# Parallel External Sort of Floating-Point Data by Integer Conversion

CHANGSOO KIM, SUNGROH YOON, and DONGSEUNG KIM
Department of Electrical Engineering
Korea University
Anamdong 5Ga, Seoul
REP. OF KOREA

*Abstract:* - This paper introduces a fast external sorting algorithm of floating point numbers with integer operations only, which shortens the computing time significantly. Conversion overhead to integer can be avoided if the floating point data are stored in the disk since integer conversion is made while they are read from the data file. Experimentally large-scale data stored in the disk are sorted in a cluster computer with various data distributions, where speedups over two fold or more are observed.

*Key-Words:* sort, floating- point arithmetic, NOW- sort, cluster computer, load balancing

## 1 Introduction

Sorting is a fundamental computation widely used in many applications such as data searching, job scheduling, database management, etc [1]. There are lots of high performance sorting algorithms in the literature [2-8]. Most sorting algorithms focus on integer data since the processing is easy and flexible. If we think of comparison based sorting algorithms, they can be equally applied to real (floating-point) numbers as well. However, if floating point data can be translated to integers and their relative order is preserved, the running time could be significantly reduced by those algorithms. To devise a detailed way of avoiding floating-point arithmetic is the subject of this research.

While *internal sort* refers to ordering of as many data as fit in main memory, *external sort* orders large-scale data often stored in the disk [1]. Hence, external sort demands multiple iterations of data retrieval from the disk, ordering computation in main memory, and write back to the disk. Thus, it usually takes longer time due to slow performance of disk memory [9]. *NOW-sort* [10,11] is a well known external sorting algorithm on parallel computers like networked workstations. The algorithm runs in two steps; the first phase roughly sets boundary values (i.e. splitters/ pivots) and relocates all data to processors based on the pivots. Then, the second phase performs local sorting computation without interprocessor communication. The sort is fast since data exchange is no longer needed after the first phase. However, if there is unbalance in the data distribution among processors, the execution time is lengthened by the most heavily loaded processor.

The following sections introduce the idea on sorting by integer translation, explain parallel external sorting by NOW-sort, and report experimental results with discussion.

## 2 Integer Translation

Many sorting algorithms use *comparison* to order values. If two real numbers $A$ and $B$ are compared, $A$-$B$ is computed by floating point arithmetic, thus, if the subtraction results in a positive value, for example, we decide that $A$ is greater than $B$. Low level format of a floating point number consists of different fields such as exponent and mantissa as shown in Fig.1. However, we may attempt to interpret the whole word of a floating point number (we assume it uses IEEE-754 standard [12]) as an *integer*, which means we ignore the meaning of different fields and regard them in one homogeneous word. Then, the comparison (subtraction) can be done using integer arithmetic, since the interpretation maintains the same relative order of the two floating point numbers because the upper part of the floating point format consists of exponent, and the lower part mantissa. Using this property, sorting can be done *without* the floating point arithmetic. This *integer translation sort* introduces savings of computing time and flexibility of computation, thus, fast sorting can be achieved.

As an illustration, consider two real values $A$=3.0 = $1.5\times 2^1$ and $B$=5.0 = $1.25\times 2^2$, written in decimal notation for simplicity. Their 32 bit representation follows a floating point number format such as IEEE-754 standard format. In case of $A$, the sign bit is zero, exponent should be 127+1= 128 with bias 127, and mantissa will be $1000\ldots00_B$ (in 23 bits) since the decimal value of 0.5 is 0.1 in binary representation. (We refer the subscripts of B and H to binary and hexadecimal representation, respectively.) Similarly,

0, 129, and $0100\ldots0_B$ are the sign bit, exponent, and the mantissa of *B*, respectively. Their integer translation gives $40400000_H$ and $40A00000_H$, and the subtraction of *A*-*B* with integer arithmetic produces a negative value, thus, we can tell that *A* is smaller than *B*.

