# An Efficient Partial Sort Algorithm for Real Time Applications

DAVID DOMINIQUE[(1)*], FAITH DOMINIQUE[(2)]
[(1)] University of California, Santa Cruz, U.S.A
[(2)] Sunnyvale, U.S.A
[*] ddominiq@ucsc.edu

**Abstract:** An efficient partial sort algorithm that is less complex, and more practical to implement compared to existing partial sort algorithms especially for real time applications is presented. The proposed partial sort algorithm does not use pivots, but instead makes use of past sorting decisions to narrow the search for each new sorted element. A major advantage of the proposed partial sort algorithm is that its complexity and sorting time are fixed, and are not dependent upon the data to be sorted, unlike other algorithms. This makes it a good candidate for real time applications.

**Keywords:** partial sort, pivots, complexity

## 1. INTRODUCTION

Sorting is one of the most fundamental signal processing algorithms used today in a variety of applications and many algorithms have been developed for a full sort of a list of *L* elements. However, in many applications, only a partial sort of the *L* elements is required. An obvious solution is a complete sort of all the *L* elements. However, this is overkill and many partial sort algorithms have been proposed [1][2][3]. Reference [1] implements a partial heapsort by building a heap of size *N* and then picking the *N* minimums or maximums, which is not optimal from a complexity point of view. Another category of algorithms uses pivots [2][3] to select the $N^{th}$ element and then sort the elements based on the pivot. Based upon the choice of the pivot, complexity can range from $O(L)$ to $O(L^2)$. An additional source of complexity that is seldom addressed is that associated with re-arranging the elements with respect to the pivots. This re-arrangement results in increased processing latency and memory bus usage. These are major disadvantages for real time applications since the complexity of the algorithm and sorting time are data and pivot choice dependent.

The proposed partial sort algorithm does not use pivots, but instead makes use of past sort decisions to narrow the search for each sorted element. This reduces the complexity of the algorithm. A major advantage of the proposed partial sort algorithm is that its complexity and sorting time are fixed for a given *N* and *L*, and are not dependent upon the data to be sorted, unlike other algorithms [2][3]. This makes it a good candidate for real time applications where a fixed processing time budget and computing power have to be used. The proposed algorithm is especially well suited for implementation on a Digital Signal Processing (DSP) processor due to its inherent pipelining and parallelization potential. The complexity of the proposed algorithm is derived and comparisons of the complexity of the proposed algorithm to well-known state of the art partial sort algorithms are presented.

## 2. ALGORITHM DESCRIPTION

When the number of elements to be partially sorted, *N*, is small compared to the total number of elements, *L*, it is less complex to find the *N* partially sorted elements by finding *N* minimums instead of a regular sorting operation, provided these *N* minimums are found by limiting the search to a smaller set of numbers as opposed to searching all the original *L* elements. The key to reducing complexity in the proposed partial sort algorithm is to split the input set of *L* elements into multiple smaller data sets, using a suitable mechanism to limit the search for the $J^{th}$ sorted element to one of those smaller sets and then selectively updating the minimums associated with that data set.

The processing architecture of the proposed partial sort algorithm is shown in Fig. 1 and is described by the pseudocode shown below. In the pseudocode description, *(a:b)* means elements from position *a* to position *b* in a particular row or column. The input has *L*

elements, out of which the smallest (or largest) *N* are to be selected. These *L* elements are partitioned into a two dimensional array of *R*C* number of buffers, where *R* is the number of row buffers and *C* is the number of column buffers. Each buffer holds *K = L/(R*C)* number of elements. This partitioning can be done in any arbitrary manner. The partial sort is implemented through a series of successive minimum (or maximum) operations as shown in Fig. 1. The partial sort is accomplished in 2 distinct steps. In the first step, the first partially sorted element is determined. In the second step, the rest of the *N-1* sorted elements are determined re-using the minimums computed in the first step. To compute the first partially sorted element, the minimums of each of the *R*C* buffers are determined. These minimums are labelled "Buffer Minimums" in Fig. 1. Each of these minimums is determined using *K-1* element comparisons. In the next stage, the minimums of these Buffer Minimums for each row are determined. These minimums are called the Row Minimums. In the final stage, the minimum of these *R* row minimums is determined, and this becomes the first partially sorted element. At this stage, the buffer *B(x,y)* where this sorted element was selected from is identified.

