

High-Speed Parallel External Sorting of Data with Arbitrary Distribution*

Minsoo Jeon and Dongseung Kim

Department of Electrical Engineering, Korea University

Seoul, 136-701, Korea

{msjeon, dkim}@classic.korea.ac.kr

Tel. nos.: +822 3290 3232(voice), +822 928 8909(fax)

Abstract. Many parallel sorting algorithms of (external) disk data have been reported such as NOW-sort, SPsort, and hill sort, etc. They all reduce the execution time compared to some known sequential sort; however, they differ in terms of the speed, throughput, and cost-effectiveness. Mostly they deal with data that are *uniformly* distributed in their value range. If we divide and redistribute data to processors using the fixed and equal division of the key range, all processors will have about equal number of keys to sort and store. But if irregularly distributed data are given, the performance will suffer severely since the partitioning would no longer produce balanced load among processors. Few research results have been yet reported for parallel external sort of data with *arbitrary distribution*. In this paper, we develop two distribution-insensitive scalable parallel external sorting algorithms that use sampling technique and histogram counts to achieve even distribution of data, which eventually contribute to achieve superb performance. Experimental results on a cluster of 16 Linux workstations show up to three-fold enhancement of the performance compared to previous NOW-sort for sorting 16GB integer keys.

Keywords: external sort, NOW-sort, sample sort, cluster, load balancing, and histogram

1 Introduction

Due to the growing internet, we share the greatest data of the world in the history ever. It is reported that 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 [13] of the growth of computing power during the same period [5], thus, more data will be generated as the information era continues [15]. 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 big compared to

* This research was supported by KOSEF Grant (no. R01-2001-000341-0). A preliminary version of the paper was presented in ISHPC-V, Tokyo, Japan, Oct. 20-22, 2003.

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 computation grows in general as a function of $O(N \log N)$ as the input size of N increases. Dealing with data stored in disk memory additionally needs a great deal 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* [17] and *external hillsort* [18] are not fast 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, 8]. Although they are flexible and can make even load distribution, they are slow due to $O(N)$ data accesses in each of $O(\log^2 P)$ *merge-split operations*[1].

Partition-based external sorts such as *sample sort* [11] and *NOW-sort* [2] execute in two phases regardless of the size of the data: key partitioning into P processors using $P-1$ *pivots* (or *splitters*), then local sort in each processor. The first phase classifies and relocates all data according to their key value using $P-1$ pivots. It results in inter-processor data allocation in such a way that all keys in processor P_i are less than or equal to any key stored in P_j if $i < j$ ($i, j = 0, 1, \dots, P-1$). The second phase needs only local sorting computation without interprocessor communication. These sorts are generally fast compared to merge based sorts. 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). Hence, finding a set of proper splitters leading to even partitioning is extremely important to make the best use of the computing power of all processors. In other words, their performance is *sensitive* to the input distribution. The selection of proper splitters can not be done easily if data have non-uniform distribution.

NOW-sort is known as a fast and efficient external sorting method implemented on a network of workstations with large disk data (from dozens of gigabytes to terabytes) [2,4]. 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 the disk is $2R 2W$ (2 read accesses, and 2 write accesses) *per key*, which is the fewest disk-accesses among algorithms mentioned above [4, 14]. The detailed algorithm is described in section 2. Various dialects of NOW-sorting algorithm have been devised to achieve high speed, high throughput, or cost-effectiveness. SPsort [19] 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 [20] are to achieve respectively 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 do not concern about the distribution of the data, and they usually

considered only *uniform* distribution for simplicity [20]. 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 partitioning of data. Conditions to get the best performance are investigated, and experimental results on a cluster system 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 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. For simplicity and best performance, we assume that each workstation is identical, having a fast CPU, main memory, and a local (non shared) disk drive. There are N/P keys stored in each disk in the beginning. The sorting consists of two phases as described below: a data redistribution phase and a local sort phase.