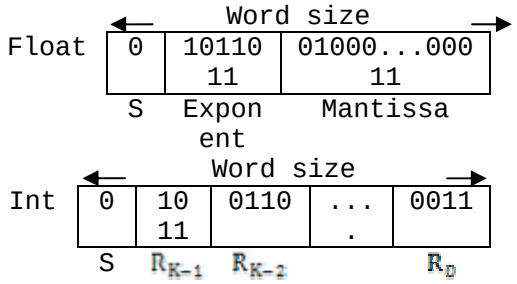


Fig.1. Floating-point and integer formats

Besides the speedup of integer arithmetic, an integer can utilize radix sorting method that delivers O(*N*)-time performance[1], which outperforms the general comparison based sorting demanding O(*N* log *N*) time.

## 3 External Sort by Integer Translation

External sorting is quite suitable for the integer translation sort since there will be no extra cost of conversion from floating point to integer. In external sort, every data in the disk should be loaded in the main memory for processing. When the data are loaded in the main memory, floating-point binary data are converted to integers with the same word size. From the point on, the integers represent the corresponding floating point numbers. All processing after the load use the integer data, and when the data are written to disk, the original floating point numbers are recorded.

NOW-sort is modified for our experiments. We add sampling for determining the pivots, and develop the code to have minimal disk I/O. It produces sorted result with data redistribution such that all keys in a processor $P_i$ is less than or equal to any key stored in $P_j$ if $i<j$ ($i, j$ =0,1, …, $P$-1). The sorting algorithm first performs splitter selection by sampling, then, loads and distributes keys, and finally does local sort and output, as described below:

(S1) Each node simultaneously reads and sends to the root node a fixed number of keys (for example, *Q* keys). Then, root node sorts the *PQ* keys, and chooses *P*-1 pivots/splitters by selecting every $Q^{th}$ keys. The pivots are broadcast to all other processors.

(S2) Each node in parallel reads keys from disk, partitions them into *P*-1 bins by the splitters, and sends them to the corresponding nodes. At the same time it accepts from others, moves to buffers, and stores into disk the arriving keys.

(S3) Once the distribution is complete, each node reloads and locally sorts its keys in the disk, and writes back to disk.

## 4 Experimental Results and Discussion

The algorithms have been implemented on a PC cluster. It consists of 8 PCs with 1.83 GHz AMD Athlon XP 2500+ CPUs interconnected by a Myrinet switch. Each PC runs under Linux with 1GB RAM and 80GB hard disk. Code is written in C language with the MPI communication library. Input keys are 32-bit single-precision floating point numbers synthetically generated with two distribution functions (*uniform* and *gauss*). NOW-sort is applied as a framework of the external sorting, where the integer conversion is performed in the early phase, then the remaining part of the computing uses the integers.

Figs. 2 and 3 show the comparison of our sorting to the generic method, marked *integer* and *floating point*, respectively. Execution times with various input sizes and processor counts are shown simultaneously, where left bars correspond to the integer translation, and right generic. The overall speed of integer translation external sort is at least two times faster than the generic method. If only the internal sorting computation time is focused, the integer translation sort is about four times faster than that of floating point, as listed in Table 1. It occupies a dominant portion of the overall time as found in Fig.4, where the times spent by individual functions such as internal sorting computation, data communication, disk read/write are shown. Load imbalance due to uneven data distribution among processors is very small (below 1% deviated from perfect balance), as observed in Fig.5. We have extracted as many as $\sqrt{N}$ samples for selecting pivots for partitioning as in [13]. The overhead of the sampling turns out to be negligible, thus it does not hurt the performance.

Figs.6 and 7 plot the speedup and corresponding efficiency measure. They show that as more data are given, the efficiency drops since the communication overhead grows due to the limited bandwidth of Myrinet.