---

// 1st STEP – Determine 1st Partially Sorted Element. The subscript indicates the partially sorted element currently being processed

// Find Buffer Minimums

*For i = 1:R* // Row Index

*For j = 1:C* // Column Index

$M_1(i, j) = Minimum \{B_1 (i, j)\}$

*End*

*End*

// Find the Minimum for each Row

*For i = 1:R*

$D_1(i) = Minimum \{M_1 (i, 1:C)\}$

*End*

// Find Column Minimum which is 1st sorted element

$S_1 = Minimum \{D_1 (1:R)\}$

$(x_1, y_1)$ = *Index of buffer where S1 was selected from*

*Remove* $S_1$ *from its buffer* $B(x_1,y_1)$

// 2nd STEP - Determine rest of the N-1 sorted elements

*For j = 2: N-1*

// Steps to determine the jth sorted element

// Update minimum for just 1 buffer

$M_j ( x_{j-1}, y_{j-1} ) = Minimum \{B_j (x_{j-1}, y_{j-1})\}$

// Update minimum for just 1 Row

$D_j (x_{j-1}) = Minimum \{M_j (x_{j-1}, 1:C)\}$

// Find Column Minimum which is the $j^{th}$ sorted element

$S_j = Minimum \{D_j (1:R)\}$

$(x_j, y_j)$ = *Index ofbuffer where* $S_j$ *was selected from*

*Remove* $S_j$ *from its buffer* $B(x_j, y_j)$

*End*

Pseudocode Description of ProposedAlgorithm

---

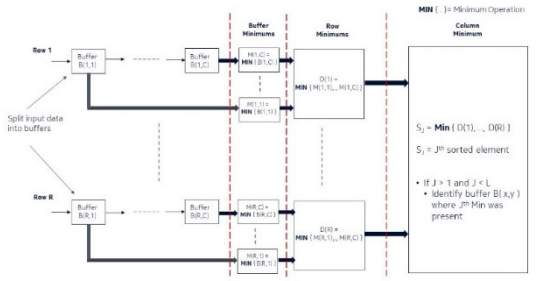


Fig. 1 Determining the first sorted element

Once the first sorted element has been determined, the rest of the *(N-1)* sorted elements are determined by reusing the minimums calculated in the calculation of the first sorted element as well as through highly selective updates of various minimums. To determine the $J^{th}$ sorted element, the sorted element determined in the previous iteration, $S_{J-1}$, is removed from its buffer *B(x,y)*. A new minimum for only this buffer *B(x,y)* is then determined since its contents have changed due to the removal of $S_{J-1}$. The rest of the *(R*C-1)* buffer minimums are left unchanged since their values have not changed, as shown by the empty blocks with the dashed outlines in the "Buffer Minimums" column in Fig. 2. Figure 2 shows the example where the previously determined sorted element is found in buffer *B(1,1)*. A new Row Minimum is now determined, corresponding to the row

index "x" where this buffer *B(x,y)* is located. The rest of the row minimums are left unchanged since their values have not changed, as shown by the empty blocks with the dashed outlines in the "Row Minimums" column in Fig. 2. The minimum of all the Row Minimums is now the next sorted element $S_J$. Hence, after the first sorted element has been determined, finding each of the remaining *(N-1)* sorted elements requires only 3 minimum operations since we have narrowed each search to one buffer and re-use previously determined minimums. This reduces the number of element comparisons required

