The CSI Journal on
Computer Science and Engineering
Vol. 14, No. 1, 2016
Pages 1-5
Regular Paper

A Comparison-Based Algorithm for Hardware- and Software-Based Median-Finding in Signal Processing Applications

Behrooz Parhami[1] Saleh Abdel-hafeez[2] Arwa Damir[2]

[1]Department of Electrical and Computer Engineering, University of California, Santa Barbara, California, USA
[2]Faculty of Computer and Information Technology, Jordan University of Science and Technology, Irbid, Jordan

Abstract

Finding the median value in a list of numbers is an important computational task that is useful for a variety of problems in domains such as computing, networking, signal processing, and remote sensing. Algorithms with worst-case running times that vary linearly with the problem size are known. As is the case for sorting, however, algorithms with non-optimal worst-case running times but with better average performance for problem sizes of practical interest do exist. We devise one such algorithm based on comparisons, analyze its running time, and experimentally verify its performance for applications with small-to-moderate input lists.

Keywords: Algorithm, Average-Case Running Time, Computational Complexity, Median, Worst-Case Analysis.

1. Introduction

Algorithms for selection, also known as "order statistics," particularly the special case of median-finding, are of great interest in computer science and engineering, and are thus taught widely [1] [2]. One application of median selection is in sorting via dividing the array into two halves. Perhaps the most widespread application of median finding in signal processing is for smoothing of digital images, where each pixel value is replaced by the median of pixel values in a predefined neighborhood [3]. In this context, the median value provides a good alternative to the mean value, given its lower sensitivity to outliers [4]. In multichannel redundant computations, median voting comes with similar advantages.

We have performed an extensive survey of median-finding algorithms for software implementation [5] [6] [7] [8] [9] and their associated complexity results [10] [11] [12] [13] [14]. Here is a brief summary. For very large n, algorithms with optimal running time $\Theta(n)$ exist. Current lower and upper bounds for median finding are 2n + o(n) and 3n + o(n), respectively (the best bounds today are a tad closer to each other, but the approximations just cited suffice for our purposes). For certain small values of n, ad-hoc algorithms are known that do better in terms of the leading constant of $\Theta(n)$**,** but they cannot be generalized.

Median-finding with small input lists can be easily realized by taking advantage of hardware sorting/selection networks [15] [16], which are particularly efficient for 0-1 inputs. The special case of 9 inputs, which is useful in smoothing operations for image processing, is readily realizable on FPGAs [17] to provide hardware assist for performance enhancement. Similar hardware methods would be applicable to other small values of n. We have been unable to find software algorithms that are particularly efficient for small- to moderate-size lists.

On occasion, an approximate or probabilistically correct median value may suffice. For example, the algorithm of Battiato et al. [18] usually provides a good estimate of the median in linear time; on average, it performs 4n/3

comparisons and n/3 exchanges. Sen [19] considers the theoretical PRAM CRCW parallel model for a constant-time solution, with high probability. Online or incremental selection algorithms have also been considered [20] [21].

The aforementioned efforts and variations foretell the importance of computationally efficient median finding in a wide array of contexts. Hence new algorithms that might prove efficient in some settings or applications are useful. After describing our new median algorithm in Section 2, we outline a simulation strategy for assessing it in Section 3. Sections 4-6 are devoted to experimental results, theoretical considerations, and conclusions, respectively.

```
A comparison-based median-finding algorithm
 1. Input: integer list Data[0 : n − 1] of odd length n
 2. Output: integer median value m in Data
 3. for i = 0 to n − 1 do
 4.     x[i] ← n × Data[i] + i
 5. endfor
 6. k ← (n − 1)/2; m ← x[k]; left ← k; right ← k
 7. while left ≠ 0
 8.     left ← left − 1; right ← right + 1
 9.     if m < x[right]
10.     then if m < x[left]
11.         then if x[left] < x[right]
12.             then m ↔ x[left]; left ← k; right ← k
13.             else m ↔ x[right]; left ← k; right ← k
14.             endif
15.         endif
16.     else if x[left] < m
17.         then if x[left] < x[right]
18.             then m ↔ x[right]; left ← k; right ← k
19.             else m ↔ x[left]; left ← k; right ← k
20.             endif
21.         endif
22.     endif
23. endwhile
24. m ← ⌊m/n⌋
```

Figure 1. Our median-finding algorithm in pseudocode

2. Algorithm Description

A pseudo-code description of our algorithm is seen in figure 1. We have implemented the algorithm and used the resulting program to obtain the experimental results described in Section 4 of the paper.

