Distribution-Insensitive Parallel External Sorting on PC Clusters*

Minsoo Jeon and Dongseung Kim

Department of Electrical Engineering, Korea University
Seoul, 136-701, Korea
{msjeon, dkim}@classic.korea.ac.kr
Tel. no.: +822 3290 3232, Fax. no.: +822 928 8909

Abstract. There have been many parallel external sorting algorithms reported such as NOW-Sort, SPsort, and hill sort, etc. They are for sorting large-scale data stored in the disk, but they differ in the speed, throughput, and cost-effectiveness. Mostly they deal with data that are *uniformly* distributed in their value range. Few research results have been yet reported for parallel external sort for data with *arbitrary distribution*. In this paper, we present two distribution-insensitive parallel external sorting algorithms that use sampling technique and histogram counts to achieve even distribution of data among processors, which eventually contribute to achieve superb performance. Experimental results on a cluster of Linux workstations show up to 63% reduction in the execution time compared to previous NOW-sort.

1 Introduction

The capacity of digital data storage worldwide has doubled every nine months for at least a decade, at twice the rate predicted by Moore's Law [12] of the growth of computing power during the same period [5], thus, more data are generated as the information era continues [14]. Processing and storing data will take longer time than before. *External sort* orders large-scale data stored in the disk. Since the size of data is so large compared to the capacity of the main memory, a processor can process only a fraction of the data at a time by bringing them into main memory from the disk, sorting, then storing back onto the disk to make room for the next data. Hence, the sorting usually demands multiple iterations of data retrieval from the disk, ordering computation in main memory, and writing-back to the disk. The time for pure sorting

* This research was supported by KOSEF Grant (no. R01-2001-000341-0).

computation grows in general as a function of $O(N)$ or $O(N \log N)$ as the input size of N increases. Dealing with data stored in disk memory additionally needs a great amount of time since disk memory is very slow compared to the speed of recent CPUs. So, keeping the frequency of disk I/Os as small as possible is very important to achieve high performance.

Many parallel/distributed sorting algorithms have been devised to shorten the execution time. They can be classified into either *merge-based* sorts or *partition-based* sorts. Most merging based algorithms such as *parallel binary merge sort* [16] and *external hillsort* [17] are not fast algorithms because they have to access disk many times in reading and writing data during $\log P$ iterations of merging, where P is the number of processors used in the sorting. Similarly, neither are those bitonic sort [3], odd-even merge sort [3, 7] since they need $O(\log^2 P)$ iterations of *merge split* operations [1]. Although they are flexible and keep even load among processors, they are slow due to $O(N)$ data accesses in each merge split operation.

Partition-based external sorts such as *sample sort* [10] and *NOW-sort* [2] execute in two phases regardless of the data amount: key partitioning into P processors using P-1 pivots, then local sort in each processor. The first phase roughly relocates all data in the order of the indexes of processors by classifying and exchanging keys among processors. Then, all keys in a processor P_i are less than or equal to any key stored in P_j if $i < j$ ($i, j = 0, 1, \ldots, P$-1). The second phase needs only local sorting computation without interprocessor communication. These sorts are generally fast compared to merge based sort. However, if there is imbalance in the number of data among processors, the completion time is lengthened by the heaviest loaded processor (having the maximum data amount). Hence, finding a set of proper *pivots* leading to even partitioning is extremely important to make the best use of the computing power of all processors. These sorts have superior performance due to a small number of disk accesses, but their performance is *sensitive* to the input distribution.

NOW-Sort is known as a fast and efficient external sorting method implemented on a network of workstations with a large scale of disk data (from dozens of gigabytes to terabytes). It uses partial radix sort and bubble sort algorithms, communication primitives called active message, and fast disk I/O algorithms. It is fastest external sorting algorithm in that the number of accesses to hard disk is *2R 2W* (2 read accesses, 2 write accesses) *per key*, which is the fewest disk-accesses among algorithms mentioned above [4, 13]. The detailed algorithm is described in section 2.

