A NEW PIVOT SELECTION SCHEME FOR QUICKSORT ALGORITHM

Aminu Mohammed[1] and Mohamed Othman[2*]

Received: Jul 9, 2004; Revised: Apr 26, 2004; Accepted: Apr 28, 2004

Abstract

Data sorting is one of the most intensively studied problems in computing science for both its theoretical importance and its use in many applications. Quicksort which depends on an appropriate pivot selection technique for its performance is widely considered to be one of the most efficient sorting techniques. Brest *et al.* (2000) has implemented a parallel quicksort algorithm on PC-cluster using a Median5 function as a pivot selection scheme. In this paper, a sequential quicksort was implemented using Median5 function as a pivot selection scheme and subsequently a new pivot selection scheme for minimizing the execution time of quicksort algorithm sequentially is proposed. The two schemes were tested together using integer and double array data types. From the results obtained, the execution time of quicksort algorithm was reduced by about 23-28% for integer array and 17-22% for double array when compared with Median5 function (median-of-five with random index selection scheme).

Keywords: Data sorting, partitioning, pivot selection scheme, sequential quicksort algorithm

Introduction

Sorting data is one of the most intensively studied problems in computing science and it continues to be an interesting theoretical and difficult practical problem. Many sorting algorithms have been proposed (Scowen, 1965; Motzkin, 1983; Roger, 1985; Moh *et al.*, 1999).

Although there is no internal sorting algorithm that is best for every situation, the Quicksort algorithm introduced by Hoare (1961; 1962) is widely accepted as the most efficient internal sorting technique. Quicksort sorts a list of keys A[1], A[2], . . . , A[n] recursively by choosing a key "*m*" in the list as a pivot key around to rearrange the other keys in the list. Ideally, the pivot key is near the median key value in the list, so that it is preceded by about half of the keys and followed by the other half. The keys of the list are rearranged such that for some j, A[1], A[2], . . . , A[j] contain all the keys with values less than *m*, and A[j+1], A[j+2], . . . , A[n] contain all the keys with values greater than or equal to *m*. The elements A[1], A[2], . . . , A[j] are called the left sub list, and the elements A[j+1], A[j+2], . . . , A[n] are the right sub list. Thus, the original list is partitioned into two sub

[1] *Department of Mathematics, Faculty of Science Usmanu Danfoduyo University, P.M.B. 2346 Sokoto, Sokoto-State. Nigeria. Tel: 234-060-230867 E-mail: mamunuus@yahoo.com*

[2] *Department of Communication Technology and Network, Faculty of Computer Science and Information Technology, University Putra Malaysia 43400 UPM Serdang, Selangor D.E., Malaysia Tel: 603-8946-6565 Fax: 603-8946-6577, 603-8948-3745 E-mail: mothman@fsktm.upm.edu.my*

[*] *Corresponding author*

lists where all the keys of the left sub list precede all the keys of the right sub list. After partitioning, the original problem of sorting the entire list is now reduced to the problem of sorting the left and right sub lists independently. Quicksort is then applied recursively to each of these sub lists until the sub list consists of just a single item. Not only is this algorithm simpler than many other sorting algorithms, but empirical (van Emden, 1970; Sedgewick, 1977) and analytical (Knuth, 1998) studies show that quicksort can be expected to be up to twice as fast as its nearest competitors, with expected time complexity of *O(n log n)* and a worst case of $O(n^2)$.

There is a continual demand for greater computational speed from a computer system than currently possible (Moh *et al.*, 1999). Areas requiring great computational speed include numerical modeling and simulation of scientific and engineering problems. Such problems often need huge repetitive calculation on a large amount of data to give valid result. Thus, it is appropriate to study how the performance of quicksort algorithm can be improved.

The rest of the paper is organized as follows. In section 2, an overview of the pivot selection techniques is given while in section 3 the proposed pivot selection scheme is described. Experimental results of the implemented pivot selection scheme on a qucksort algorithm are presented in section 4 and concluding remarks are given in section 5.

Pivot Selection Techniques

There are several ways to make the worst case for a quicksort algorithm very unlikely in practical situations (Sedgewick, 1978). Instead of using the first element in the list as the partitioning element, one may use some other fixed elements, like the middle element. This helps some, but simple anomalies can still occur. Using a random partitioning element will virtually prevent the anomalous cases from happening in practical sorting situations, but random number generation can be relatively expensive and does not reduce the average running time of the rest of the algorithm (Singleton, 1969; Roger, 1985).

Median-of-Three Method

The best choice of pivot would be the median of the array; unfortunately this is hard to calculate and would slow down quicksort considerably. The median of three modifications (Hoare, 1962) will actually improve the average performance of the algorithm while at the same time making the worst case unlikely to occur in practice (Weiss, 1999).