In the first phase, each processor (node) classifies and transmits its data to P processors according to their key values using $P-1$ pivots. For example, if the keys are *integers*, classification is accomplished by bucket sort using a few *most significant bits* (MSBs) of their binary representation. If the keys have uniform distribution, a prefix of $\log_2 P$ MSBs can be used to send the keys to proper processors for further processing. Thus, P_0 will receive 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. When processors receive keys from others, they again bucket-sort the incoming keys before storing them into disk to reduce the time for local sort in the second phase. The bucket sort also uses a 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. Each bucket is stored as a separate file according to the index for future references.

Although keys have been ordered with respect to the order of processors at the end of the first phase, 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, and write back to disk. *In-place sorting* is possible since files are loaded and stored with same names and sizes. The sorting completes when all files are processed.

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 for classification, 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, 14].

In the benchmarking reports on NOW sort 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 Sorts

3.1 Parallel Sample Sort

In this section we expand NOW-sort using the sample sort. 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 are obtained by *sampling*. Each node randomly samples a fixed number of keys in the local disk, and then they are gathered and sorted in a master processor. $P-1$ splitters are picked up that have an 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 becomes aggravated by the most heavily loaded processor, in which case the execution time cannot be shorter than the balanced one.

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 imprecise data distribution, thus it lengthens the computation time too. In section 4, three representative 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, 10]. 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 its own 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 the data into disk files bucket by bucket. By bucket sort, keys are ordered with respect to the indexes of buckets; however, they have not been yet ordered within each bucket. Histogram data in each processor are sent to a master processor (for example, P_0), and the global histogram is computed. $P-1$ pivots are selected from the boundary values of the histogram bins that will divide keys evenly to all processors. Then, 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, too². Detailed description of the algorithm is given 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 selects splitters. N/P keys are initially stored in the disk of each node.

Phase 1: Histogram computation and bucket sort

(S1) Each node iterates the following computation:

1. Each node reads a fixed number of keys from disk.
2. Each node calculates local histogram. Simultaneously, 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 $P-1$ splitters from P_0 .

The master node performs the following computation additionally:

1. It collects local histogram data 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.

² It is also possible to sort in $2R1W$ accesses if more elaborated preprocessing and data movement scheduling are made. In fact, due to the delay and overhead in synchronization, the performance we obtained is not better than the $2R2W$ scheme.

Phase 2: Data redistribution and post-processing

(S3) Each node reads data from disk bucket by bucket, partitions them into P 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 bucket by bucket. The process continues until all data in the disk are read out and no more incoming data arrive.

In the algorithm, we check whether data are overcrowded in specific histogram bins at the end of S1, which is omitted for simplicity. If needed, they can be splitted further. Splitters obtained from very fine-grain histogram will produce quite even partitioning, however, moderate histogram with less overhead suffices for good load balancing in practice.

Now, an example is given in Fig. 1 to illustrate the pivot/splitter selection for two processors ($P=2$) with $N=40$ keys stored in each processor. The master node gathers histograms (Fig. 1(a)), and computes the accumulated sums (Fig. 1(b)). The pivot (in this case only one pivot is needed) is the upper boundary value of the 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 between the two processors, after which P_0 and P_1 have 41 and 39 keys, respectively.

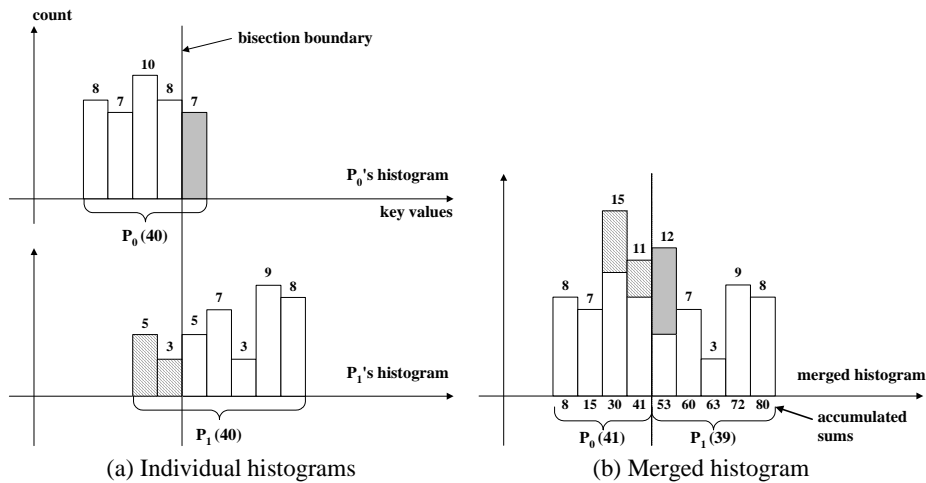


Fig. 1. Histogram counts are used to partition the input such that the two processors will have load as equal as possible after the data exchange.

