Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Fast sort of floating-point data for data engineering

Changsoo Kim, Sungroh Yoon, Dongseung Kim *

School of Electrical Engineering, Korea University, Seoul 136-713, Republic of Korea

ARTICLE INFO

Article history:
Received 13 May 2010
Received in revised form 19 September 2010
Accepted 26 October 2010

Keywords:
Parallel sort
Floating-point arithmetic
External sort
Engineering simulation
Workstation cluster
Message passing interface

ABSTRACT

In this paper, a novel external sort algorithm that improves the speedup of the sorting of floating-point numbers has been described. Our algorithm decreases the computation time significantly by applying integer arithmetic on floating-point data in the IEEE-754 standard or similar formats. We conducted experiments with synthetic data on a 32-processor Linux cluster; in the case of the internal sort alone, the Giga-byte sorting achieved approximately fivefold speedups. Furthermore, the sorting achieved two-fold or greater improvements over the typical parallel sort method, network of workstations (NOW)-sort. Moreover, the sorting scheme performance is independent of the computing platform. Thus, our sorting method can be successfully applied to binary search, data mining, numerical simulations, and graphics.

1. Introduction

Sorting is a fundamental operation widely used in many applications such as data searching, job scheduling, database management, and engineering simulations [1]. Numerous high-performance sorting algorithms have been previously developed to decrease the time required [2–5], including one algorithm that employed fast graphic processor units [6]. The sorting of integer keys is simple and flexible, but the sorting of real numbers needs floating-point arithmetic for comparison, which usually takes longer time than the integer sorting. If floating-point data can be sorted by integer arithmetic, the execution time can be significantly shortened and the overall task can become considerably faster.

Internal sort refers to the ordering of the amount of data that will fit in the main memory, while in external sort, large-scale data that are often stored in storage disks are ordered. Hence, external sort demands multiple iterations of data retrieval from the disk first, ordering computations in the main memory next, and finally, writing back to the disk. Here, slow disk memory and large computations make the process time consuming. Speeding up of such computations can be achieved by parallel external sort algorithms such as the well-known network of workstations (NOW)-sort [7], which runs on networked workstations. NOW-sort consists of two phases—the first phase sets approximate boundary values (pivots) and relocates all data to processors based on them. Subsequently, the second phase performs local sorting in parallel; this part of the sorting is fast since data exchange is no longer required after the first phase. However, load balance by even data distribution among the processors should be maintained.

In this research, a better sort algorithm than NOW-sort has been developed by avoiding floating-point arithmetic. Further, the degree of performance enhancement in previous methods such as parallel sort by regular sampling (PSRS) [8] and partitioned parallel radix sort [3] is not comparable to ours. In addition, the improvement by our algorithm is independent of processor architecture and computer hardware.

2. Comparison by integer translation

As mentioned previously, many sorting algorithms use comparisons to order values. For comparing two *real* numbers A and B, $A - B$ is computed first; A is greater (smaller) than B if the subtraction results in a positive (negative) value. In such comparisons, we must eliminate floating-point operations to avoid the complicated computations that are not present in integer arithmetic. Hence, we will treat the n-bit data as one homogeneous word, i.e., an n-bit integer. Suppose there are two real numbers (IEEE-754 standard) [9] as follows: $F_A = S_A \times M_A \times 2^{E_A}$ and $F_B = S_B \times M_B \times 2^{E_B}$ (S, M, and E are sign, mantissa, and exponent, respectively). Let I_A and I_B be the corresponding integers obtained by copying all the bits of the floating-point words, as shown in Fig. 1. Instead of computing $F_A - F_B$, we compute $I_A - I_B$ and decide the order. This decision is *correct* except when both are negative, in which case we simply reverse the order found by the integer comparison. The proof is briefly given in the appendix, and a comprehensive description can be found in [10].

* Corresponding author. Tel.: +82 232903232; fax: +82 29288909.
E-mail address: dkimku@korea.ac.kr (D. Kim).

