Finding the median on a vector computer

Marta Woodward and *Stewart A. Levin*

ABSTRACT

Median and quantile calculations are used in geophysics for discriminating against high amplitude noise bursts. While both plotting and processing applications of the median have been presented in the past by the SEP, the latter have been neglected in practice—in part because quantile calculations have been much slower than mean and variance calculations. This paper discusses the results of a literature search for median and quantile algorithms that are well-matched to array processors and vector computers. Timing comparisons are made between a vectorizable scheme and the nonvectorizable program currently used by the SEP. Although the vectorizable algorithm requires 50% more comparisons to find the median than its nonvectorizable counterpart ($3n$ as opposed to $2n$ for an array of length n), it runs six times faster on a Convex C1-XP. An algorithm that finds the median with an asymptotic worst case of $3n$ comparisons is also outlined.

INTRODUCTION

In 1973, Claerbout and Muir observed that the median is a more robust statistical average than the mean: its spike-rejecting quality makes it a more stable estimator of erratic data containing high amplitude noise bursts. Because of this property, median and quantile calculations are routinely used in the SEP for rescaling seismic data for plotting (Mehta, 1977). Schemes have also been described for using the median in stacking and deconvolution (Woodward and Dellinger, 1984; Woodward, 1985). These processing applications have not gained widespread acceptance in the SEP, partly because calculation of quantiles is more time consuming than calculation of means and standard deviations. The problem has been exacerbated by the SEP's use of *quant* (Canales, 1977) as a quantile program: nonvectorizable, the algorithm fails to exploit modern vector computer architecture. This paper describes a vectorizable quantile algorithm *vquant* based on a vector sorting scheme presented

by Stone (1978). Timing comparisons are made between *quant* and *vquant* which show the latter to run approximately six times faster than the former on a Convex C1-XP. Hopefully, the availability of this faster median algorithm will encourage use of the median in processing applications.

QUICKSORT

The median (or, equivalently, quantile) programs presented in this paper are based on Hoare's *quicksort* algorithm—a full-sort, partition-exchange scheme (Hoare, 1962). Elements smaller and larger than a selected element (or key) are placed at opposite ends of an array through a series of moves or exchanges. When the key is finally placed in its sorted position, the array is partitioned: elements larger or smaller than the key lie on either side of the key. The procedure is recursively repeated on the smaller, partitioned arrays until the entire array has been sorted. Assuming a reasonable (linear-time) partition method, the approach uses on the order of $n\log_2 n$ comparisons to complete a full sort, where n denotes array length.

Stone (1978) discusses partition algorithms that are well-suited for vector computers. A ratfor subroutine coded from one of these algorithms and compatible with *quicksort* is shown in Figure 1 as *jpart*.

VQUANT AND QUANT

Quicksort and *jpart* may be directly adapted for quantile calculation by successively discarding those partitions which do not contain the desired quantile—thereby partitioning a progressively smaller array until the quantile is found. A ratfor version of this algorithm is shown in Figure 2 as *vquant*. The algorithm requires on average $3n$ comparisons for finding the median—as shown in Figure 3a.

The median routine used in the past by the SEP is shown in Figure 4 as *quant*. This in-place scheme is more clever than *vquant* in that the array is completely partitioned only once, being divided around the chosen quantile position k in its last step. The algorithm recognizes that an array need not be fully partitioned in order to rule out possible quantile candidates; it prunes the array as the calculation proceeds—immediately discarding those ends that cannot contain the kth element. This trick reduces the average number of comparisons for finding the median to only $2n$ (see Figure 3a); unfortunately, it also prevents *quant* from vectorizing.

Timing comparisons between *quant*, and vectorized and nonvectorized implementations of *vquant*, are shown in Figure 3b. The algorithms were used to select the median from odd-length arrays of random numbers on a Convex C1-XP computer. Each array length was run for 2400 different arrays; the same random arrays were used for each algorithm. The times shown on the vertical axis are in seconds of cpu time for 2400 runs. Because the Convex Fortran compiler vectorized *jpart* inefficiently, an assembly code version was used for the timing runs. Although

```
#
# partition subroutine
#
# j=jpart(x,y,n)
#
# x is input vector of length n
# y is output vector of length n
# n is vector length
#
# the elements of x move to y such that: x(1) moves
# to y(j); y(1)...y(j-1) are all less than y(j); and
# y(j+1)...y(n) are all not less that y(j). j is returned.
#
# because the Convex Fortran compiler vectorized this code inefficiently,
# our timings were done for an assembly language version instead.
#
      integer function jpart(x,y,n)
      integer n
      real x(n), y(n)

      integer i, ibot, itop
      real x1, xi

      ibot=1
      itop=n
      x1=x(1)
      do i=2,n {
          xi=x(i)
          if(x1.gt.xi) {
              y(ibot)=xi
              ibot=ibot+1
          }
          else {
              y(itop)=xi
              itop=itop-1
          }
      }
      y(ibot)=x1
      jpart=ibot
      return
      end
```