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 and splitters information ($T_{comm.hist}$). 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

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

It is simplified as

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

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 regarded constants in general because they represent worst-case lower/upper bounds. Since the algorithm makes the load of processors reasonably balanced, there are about $\frac{N}{P}$ keys per node for the local sort in Phase 2. Under the balanced condition Eq. 1 is rewritten with the parameter substitution as follows:

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

where $\left\{ \frac{2}{B_r} + \frac{2}{B_w} + \dots + \frac{1}{B_b} \right\}$ is regarded a constant with many processors. The running time of $O\left(\frac{N}{P}\right)$ tells the important characteristic that the algorithm is *scalable* in terms of the number of processors and the data size.

4 Experiments and Discussion

The algorithms are implemented on a PC cluster. It consists of 16 linux workstations 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 $N=64$ MB to $N=1$ GB per node. Input keys are 32-bit integers synthetically generated with three distribution functions (*uniform*, *gauss*, and *stagger*)[6]. More distribution will be included in the near future.

At first, the performance of distribution insensitive sorting algorithms is checked in a straightforward manner by observing how well they actually divide data to processors after Phase 1. Compared to generic NOW-sort and coarsely sampled sample sort, histogram-based parallel sort and *optimal* sample sort produce very even key distribution as shown in Fig. 2. (A processor has load with 0% deviation if it is allocated with N/P keys.) Of course generic NOW-sort shows the worst load allocation since it does not include any routine to take care of nonuniform distribution at all.

In the implementation of sample sort, three different sample counts of $2P(P-1)$, $\log N$, and \sqrt{N} of the total N data are employed to represent *light* [16], *mid-range* [9], and *optimal rates* [12], respectively. Work load is best balanced with \sqrt{N} sampling rate among the three. The term *optimal* 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} [12]. With light sample rate the algorithm judges with insufficient information, thus, the load allocation is very poor, whereas mid-range and optimum rates maintain well-balanced load among processors. We can easily tell the reason from Fig. 3 which analyzes the execution times of the heaviest loaded processors: although the pure sorting times are nearly equal for all cases, the light sampling has the greatest computation time, the most data communication, and the longest disk access time, all due to the largest number of data allocated to it.

Load balancing of histogram-based sort is as good as that of sample sort with \sqrt{N} sampling rate. 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. 2(b). (Note that the scale in Fig.2(b) is magnified compared to 2(a).) The load imbalance is minimal in the histogram-based algorithm for other distribution functions.

Figs. 4,5 and 6 show the execution times of the algorithms for uniform, gaussian, and staggered distribution with a constant data of 1GB per node. *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. 4 and 6, generic NOW-sort outperforms the two distribution-insensitive sorting schemes since it does not have any overhead to detect the data distribution characteristics during the sort. However, it delivers quite inferior performance for gaussian distribution when more than two processors are used³ (see Fig. 5).

Figs. 7,8 and 9 also show the execution times when the number of processors is fixed to 16. They show that the execution time is *linear* with respect to the data size, as expected in the analysis in section 3. Generic NOW-sort is comparable to the two distribution-insensitive sorting schemes for uniform and staggered distribution, but it shows quite inferior performance for gaussian distribution.

