Journal of Applied Sciences

ISSN 1812-5654

science alert

ANSInet
an open access publisher
http://ansinet.com

Journal of Applied Sciences 8 (12): 2341-2343, 2008
ISSN 1812-5654
© 2008 Asian Network for Scientific Information

New Parallel Sorting Algorithm Based on Partitioning and Redistribution

Pushpa Rani Suri and Sudesh Rani
Department of Computer Science and Applications, Kurukshetra University,
Kurukshetra-136119, Haryana, India

Abstract: This study describes a new parallel sorting algorithm PPS based on the techniques of partitioning and redistribution, where the sorting process is split into two stages: partitioning and independent local work. In parallel partitioned sort, first we partition local data using range partitioning. Then local sort is carried out. The main benefit of parallel partitioned sort is that no merging is needed for the final result because the results produced by the local sort are already the final results.

Key words: External sorting, internal sorting, parallel sorting, quicksort, range redistribution

INTRODUCTION

Sorting is one of the most important operations in database systems and its efficiency can influences drastically the overall system performance. Sorting is frequently used in database systems to produce ordered query results. It is also the basis of sort-merge join, a join algorithm employed by many existing DBMSs and it is used in some systems for processing group-by queries. Therefore its performance influences dramatically the performance of the whole database system (Iyer and Dias, 1990; Surajit and Kyuseok, 1996). With the advent of parallel processing, parallel sorting has become an important area for algorithm research.

Generally sorting algorithms can be divided into internal and external algorithms (Selim, 1990). An external sorting algorithm is necessary if the data set is too large to fit in main memory. In databases, since data is stored in tables (or files) and is normally very large, database sorting is therefore an external sorting. Obviously this is the common case in database systems. Sorting algorithms are very well analyzed and examined for sequential architectures. Also internal sorting algorithms for parallel architectures are investigated (Alok and Plaxton, 1994; Dusseau *et al*., 1996; Manzur and Brent, 2000), but the topic of external sorting for parallel computer architectures has not yet received adequate consideration. Parallel external sorting has not been fully explored. The traditional approaches of parallel external sorting have been to perform local sort in each processor in the first stage and to carry out merging by the host or using a pipeline hierarchy of processors in the second stage (Mohan *et al*., 1994; Selim, 1990).

SERIAL EXTERNAL SORTING: A BACKGROUND

Serial external sorting is external sorting in a uniprocessor environment. The most common serial external sorting algorithm is based on sort-merge. The underlying principle of sort-merge algorithm is to break the file up into unsorted subfiles, sort the subfiles and then merge the sorted subfiles into larger and larger sorted subfiles until the entire file is sorted. Notice that the first stage is to sort the first lot of subfiles, whereas the second stage is actually the merging phase. In this scenario, it is important to determine the size of the first lot of subfiles which are be sorted. Normally, each of these subfiles must be small enough to fit into main memory, so that sorting these subfiles can be done in main memory using any internal sorting technique. We divide a serial external sorting algorithm into two phases: sort and merge. The sort algorithm, which incorporates a partitioning of the original file, is explained as follows: First, determine R, the number of records, which we can reasonably sort internally. Second, determine K the total number of disks we can use. Third, sort R, records at a time internally, writing the results in turn onto each of K/2 disks with file markers at their ends. Finally, repeat the third step above, writing additional files onto the K/2 disks. Once sorting of subfiles is completed, merging phase starts. An algorithm for the merging process is described as follows. First, do a K/2-way merge using the first subfile from each disk, writing the output onto one of the K/2 empty disks. Second, repeat the first step above for each of the rest of the K/2 empty disks. Finally, repeat steps 1 and 2 above, merging in rotation onto the

Corresponding Author: Sudesh Rani, Department of Computer Science and Applications, Kurukshetra University, Kurukshetra-136119, Haryana, India Tel: +09416352227

K/2 subfiles until the original K/2 disks are empty. As stated in the beginning that serial external sort is the basis for parallel external sort, because in a multiprocessor system, particularly in a shared-nothing environment, each processor has its own data, and sorting this data locally in each processor is done as per serial external sort explained above. Therefore, the main concern in parallel external sort is not the local sort, but whether local sort is done first or later and how merging is performed.