FIG. 1. *Jpart.r*: a ratfor version of a vectorizable partition algorithm.

```
#
# vquant: vectorizing quantile finder
#
# x is input vector of length n
# n is vector length
# k is the position of the element of interest
# y is a work vector of length n
#
      subroutine vquant(x,n,k,y)
      integer n, k
      real x(n), y(n)

      integer first, last, i, j, jpart, m
      real t

      first=1; last=n
      while (first.lt.last) {
          m=last-first+1
          j=first+jpart(x(first),y,m)-1
          call vcopy(y,x(first),m)
          if(j.gt.k) last=j-1
          else if(j.lt.k) first=j+1
          else first=last
      }
      return
      end

#
# vector copy subroutine
#
# x is input vector of length n
# y is output vecttor of length n
# n is vector length
#
      subroutine vcopy(x,y,n)
      integer n, i
      real x(n), y(n)

      do i=1,n
          y(i)=x(i)
      return
      end
```

FIG. 2. *Vquant.r*: a vectorizable, ratfor, quantile-finding program. The call to *jpart* refers to Figure 1.

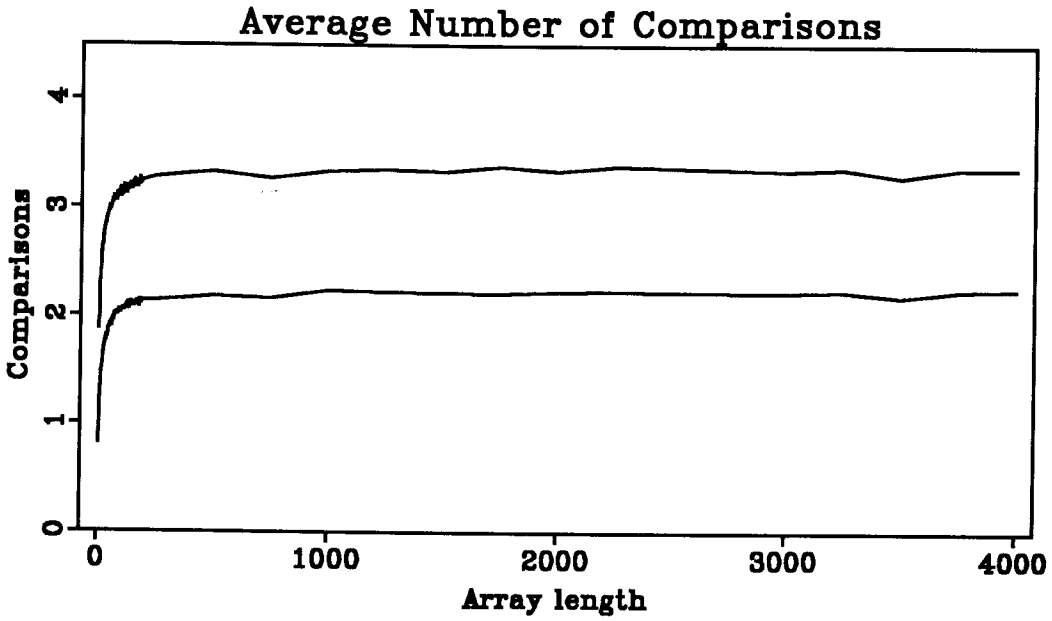

FIG. 3a. The number of comparisons required by *vquant* (upper curve) and *quant* (lower curve) for finding the median—averaged over 2400 runs for each array length.