0965-9978/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.advengsoft.2010.10.017

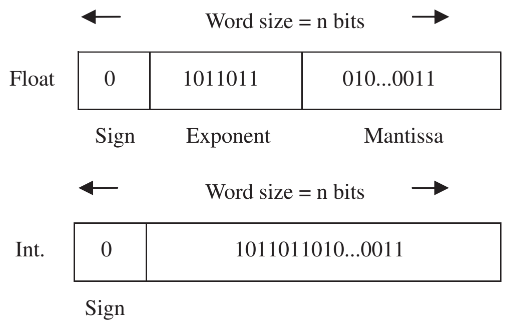

Fig. 1. Floating-point number and corresponding integer.

This comparison by *integer translation* saves sorting time and provides flexibility of computation. In addition, the performance is significantly boosted by the application of O(N)-time radix sort algorithm on real numbers, since the algorithm only accepts integers. It is well known that the best comparison-based sorting algorithms such as quicksort require at least O(N log N) time. The results of both integer-translation-only quicksort and integer-translation-and-radix-sort are compared with those of the previous generic sort in Table 1; Table 1 shows the execution times on various processor platforms for 64-MB floating-point keys [11]. It can be observed that enhancements with integer arithmetic on more advanced processors are less (probably due to pipelined architecture), but a two-to-three folds improvement can be achieved with radix sort on all processors.

3. External sort by integer translation

External sort of floating-point numbers usually involves large amounts of data in the disk and takes a long time to complete. Hence, it is extremely challenging to reduce the cost by applying the integer translation in the sort. In this study, we will apply the radix sort that has never been used previously for floating-point numbers. When data in the disk are loaded in the main memory, the data type is changed to integer. All the subsequent computations use the integer data until they are written back to the disk storage. Further, there will be little overhead of conversion. Technically, the conversion (to integers at the start, and the recovery to original values after the completion) can be performed using a string-copy function in the C language library as shown below:

(1) Integer translation
```
float src[SIZE];
int dest[SIZE];
memcpy(dest, src, sizeof(int) * SIZE);
```
(2) Recovery from integer
```
float recovered[SIZE];
int sorted[SIZE];
memcpy(recovered, sorted, sizeof(float) * SIZE);
```

Table 1
Sorting times for 64 MB floating-point keys are shown in seconds based on generic (floating-point) computation, integer arithmetic, and radix sorting operations.

Processor type	Quicksort with floating-point arithmetic	Quicksort with integer arithmetic	Radix sort
Intel-Core2 Duo 3.0 GHz	4.61	4.45	0.94
Intel-P4 2.0 GHz	10.8	9.27	3.81
AMD-Athlon 1.92 GHz	9.16	7.99	3.12
AMD-Athlon 1.83 GHz	9.91	8.43	3.38
Intel-P4 1.6 GHz	13.3	11.2	4.03

NOW-sort [7] is modified to adopt the integer conversions in our experiments. Keys are evenly stored in all the processors (nodes) initially, and the algorithm outputs the ordered results such that all the keys in a processor P_i are less than or equal to any key stored in P_j if $i < j$ ($i, j = 0, 1, \ldots, P - 1$). Of course, in each processor, the data becomes sorted internally.

Unlike the original NOW-sort algorithm that dealt only with the data having uniform distributions, in our sort, those with non-uniform distributions can also be sorted [11]. The algorithm first chooses pivots by sampling the keys; all the processors use common pivots to partition and relocate keys; and finally, the local sort is performed. All the computations are performed with integer arithmetic. The procedure is described as follows:

(S1) Each node simultaneously samples and sends a fixed number of keys (for example, Q keys) to the root node P_0. The root node then sorts PQ keys and chooses $P - 1$ pivots by selecting every Qth key. The pivots are broadcast to all processors.
(S2) Each node in parallel reads keys from the disk, partitions them into $P - 1$ buffers using the pivots, and sends them to the corresponding nodes. At the same time, it accepts incoming keys from others, moves them to buffers, and stores them into disks (see Fig. 2).
(S3) Since the keys are ordered among P processors by S2, the data in the local disk should be ordered; each node reloads and locally radix-sorts its keys, and writes back to its own disk.

