A Practical Performance Comparison of Parallel Sorting Algorithms on Homogeneous Network of Workstations

Haroon Rashid and Kalim Qureshi*
COMSATS Institute of Information Technology, Abbottabad, Pakistan
***Math. and Computer Science Department, Kuwait University, Kuwait**

Three parallel sorting algorithms have been implemented and compared in terms of their overall execution time. The algorithms implemented are the odd-even transposition sort, parallel merge sort and parallel rank sort. A homogeneous cluster of workstations has been used to compare the algorithms implemented. The MPI library has been selected to establish the communication and synchronization between the processors. The time complexity for each parallel sorting algorithm will also be mentioned and analyzed.

Keywords: Parallel sorting algorithms, performance analysis, network parallel computing.

1. Introduction

Sorting is one of the most important operations in database systems and its efficiency can influences drastically the overall system performance. To speed up the performance of database system, parallelism is applied to the execution of the data administration operations. The workstations connected via a local area network allow to speed up the application processing time [1]. Due to the importance of distributed computing power of workstations or PCs connected in a local area network [2] we have been studying the performance evaluation of various scientific applications [1-3]. The dedicated parallel machines are used for parallel database systems and lot of research have already addressed the issues related to dedicated parallel machines [4]. Little research has been carried out on performance evaluations of parallel sorting algorithms on cluster of workstations.

2. Parallel Sorting Algorithms

In this paper, 3 parallel sorting algorithms will be implemented and evaluated. These algorithms are:
1. Odd-even transposition sort.
2. Parallel rank sort.
3. Parallel merge sort.

2.1 Odd-Even Transposition

The Odd-even transposition sort algorithm [5,6] starts by distributing n/p sub-lists (p is the number of processors) to all the processors. Each processor then sequentially sorts its sub-list locally. The algorithm then operates by alternating between an *odd* and an *even* phase, hence the name *odd-even*. In the even phase, even numbered processors (processor i) communicate with the next odd numbered processors (processor i+1). In this communication process, the two sub-lists for each 2 communicating processes are merged together. The upper half of the list is then kept in the higher number processor and the lower half is put in the lower number processor. Similarly, in the odd phase, odd number processors (processor i) communicate with the previous even number processors (i-1) in exactly the same fashion as in the even phase. It is clear that the whole list will be sorted in a maximum of p stages. Figure 1 shows an illustration of the odd-even transposition algorithm.

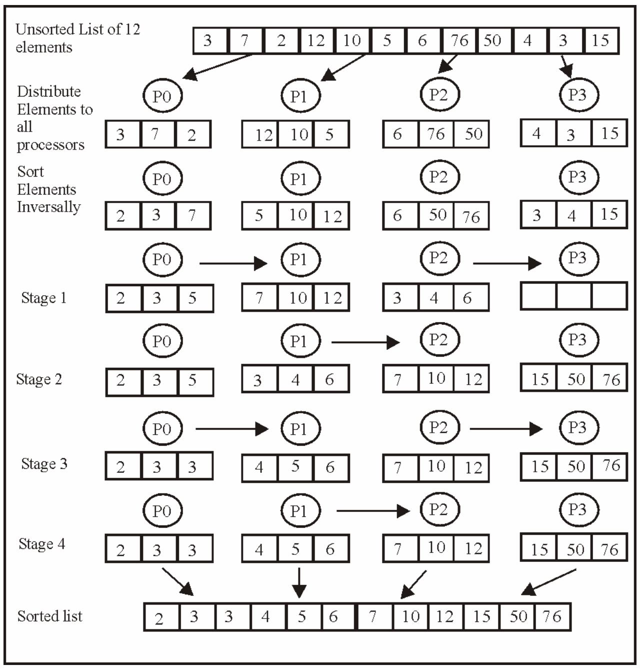

Figure 1: Odd-even transposition sort, sorting 12 elements using 4 processors.