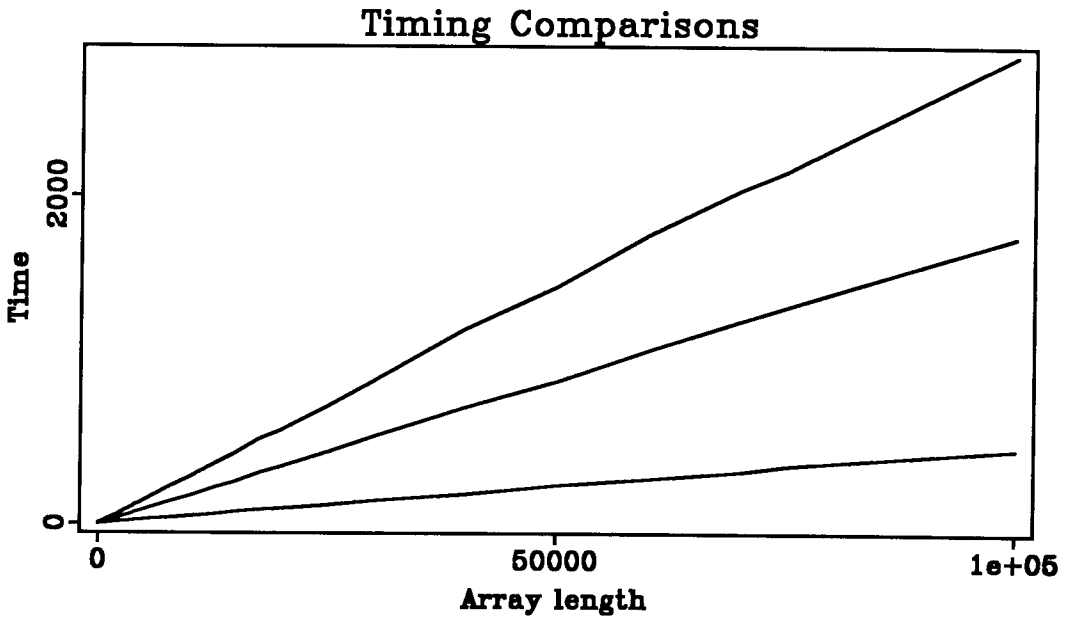

FIG. 3b. Timing comparisons between vectorized *vquant*, scalar *vquant* and *quant* (bottom, middle and top curves, respectively). The times shown on the vertical axis are in seconds of cpu time for 2400 runs.

```
#
# quant: nonvectorizing quantile finder
#
# k is the position of the element of interest
# a is input vector of length n
# n is vector length
#
      subroutine quant(k,a,n)
      integer k, n
      real a(n)

      integer low, hi, i, j
      real aa, ak

      low=1; hi=n
      while (low.lt.hi) {
          ak=a(k)
          i = low
          j = hi
          repeat
              if (a(i).lt.ak)
                  i = i+1
              else {
                  while (a(j).gt.ak)
                      j = j-1
                  if (i.gt.j)
                      break 1
                  aa = a(i)
                  a(i) = a(j)
                  a(j) = aa
                  i = i+1
                  j = j-1
                  if (i.gt.j)
                      break 1
              }
          if (j.lt.k)
              low = i
          if (k.lt.i)
              hi = j
      }
      return
      end
```

FIG. 4. *Quant.r*: a nonvectorizable, ratfor, quantile-finding program.

vquant requires 50% more comparisons per array length than *quant*, it runs almost six times faster. Somewhat more than half of this improvement results from the exploitation of vector computer architecture; the remainder is due to optimized assembler coding.

OTHER ALGORITHMS

Vector algorithms have a problem in that they become increasingly less efficient for shorter vectors. A standard countermeasure is to process many such vectors simultaneously. By forming new arrays with one element from each short vector, new opportunities open up for vectorization. For the above *quicksort*-based routines this strategy does not help. These quantile-finders are linear-time methods only on average; for some inputs they can run much more slowly, with worst-case performance proportional to n^2 instead of n. Computing many medians simultaneously produces execution times proportional to the number of medians multiplied by the slowest individual median calculation. Consequently, in computing many short medians simultaneously the average number of calculations increases—eating up the savings of reduced vector overhead.

Fortunately, other variations on the *quicksort* methods do guarantee linear performance. Bleich and Overton (1983) give such an algorithm for the more general weighted median problem. By making an intelligent choice for the trial median (key) at each partition, they avoid selecting extreme outliers and insure the elimination of a certain fraction of the elements at each step. This selectivity in choosing the keys guarantees linear performance.

At present, the method requiring the fewest number of comparisons is the non-*quicksort* algorithm of Schönage et al. (1973)—with a worst-case performance asymptotically proportional to $3n$. The scheme in effect preprocesses its input array: it creates a group of partially ordered subsets; sorts these subsets by their medians; discards the outlying elements, then passes a greatly reduced array to an algorithm like *quant* or *vquant*. While the algorithm is adaptable to computing simultaneous medians on a vector machine, it has a high overhead cost which probably makes it unsuited for the small arrays usually encountered in geophysical applications.

CONCLUSIONS

Following suggestions from the literature, we were able to markedly improve the speed of median calculations on our Convex vector computer. The majority of the improvement came from using a less efficient, but vectorized method of partitioning an array into those elements less than and those greater than a given value. This approach will also speed up the computation of quantiles and weighted means, an important component of L^1 minimization. We hope this will encourage a more

widespread use of these robust statistical measures in geophysical data processing, both here at the SEP and elsewhere throughout the exploration industry.

REFERENCES

Bleich, C., and Overton, M.L., 1983, A linear-time algorithm for the weighted median problem: Computer Science Department Technical Report no. 75, Courant Institute.

Canales, L., 1976, A quantile finding program: SEP-10, pp. 99-100.

Claerbout, J.F., and Muir, F., 1983, Robust modeling with erratic data: Geophysics, vol. 38, no. 5, pp. 826-844.

Hoare, C.A.R., 1962, Quicksort: The Computer Journal, vol. 5, no. 1, pp. 10-15.

Mehta, S., 1977, Equalizing gain on seismic sections by quantiles: SEP-11, pp. 205-210.

Schönhage, A., Paterson, M., Pippenger, N., 1973, Finding the median: Theory of Computation Report no. 6, Department of Computer Science, University of Warwick.

Stone, H.S., 1978, Sorting on STAR: IEEE Transactions on Software Engineering, vol. SE-4, no. 2, pp.138-146.

Woodward, M.J., 1985, Statistical averages for velocity analysis and stack: median vs. mean: SEP-42, pp. 97-111.

Woodward, M., and Dellinger, J., 1984, Median spectra: SEP-41, pp. 35-50.