In this method, a sample of size three is used at each particular stage. Primarily, this sampling method ensures that the partitioning elements do not consistently fall near the ends of the sub lists. To make the worst case unlikely the method uses the first, middle, and the last elements as the samples and the median of those three as the partitioning element. Using this method clearly eliminates the bad case for sorted input and actually reduces the running time of quicksort by about 5% (Hoare, 1962; Singleton, 1969).

Median-of-Five Method

As a larger sample size gives better estimates of the median, many researchers try to improve on the running time of quicksort by proposing different pivot selection techniques. Thus, the median-of-five method was used (Brest *et al.*, 2000; Cerin, 2002). The method uses a sample size of five elements, i.e. the first, middle, last, and two other elements randomly picked through a random number generation function between the first and the last elements. This technique gives a better load balancing and reduces the execution time of quicksort by more than 5%, but there is an overhead associated with random number generation. A snapshot of this method is presented in Figure 1.

Proposed Method

In this section we propose a new pivot selection scheme for quicksort algorithm, which does not involve any use of random index selection. This method also uses a sample size of five elements as in the median-of-five method with random index selection scheme. The five elements are selected as follows: first, middle, last, and other two are at positions $\left[(low+high)/4\right]$ and $\left[3*(low+high)/4\right]$, where *low* and *high* are indexes of the first and last elements in the original array. Thus, it forms a new array of five elements, which will be used by the algorithm. It then returns the middle element after sorting the new array. The returned middle element is the median of those five elements and is used as the partitioning element. Thus, it eliminates the use of random index selection.

The snapshot of the new pivot selection scheme is presented in Figure 2. All of the five elements are selected explicitly as shown in the snapshot. It therefore shows the idea of our scheme. First, the five elements are picked from the unsorted array. Then the elements are sorted using a sample sort. Finally, the median of the five elements i.e. V [2] is returned, which will be used as the pivot element.

Results and Discussion

In this section, the experimental results of the

```
Median5 Scheme (int A[], int low, int high)
{
int  V[5] ;
V[0] = low;
V[1] = high;
V[2] = (low + high)/2 ;
V[3] = (int)(low - high) * ((double)(rand ( ) / (double) (RANDMAX + 1))  ;
V[4] = (int)(low - high) * ((double)(rand ( ) / (double) (RANDMAX + 1))  ;

for(int i=0; i< 5; i++)
for(int j=0; i< 4; i++)
   if(A[V[j]] > A[V[j+1]])
     {
       int temp = V[j] ;
            V[j] = V[j+1] ;
            V[j+1] = temp ;
      }
    return V[2] ;
}
```

Figure 1. A snapshot of median5 function scheme.

```
Proposed Scheme (int A[], int low, int high)
{
int  V[5] ;
V[0] = low;
V[1] = high;
V[2] = (low + high)/2 ;
V[3] = (low + high)/4 ;
V[4] = 3((low + high)/4) ;

for(int i=0; i< 5; i++)
for(int j=0; i< 4; i++)
   if(A[V[j]] > A[V[j+1]])
     {
       int temp = V[j] ;
            V[j] = V[j+1] ;
            V[j+1] = temp ;
      }
    return V[2] ;
}
```

Figure 2. A snapshot of the proposed scheme.

implemented sequential quicksort algorithm using our scheme and the Median5 function (median-of-five with random index selection) as pivot selection techniques are presented. The algorithm was implemented in C++ programming language on a PC-cluster machine. Our implementation was sequential and thus the parallel capability of the PC-cluster was not utilized. The purpose of using a PC-cluster as a normal PC was to utilize its larger array size capability. The maximum array size that could be declared on the PC-cluster was two million elements. In the experiment, integer and double array data types were used. The data were randomly generated and 40 independent measurements were performed in each case. Thus, we operated on average values. The obtained results are shown in Tables 1 and 2.

Table 1. Execution time of the two schemes using integer array type.

n	Proposed scheme (sec)	Median5 scheme (sec)
100K	0.16	0.21
200K	0.34	0.47
300K	0.53	0.72
400K	0.72	0.10
500K	0.91	1.26

Table 2. Execution time of the two schemes using double array type.

n	Proposed scheme (sec)	Median5 scheme (sec)
100K	0.27	0.32
200K	0.58	0.68
300K	0.89	1.06
400K	1.22	1.46
500K	1.46	1.83

The graphs on Figures 1 and 2 show the execution time of the two methods when sorting integer and double array types. Based on the results, the proposed scheme i.e. the median-of-five method without random index selection was faster than Median5 scheme (median-of-five with random index selection). Thus, the proposed scheme was faster and therefore minimized the execution time of quicksort algorithm sequentially by about 23-28% and 17-22% for integer and double array data types respectively.

