

Verification: Each base processor sends up a 1 if its item is less than or equal to W . These 1's are summed on their way up to the apex, which determines if W is the k'th item or is too large or too small. In the last two cases it sends down a message which deactivates all items as large as W , if W is too large, or all items as small as W if it is too small.
until the k'th item is found.

(This algorithm is similar to one in [14], which was for a mesh-connected computer with broadcasting.) An important feature of this algorithm is the fact that on each iteration at least 1/4 of all active items become inactive, and hence at most $\log_{4/3}(n)$ iterations are required. To see why the 1/4 appears, suppose on some iteration W was too high. The weights guarantee that at least 1/2 of all active items are beneath a processor of height lg(n)/2 which, during stage 1, picked an item at least as large as W . For each such processor on the middle level, at least half of the items beneath it are as large as W .

To see how much time this algorithm takes it is easiest to work with the height of the tree (ie, lg(n)) instead of its width. If T(h) denotes the worst case time on a 1-PC of height h , then

$$T(0) = 0$$
$$T(h) \le h^{*}\log_{4/3}(2) * [2*T(h/2) + B*h] \qquad h > 0$$

The solution of this is $0([C*h]^{**}[\lg(h/2)])$, where $C = 2*[\log_{4/3}(2)]^{**}2$, which is $o(n^{**}\varepsilon)$ for any $\varepsilon > 0$.

Theorem 3 In $o(n^{**}\varepsilon)$ time, for any $\varepsilon > 0$, the above algorithm finds the k'th item among n items stored in the base of a 1-PC of size n .

We make no claims about the optimality of our algorithm. In fact, if one uses lg(lg(n))/2 stages, instead of the 2 used above, each determining the weighted median of items determined below in the previous stage, then one can do selection in $o(\lg(n)^{**}[\lg(\lg(n))/\lg(\lg(\lg(n)))])$ time. Further fine-tuning of this approach does not seem very interesting. We conjecture that $\theta(\lg(n))$ is unattainable as a worst case time for selection, but have been unable to prove this. There is also the question of expected case time, and we do not even know the expected time of our algorithm.

5. MEDIAN FILTERING

In this section we consider the problem of median filtering of a noisy digitized picture stored one pixel per PE in the base of a 2-PC . In median filtering the idea is to replace each pixel with the median of the pixel values

in a $D \times D$ square, D odd, whose center is the original pixel. We call this square a window. Median filtering is a well-known technique (see, for example, [20] and the references therein) and is applicable to data of any dimension. Our reason for concentrating on the 2-dimensional problem is a paper of Tanimoto [16] which gives a simple algorithm. Our goal is to minimize the running time as a function of D, where we show below how to eliminate any dependence on the size of the 2-PC. We give an optimal algorithm, but we must mention that our asymptotic result is misleading since in practice D is quite small.

Tanimoto [16] noted that any dependence on the size of the pyramid could be eliminated by partitioning the image into a set of nonoverlapping regions of area $\theta(D^{**}2)$. Processors at height $\lceil \lg(D) \rceil$ are viewed as the apex of a subpyramid in which filtering is performed, with each subpyramid responsible for computing the new value for each pixel within it. Windows around pixels in one subpyramid can include pixels from an adjacent subpyramid, so first all necessary information is exchanged between adjacent subpyramids. This exchange can easily be done in $\theta(D)$ time.

Within each subpyramid Tanimoto computed the new pixel values one at a time. The total time for this method is $\theta(D^{**}2 * T(D))$, where $T(D)$ is the time needed to compute the median in a 2-PC of size $D^{**}2$. The procedure he first mentions finds the median in $\theta(D^{**}2)$ time, giving a total time of $\theta(D^{**}4)$. He also notes that by just sorting in the base the median can be found in $\theta(D)$ time, giving $\theta(D^{**}3)$ total time. By instead using the selection algorithm of the preceeding section one can perform median filtering in $o(D^{**}[2*\varepsilon])$ time for all $\varepsilon > 0$.

However, Tanimoto's approach is not very efficient as it ignores the fact that calculating the median at one point is closely related to calculating the median at any adjacent point. By exploiting this we are able to give an algorithm taking $\theta(D)$ time. A straightforward data movement analysis, as in section 3, shows that this is an optimal worst-case time. It seems that this is also an optimal expected-case time.