4. Experimental results and discussion

The proposed algorithm was implemented on a cluster, i.e., a collection of networked workstations. The cluster consisted of 16 PCs with 2.53 GHz Intel Core 2 Duo E7200 CPUs interconnected by a Gigabit Ethernet switch. Each PC ran under Linux OS with 2 GB RAM and 320 GB hard disk. The algorithm was coded in C language using the message passing interface (MPI) communication library. The input keys were 32-bit single-precision floating-point numbers synthetically generated and having both *uniform* and *Gaussian* distributions.

Figs. 3 and 4 show comparisons of our sorting results with those of the generic method that uses floating-point arithmetic for both uniform and non-uniform data. The execution times with various input sizes and processor counts—denoted by marked *integer* and *floating-point*—are shown simultaneously. The sorting performance of the proposed algorithm is at least twice as good as that of the generic method for both Gaussian and uniform distributions. For only the internal sorting, the integer translation sort is approximately five times faster than the sort with the floating-point computation. Nevertheless, I/O times such as disk access time and interprocessor communication time remain unchanged, as shown in Fig. 5, which results in the maximum speedup of two. In order to obtain even load partitions, as many as $\sqrt{N}$ samples were used for selecting pivots, as described in [12]. The overhead of the sampling is negligible. Load imbalance due to uneven data distributions among the processors has a deviation of less than 1% from the perfect balance [11]. Finally, significant speedups were achieved in the experiments, as shown in Fig. 6, when the load (data size) per processor is not significantly low, since the network and I/O overheads have slight effects as compared to the overall sorting time.

5. Potential applications

The sorting algorithm can significantly save computing costs in fields wherein intensive data ordering is required; for example,