Various dialects of NOW-sorting algorithm were devised to achieve high speed, high throughput, or cost-effectiveness. SPsort [18] is a large-scale SMP version of NOW-Sort implemented on IBM RS/6000 SP with 1952 processors, and has a record of sorting a terabyte of data in 1,057 seconds. Datamation, MinuteSort, PennySort, TerabyteSort benchmarks [19] are respectively to achieve a maximum throughput, greatest size of data sorted in a minute, cost-effectiveness, and a minimum time for sorting a fixed number of data. Most of them did not concern about the distribution of

the data, and they usually included only *uniform* distribution for simplicity [19]. Few research results have been yet reported for *arbitrary* distribution.

In this paper, we develop a distribution-insensitive NOW-Sort algorithm together with sample sort to solve the problem. They employ sampling technique and histogram counts for the even distribution of data among all processors. Conditions to get the best performance are investigated, and experimental results on PC cluster are reported.

The rest of this paper is organized as follows. In section 2 and section 3 NOW-Sort and our algorithms are described, respectively. In section 4 the analysis and discussion on the experimental results of the proposed algorithms are given. Conclusion is given in the last section.

2 Partition Based Sort: NOW-Sort

NOW-Sort is a distributed external sorting scheme, implemented on a network of workstations (NOWs) that are interconnected via fast ethernet or Myrinet. The sorting consists of two phases: a data redistribution phase and an individual local sort phase. Detail description is given below.

In the first phase, each processor (node) classifies its data into P groups according to their key values, and it sends them to the corresponding nodes. In the process, keys are classified and replaced according to the order of processor indexes: P_0 contains the smallest keys, P_1 the next smallest, and so on. If the keys are integers, classification is accomplished by bucket sort using a fixed number of *most significant bits* (MSBs) of their binary representation. With uniform distribution a prefix of $\log_2 P$ MSBs is sufficient. It determines where the key should be sent and stored for further processing. Thus, P_0 will be in charge of the smallest keys whose log P MSBs are 0…00 in the binary representation, P_1 the next smallest keys with the prefix of 0…01, … , and P_{P-1} the biggest keys with the prefix of 1…11. To reduce the communication overhead, keys are sent to other processors only after the classified data are filled up in temporary buffers. While processors receive keys from others, they again bucket-sort the incoming keys before storing them into disk to reduce the time for final local sort. The bucket sort uses the next few MSBs after the first $\log_2 P$ bits of each key. Incoming keys are ordered according to the indexes of the buckets, and each bucket will include keys that do not exceed the main memory capacity. The buckets are stored as separate files according to their indexes for future references.

Although keys have been ordered with respect to the order of processors at the end of the phase 1, they are not yet sorted within each processor. As soon as all processors finish the redistribution, the second phase begins: each node repeats the following: load a file of keys to main memory from disk, sort it, and write back to disk. In-place

sort is possible since files are loaded and stored with same names and sizes. If all files are processed, the sorting completes.

To shorten the execution time in the external sort, reducing the number of disk accesses per key is critical. In the first phase of NOW sort, each key is read once from the disk, sent to a certain node, and written back into the disk there. In the local sort phase there is also one disk-read and one disk-write operations per key. Thus, there are 2R2W (2 read accesses, 2 write accesses) disk accesses per key, no matter how many data there are and how many processors the sort includes. 2R2W is the fewest disk accesses of external sort algorithms known so far [4, 13].

In the benchmarking reports such as Datamation and MinuteSort, they do not include all kinds of data distribution. To make the sort simple and fast, they only use keys with *uniform* distribution in developing a best sorting algorithm. In case of non-uniform distribution processors may have uneven work load after redistribution, thus, the execution time may increase considerably. The following section describes our solutions for the problem that are insensitive to the data distribution.