Figs. 10,11 and 12 plot the speedups for three data distribution. They show almost-linear speedup curves with respect to P . The generic NOW-sort shows the lowest speedup for gaussian distribution. Finally, optimal sample sort and histogram-based sort have about an order of magnitude improvement, i.e. 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 Figs. 11&13. The superior performance comes from the way to classify keys. While the histogram-based algorithm classifies keys using the prefix of each key, with the complexity of $O(N)$, the sampling employs binary search for the classification with the complexity of $O(N \log P)$. The job consumes up to 20% of the overall execution time as observed in Fig 3. In all cases the histogram-based algorithm performs better than the sample sort.

³ When two processors are used, NOW-sort outperforms although it deals with nonuniform distribution, since the pivot is set in the half point of the key range, which divides data into two equal parts, resulting in even load distribution.

5 Conclusions

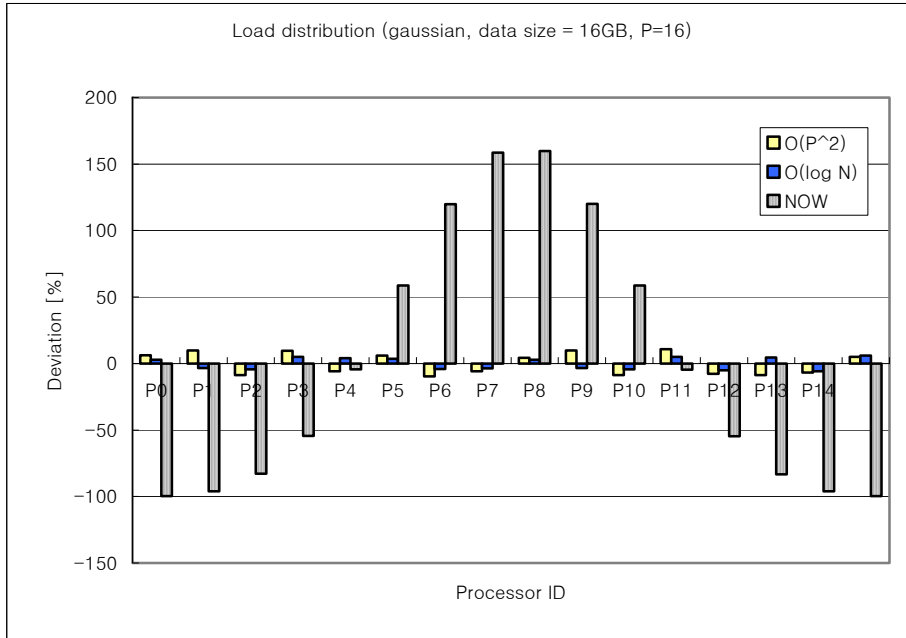
In this paper we present two enhanced external sorting algorithms that can be applied to sort data with arbitrary distribution. Sample sorting is used, and histogram-based NOW-sort is developed. Both algorithms partition and redistribute data evenly to all processors with no dependence on their distribution. They differ in the way to select the splitters for the distribution. The algorithms require only 2R2W disk accesses regardless of the size of data and the number of processors. The reduction of execution times reaches up to an order of magnitude compared to NOW-sort especially for *non-uniform* data, and the improvement is significant since external sort usually deals with huge data and a number of lengthy disk accesses.

The future research of the algorithms is to develop a better way to adaptively set the number of histogram bins and the size of buckets in the redistribution process, both of which should be chosen at runtime to maximally utilize the main memory. We are also developing a hybrid method to classify data in the first phase of the sample sort, and extending the experiments to include various distribution functions.