The algorithm modifies our original input list Data into the internal list x with no repeated elements. We remove repeated elements to avoid getting into infinite loops as a result of exchanging equal values back and forth. The modification is performed on lines 3-5 of the algorithm.

The rest of the algorithm makes multiple passes through list x of odd length n, scanning it outward in both directions, starting at the center. Once it finds a pair of elements x[k – j] and x[k + j] such that the middle element m = x[k] does not fall between them, it swaps one of the elements with m to meet the condition and restarts from the middle. If a scan reaches the edges of the list, the middle element is the desired median.

Here is a line-by-line description of the algorithm. Line 6 initializes the center index k, the median m and the pointers left and right used to scan the list from the center to its ends. The while-loop on lines 7-23 is where the scanning passes occur. If a scan is completed (ending with left = 0), the execution ends.

Line 8 updates the pointers so that the scan can move outward. The first execution of this line sets the pointers to start values k – 1 and k + 1, respectively. Subsequently, the pointers are updated in each iteration, both getting reset to k when swapping occurs on one of the lines 12, 13, 18, or 19.

Lines 9-10 of the algorithm are used to establish whether x[left] < m < x[right]. If so, then no action is needed and we continue with the scan. Otherwise (corresponding to the "then" part on lines 11-15), the smaller of the two values x[left] or x[right], that is, the one that is closer to m, is exchanged with m.

Lines 16-21 serve a similar purpose to lines 10-15, except that they detect the condition x[left] > m > x[right], where no action is required, and perform the required swap if the condition is not met.

3. Simulation Methodology

We used exhaustive evaluation for n up to 17, taking advantage of symmetries to reduce the running time towards the upper end of this range. For n ≥ 19, we used randomly generated samples to estimate the average and worst-case performance. We began by taking a sample of 10,000 randomly generated input lists and ran simulations to find the worst-case and average-case numbers of comparisons. We evaluated the quality of the results by noting whether multiple runs with the same sample size led to nearly identical results, increasing the sample size gradually, up to 1,500,000, in case of discrepancies.

We also ran such randomized experiments for smaller values of n for which exact results had already been obtained, to provide a kind of cross-checking for the quality of simulation results.

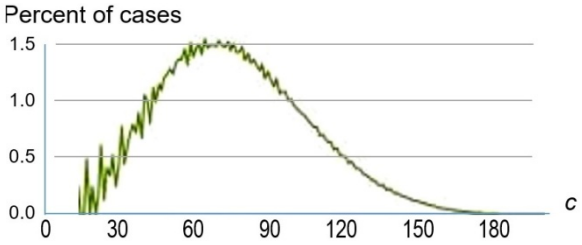


Figure 2. Distribution of the number c of comparisons needed for median-finding in an input list of size n = 15

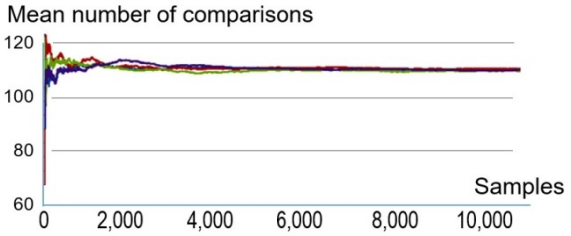


Figure 3. Variations in the average running time, as more samples are added to the simulation run for n = 19. All four runs converge to the same average, indicating robustness

Initial estimate for the required sample size was derived by noting the distribution of the number of comparisons performed by our algorithm, which in every case was very

similar to the bell-shaped curve in figure 2 (which pertains to n = 15). The curves all have peaks very close to the average number of comparisons and slightly longer tails than an ordinary normal distribution. The high peaks and relatively small variances bode well for accurate assessment of the average number of comparisons from a moderately sized random sample of inputs.

Figure 3 shows quick convergence to the actual average running time for n = 19, as more samples are introduced on an incremental basis.

4. Experimental Results

Our experimental results based on the simulation methodology of Section 3, are shown in Table 1 and plotted in figure 4. From figure 4, the quadratic lower bound for the worst-case running time and the super linear average case are evident. It is also seen that the mean/max ratio declines with increasing n. The latter trend is good news, as it indicates a sub quadratic average running time or, at least, quadratic time with a fairly small coefficient.