Time Complexity of Odd-Even transposition: At first glance of parallelizing the bubble sort algorithm it seems that the performance will increase a factor of p. However, careful analysis of the complexity reveals that it is actually much more than the stated value. Below is the analysis of the time complexity for the odd-even transposition sorting algorithm [5]:

The performance of the sequential bubble sort algorithm is:

$$\sum_{i=1}^{n} i = 1+2+3+...+n = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = O(an^2), \text{ where } a = \frac{1}{2}.$$

The performance of the odd-even transposition algorithm is:

$$\sum_{i=1}^{n/p} i = 1+2+3....+\frac{n}{p} = \frac{n/p(n/p-1)}{2} = \frac{n^2}{2p^2} - \frac{n}{2p} = o(bn^2)$$

$$where \quad b = 1/2p^2$$

This means that theoretically speaking the time will reduce by $1/p^2$.

2.2 Parallel Merge Sort

The merge sort algorithm uses a divide and conquer strategy to sort its elements [7].The list is divided into 2 equally sized lists and the generated sub-lists are further divided until each number is obtained individually. The numbers are then *merged* together as pairs to form sorted lists of length 2. The lists are then merged subsequently until the whole list is constructed. This algorithm can parallelized by distributing n/p elements (where n is the list size and p is the number of processors) to each slave processor. The slave can sequentially sort the sub-list (e.g. using sequential merge sort) and then return the sorted sub-list to the master. Finally, the master is responsible of merging all the sorted sub-lists into one sorted list. Figure 2 shows an illustration of the parallel merge sort algorithm.

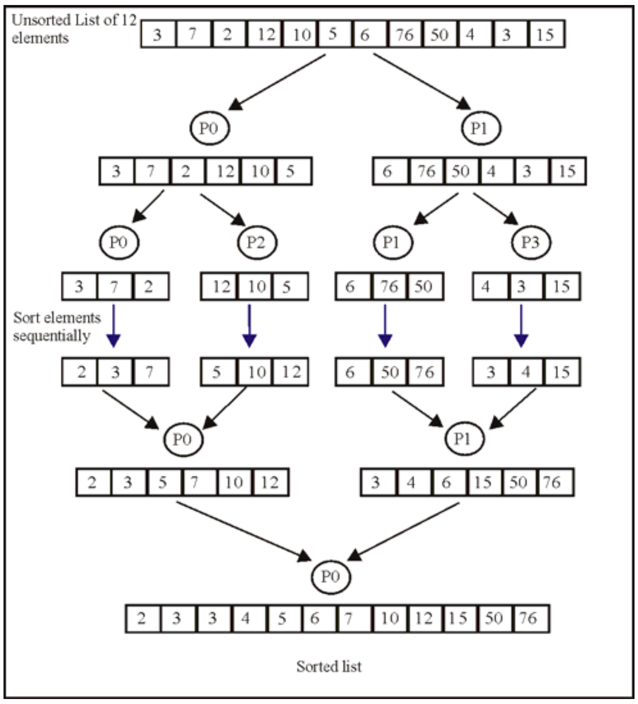

Figure 2: Parallel merge sort algorithm, sorting 12 elements using 4 processors.

Time Complexity of parallel merge sort: Sequential merge sort time complexity is ***O (n log n)***. when parallelizing the merge sort algorithm the time complexity reduces to ***O(n/p log n/p)*** as stated in [5].

2.3 Parallel Rank Sort

In the sequential rank sort algorithm (also known as enumeration sort), each element in the list to be sorted is compared against the rest of the elements to determine its *rank* amongst them [8]. This sequential algorithm can be easily parallelized by enabling the master processor to distribute the list amongst all the processors and assigning each slave processor n/p elements (where n is the list size and p is the number of processors). Each processor is responsible of computing the rank of all the n/p elements. The ranks are then returned from the slaves to the master who in turn is responsible of constructing the whole sorted list (see figure 3).