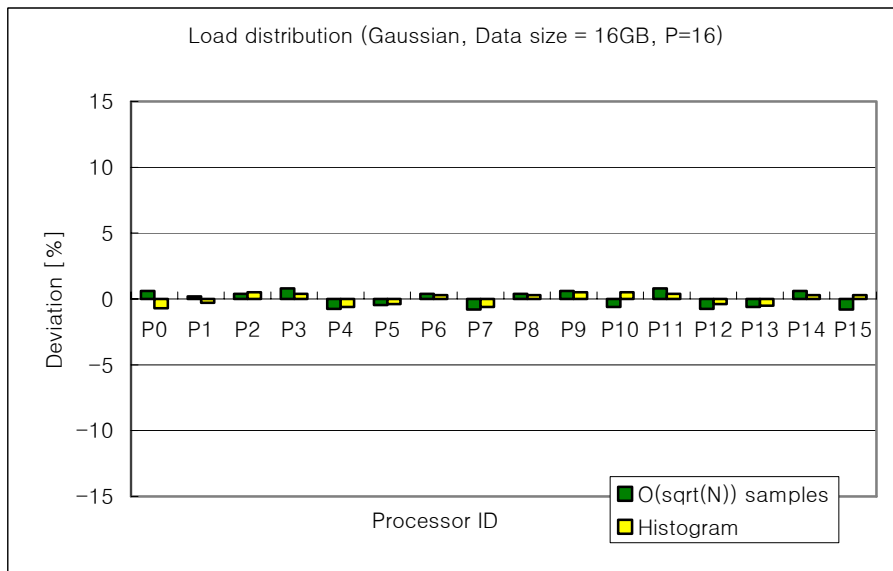
References

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. D.R. Hellman, D.A. Bader, and J. JáJá, Parallel algorithms for personalized communication and sorting with an experimental study, Procs. of the ACM Symp. on Parallel Algorithms and Architectures, Padua, Italy, June 1996, pp.211-220.
7. 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

8. D. E. Knuth: The Art of Computer Programming, Volume III: Sorting and Searching, Addison-Wesley (1973)
9. 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
10. 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
11. H. Shi and J. Schaeffer, Parallel sorting by regular sampling, Journal of Parallel and Distributed Computing, Vol.14 (1992) 361-372
12. C. C. Mcgeoch and J. D. Tygar: Optimal sampling strategies for quicksort. Random Structures and Algorithms, Vol. 7 (1995) 287-300
13. G. E. Moore: Cramming more components onto integrated circuits. Electronics, Vol. 38, No. 8 (1965)
14. F. Popovici, J. Bent, B. Forney, A. A. Dusseau, R. A. Dusseau: Datamation 2001: A Sorting Odyssey. In Sort Benchmark Home Page.
15. J. Porter: Disk trend 1998 report. <http://www.disktrend.com/pdf/portrpkg.pdf>
16. R. Raman: Random sampling techniques in parallel computation. Proc. IPPS/SPDP Workshops (1998) 351-360
17. 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
18. L. M. Wegner, J. I. Teuhola: The external heapsort. IEEE Trans. Software Engineering, Vol. 15, No. 7 (1989) 917-925
19. J. Wyllie: SPsort: How to sort a terabyte quickly. Technical Report, IBM Almaden Lab., <http://www.almaden.ibm.com/cs/gpfs-spsort.html> (1999)
20. <http://research.microsoft.com/barc/SortBenchmark>, In Sort Benchmark Home Page.



(a) NOW-sort and sample sort with two different sample counts.



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

Fig. 2. Degree of load imbalance in participating processors for *gaussian* distribution.

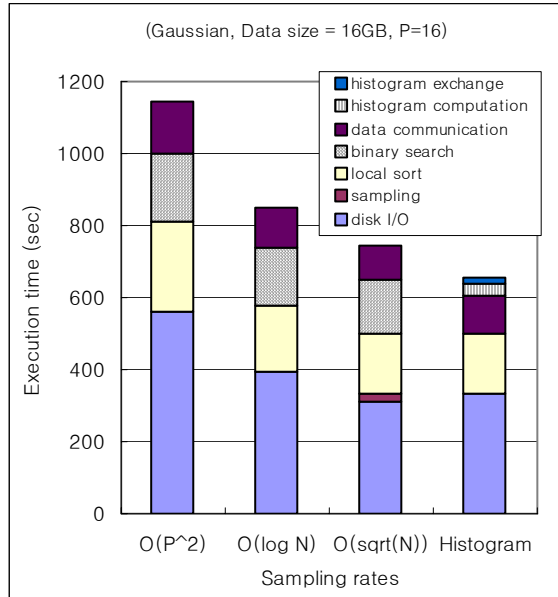


Fig. 3. Analysis of execution time of the heaviest loaded processors for *gaussian* distribution.