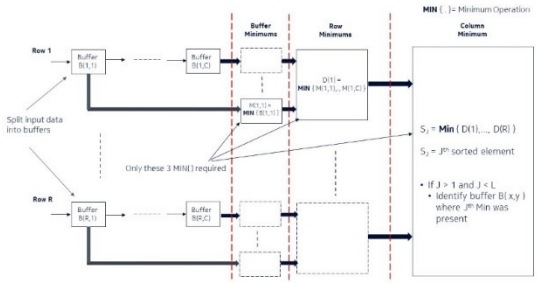


Fig. 2 Determining the J[th] sorted element

## 3. DETERMINING THE OPTIMAL PARAMETERS

The number of element comparisons for determining the first of the N partial sorted elements in Step 1 can be written as

$$Number\ of\ comparisons\ \ for\ 1st\ sorted\ element = = L - 1 \qquad (1)$$

The number of comparisons for determining the remaining *(N-1)* sorted elements can be written as

$$Number\ of\ comparisons\ for\ (N-1)\ elements =$$

$$(N-1) * \{(L/(R*C) - 2) + (C-1) + (R-1)\} (2)$$

The total number of comparisons for all *N* out of *L* elements can be written as

$$Total\ number\ of\ comparisons\ for\ N\ out\ of\ L\ elements = (L-1) + (N-1) * \left\{\left(\frac{L}{R*C}\right) + R + C - 4)\right\} \qquad (3)$$

Eqn. (3) shows that the complexity of the proposed algorithm is heavily dependent upon the choice of *R* and *C*. An arbitrary choice of *R* and *C* will not result in the lowest complexity solution. A suitable trade-off has to be determined between the size of *K,* and the values of *R* and *C* in order to minimize complexity. Since Eqn. (3) relates the total complexity to *R* and *C*, it can be used to as a starting point to determine the optimal values of *R* and *C* that minimize the number of comparisons for a given partial sort of *N* out of *L* elements.

Eqn. (3) can be expressed as a function as follows

$$f(R,C) = (L-1) + (N-1) * \left\{\left(\frac{L}{R*C}\right) + R + C - 4)\right\} \qquad (4)$$

Since L and N are known apriori and are constants, R and C can be determined as follows

$$(R,C) =$$

$$arg\ min_{(R,C)\in\mathbb{Z}} \{(R,C) \in \mathbb{Z} \cap \forall (x,y) \in \mathbb{Z} : f(R,C) \leq f(x,y)\} \qquad (5)$$

Subject to the constraint

$$\frac{L}{R*C} \geq A, where\ A \in \mathbb{Z}, and\ A \geq 1 \qquad (6)$$

Where *Z* is the set of integers. The values of *R* and *C* are determined from a numerical evaluation of Eqn. (4) for different values of *R* and *C* subject to the constraint given in Eqn. (6), and finding that set of values, (R,C), which results in the smallest value for Eqn. (4). The constraint given by Eqn. (6) ensures that we end up with the same number of elements in each buffer, which simplifies the implementation. However, this constraint can be removed, in which case there will be at least 1 buffer that will have a different number of elements compared to the rest. Differing buffer sizes have no impact on the results of the partial sort. In an actual implementation, some additional housekeeping will have to be done to keep track of these different buffer sizes.

A sample determination of *R* and *C* is shown in Table 1 for a partial sort of *N* = 16 out of *L* = 1024 elements. Table 1 shows the number of element comparisons determined from Eqn. (4) for different values of *R* and *C*. The

minimum number of comparisons is 1443, and it can be seen there are 3 sets of *(R,C)* values which result in this minimum value. Any one of these 3 sets of *(R,C)* values can be used.

Table 1. Number of element comparisons for different values of R and C for a partial sort of 16 out of 1024

| Number of Element Comparisons | | | | |
|---|---|---|---|---|
| | **R = 4** | **R = 8** | **R = 16** | **R = 32** |
| **C = 4** | 2043 | 1623 | 1503 | 1623 |
| **C = 8** | 1623 | **1443** | **1443** | 1623 |
| **C = 16** | 1503 | **1443** | 1503 | 1713 |
| **C = 32** | 1623 | 1623 | 1713 | 1938 |

## 4. PERFORMANCE

The performance of the proposed partial sort algorithm is compared against the Partial QuickSort (PQS) algorithm of Martinez [2] and is shown in Fig. 3. The metric used for the performance comparison is the number of element comparisons for a partial sort of *N* out of L= 1024 and L = 512 elements for different values of *N*. The number of element comparisons for PQS is given by Eqn. (3.6) of [2] and is plotted in Fig. 3 for the chosen configurations. PQS was chosen for comparison as it and its variants are among the most efficient partial sort algorithms. Another efficient partial sort algorithm is the Incremental QuickSelect (IQS) algorithm proposed in [3]. IQS requires the same number of element comparisons as the PQS algorithm.