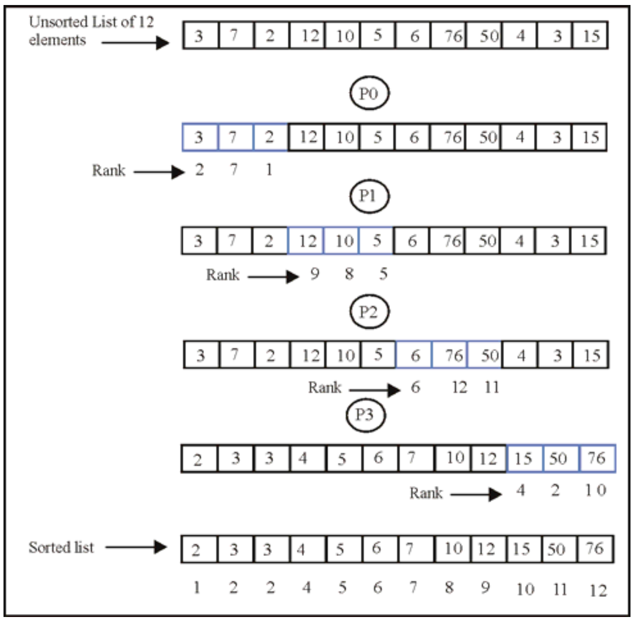

Figure 3: Parallel sort algorithm sorting 12 elements using 4 processors.

Time Complexity of parallel merge sort: In the sequential version of the rank sort algorithm. Each element is compared to all the other elements. The complexity of the algorithm can be expressed as:

$$\sum_{i=1}^{n} n = O(n^2)$$

When parallelizing this algorithm, it can be easily seen that the complexity reduces to:

$$\sum_{i=1}^{n/p} n = O(cn^2).\ c = \frac{1}{p}$$

This means that if n number of processors is used then the sorting time will become almost linear ***O (n)***.

3 Implementation

Each of the parallel algorithms stated above will be compared to its sequential implementation and evaluated in terms of its overall execution time, speedup and efficiency. The speedup is used to measure the gain of parallelizing an application versus running the application sequentially and can be expressed as:

Speed = Execution time using one processor / Execution time using p processor (1)

On the other hand, the efficiency is used to indicate how well the multiple processors are utilized in executing the application and can be expressed as:

Efficiency = Execution time using p processor / Total number of processor (2)

The C programming language used to develop the sorting algorithms. The MPI library routines used to handle the communication and synchronization between all the processors. The performance of the sorting algorithms was evaluated on a homogeneous cluster of SUN workstations, with SUNOS operating system. Each sorting algorithm performance was evaluated for 2, 4, 6, 8, 10, 12 machines. The speedup and efficiency will be calculated based on the previous records. An array of 10,000 random integers was used to test the parallel algorithms.

3. Results and Discussions

Odd-Even Transposition: Figure 4 shows the total execution time for the odd-even transposition sorting algorithm. It can easily be seen that parallel algorithm is by far faster than the sequential bubble sort algorithm. The speed up for the odd-even transposition sorting algorithm is also displayed in figure 5 along with the efficiency in figure 6.

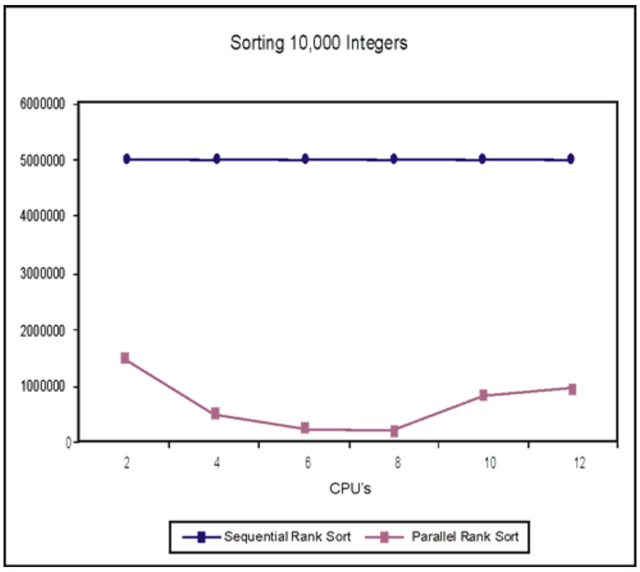

Figure 4: Total execution time of the odd-even transposition sort algorithm

Parallel Merge Sort: Parallel merge sort is one of the most efficient algorithms for sorting elements. In figure 7 an illustration of the total execution time of the algorithm is displayed. It shows that sorting using up to 8 processors is helpful in reducing the total time required to sort the elements. However, increasing the processors to more than 8 processors will result in lower performance compared to the sequential merge sort algorithm as shown in figure 7. This of course is due to the communication overhead that occurs between the processors to merge the result in to one sorted list. The speedup and efficiency of the parallel merge sort algorithm are displayed in figure 8 and figure 9 respectively.