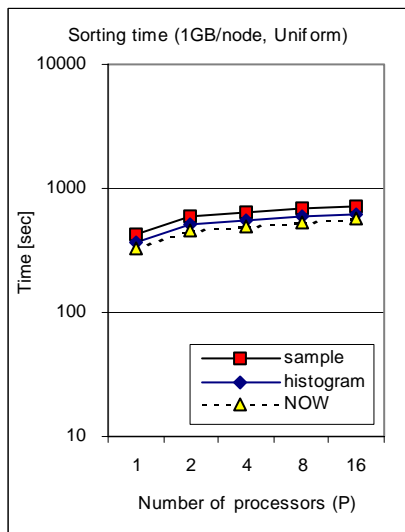


Fig. 4. Execution times of three sorts (sample, histogram, and NOW) for *uniform* distribution.

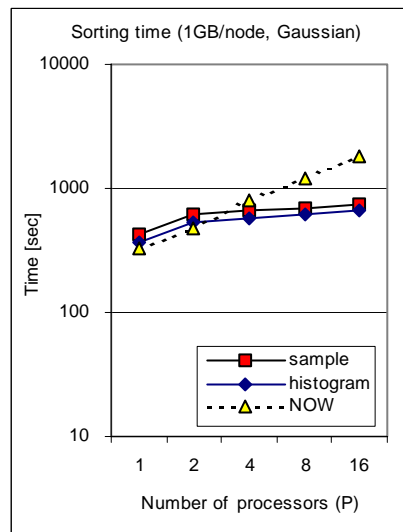


Fig. 5. Execution times of three sorts for *gaussian* distribution

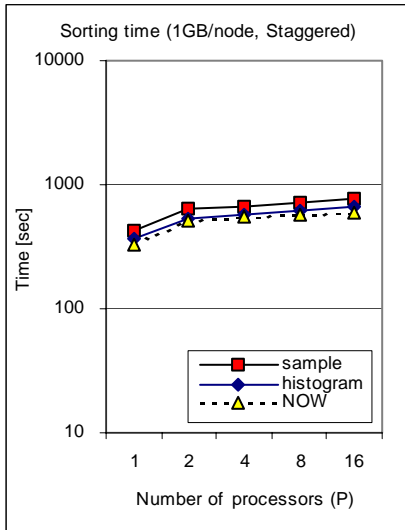


Fig. 6. Execution times of three sorts for *staggered* distribution.

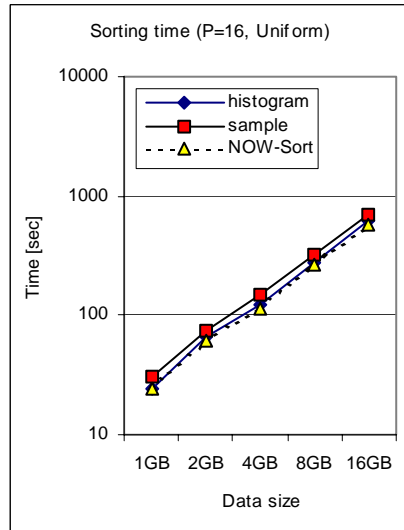


Fig. 7. Execution times of three sorts (sample, histogram, and NOW) for *uniform* distribution.