The speedup of a parallel sort achievable on a multiprocessor depends largely on how well the average memory latency and overhead of scheduling and synchronization can be minimized. Based on the general strategies utilized, most parallel sorts suitable for multiprocessor computers can be placed into one of two rough categories: merge-based sorts and partition-based sorts. Merge-based sorts consist of multiple merge stages across processors, and perform well only with a small number of processors. When the number of processors utilized gets large, so does the overhead of scheduling and synchronization, which reduces the speedup. Partition-based sorts consist of two phases: partitioning the data set into smaller subsets such that all elements in one subset are no greater than any element in another, and sorting each subset in parallel. The performance of partition-based sorts primarily depends on how well the data can be evenly partitioned into smaller ordered subsets. Unfortunately, no general, effective method is currently available and it is an open question of how to achieve linear speedup for parallel sorting on multiprocessors with a large number of processors.

DESCRIPTION OF THE ALGORITHM

Parallel Partitioned Sort (PPS) is influenced by the techniques used in parallel partitioned join, where the process is split into two stages: partitioning and independent local work (Mishra and Eich, 1992; Wolf *et al*., 1993). In parallel partitioned sort, first we partition local data according to range partitioning used in the operation. In this method, the first phase is not a local sort. Local sort is not carried out here. Each local processor scans its records and redistribute or repartition according to some range partitioning.

After partitioning is done, each processor will have an unsorted list in which the values come from various processors. It is then local sort is carried out. Thus, local sort is carried out after the partitioning, not before. It is also noticed that merging is not needed. The results produced by the local sort are already the final results. Each processor will have produced a sorted list and all processors in the order of the range partitioning method used in this process are also sorted. Figure 1 shows an illustration of this method.

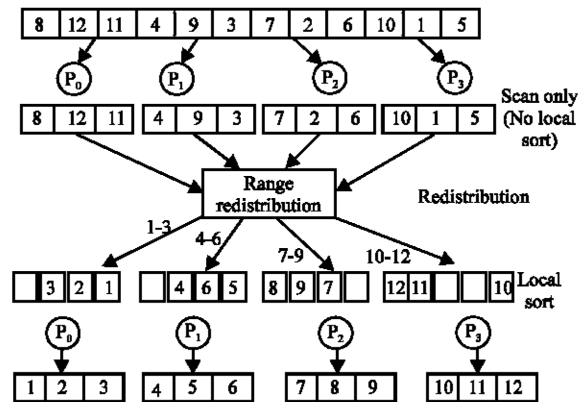

Fig. 1: Parallel partitioned sort algorithm

The main benefit of parallel partitioned sort is that no merging is necessary, and hence the bottleneck in merging is avoided. It is also a true parallelism, as all processors are being used in the two phases. And most importantly, it is a one-level tree reducing unnecessary overhead in the pipeline hierarchy.

Example: Figure 1 gives an illustration of the PPS algorithm. The data sequence to be sorted contains 12 entries, as follows, S = {8, 12, 11, 4, 9, 3, 7, 2, 6, 10, 1, 5}, that will be sorted using four processors (N = 4). The sequence S is initially scanned and partitioned into four 3 entry subsequences. Each local processor scans its records and redistribute or repartition according to some range partitioning. After partitioning is done, each processor will have an unsorted list in which the values come from various processors. It is then local sort is carried out. Thus, local sort is carried out after the partitioning, not before. No merging is needed as the results produced by the local sort are already the final results.

Notice from the illustration that after redistribution phase, some of the boxes are empty. Empty boxes are actually empty list as a result of data redistribution. They indicate that they do not receive any values from the designated processors. For example, the first empty box on the left is because there are no values ranging from 1-3 from processor 1.