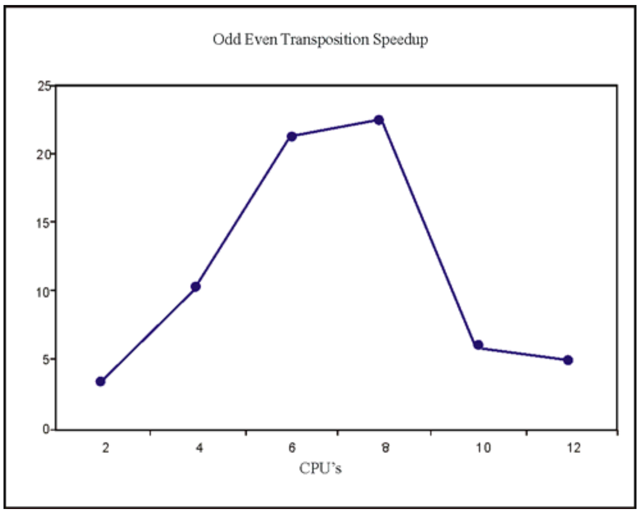

Figure 5: Speedup of the odd-even transposition sort algorithm

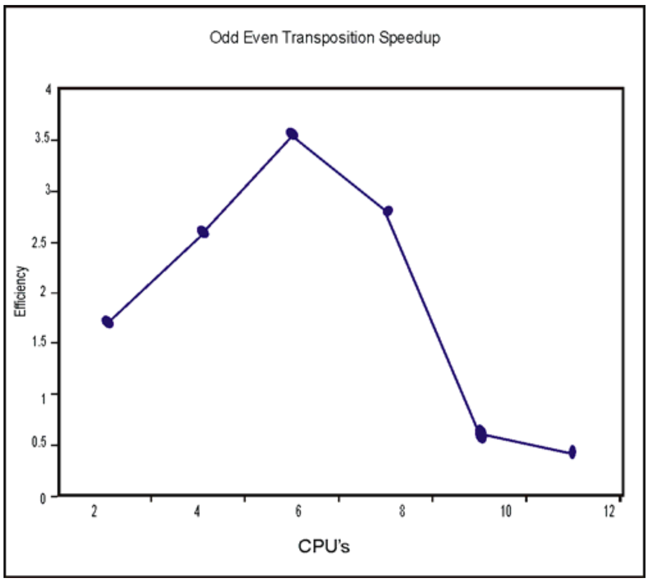

Figure 6: Efficiency of the odd-even transposition sort algorithm

Parallel RankSort: Running the parallel rank sort algorithm on 2 processors to sort 10,000 integers is slower than the sequential implementation due to the communication overhead needed to distribute the whole unsorted list to all the processors. However, the benefit of parallelization kicks in after increasing the number of processors. Using 2 processors run parallel rank sort should increase the performance of the algorithm conditioned the number of elements to be sorted are greater than 10,000 elements (e.g. 1,000,000 elements).