**Table 1. Comparison of computing time of internal sort only with P=8**
(U=Uniform, G=Gaussian)

| Data size / Algorithms | 1GB | | 2GB | | 4GB | |
|---|---|---|---|---|---|---|
| | U | G | U | G | U | G |
| Integer translated (a) | 5.5 | 5.7 | 11.3 | 11.3 | 22.4 | 22.1 |
| Floating point (b) | 24.4 | 25.5 | 45.8 | 48.9 | 99.9 | 98.6 |
| b/a | **4.4** | **4.5** | **4.1** | **4.3** | **4.5** | **4.5** |

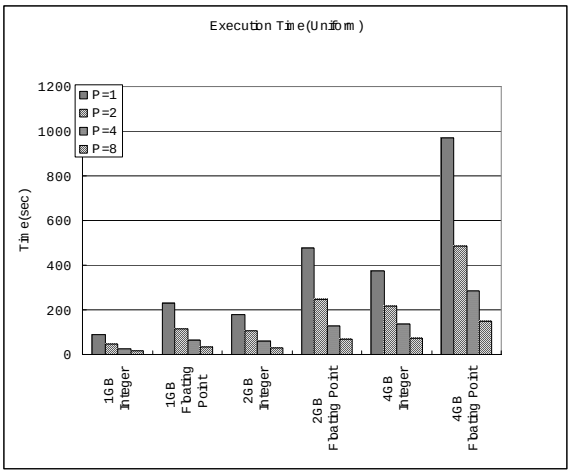


Fig. 2 Execution time of sorts with and without integer translation (*uniform* distr.)

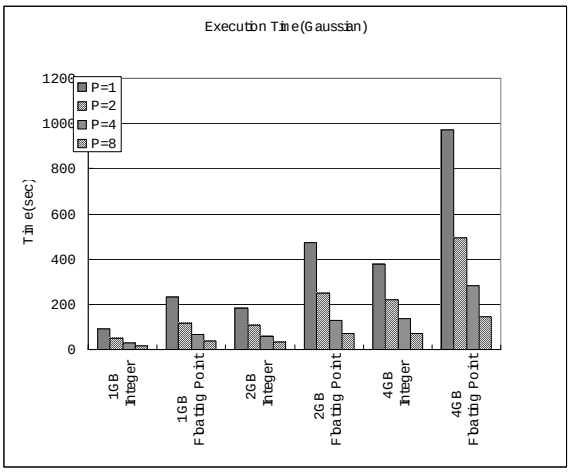


Fig. 3. Execution time of sorts with and without integer translation (*gaussian* distr.)

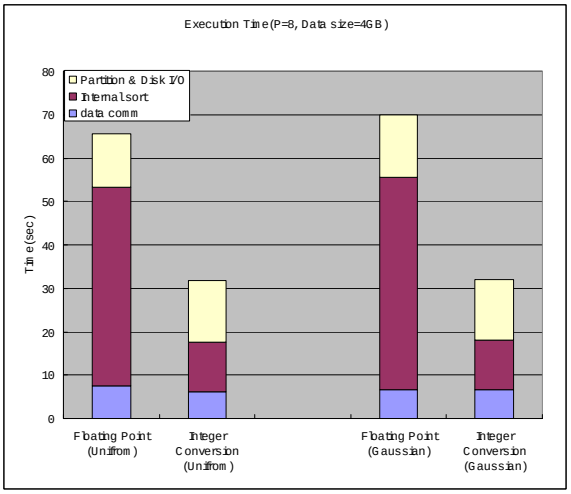


Fig. 4. Components of execution time in the parallel sort

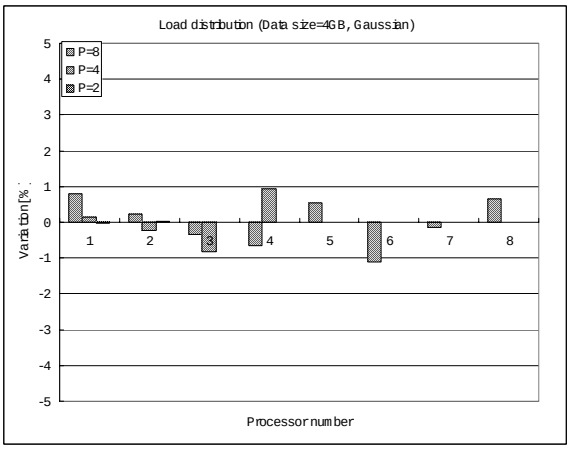


Fig. 5. Degree of load imbalance among processors shown in percentage.