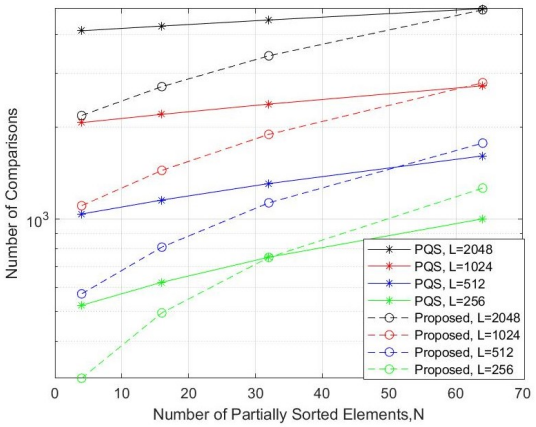


Fig. 3 Complexity of proposed partial sort algorithm

It can be seen from Fig. 3 that the proposed algorithm is significantly more efficient in terms of the number of element comparisons, especially when the number of elements to be partially sorted, *N*, is small compared to the total number of elements *L*. It should be noted that PQS and IQS have additional complexity due to the use of pivots, which is not shown in these curves.

Figure 4 shows the improvement of the proposed algorithm compared to PQS as a function of the partial sort percentage which is the percentage of the ratio of (N/L). As mentioned before, the proposed algorithm is more efficient compared to its peers when the partial sort percentage is small. It is instructive to characterize this behavior as this would be one of the metrics to use when deciding upon the choice of the partial sort algorithm.

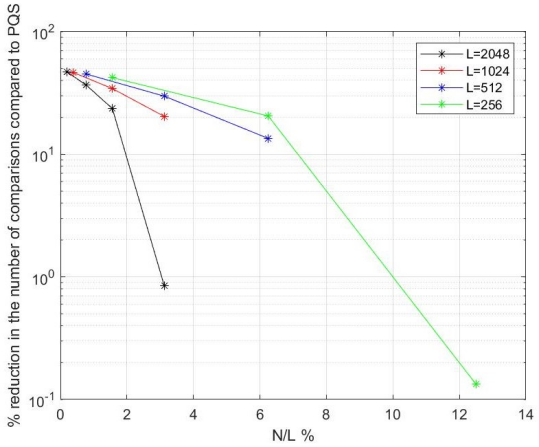


Fig. 4 Percentage reduction in the number of comparisons

A standard measure of performance is the efficiency of the sorting algorithm. Efficiency is usually measured by the number of comparisons expressed in big-O notation. While this is sufficient for academic analysis, the authors feel that a big-O comparison is too high level when choosing between algorithms that might have the same big-O complexity but whose practical implementation complexities could be quite different. A more useful metric for efficiency is to see how close a sorting algorithm comes to the number of comparisons required to pick the smallest element for a given data set. The number of comparisons required to pick the minimum of a data set is a lower bound that cannot be bettered unless *apriori* information about the data set is available. Figure 5 shows this efficiency for the sorting configurations shown before. The Y-axis shows the additional comparisons required by various sorting algorithms for a partial sort of *N* out of *L* elements, relative to the number of comparisons required to find the minimum of that data set. The number of comparisons required to find the global minimum for a data

set of length *L* elements is simply *(L-1)*. It can be seen from Fig. 5 that the proposed algorithm gets very close to the minimum bound when the number of elements to be partially sorted, *N*, is small compared to *L*. For a partial sort of 4 out of 512, the number of additional comparisons for the proposed algorithm is only about 8%, compared to almost 100% for PQS and IQS.