Lines 1-3 of Table 1 yield the following quadratic fit to the number of comparisons: $0.295n^2 + 0.440n + 0.025$. Lines 4-6 provide $0.305n^2 – 0.315n + 4.46$. Because there is good agreement on the second-degree term, we hypothesize the function $c(n) = 0.3n^2 + bn + c$ for the combined range interval [3, 13] of n, leading to $0.3n^2 + 2.04$ as a reasonable approximation with a maximum error of ±0.83 in the range above. Checking the expression against the result for n = 15 and 17, we see that c (15) and c (17) have comparable errors of 0.94 and 0.53, respectively.

Table 1. Best-, average-, and worst-case number of comparisons for small input lengths

n	Min	Max	Mean	Mean/Max
3	2	5	4.00	0.80
5	4	15	9.60	0.64
7	6	32	16.97	0.53
9	8	55	26.33	0.48
11	10	85	37.90	0.45
13	12	126	51.91	0.41
15	14	179	68.60	0.38
17	16	242	88.21	0.36

Table 2. Number of comparisons for larger input lengths, obtained via running random samples of cases

n	Min	Samples/Trials	Max	Mean	Mean/n^2
11	10	10K / 4	83	37.91	0.31
13	12	10K / 4	123	51.98	0.31
15	14	10K / 4	171	68.47	0.30
17	16	10K / 4	230	88.13	0.30
19	18	10K / 4	286	111.11	0.31
21	20	10K / 4	338	137.48	0.31
23	22	10K / 4	432	166.46	0.31
25	24	100K / 4	552	200.52	0.32
35	34	150K / 4	1267	434.86	0.35
45	44	150K / 4	2181	800.67	0.40
55	54	150K / 4	3775	1328.36	0.44
65	64	300K / 4	5722	2046.75	0.48
75	74	450K / 4	7884	2987.14	0.53
85	84	1500K / 4	11,085	4178.53	0.58
95	94	1500K / 4	14,427	5648.92	0.63

We also conducted randomized simulation experiments for n = 19, 21, 23, 25, and for other selected values of n under 100. We performed the same kind of randomized experiments for n = 11, 13, 15, 17, as a way of cross-checking the viability of the randomized approach by comparing its results to known exact values in Table 1. The cross-checked results (exact versus randomized) showed excellent agreement in every case.

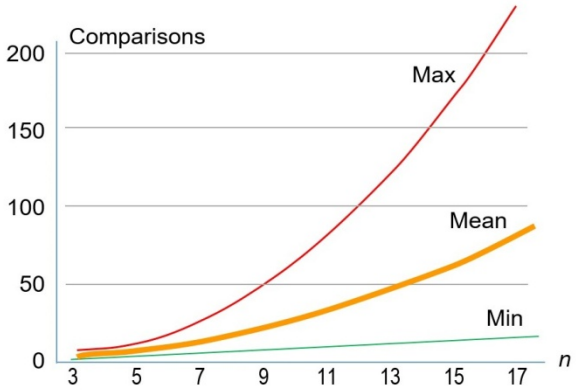


Figure 4. Number of comparisons needed for finding the median in a list of size n

The results of our randomized simulations are shown in Table 2. In all these control cases (first 4 lines of Table 2), the maximum number of steps obtained for the random experiment was within 5.0% of the actual value and the aggregate mean was within 0.2%, with no more than 0.9% change from one run to another. The larger error for the maximum is to be expected, given the long tail of the example distribution of the number of comparisons depicted in figure 2.

5. Theoretical Considerations

Even though our presented algorithm is intended for use on relatively small data sets, it is nevertheless instructive to analyze its behavior theoretically and asymptotically. We begin by addressing the issue of termination, which is important, regardless of performance issues.

Theorem 1: Our median-finding procedure always stops after a finite number of steps.

Proof: Considering the middle element y and the two symmetrically located elements x and z that are involved in an exchange, we note that the exchange transforms xyz to yxz when x > y and x < z (other cases are similar). In this case, the new elements ending up in the locations originally occupied by x and z have a larger difference than |x – z|. Thus, using k as the midpoint index, every exchange will increase the value of

$$D = \Sigma_{1 \le i \le (n-1)/2} |x_{k+i} - x_{k-i}|$$

by the difference of two of the input numbers. For the example input (0, 7, 4, 5, 1, 8, 3, 6, 9), the sum above assumes the values 14, 18, 19, 21, and 22, respectively. Given that the sum of the symmetric differences in the expression above has a finite maximum possible value, the procedure cannot go on indefinitely.