3 Distribution-Insensitive External Sort

3.1 Sample Sort

In this section we modify NOW-Sort algorithm in order to use sampling techniques. Sample sort is a two-stage parallel sorting algorithm capable of ordering keys with *arbitrary* distribution. Keys initially stored evenly throughout the nodes are partitioned using *splitters* and relocated according to their values so that P_0 will have the smallest keys, P_1 the next smallest, …, P_P the largest. Thus, no more key exchanges among nodes are needed in the later stage. Now each node sorts its keys locally, then, the whole keys are ordered among processors and within the processors as well. More explanation follows.

The splitters (or pivots) are obtained by *sampling*. Each node samples a fixed number of keys, and then they are collected and sorted in a master processor. P-1 splitters are collected from them having equal interval in the sorted list, and they are broadcasted to all nodes. Now, all nodes read their own keys, classify them into P groups using the pivots, and transmit them to P nodes according to the indexes of the groups. At the same time, each node receives keys from others. After the exchange, each node individually sorts its own keys. In this parallel computation, if the load of each node is roughly same, all nodes can finish the job at about the same time, which gives the best performance in the sample sort. If not, the performance is aggravated by the most heavily loaded processor, in which case the time should be longer than the balanced case.

Sample sort is a very excellent sorting scheme when the distribution of keys is not uniform, and especially when the information is not known ahead. Depending on how many samples are extracted from the input, the quality of the resultant partitions of the keys varies. Although the use of many samples produces a good result, it consumes computation time significantly. On the other hand, too few samples may give coarse data distribution. In section 4, three different sampling rates are experimented and compared.

3.2 Histogram-Based Parallel External Sort

This algorithm is similar to the sample sort in the aspect that it classifies and distributes using P-1 splitters, and then local sort follows. It differs from the sample sort in the way to find the splitters: it employs histogram instead of sampling [6, 9]. The algorithm executes in two phases too: histogram computation and bucket sort phase (phase 1) and data distribution and post-processing phase (phase 2). In phase 1, each of the P nodes reads data from disk, computes local histogram to get the information about the distribution of its keys. While a node computes histogram using a fixed number of MSBs of the keys, it also performs bucket sort and places data into disk files bucket by bucket. Although keys have been ordered with respect to the order of buckets, they have not been sorted yet within each bucket. Histogram data in each processor are sent to a master processor (for example, P_0), and then the global histogram is computed by summing up them. P-1 pivots are selected from the boundary values of the histogram bins that divide keys evenly to all processors. The pivots are broadcasted to all P nodes from the master. In phase 2, each node reads a fixed number of data from disk, classifies the data into P buffers in memory, and sends each buffered data to the corresponding node. While it transmits its data to other nodes, it also receives data from other nodes. Incoming data are gathered, sorted, and stored into disk bucket by bucket. This process continues until all disk data are consumed. This sorting demands 2R2W accesses per key, too. The detailed algorithm is described below.

Let N, P denote the number of total keys and the number of processors, respectively. P_0 is the master node that calculates the global histogram and finds splitters. N/P keys are initially stored in the disk of each node.

Phase 1: Histogram computation and bucket sort phase

(S1) Each node iterates the following computation:

1. Each node reads a fixed number of keys from disk.
2. Each node calculates local histogram. While a node computes histogram, it also performs bucket sort such that each key is assigned to the corresponding histogram buffer where the key belongs. It stores (appends) the buffered data into disk files according to the indexes whenever they are full.

(S2) Each node sends local histogram information to the master node (P_0). It later receives splitters from P_0.

The master node performs the following computation additionally:

1. It collects local histograms and calculates the global histogram.
2. It computes the accumulated sums of the histogram counts and selects P-1 splitters: The first splitter is chosen to the histogram boundary to include the $\frac{N}{P}$ smallest keys in the range, the next splitter to include the next $\frac{N}{P}$ smallest keys, and so on. Notice that splitters derived from very fine-grain histogram will produce quite even partitioning, however, moderate histogram suffices for good load balancing in practice.