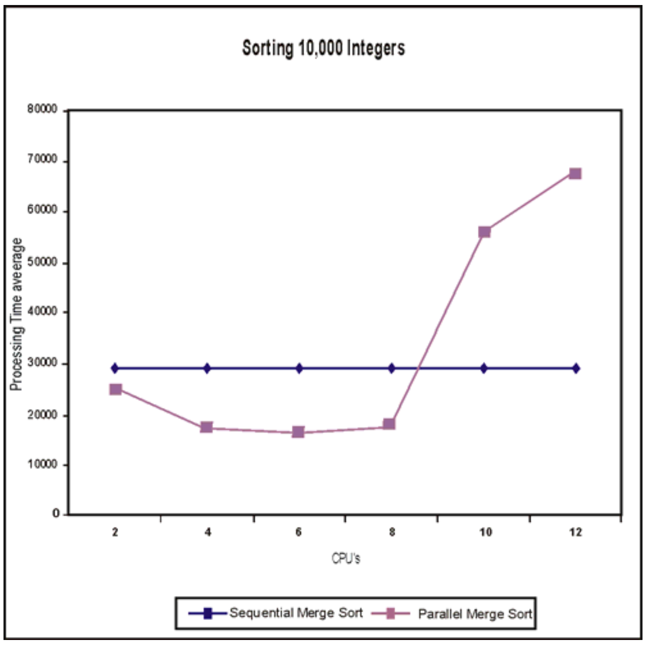

Figure 7: Total execution time of the parallel merge sort algorithm.

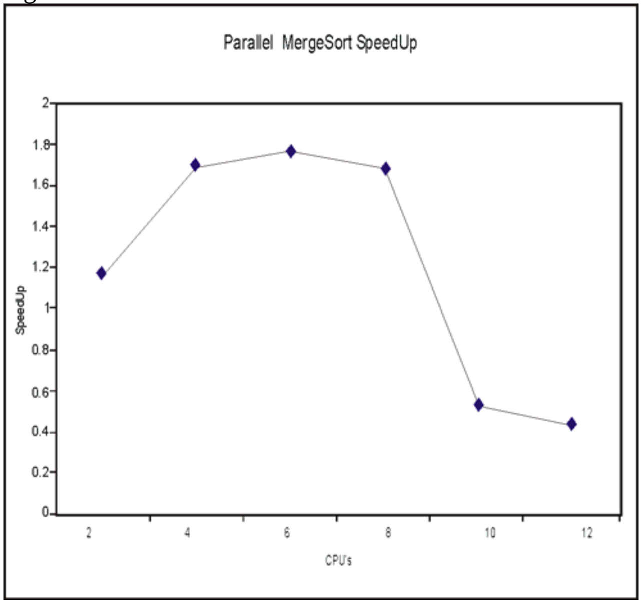

Figure 8: Speedup of the parallel merge sort algorithm.

The limitation of the parallel rank sort is the memory it requires in order to sort its elements. Each processor needs a copy of the whole unsorted list for it to rank its portion of elements. Another memory requirement is to construct an array proportional to the unsorted list size to enable the algorithm of sorting lists with repeated elements. The parallel rank sort algorithm can be considered as a *memory intensive* algorithm. Figure 10 shows the total execution time for the parallel sort algorithm compared to its sequential implementation. When this algorithm runs on 6 processors it can improve the total execution time by a factor slightly greater than 2. However, lot of communication overheads and data transfer is required which prevents us from increasing the performance beyond this factor. The speedup for the parallel rank sort algorithm is also displayed in figure 11 along with the efficiency in figure 12.

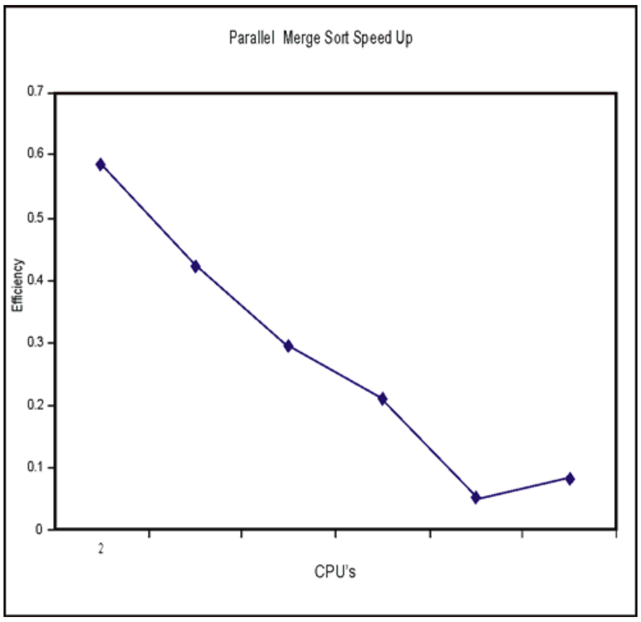

Figure 9: Efficiency of the parallel merge sort algorithm.