We next show that for suitably large input size n, the worst-case number of comparisons in our algorithm is at

least quadratic in n. This result essentially rules out the use of our algorithm on large input lists, hence our emphases in the title, abstract, and body of the paper that the algorithm is suitable only for relatively small input lists found in certain signal processing applications. But there are two redeeming factors: (1) The average case is usually much better than the worst case, as reflected in our experimental results; (2) We are working on hardware implementations for which the latency isn't proportional to the number of comparisons, but to the number of passes, which is a factor of n smaller than the worst-case number of comparisons.

Theorem 2: For large n, our median-finding algorithm requires at least $\Omega(n^2)$ comparisons in the worst case.

Proof: We actually prove the worst-case lower bound for finding the median of a sequence consisting of 0s and 1s only, that is, a potentially simpler problem which is a special case of general median-finding. In each iteration of the algorithm, the middle value is compared against two elements at equal distance to its left and right. When the three elements are 000, 001, 011, 100, 110, or 111, no exchange takes place. On the other hand, for the patterns 010 and 101, exchange will take place, turning the pattern into 001, 100, 011, or 110. The new patterns 0y1 and 1y0 will not lead to any additional exchanges in future passes, regardless of the value y that appears in the middle. On the other hand, we can construct an initial string of 0s and 1s so that exchanges occur at positions ±1, ±2, ±3, ... from the middle element in consecutive scans. Thus, the total number of comparisons will be exactly

$$C = \Sigma_{1 \le i \le (n-1)/2} (2i) + (n - 1)/2 + (n - 1) = (n + 7)(n - 1)/4,$$

where 2i represents the i comparison pairs until we get to the point where an exchange is needed, the second term accounts for the extra (n – 1)/2 comparisons for the single exchange in each pass, and the last term, n – 1, is for comparisons in the final scan without any exchanges. Consider as an example the input 10101. We need 3 comparisons in the first pass (after which, the sequence becomes 10011), 5 in the second pass (00111), and 4 in the final pass, for a total of 12 comparisons, which agrees with (5 + 7) (5 – 1)/4, suggested by the formula above.

6. Conclusion and Future Work

Our median-finding algorithm is suitable for hardware and/or software implementation for input lists of small to moderate sizes, for which the benefits of asymptotically linear-time algorithms do not kick in.

A natural question is whether other median-finding algorithms with similar asymptotic time complexities exist and if so, how our algorithm compares with them. The simple bubble-sort and selection-sort algorithms have quadratic complexities and can be used to find the median value in a list, which will be the middle element after sorting. Selection-sort is particularly suitable for this purpose, because it can be terminated once half of the list has been sorted. In fact, sorting-based median-finding algorithms are quite common in practical applications.

A possible direction for extending this research is to conduct detailed comparisons between our algorithm and competing ones to better delineate the conditions (input list size, distribution of values in the list, etc.) under which each algorithm has a performance edge. It is likely that the relative performance benefits of each approach will end up being language-and machine-dependent; nevertheless, it may be possible to draw some general conclusions and practical guidelines from the data.

We are also considering variations on our algorithm to see if its average-case or worst-case performance can be improved. One such variation is to continue the outward scan after an exchange has occurred, rather than begin a new scan right away. We have observed that this variation improves the performance in some examples formerly constituting the worst case, but it is not clear that new worst-case patterns won't arise for the modified version.

References

[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. New York: McGraw-Hill, 2nd ed., 2003. (Chapter 9: "Medians and order statistics," pp. 183-196).

[2] C. Gurwitz, "On Teaching Median-Finding Algorithms," *IEEE Trans. Education*, vol. 35, no. 3, pp. 230–232, 1992.

[3] J. C. Ross, *the Image Processing Handbook*, Boca Raton: CRC Press, 6th ed., 2011.

[4] I. Pitas, and A. N. Venetsanopoulos, *Nonlinear Digital Filters: Principles and Applications*. Berlin: Springer, 1990.

[5] D. J. Bernstein, "Fast Multiplication and Its Applications," in *Algorithmic Number Theory*. London: Cambridge, 2008, pp. 325–384.

[6] D. Dor, and U. Zwick, "Selecting the Median," *SIAM J. Computing*, vol. 28, no. 5, pp. 1722–1758, 1999.

[7] K. C. Kiwiel, "On Floyd and Rivest's SELECT Algorithm," *Theoretical Computer Science*, vol. 347, pp. 214–238, 2005.