Phase 2: Data redistribution and post-processing phase

(S3) Each node reads data from disk bucket by bucket, partitions them into P-1 bins by the splitters, and sends them to the corresponding nodes. At the same time it gathers, sorts, and stores into disk the incoming data from other nodes bucket by bucket. The process continues until all data in the disk are read out and there are no more incoming data.

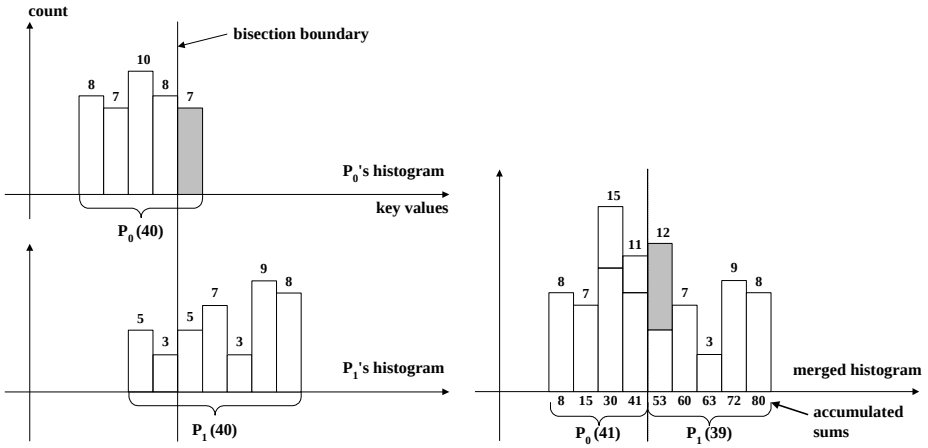
An example is given in Fig. 1 to illustrate the pivot selection for two processors (P=2) with N=40 keys stored in each processor. The master node gathers histograms (Fig. 1(a)), merges them, and computes the accumulated sums (Fig. 1(b)). The pivot (in this case only one pivot is needed) is the upper boundary value of fourth histogram bin with the accumulated sum of 41, which is closest to the even partitioning point of 40. Now, those keys in the shaded bins are exchanged with the other processor. P_0 and P_1 then have 41 & 39 keys after redistribution, respectively.

The complexity of the algorithm will now be estimated. In phase 1, step S1.1 requires the time to read all data from local disk (T_{read}). S1.2 requires the time to calculate histogram ($T_{comp.hist}$), to bucket-sort (T_{bucket}), and to write data to disk (T_{write}). S2 requires the communication time to send and receive histogram information and splitters ($T_{comm.hist}$). In phase 2, S3 requires the time to read data from disk bucket by bucket (T_{read}), the time to classify and send to the corresponding nodes (T_{comm}), the time to sort the incoming data (T_{sort}), and the time to write the sorted data onto disk (T_{write}). The expected time can be estimated as below:

$$T_{total} = (T_{read} + T_{comp.hist} + T_{bucket} + T_{write} + T_{comm.hist}) + (T_{read} + T_{comm} + T_{sort} + T_{write})$$

It is simplified as

$$T_{total} = 2T_{read} + 2T_{write} + T_{comm} + T_{sort} + T_{etc} \quad (\mathbf{1})$$



(a) Individual histograms (b) Merged histogram

Fig. 1. Example of pivot selection.