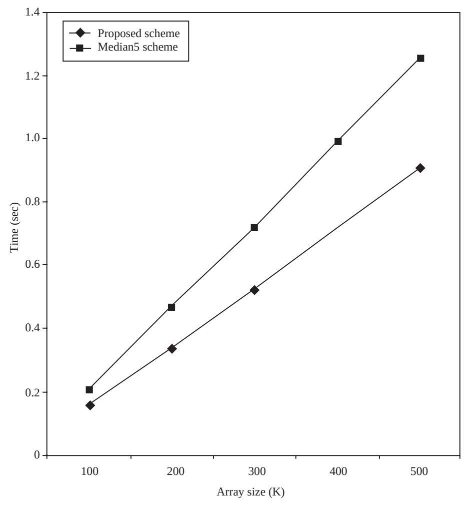

Figure 3. Average time against array size of the two schemes-using integer array.

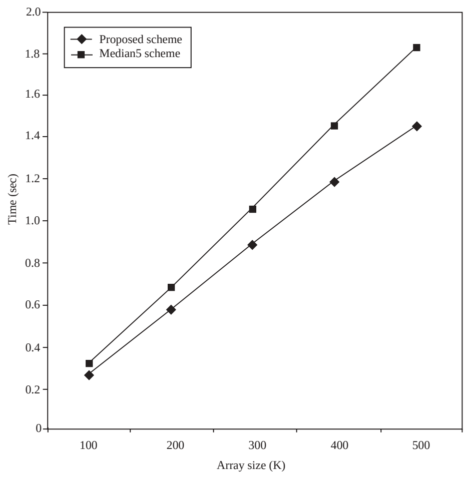

Figure 4. Average time against array size of the two schemes-using double array.

Conclusion

In this paper, the pivot selection methods in sequential quicksort algorithm and experimental study of their performance were presented. Tables 1 and 2 show that the median-of-five method without random index selection (proposed scheme) was faster than Median5 scheme (median-of-five with random index selection) for both integer and double array type. This has been graphically shown in Figures 3 and 4. Thus, the execution time of sequential quicksort algorithm was minimized by about 23-28% and 17-22% for an integer and double array types respectively.

The work is constrained by our inability to generate larger array size that could be in tens or hundreds of million in sizes. Likewise, the proposed scheme needs to be implemented using parallel quicksort in order to ascertain its performance in that direction.

References

Brest, J., Vreze, A., and Zumer, V. (2000). A Sorting algorithm on PC cluster. Proceedings of the ACM Symposium on Applied Computing; March 19-21, 2000; Como, Italy. ACM Press, New York, NY, USA, p. 710-715.

Cerin, C. (2002). An out-of-core sorting algorithm clusters with processors at different speed. IEEE Proceedings of the Int. Parallel and Distributed Processing Symposium; April 15-19, 2002; Fort Lauderdale, Florida, USA. IEEE Computer Society, Washington, DC, USA, p. 681-686.

Hoare, C.A.R. (1961). Algorithm 64; Quicksort. Comm. ACM. 4(7):321.

Hoare, C.A.R. (1962). Quicksort. Computer Journal, 5:10-15.

Knuth, D.E. (1998). The Art of Computer Programming. 2nd ed. Addison Wesley, Boston, USA, 3:73-80.

Loeser, R. (1974). Some performance test of quicksort and descendents. Commun. ACM., 17(3):143-152.

Moh, S., Kim, S., Lee, M., Yu, C., and Han, D. (1999). A new parallel quicksort with efficient processor allocation and minimal communication. SIG on Parallel Processing System Conference; September 10-11, 1999; Korea Information Science Society, Seoul, Korea, p. 83-90.

Motzkin, D. (1983). Meansort. Comm. ACM., 26(4):250-251.

Roger, L.W. (1985). A class of sorting algorithms based on quicksort. Commun. ACM., 28(4):396-402.

Scowen, R.S. (1965). Algorithm 271; Quickersort. Comm. ACM. 8, 11:669-670.

Sedgewick, R. (1975). Quicksort. [PhD. thesis]. Stanford Comptr. Sci. Rep. STAN-CS-75-492, Stanford U., Stanford, California, p. 25-251.

Sedgewick, R. (1977). Quicksort with equal keys. Siam Journal on Comput., 6(2): 240-267.

Sedgewick, R. (1978). Implementing quicksort program. Commun. ACM., 21(10):847-857.

Singleton, R.C. (1969). Algorithm 347; An efficient algorithm for sorting with minimal storage. Comm. ACM., 12(3):185-187.

van Emden, M.H. (1970). Increasing the efficiency of quicksort. Comm. ACM., 13(9):563-567.

Weiss, M.A. (1999). Data Structure and Algorithm Analysis in C++. 2nd ed. Addisson-Wesley Publishing Inc, Boston, USA, p. 250-467.

Youran, L., and Magdi, A.M. (1992). Parallel quicksort in hypercube. ACM/SIGAPP Symposium on Applied Computing; March 1-3, 1992; Kansas City, Missouri, United States. ACM Press, New York, NY, USA, p. 740-746.

Zumer, V., Ojstersek, M., Vreze, A., and Brest, J. (1999). Sorting on heterogeneous computing System. Proceeding of MIPRO'99: 10th International Conference on Computers in Intelligent Systems; May 17-21, 1999; Opatia, Croatia, p. 1-4.