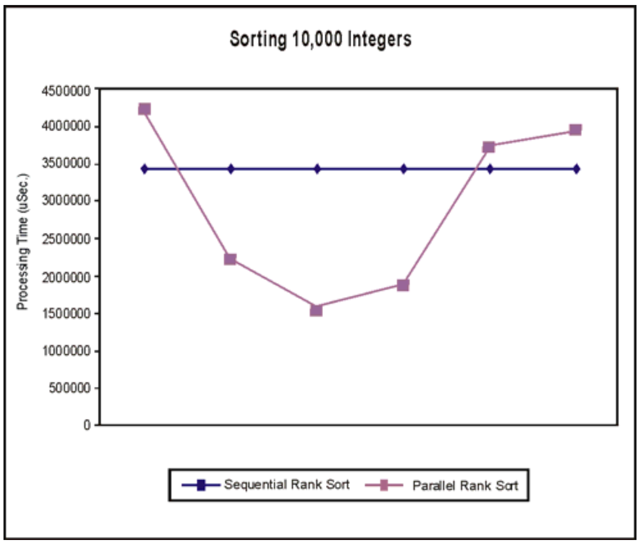

Figure 10: Total execution time of the parallel rank sort algorithm.

4. Conclusions

Three parallel sorting algorithms have been developed and executed on a homogeneous cluster of machines. The parallel algorithms implemented are the odd even transposition sorting algorithm, the parallel rank sort algorithm and the parallel merge sort algorithm. Figure 13 shows a comparison between the 3 parallel sorting algorithms when sorting 10,000 integers on 2, 4, 6, 8, 10, and 12 workstations.

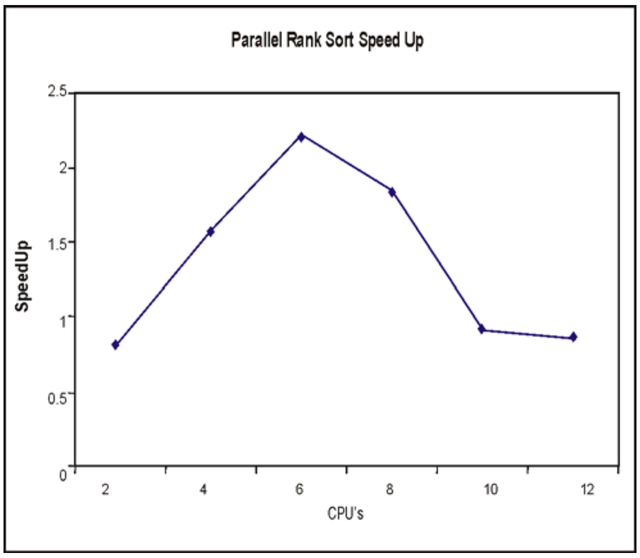

Figure 11: Speedup of the parallel rank sort algorithm.

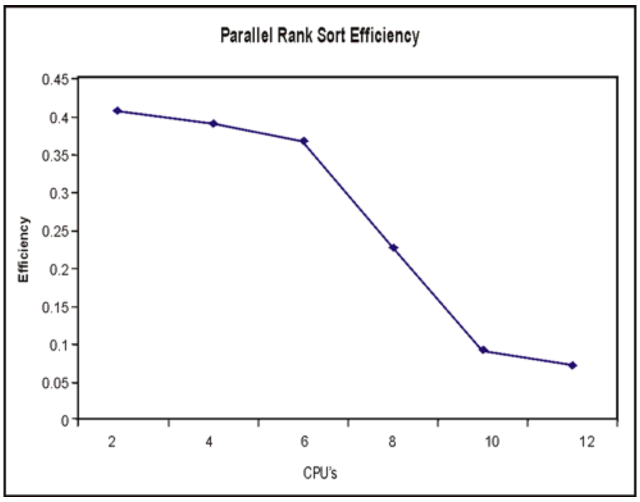

Figure 12: Efficiency of the parallel rank sort algorithm.