Theorem 4 In a 2-PC or a 2-MCC, using a $D \times D$ window, median filtering can be accomplished in $\theta(D)$ time.

The theorem is proved by the following algorithm, which is only briefly sketched. We use only the base of a

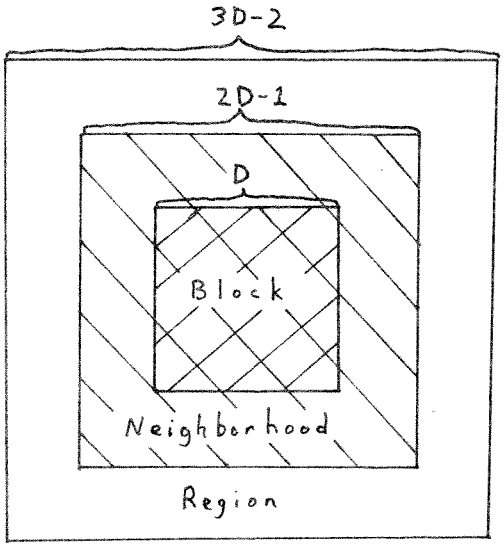


Figure 3

2-PC, partitioned into $D \times D$ blocks. Within each block each processor will determine its new pixel value. Because windows of pixels in the block can fall outside the block, the block needs to know all pixel values in a $(2D-1) \times (2D-1)$ square called a neighborhood. Also, for reasons that will be explained later, the block actually must simulate the actions of processors lying in a $(3D-2) \times (3D-2)$ square called a region. (See Figure 3.) Each processor in the block simulates the actions of a square of 9 processors from the region, with the algorithm being described as if all of the processors in the region are assisting the block.

Via sorting, each neighborhood processor determines the order position of its pixel value, e.g., a processor may determine that its pixel value is the fifth smallest in the neighborhood. Each neighborhood processor x sends this order position to the form processors c1,c2,c3,c4 indicated in Figure 4. These four processors are the corners of the square of all processors whose windows include x. Notice that the region has been chosen so that if x is in the neighborhood then c1,c2,c3, and c4 all lie within the region.

Using these corners, given any order interval there is a "spreading wave" process taking $\theta(D)$ time, after which each processor in the block will know how many pixels values in its window fall in the order interval. There are $4D^{**}2 - 2D + 1$

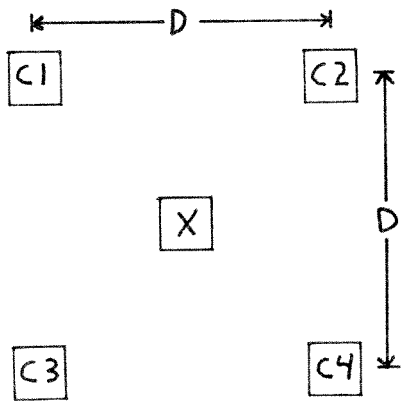


Figure 4

pixels in the neighborhood, so by using $4D$ order intervals each has at most D values. We repeat the spreading wave process $4D$ times, pipelining the waves so that it takes only $\theta(D)$ time. When finished, each processor in the block knows which order interval contains the median of its window. The processor also knows a relative description of which pixel value in the order interval is the median of its window, e.g., the processor may know that in the appropriate order interval it is looking for the third smallest pixel value belonging to its window. A final combination of sorting and searching, also taking $\theta(D)$ time, finishes the algorithm.

The algorithm sketched above can be modified to provide an optimal $\theta(D/\log(D))$ median filtering algorithms using a window of size D, for 1-dimensional data, stored in the base of a 1-PC. It can also be adapted to d-PCs and d-MCCs, $d \geq 2$, using windows of size $D^{**}d$, but the adaptions give $\theta(D^{**}(d/2))$ algorithms. The only lower bound known to the author is $\theta(D)$, so it seems that optimal algorithms for higher dimensions will require different techniques.

6. CONCLUSION

Our results can be rearranged to allow a better comparison between pyramids of different dimensions. If we have a d-PC of size n, with one item per base processor, then these n items can be sorted in $\theta(n/\lg(n))$ time if $d=1$, and $\theta(n^{**}(1/d))$ time if $d > 1$. (This holds for both worst case and expected case times.) The 1-PC algorithm is new, while the others are well-known, and all are optimal to within a constant multiple. Our 1-PC sorting algorithm depended on a new optimal merging algorithm. While there was no time to explore further here, the 1-PC sorting algorithm gives several other optimal, $\theta(n/\log(n))$ algorithms.