where $T_{etc} = T_{comp.hist} + T_{comm.hist} + T_{bucket}$ Let B_r, B_w, B_c, B_s, and B_b be the disk read bandwidth, disk write bandwidth, network bandwidth, the number of data that can be sorted by radix sort per unit time, and the number of data that can be processed by bucket sort in unit time, respectively. They are all constants in general. By substituting the parameters, Eq. 1 is rewritten as follows:

$$\begin{aligned} T_{total} &= \frac{2N}{P}\left(\frac{1}{B_r}+\frac{1}{B_w}\right)+\left(\frac{P-1}{P}\right)\cdot\frac{2N}{P}\cdot\frac{1}{B_c}+\frac{N}{P}\cdot\frac{1}{B_s}+\frac{N}{P}\cdot\frac{1}{B_b} \\ &= \frac{N}{P}\cdot\left\{\frac{2}{B_r}+\frac{2}{B_w}+\left(\frac{P-1}{P}\right)\cdot\frac{2}{B_c}+\frac{1}{B_s}+\frac{1}{B_b}\right\} \\ &\approx O\left(\frac{N}{P}\right) \end{aligned} \quad (2)$$

In Eq. 2 since $\left\{\frac{2}{B_r}+\frac{2}{B_w}+\cdots+\frac{1}{B_b}\right\}$ is a constant with many processors, the running time of the algorithm is approximately $O\left(\frac{N}{P}\right)$.

4 Experiments and Discussion

The algorithms are implemented on a PC cluster. It consists of 16 PCs with 933 MHz Pentium III CPUs interconnected by a Myrinet switch. Each PC runs under linux with

256MB RAM and 20GB hard disk. The programs are written in C language with MPI communication library (MPICH-GM). Experiments are performed on the cluster of up to P=16 nodes, and the size of input data ranges from N=256MB to N=1GB per node. Input keys are 32-bit integers synthetically generated with three distribution functions (*uniform*, *gauss*, and *stagger*). More distribution will be included in the near future.

Fig. 2, 3, and 4 show the execution times of the algorithms for uniform, gaussian, and staggered distribution. *Sample*, *histogram*, and *NOW* in the figures represent sample sort, histogram-based sort, and generic NOW-Sort, respectively. For uniform and staggered distribution as observed in Fig. 2 and 4, generic NOW-Sort outperforms the two distribution-insensitive sorting schemes since it does not have any overhead to reflect data distribution characteristics during the sort. However, it delivers quite inferior performance for gaussian distribution when more than two processors are used (see Fig. 3). *Optimum* sample sort and histogram-based sort have about an order of magnitude speedup in the case, up to 63% reduction in the execution time compared to NOW-sort, and the histogram-based algorithm has a shorter execution time by 13.5% to 20% than sample sort, as observed in Fig. 5. The performance improvement comes from the way to classify keys. While the histogram-based algorithm classifies keys using prefix of each key, with the complexity of $O(N)$, the sampling adopts binary search for the classification with the complexity of $O(N \log P)$. The job consumes up to 20% of the overall execution time. In all cases the histogram-based algorithm has better performance than the sample sort.

Three different sample counts of $2P(P\text{-}1)$, $\log N$, and $\sqrt{N}$ of the total N data, are chosen for *regular* [15], *mid-range* [8], and *optimum* degrees [11], respectively. Fig. 5 and 6 show the respective execution times and the degree of load imbalance of the sorting. Load distribution is well balanced with $\sqrt{N}$ sampling rate among them. Load balancing of histogram-based sort is as good as that of sample sort with $\sqrt{N}$ sampling rate. The term *optimum* comes from the fact that it produces the best result in our experiments, hinted by quicksort and quickselect achieving the shortest execution time when the sample count is $\sqrt{N}$ [11]. Regular sample judges with insufficient information, thus, the load allocation is very poor, whereas mid-range and optimum maintain well balanced load among processors. We can easily tell the reason from Fig. 7 which analyzes the execution times of the heaviest loaded processors: The regular sampling has the greatest computation time, the most data communication, and the longest disk access time, all due to having the largest number of data to process.

Load balancing in the two parallel sorting methods is maintained fairly well when histogramming and $\sqrt{N}$ samples are used. In case of gaussian distribution, the maximum deviation of load from the perfectly even distribution for all three distributions is lesser than 1% as observed in Fig. 6(b). The load imbalance is also minimum in the histogram-based algorithm for other distribution functions.