The efficiency of the proposed algorithm especially for small partial sort percentages (i.e., small *N*/*L*) is one of the major advantages of the proposed algorithm against other partial sort algorithms, all of which are very inefficient when the partial sort percentage is small. This is clearly seen in Fig. 5 where the number of additional comparisons required by the proposed algorithm is a gradual function of *N/L*, as opposed to PQS and IQS where the number of additional comparisons starts at a much higher number. Figure 5 also shows that as *N/L* increases, traditional partial sort algorithms start to become more efficient after a certain point.

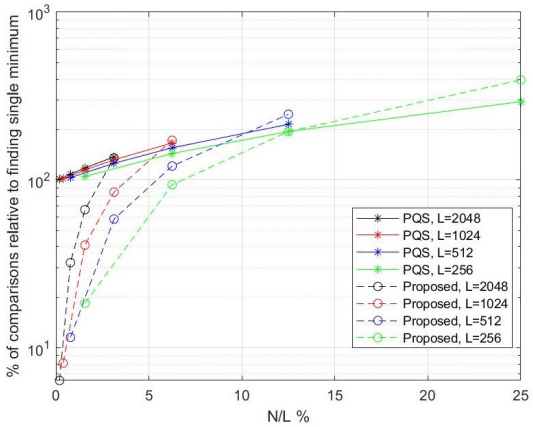


Fig. 5 Sorting efficiency against the single minimum bound

Most of the partial sort algorithms [2][3][4][5] including PQS and IQS that have been proposed in the open literature require the use of pivots. It should be noted that the use of pivots requires additional processing on top of the element comparisons shown for PQS in Fig. 3. Pivot selection algorithms [6][7] also increase the latency of the overall sorting operation since a pivot has to be found before the next sorting iteration can begin. The complexity of pivot selection is usually dependent upon the data being sorted, resulting in varying sorting latency. This varying processing latency can become a major issue for real time operations when strict processing time deadlines have to be met. The proposed algorithm does not use any pivots, and hence does not require the explicit data exchanges or re-arrangement associated with the use of pivots. The initial splitting of the data into the buffers in Step 1 of the proposed algorithm can be easily handled through the use of appropriate pointer indices, without having to explicitly move or re-arrange data. All of these further increase the complexity differential between the proposed algorithm and its contemporaries.

Another feature of the proposed algorithm that makes it suitable for real time applications is its parallelizability. Most of the operations in Step 1 as shown in Fig. 1 can be implemented in parallel. This can improve the processing latency of the overall sorting process.

## 5. CONCLUSION

This paper has detailed a new efficient partial sort algorithm that has lower complexity compared to other competitive partial sort algorithms. A significant reduction in complexity can be seen especially in scenarios when the number of elements, *N*, to be partially sorted is small compared to the total number of elements, *L*. The simplicity of the processing steps combined with the constant processing latency makes it a good candidate for many real time sorting applications. Further enhancements are possible, for example, when operating on already partially sorted arrays.

### *References*

[1] N. M. Josuttis, The C++ Standard Library: A Tutorial and Reference Guide, Addison-Wiley, 1999.

[2] C. Martinez, “Partial Quicksort,” Proceedings 6th ACM-SIAM Workshop on Algorithm Engineering and Experiments and 1st ACM-SIAM Workshop on Analytic Algorithmics and Combinatorics (ALENEX-ANALCO ’04), New York, pp. 224-228, 2004.

[3] R. Paredes and G. Navarro, “Optimal Incremental Sorting,” Proceedings Eighth Workshop on Algorithm Engineering and Experiments (ALENEX), pp. 171-182, 2006.

[4] C.A.R. Hoare, “FIND (Algorithm 65),” Comm. ACM, Vol.4, pp. 321-322, 1961.

[5] C.A.R. Hoare, “Quicksort,” Computer Journal, Vol.5, pp. 10-15, 1962.

[6] P. Kirschenhofer, H. Prodinger, and C. Martınez, Analysis of Hoare’s Find algorithm with median-ofthree partition, Random Structures & Algorithms, 10(1), pp. 143–156, 1997.

[7] C. Martinez and S. Roura, “Optimal sampling strategies in quicksort and quickselect,” SIAM J. Comput., 31(3), pp. 683–705, 2001.