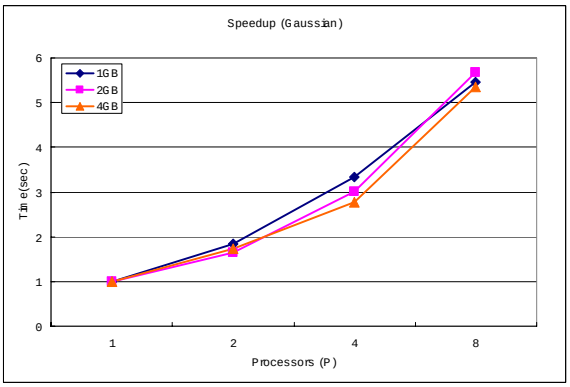


Fig. 6. Speedup of integer translation sort

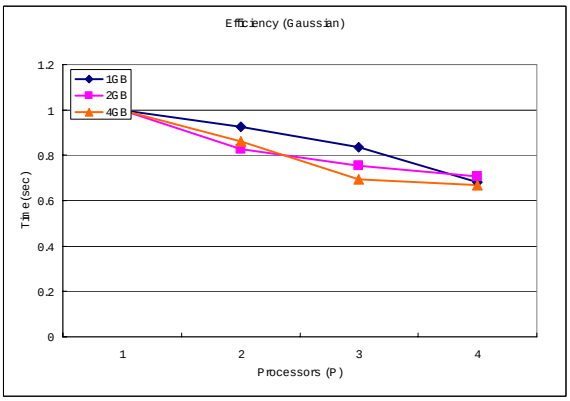


Fig. 7. Efficiency of integer translation sort

## 5 Summary

In this research we devise an enhanced sorting algorithm of real numbers that uses integer format instead of the original floating-point one. To verify the enhancement the new method is applied to an external sorting application where the data conversion does not require additional cost. Experimental results on an 8-node cluster show over two-fold speedup. Besides the computing speedup, we are studying the integer translation for fast data partitioning as often used in radix sort.

**Acknowledgement**: This paper was supported in by KOSEF Grant (R01-2006-000-11167-0), and in part by Korean Government (MOEHRD) (KRF-2005-041-D00670).

**References:**

[1] D.E. Knuth, *The art of computer programming, volume 3: (2nd ed.) sorting and searching*, Addison Wesley Longman Publishing Co., Inc., 1998.

[2] K. Batcher, Sorting networks and their applications, *Proc. AFIPS Spring Joint Computer Conference 32*, Reston, VA, 1968, pp. 307-314.

[3] M. Jeon and D. Kim, Parallel merge sort with load balancing, *Int'l Journal of Parallel Programming*, Kluwer Academic Publishers, Vol. 31, No.1, Feb. 2003, pp. 21-33.

[4] S-J Lee, M. Jeon, D. Kim, and A. Sohn, Partitioned parallel radix sort, *Journal of Parallel and Distributed Computing*, Academic Press, Vol. 62, April 2002, pp. 656-668.

[5] F. Popovici, J. Bent, B. Forney, A. A. Dusseau, R. A. Dusseau, Datamation 2001: A Sorting Odyssey, In *Sort Benchmark Home Page*.

[6] L. Rivera, X. Zhang, A. Chien, HPVM Minutesort, *Sort Benchmark Home Page*.

[7] J. Wyllie, SPsort: How to sort a terabyte quickly, *Technical Report*, IBM Almaden Lab., Feb. 1999, http://www.almaden.ibm.com/cs/gpfs-spsort.html.

[8] C. Cerin, An out-of-core sorting algorithm for clusters with processors at different speed, *Proc. 2002 Parallel and Distributed Processing Symp.*, April 15-18, Fort Lauderdale, FL, USA, 2002.

[9] J. Porter, Disk trend 1998 report, http://www.disktrend.com/pdf/portrpkg.pdf

[10] 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, May 1997.

[11] 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, pp. 124-133.

[12] American National Standard Institute, An American National Standard: IEEE Standard for Binary Floating-Point Arithmetic, 1988. ANSI/IEEE Standard No. 754.

[13] R. Raman, Random sampling techniques in parallel computation, *Proc. IPPS/SPDP Workshops*, 1998, pp. 351-360.

[14] JS Vitter, External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209-271, June 2001