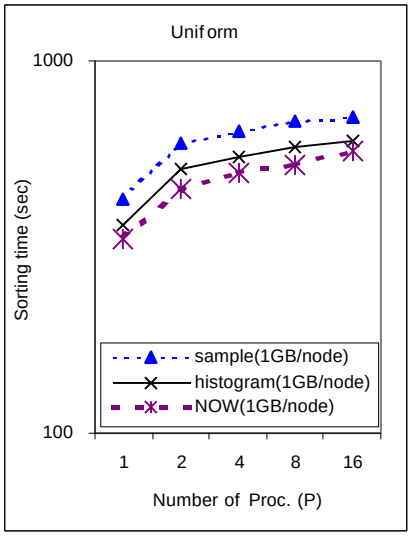


Fig. 2. Execution times of three sorts (sample, histogram, and NOW) for *uniform* distribution.

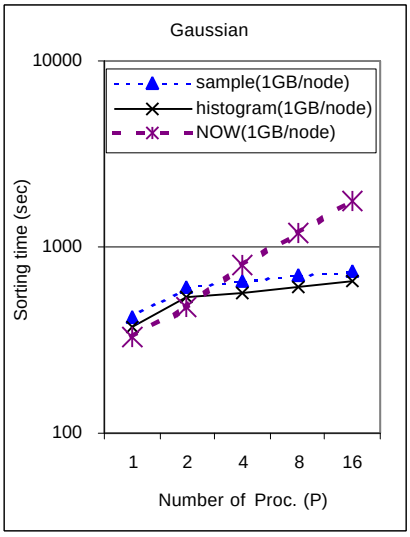


Fig. 3. Execution times of three sorts (sample, histogram, and NOW) for *gaussin* distribution.

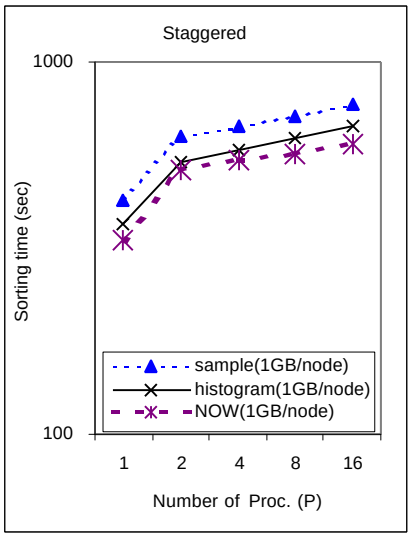


Fig. 4. Execution times of three sorts (sample, histogram, and NOW) for *staggered* distribution.