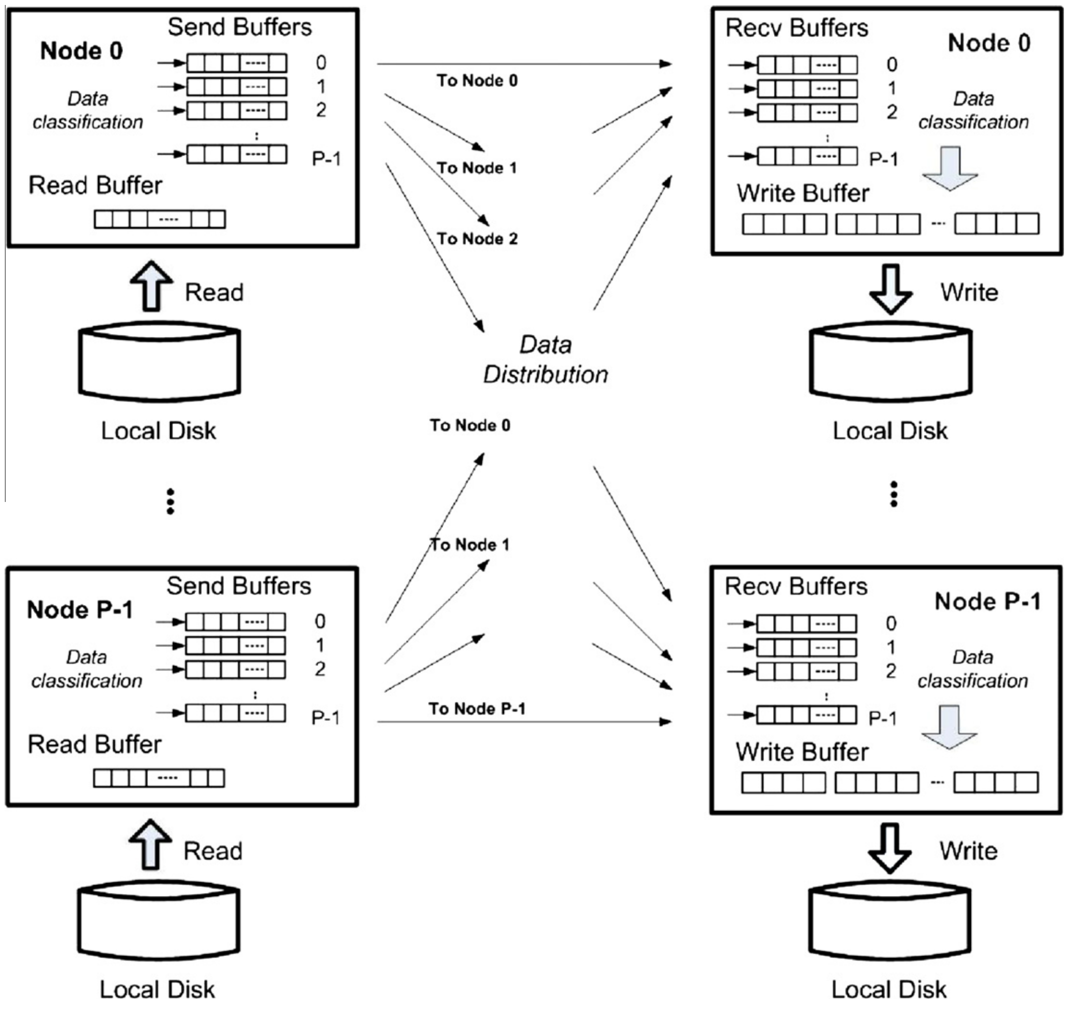

Fig. 2. Data flow among nodes (stage S2).

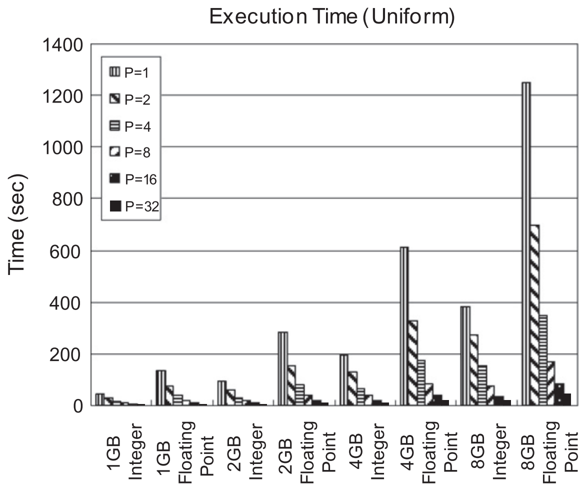

Fig. 3. Execution time of sorting with and without integer translation (*uniform* distribution).

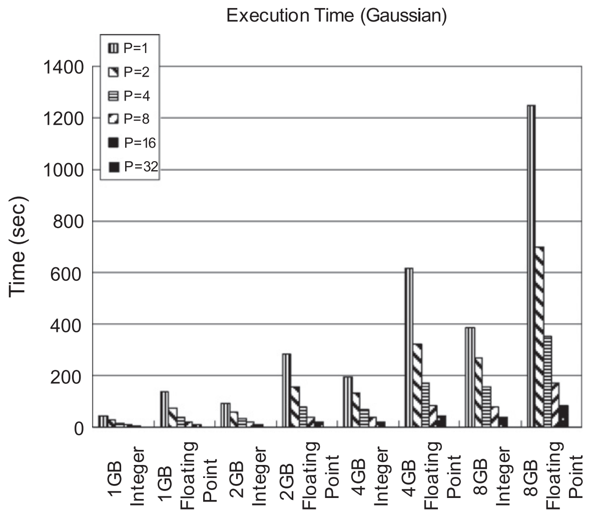

Fig. 4. Execution time of sorting with and without integer translation (*Gaussian* distribution).

they can be successfully employed in the following scientific and engineering applications:

(1) Internet search—*binary search*, widely used for fast searches, requires the data (keys) to be in order. World Wide Web data is growing at an enormous rate every day and needs to be updated and reordered frequently.
(2) Database search and data mining—data engineering involves enormous data processing for transactional databases. Queries that need sorting with respect to specified fields require considerable computation time.

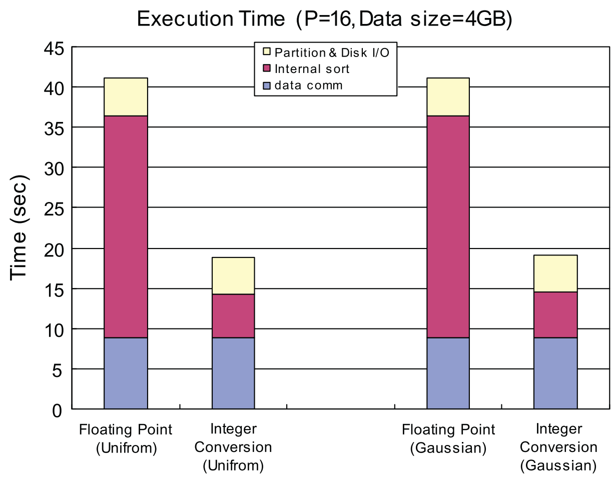

Fig. 5. Components of execution time in parallel sort.