We also considered selection, providing an algorithm much faster than previous ones. Previous algorithms utilized sorting, which does not fit the pyramid structure very well. Pyramids provide logarithmic paths between any two processors, but if there is too much data movement, such as occurs with sorting, then the apex becomes a bottleneck. By reducing the amount of data being moved we reduced the time to $o(n^{**}\varepsilon)$ for any $\varepsilon > 0$. Thus seletion is faster than sorting on serial computers [1], mesh-connected computers with broadcasting [14], and pyramids.

Our selection algorithm also gives a faster median filtering algorithm which, when using a $D \times D$ filtering window, takes $o(D^{**}[2+\varepsilon])$ time for any $\varepsilon > 0$. However, by making better use of the fact that windows of adjacent processors overlap extensively we are able to give an optimal $\theta(D)$ algorithm. In a future paper the author will consider a variety of windowing operations on data stored in mesh-connected computers and in pyramid computers.

REFERENCES

1. A.V. Aho, J.E. Hopcroft, and J.D. Ullman, The Design and Analysis of Algorithms, Addison-Wesley, 1974.

2. M.J. Atallah and S.R. Kosaraju, A generalized dictionary machine for VLSI, Dept. E.E. Comp. Sci., Johns Hopkins Univ.

3. J.L. Bentley and H.T. Kung, Two papers on a tree-structured parallel computer, Dept. Comp. Sci., Carnegie-Mellon Univ., Rep. CMU-CS-79-142, 1979.

4. C.R. Dyer, A fast parallel algorithm for the closest pair problem, Info. Proc. Let. 11 (1980), 49-52.

5. C.R. Dyer, A quadtree machine for parallel image processing, Dept. Comp. Sci. Univ. Illinois at Chicago Circle, Tech. Rep. KSL 51, 1981.

6. A.R. Hanson and E.M. Riseman, Preprocessing cones: a computational structure for scene analysis, Univ. Mass., COINS Tech. Rep. 74C-7, 1974.

7. C.E. Kim and A. Rosenfeld, Convex digital solids, IEEE Trans. P.A.M.I. 4 (1982), 612-618.

8. C.E. Leiserson, Systolic priority queues, Dept. Comp. Sci., Carnegie-Mellon Univ., Rep. CMU-CS-79-115, 1979.

9. R. Miller, An efficient data movement technique for the pyramid computer, to appear.

10. D. Nassimi and S. Sahni, Finding connected components and connected ones on a mesh-connected parallel computer, SIAM J. Comput. 9 (1980), 744-757.

11. T.A. Ottmann, A.L. Rosenberg, and L.J. Stockmeyer, A dictionary machine (for VLSI), IEEE Trans. Comput. 31 (1982), 892-897.

12. B. Sakoda, Parallel construction of polygonal boundaries from given vertices on a raster, Dept. Comp. Sci., Penn. State Univ., Tech. Rep. CS81 1-21, 1981.

13. Q.F. Stout, Drawing straight lines with a pyramid cellular automation, Info. Proc. Let. 15 (1982), 233-237.

14. Q.F. Stout, Mesh-connected computers with broadcasting, IEEE Trans. Comp., to appear.

15. S.L. Tanimoto, Towards hierarchical cellular logic: design considerations for pyramid machines, Dept. Comp. Sci., Univ. Washington, Tech. Rep. 81-02-01, 1981.

16. S.L. Tanimoto, Sorting, histogramming, and other statistical operations on a pyramid machine, Dept. Comp. Sci., Univ. Washington, Tech. Rep. 82-08-02, 1982.

17. S.L. Tanimoto and A. Klinger (eds.), Structured Computer Vision: Machine Perception Through Hierarchical Computation Structures, Academic Press, 1980.

18. C.D. Thompson and H.T. Kung, Sorting on a mesh-connected parallel computer, Comm. A.C.M. 20 (1977), 263-270.

19. L. Uhr, Layered "recognition cone" networks that reprocess, classify, and describe, IEEE Trans. Comp. 21 (1972), 758-768.

20. Two Dimensional Digital Signal Processing II, Transforms and Median Filters, Springer-Verlag, 1981, particularly the articles by B.I. Justusson and S.G. Tyan.