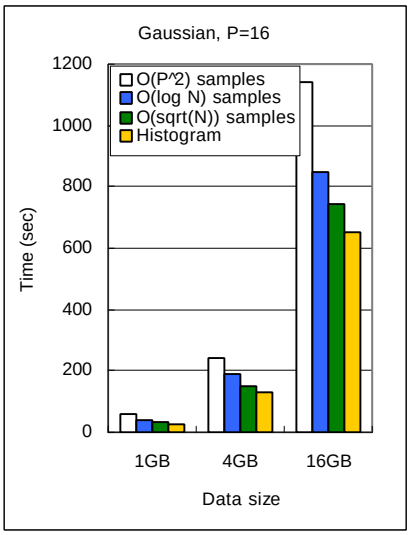
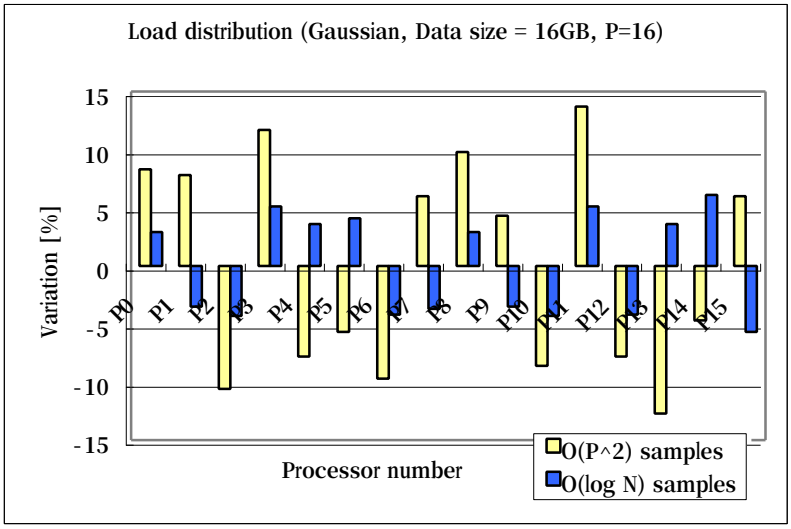
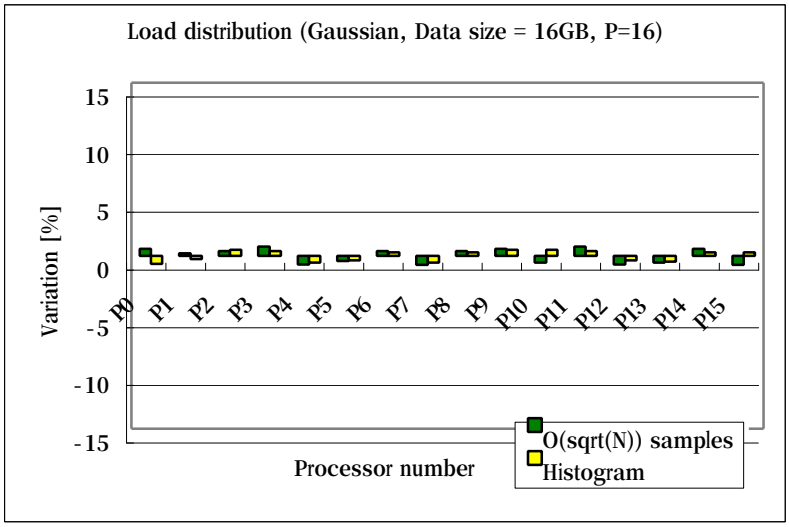


Fig. 5. Execution times of histogram- based sort and sample sort for *gaussian* distribution. Sample sort uses three different sampling rates.



(a) Sample sort with different sample counts.



(b) Optimal sample sort and histogram-based sort.

Fig. 6. Degree of load imbalance in participating processors for *gaussian* distribution.

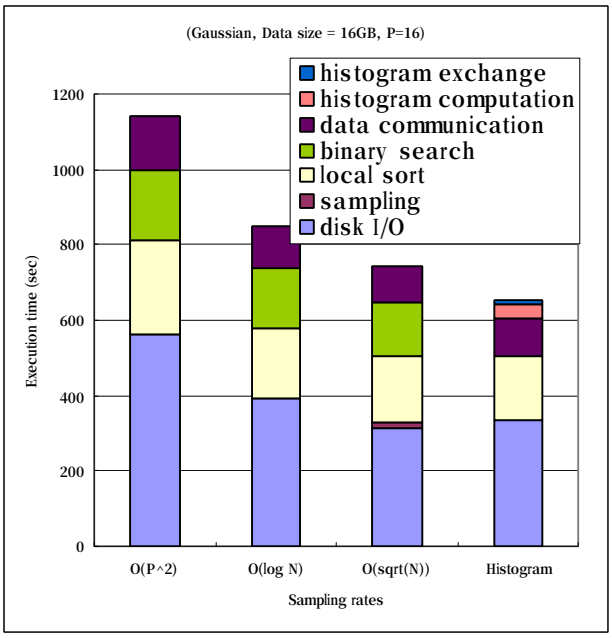


Fig. 7. Analysis of execution time for *gaussian* distribution.