From figure 13 it is obvious that the parallel merge sort is the fastest sorting algorithm followed by the odd-even transposition sorting algorithm then the parallel rank sorting algorithm. The parallel rank sort algorithm is the slowest algorithm because each processor needs its own copy of the unsorted list thus, in turn, raises a serious communication overhead. A solution has also been developed and successfully tested to allow parallel rank sort for sorting a list of integers with repeated elements. The odd-even sorting algorithm comes in second place because of the time it takes to initially sort its elements locally in each processor using sequential bubble sort which has a performance of $O(n^2)$. The odd-even transposition sorting algorithm can be improved by adapting a faster sequential sorting algorithm to sort the elements locally for each processor in the order of $O(n \log n)$ (e.g. sequential merge sort or quick sort).

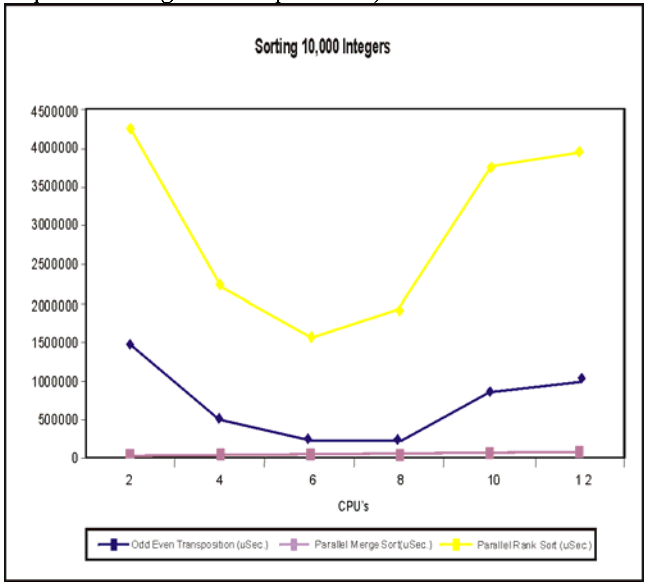

Figure 13: A comparison of the total execution time required for sorting 10,000 integers using parallel merge sort, parallel rank sort and odd-even transposition.

References

---

[1] Kalim Qureshi and Haroon Rashid,"A Practical Performance Comparison of Parallel Matrix Multiplication Algorithms on Network of Workstations.", IEE Transaction Japan, Vol. 125, No. 3, 2005.
[2] Kalim Qureshi and Haroon Rashid," A Practical Performance Comparison of Two Parallel Fast Fourier Transform Algorithms on Cluster of PCS", IEE Transaction Japan, Vol. 124, No. 11, 2004.
[3] Kalim Qureshi and Masahiko Hatanaka, "A Practical Approach of Task Partitioning and Scheduling on Heterogeneous Parallel Distributed Image Computing System," Transaction of IEE Japan, Vol. 120-C, No. 1, Jan., 2000, pp. 151-157.
[4] K. Sado, Y. Igarashi, Some Parallel Sorts on a Mesh-Connected Processor Array and Their Time Efficiency, Journal of Parallel and Distributed Computing, 3, pp. 398-410, 1999.
[4] D. Bitton, D. DeWitt, D.K. Hsiao, J. Menon, A Taxonomy of Parallel Sorting, ACM Computing Surveys, 16,3,pp. 287-318, September 1984.
[5]. Song, Y.D., Shirasi, B. A Parallel Exchange Sort Algorithm. *South Methodist University, IEEE 1989.*
[6] B.R. lyer, D.M. Dias, System Issues in Parallel Sorting for Database Systems, Proc. Int. Conference on Data Engineering, pp. 246-255, 2003.
[7] F. Meyer auf der Heide, A Wigderson, The Complexity of Parallel Sorting, SIAM Journal of Computing, 16, 1, pp. 100-107, February 1999.