COMPUTATIONAL COMPLEXITY

This section is devoted to a detailed analysis of the computational complexity of the proposed parallel sorting algorithm. The first phase of our algorithm is a simply a

scanning and partitioning of all the records. After that records are redistributed using some range partitioning. Finally records are sorted parallely using Quicksort. Quicksort (Hoare, 1962) is a very well-known sorting algorithm that has been extensively studied in the literature. On average, the computational complexity of Quicksort is $\theta(n \log n)$, that corresponds to the number of comparisons to sort n values. However, the complexity reaches $\theta(n^2)$ in the worst case, though is significantly faster in practice. In the case of the PPS algorithm, Quicksort is used in parallel to sort subsequences of size S/N, so as the computational complexity of this algorithm is $\theta(S/N \log (S/N))$, or $\theta((S/N)^2)$ in the worst case.

COMPARISON WITH OTHER SORTING ALGORITHMS

Table 1 shows a chart with the computational complexities of some classical sorting algorithms, that allows to compare them with that of our proposed algorithm. The computational complexities have a first term that is the same for all sorting algorithms, which corresponds to a first phase of local sorting based on Quicksort. The rest of the terms in the complexities depend on the peculiarities of each algorithm.

Table 1: Complexities of sorting algorithms

Parallel partitioned sort	θ (S/N log (S/N))
Odd-even mergesort	θ (S/N log (S/N)) + θ (2S)
Bitonic sort	θ (S/N log (S/N)) + θ (2S/N $\log^2$ N)
Shell sort	θ (S/N log (S/N)) + θ (S/N log N) + θ (2S/N)
Parallel sorting by regular sampling (PSRS)	θ (S/N log (S/N)) + θ (N^2 log N) + θ (N log (S/N)) + θ (S/N)

By comparing complexity of PPS algorithm with the complexities of classical algorithms we can claim that the complexity of our algorithm is smaller than that of all other classical sorting algorithms i.e., (Odd-Even Mergesort, Bitonic Sort, Shellsort, PSRS).

CONCLUSIONS

The study presents a new fast parallel sorting algorithm called PPS (Parallel Partitioned Sort). In this algorithm we make use of redistribution and repartitioning concepts. After an initial scanning of the data sequence, the algorithm works in two phases. The first phase redistributes the data sequence using range partitioning. The second phase sorts in parallel all the subsequences using Quicksort. The performance of PPS is compared with other classical sorting algorithms. In general, the proposed PPS algorithm represents an improvement with regards to classical algorithms, like Odd-Even Mergesort, Bitonic Sort, Shellsort and Parallel Sorting by Regular Sampling. The main benefit of parallel partitioned sort is that no merging is necessary, and hence the bottleneck in merging is avoided. It is also a true parallelism, as all processors are being used in the two phases.

REFERENCES

Alok, A. and C.G. Plaxton, 1994. Optimal parallel sorting in multi-level storage. In: Proceeding of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms, January 23-25, pp: 659-668.

Dusseau, A.C., D.E. Culler, K.E. Schauser and R.P. Martin, 1996. Fast parallel sorting under LogP: Experience with the CM-5. IEEE Trans. Parallel Distributed Syst., 7: 791-805.

Hoare, C., 1962. Quicksort. Comput. J., 5: 10-15.

Iyer, B.R. and D.M. Dias, 1990. System issues in parallel sorting for database systems. In: Proceedings of the International Conference on Data Engineering, February 5-9, IEEE Computer Society, Washington, DC, USA., pp: 246-255.

Manzur, M.M. and R.P. Brent, 2000. Adaptive AT2 optimal algorithms on reconfigurable meshes. Parallel Comput. J., 26: 1447-1458.

Mishra, P. and M.H. Eich, 1992. Join processing in relational databases. ACM Comp. Survey, 24: 63-113.

Mohan, C., H. Pirahesh, W.G. Tang and Y. Wang, 1994. Parallelism in relational database systems. IBM Syst. J., 33: 349-371.

Selim, G.A., 1990. Parallel Sorting Algorithms. Academic Press, Inc. Orlando, FL, USA., pp: 32-59.

Surajit, C. and S. Kyuseok, 1996. Optimization of queries with user-defined predicates. In: Proceedings of 22nd International Conference on Very Large Data Bases, September 3-6, Mumbai (Bombay), India, pp: 87-98.

Wolf, J.L., D.M. Dias and P.S. Yu, 1993. A parallel sort merge join algorithm for managing data skew. IEEE Trans. Parallel Distributed Syst., 4: 70-86.