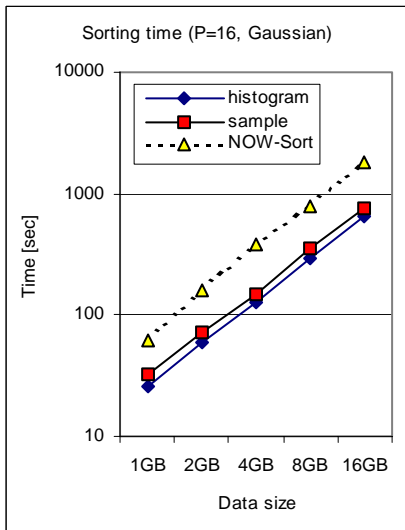


Fig. 8. Execution times of three sorts for *gaussian* distribution

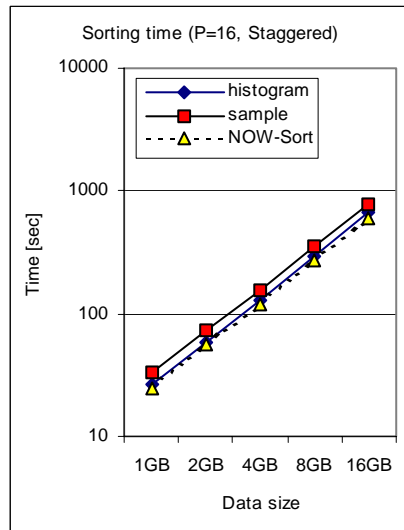


Fig. 9. Execution times of three sorts for *staggered* distribution.

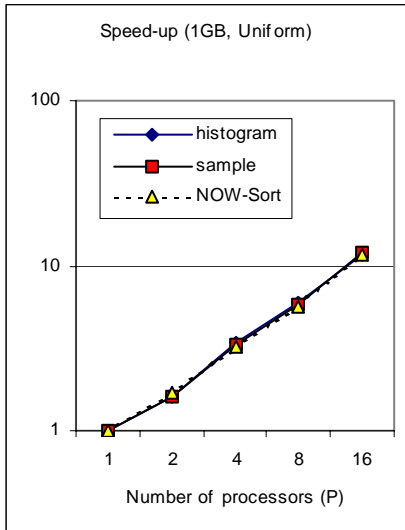


Fig. 10. Speedups of three sorts for *uniform* distribution.

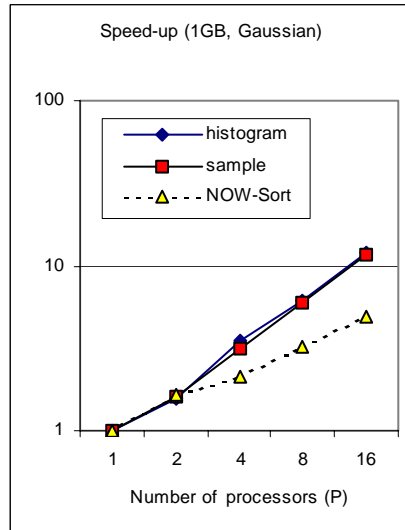


Fig. 11. Speedups of three sorts for *gaussian* distribution.

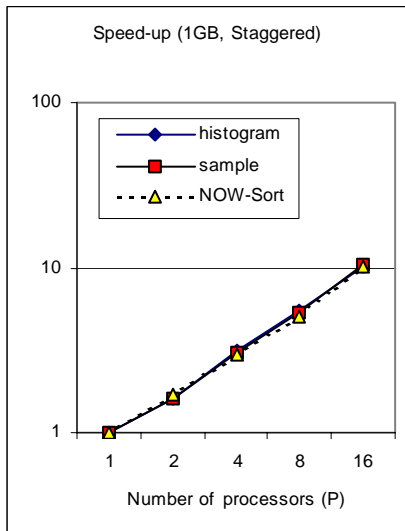


Fig. 12. Speedups of three sorts for *staggered* distribution.

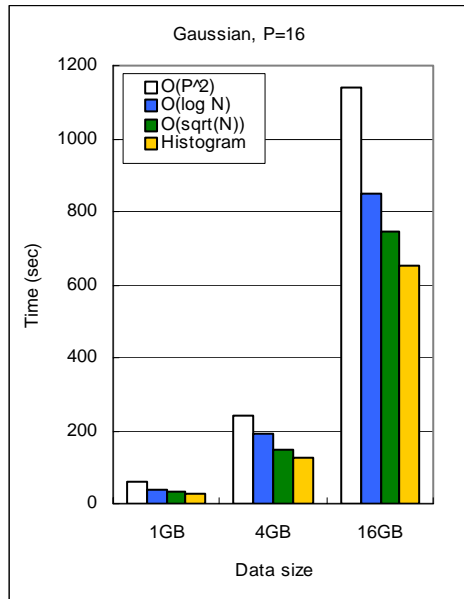


Fig. 13. Execution times of histogram-based sort and sample sort for *gaussian* distribution with 16 processors.