[8] D. R. Musser, "Introspective Sorting and Selection Algorithms," *Software Practice and Experience*, vol. 27, no. 8, pp. 983–993, 1997.

[9] A. Schonhage, M. Paterson, and N. Pippenger, "Finding the Median," *J. Computer and System Sciences*, vol. 13, pp. 184–199, 1976.

[10] S. W. Bent, and J. W. John, "Finding the Median Requires 2*n* Comparisons," in *Theory of Computing. Proc. 7th ACM Symp.* 1985, pp. 213–216.

[11] M. Blum, R. W. Floyd, V. Pratt, R. Rivest, and R. Tarjan, "Time Bounds for Selection," *J. Computer and System Sciences*, vol. 7, pp. 448–461, 1973.

[12] W. Cunto, and J. I. Munro, "Average Case Selection," in *Theory of Computing. Proc. 16th ACM Symp.* 1984, pp. 369–375.

[13] R. W. Floyd, and R. R. Rivest, "Expected Time Bounds for Selection," *Communications of the ACM*, vol. 18, no. 3, pp. 165–172, 1975.

[14] M. S. Paterson, "Progress in Selection," in *Algorithm Theory. Proc. 5th Scandinavian Workshop*. 1996, pp. 368–379.

[15] D. Knuth, *the Art of Computer Programming—vol. 3: Sorting and Searching*. Boston: Addison-Wesley, 2nd ed., 1998.

[16] B. Parhami, *Introduction to Parallel Processing: Algorithms and Architectures*. New York: Plenum, 1999.

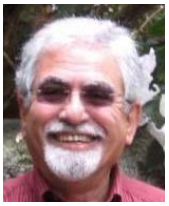
[17] J. L. Smith, "Implementing Median Filters in XC4000E FPGAs," *Xilinx XCELL Journal*, iss. 23, p. 16, 1996.

[18] S. Battiato, D. Cantone, D. Catalano, G. Cincotti, and M. Hofri, "An Efficient Algorithm for the Approximate Median Selection Problem," in *Algorithms and Complexity*. Berlin: Springer, LNCS vol. 1767, 2000, pp. 226–238.

[19] S. Sen, "Finding an Approximate Median with High Probability in Constant Parallel Time," *Information Processing Letters*, vol. 34, pp. 77–80, 1990.

[20] L. Denenberg, "Online Median Finding," Draft paper available: http://denenberg.com/omf.pdf.

[21] R. R. Mettu, and C. G. Plaxton, "The Online Median Problem," *SIAM J. Computing*, vol. 32, no. 3, pp. 816–832, 2003.



Behrooz Parhami earned a PhD degree in computer science from University of California, Los Angeles, USA, in 1973. He is currently Professor of Electrical and Computer Engineering, and former Associate Dean for Academic Personnel, College of Engineering, at University of California, Santa Barbara, USA, where he teaches and does research in computer arithmetic, parallel processing, and dependable computing. A Life Fellow of IEEE, a Fellow of IET and British Computer Society, and recipient of several other awards (including a most-cited paper award from J. Parallel & Distributed Computing), he has written six textbooks and more than 290 peer-reviewed technical papers. Professionally, he serves on journal editorial boards and conference committees and is also active in technical consulting.
Email: parhami@ece.ucsb.edu

Saleh Abdel-hafeez received a PhD in computer engineering from the University of Texas at El Paso and MS from New Mexico State University. He was a senior member of technical staff at S3 Inc. and Viatechnologies.com in the area of mixed-signal IC design. He also was an Adjunct Professor of Computer Engineering at Santa Clara University from 1998 to 2002. He has two US patents, numbered 6,265,509 and 6,356,509, with S3 Inc. His current research interests are in the areas of high speed ICs, computer arithmetic algorithms, and mixed-signal design. Dr. Abdel-hafeez is currently the Chairman of Computer Engineering Department at Jordan University of Science and Technology.
Email: sabdel@just.edu.jo

Arwa Damir earned a BS degree, with honors, in Electrical and Computer Engineering from Jordan University of Science and Technology in 2000. Since then, she has been a staff member of Computer Engineering Department, supervising instructional labs on hardware description languages and microprocessor design. **Email:** d-arwa@just.edu.jo

Paper Handling Data:

Submitted: 11.01.2017
Received in revised form: 21.02.2017
Accepted: 12.03.2017
Corresponding author: Dr. Behrooz Parhami,
Department of Electrical and Computer Engineering, University of California, Santa Barbara, California, USA.