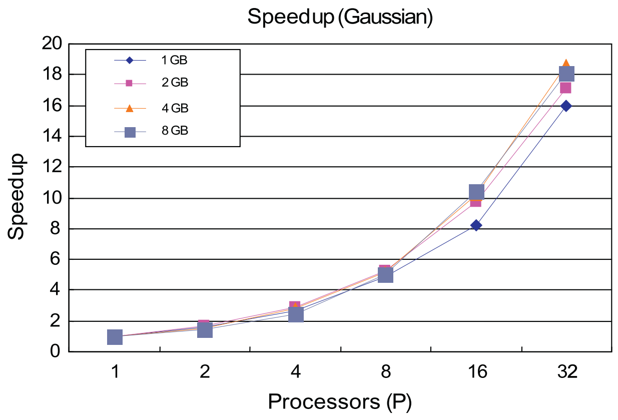

Fig. 6. Speedup of the integer translation sort.

(3) Graphics—occlusion detection and path finding in robotics can use the ordering of objects by location.
(4) Simulations—at each iteration, only those objects whose *strength* is within a given range have to be selected and used in the next iteration (strength can be the distance, force, field, etc.).

6. Summary

In this research, we have devised an enhanced external sort algorithm of real numbers by integer translation. We have conducted experiments using synthetic data on a 32-processor Linux cluster. Our results indicate that at least a twofold increase in speedup over the conventional method was achieved. Our sorting algorithm can be successfully employed in many applications such as data search, graphics, and engineering simulations.

Acknowledgements

This research was supported by KOSEF Grant (R01-2006-000-11167-0), and Korean University Grant.

Appendix A

Theorem 1. *The order of two floating-point numbers can be computed by the corresponding integers, where the integers comprise exact copies of the whole floating-point words.*

Proof. Suppose there are two floating-point numbers $F_A = S_A \times M_A \times 2^{E_A}$, and $F_B = S_B \times M_B \times 2^{E_B}$, where S_A, S_B, M_A, M_B and E_A, E_B are signs, mantissas, and exponents, respectively. Let I_A and I_B be the corresponding integers of F_A and F_B, respectively, found by string-copy operation of the floating-point words. There are two cases—the two numbers may be identical or not.

(i) If both are identical, so are their integers and the integer subtraction can give the correct order.
(ii) If both are not identical, without loss of generality, let us assume that F_A is greater than F_B i.e. $F_A > F_B$. In the ordering computation, there will be three cases with respect to signs of the two numbers.
   1. Both positive: If both numbers are positive, i.e., $S_A = S_B = 0$, then either $E_A > E_B$, or both exponents are identical and $M_A > M_B$. The higher part of the word of I_A is larger than the corresponding part of I_B. From Theorem 2 given below, the subtraction yields the result $I_A - I_B > 0$. Thus, the integer translation returns the correct order.
   2. Both negative: If F_A and F_B are negative and F_A is greater, either $E_A < E_B$, or both exponents are equal and $M_A < M_B$. Then the higher part of I_A is smaller than that of I_B. From Theorem 2, the integer computation gives the result $I_A - I_B < 0$. In this case, we simply reverse the order.
   3. Unequal sign: For the case with different signs, because $F_A > F_B$, I_A is positive and I_B is negative. Thus, the computation $I_A - I_B > 0$ returns the correct result.