5 Conclusion

In this paper we report enhanced external sorting algorithms that can be applied to data with arbitrary distribution. Sample sorting is used, and histogram-based NOW-Sort is developed. Both algorithms partition and redistribute data so that all processors have even load. They differ in the way to select the splitters for the distribution. The algorithms require 2R2W disk accesses regardless of the size of data and the number of processors. Thus, they are as fast as NOW-Sort, and the execution time reduces to an order of magnitude with respect to NOW-Sort for *non-uniform* data.

The difficulty in the algorithms is to find a proper number of histogram bins, and the size of buckets in the redistribution process, both of which should be chosen to maximally use the main memory. We are developing a better method to classify data in the first phase of the sample sort, and extending the experiments to include other distribution functions.

References

1. S. G. Akl: The design and analysis of parallel algorithms, Chap. 4, Prentice Hall (1989)
2. A. C. Arpaci-Desseau, R. H. Arpaci-Desseau, D. E. Culler, J. M. Hellerstein, and D. A. Patterson: High-performance sorting on networks of workstations. ACM SIGMOD '97, Tucson, Arizona (1997)
3. K. Batcher: Sorting networks and their applications. Proc. AFIPS Spring Joint Computer Conference 32, Reston, VA (1968) 307-314
4. A. A. Dusseau, R. A. Dusseau, D. E. Culler, J. M. Hellerstein and D. A. Patterson: Searching for the sorting record: experiences in tuning NOW-Sort. Proc. SIGMETRICS Symp. Parallel and Distributed Tools (1998) 124-133
5. U. Fayyad and R. Uthurusamy: Evolving data mining into solutions for insights. Communications of the ACM, Vol. 45, No. 8 (2002) 29-31
6. M. Jeon and D. Kim: Parallel merge sort with load balancing. Int'l Journal of Parallel Programming, Kluwer Academic Publishers, Vol. 31, No.1 (2003) 21-33
7. D. E. Knuth: The Art of Computer Programming, Volume III: Sorting and Searching, Addison-Wesley (1973)
8. J.-S. Lee, M. Jeon, and D. Kim: Partial sort. Proc. Parallel Processing System, Vol. 13, No. 1, Korea Information Science Society (2002) 3-10
9. S.-J. Lee, M. Jeon, D. Kim, and A. Sohn: Partitioned parallel radix sort. Journal of Parallel and Distributed Computing, Academic Press, Vol. 62 (2002) 656-668
10. X. Li, et. al.: A practical external sort for shared disk MPPs. Proc. Supercomputing '93 (1993) 666-675.
11. C. C. Mcgeoch and J. D. Tygar: Optimal sampling strategies for quicksort. Random Structures and Algorithms, Vol. 7 (1995) 287-300
12. G. E. Moore: Cramming more components onto integrated circuits. Electronics, Vol. 38, No. 8 (1965)
13. F. Popovici, J. Bent, B. Forney, A. A. Dusseau, R. A. Dusseau: Datamation 2001: A Sorting Odyssey. In Sort Benchmark Home Page.
14. J. Porter: Disk trend 1998 report. http://www.disktrend.com/pdf/portrpkg.pdf
15. R. Raman: Random sampling techniques in parallel computation. Proc. IPPS/SPDP Workshops (1998) 351-360
16. D. Taniar and J. W. Rahayu: Sorting in parallel database systems. Proc. High Performance Computing in the Asia Pacific Region, 2000: The Fourth Int'l Conf. and Exhibition Vol. 2 (2000) 830-835
17. L. M. Wegner, J. I. Teuhola: The external heapsort. IEEE Trans. Software Engineering, Vol. 15, No. 7 (1989) 917-925
18. J. Wyllie: SPsort: How to sort a terabyte quickly. Technical Report, IBM Almaden Lab., http://www.almaden.ibm.com/cs/gpfs-spsort.html (1999)
19. http://research.microsoft.com/barc/SortBenchmark, In Sort Benchmark Home Page.