Thus, the order of two floating-point numbers can be found by integer subtraction. ■

Let $A = (a_{n-1} \dots a_1 a_0)$ and $B = (b_{n-1} \dots b_1 b_0)$ be n-bit integers adopting 2's complement representation for negative numbers. In the proof below, the following relations will be used:

$$A = (a_{n-1} \dots a_1 a_0) = (-1)^a_{n-1} 2^{n-1} + a_{n-2} 2^{n-2} + \dots + a_1 2^1 + a_0,$$
$$B = (b_{n-1} \dots b_1 b_0) = (-1)^b_{n-1} 2^{n-1} + b_{n-2} 2^{n-2} + \dots + b_1 2^1 + b_0.$$

Theorem 2. *If A and B are two n-bit integers with the same sign, and the integer comprising only m leftmost bits of A is greater than that of B for m $\geqslant$ 2, A is greater than B.*

Proof. Denote $P_x(m) = (x_{n-1} \dots x_{n-m})$ the leftmost m bits of $X = (x_{n-1} \dots x_{n-m} x_{n-m-1} \dots x_1 x_0)$. The computation $A - B$ can be partitioned into two parts as follows:

$$\begin{aligned} A - B &= (a_{n-1} \dots a_{n-m} a_{n-m-1} \dots a_1 a_0) - (b_{n-1} \dots b_{n-m} b_{n-m-1} \dots b_1 b_0) \\ &= (P_A(m) a_{n-m-1} \dots a_1 a_0) - (P_B(m) b_{n-m-1} \dots b_1 b_0) \\ &= \{(P_A(m) 0 \dots 0) - (P_B(m) 0 \dots 0)\} + \{(a_{n-m-1} \dots a_1 a_0) \\ &\quad - (b_{n-m-1} \dots b_1 b_0)\} \\ &= (P_A(m) - P_B(m)) 2^{n-m} + \{(a_{n-m-1} \dots a_1 a_0) \\ &\quad - (b_{n-m-1} \dots b_1 b_0)\}. \end{aligned}$$

Since $P_A(m) > P_B(m)$, we have $P_A(m) - P_B(m) > 0$. Now, $P_A(m) - P_B(m) \geqslant 1$ because both are integers. Thus, $A - B \geqslant 2^{n-m} (a_{n-m-1} \dots a_1 a_0) - (b_{n-m-1} \dots b_1 b_0) \geqslant 2^{n-m} + (00 \dots 00) - (11 \dots 11) = 2^{n-m} - (2^{n-m} - 1) \geqslant 1$. It means that $A - b > 0$ therefore, A is greater than B. ■

References

[1] Knuth DE. The art of computer programming, sorting and searching. vol. 3. Addison Wesley Longman Publishing Co., Inc.; 1998.
[2] Jeon M, Kim D. Parallel merge sort with load balancing. Int J Parallel Program, vol. 31, No. 1. Kluwer Academic Publishers; 2003 (February).

[3] Lee S-J, Jeon M, Kim D, Sohn A. Partitioned parallel radix sort. J Parallel Distribut Comput, vol. 62. Academic Press; 2002. p. 656–68 (April).
[4] Rivera L, Zhang X, Chien A. HPVM minutesort. Sort Benchmark Home Page.
[5] Cerin C. An out-of-core sorting algorithm for clusters with processors at different speed. In: Proc 2002 parallel and distributed processing symp, April 15–18, Fort Lauderdale, FL, USA; 2002.
[6] Sintorn E, Assarsson U. Fast parallel GPU-sorting using a hybrid algorithm. J Parallel Distribut Comput, vol. 68, No. 10. Academic Press; 2008. p. 1381–88 (October).
[7] Arpaci-Dusseau AA, Arpaci-Dusseau RA, Culler DE, Hellerstein JM, Patterson DA. Searching for the sorting record: experiences in tuning NOW-Sort. In: Proc SIGMETRICS symp parallel and distributed tools; 1998. p. 124–33.
[8] Li X, Lu P, Schaeffer J, Shillington J, Wong PS, Shi H. On the versatility of parallel sorting by regular sampling. Parallel Comput, vol. 19. Elsevier; 1993. p. 1079–193.
[9] ANSI/IEEE Std. 754. An American national standard: IEEE standard for binary floating-point arithmetic; 1988.
[10] Dawson B. Comparing floating-point numbers. <http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm>.
[11] Kim C. Parallel external sort of floating-point data by integer conversion. Master thesis, Korea University, February 2009, also presented at WSEAS applied computing conference 2008, Istanbul, Turkey, May 27–29; 2008.
[12] Raman R. Random sampling techniques in parallel computation. In: Proc IPPS/SPDP workshops; 1998. p. 351–60.