2009 International Conference on Parallel and Distributed Computing, Applications and Technologies

An efficient parallel sorting compatible with the standard qsort

Duhu Man, Yasuaki Ito and Koji Nakano
Department of Information Engineering,
Hiroshima University
1-4-1 Kagamiyama, Higashihiroshima, Hiroshima, 739–8527 Japan
{*manduhu, yasuaki, nakano*}*@cs.hiroshima-u.ac.jp*

***Abstract*—The main contribution of this paper is to present an efficient parallel sorting "psort" compatible with the standard qsort. Our parallel sorting "psort" is implemented such that its interface is compatible with "qsort" in C Standard Library. Therefore, any application program that uses standard "qsort" can be accelerated by simply replacing "qsort" call by our "psort". Also, "psort" uses standard "qsort" as a subroutine for local sequential sorting. So, if the performance of "qsort" is improved by anyone in the community, then that of our "psort" is also automatically improved.**

To evaluate the performance of our "psort", we have implemented our parallel sorting in a Linux server with two Intel quad-core processors (i.e. eight processor cores). The experimental results show that our "psort" is approximately 6 times faster than standard "qsort" using 8 processors. Since the speed up factor cannot be more than 8 if we use 8 cores, our algorithm is close to optimal. Also, as far as we know, no previously published parallel implementations achieve a speed up factor less than 4 using 8 cores.

***Keywords*-Parallel algorithm; Sorting; Multicore processor; C standard library**

I. Introduction

Recently, software performance has improved rapidly, primarily driven by the growth in processing power. However, we can no longer follow Moore's law for performance improvements. Fundamental physical limitations such as the size of the transistor and power constraints have now required a radical change in commodity microprocessor architecture to multicore designs. Multicore processors which have two or more processing cores are now ubiquitous in home computing. Moreover, we will be able to use much more processing cores in the near future.

It is no doubt that sorting is one of the most important tasks in computer engineering, such as database operations, image processing, statistical methodology and so on. Hence, many sequential sorting algorithms have been studied in the past [1].

To speedup the sorting, multiprocessors are employed for parallel sorting. Several parallel sorting algorithms such as parallel merge sort [2], bitonic sort [3], [4], randomized parallel sorting [5], column sort [6], and parallel radix sort [7], [8] have been devised. Lately, a parallel sorting algorithm using GPUs (Graphic Processing Unit) has been shown [9].

The main contribution of this paper is to present an efficient parallel sorting compatible with "*qsort*" in C Standard Library. Therefore, any application program that uses standard "qsort" can be accelerated by simply replacing "qsort" call by our "psort". More specifically, suppose that an array of integers is sorted using "qsort" in an application program. What we need to do for accelerating the sorting is to replace library call "qsort" by our "psort" simply as follows:

```
qsort(data, num_data, sizeof(int), comp);
                    ↓
psort(data, num_data, sizeof(int), comp);
```

Also, our "psort" uses standard "qsort" as a subroutine for local sequential sorting. So, if the performance of "qsort" is improved by anyone in the community, then that of our "posrt" is also automatically improved. Further, since standard "qsort" is maintained by the community, we can minimize the bugs and security holes of our "psort" compared with the case that we use an original sequential local sorting developed by ourselves.

In our previous paper, we have shown a parallel sorting algorithm for multicore processors [10]. This parallel sorting algorithm implemented on the multicore processors. The experimental results have shown that for random 32-bit unsigned integer numbers, this parallel sorting algorithm is approximately 6 times faster than sequential sorting using 8 processors. For general purpose, however, it should be able to sort any kind of objects such as floating point numbers and strings. The advantage of our approach is to replace sequential sort with our efficient parallel sorting with less work and without skills of parallel programming.

The key idea of our parallel sorting is to select samples appropriately, and use samples in the samples as pivots to partition the input keys into groups. We have implemented and evaluated our algorithm in a Linux server with two Intel quad-core processors. The results have shown that our parallel sorting algorithm is 6 times faster than sequential sorting. Since the speed up factor cannot be more than 8 if we use 8 cores, our algorithm is close to optimal. From the experimental results, we discuss how many samples are appropriate for efficient multicore sorting.

The paper is organized as follows. In Section II, we

978-0-7695-3914-0/09 $26.00 © 2009 IEEE
DOI 10.1109/PDCAT.2009.55

present an idea of our parallel sorting algorithm for multicore processors. Section III shows an implementation of parallel sorting for multicore processors. Section IV shows an idea of our multicore sorting compatible with qsort. In Section V, we reports experimental results performed on multicore processors. We conclude in the last section.

II. Parallel Sorting by Sampling

The main purpose of this section is to show an idea of our sorting algorithm for multicore processors.

Let $A = \langle a_0, a_1, \ldots, a_{n-1}\rangle$ be a sequence of keys stored in a memory to be sorted. The outline of our sorting algorithm for p processors is as follows:

- **Step 1** Select p threshold values $d_0, d_1, \ldots, d_{p-1}$ such that d_0 is the minimum key in A.
- **Step 2** Partition A into p groups $A_0, A_1, \ldots, A_{p-1}$ using threshold values such that $A_i = \{x \in A \mid d_i \leq x < d_{i+1}\}$, where $d_p = +\infty$.
- **Step 3** Sort keys in each group A_i using one processor per group independently.

To complete the sorting, every A_i must have almost the same number of keys. We will show that, we can select threshold values such that the numbers of keys in all A_i are well balanced.

Let $A = \langle a_0, a_1, \ldots, a_{n-1}\rangle$ be a sequence of keys stored in a memory. For simplicity, we assume that every a_i is distinct. We partition A into p blocks B_i $(0 \leq i \leq p-1)$ of the same size such that $B_i = \langle a_{i\cdot\frac{n}{p}}, a_{i\cdot\frac{n}{p}+1}, \ldots, a_{(i+1)\cdot\frac{n}{p}-1}\rangle$. Suppose that each block B_i $(0 \leq i \leq p-1)$ is sorted independently, and $B_i = \langle b_{i,0}, b_{i,1}, \ldots, b_{i,\frac{n}{p}-1}\rangle$ denotes the sorted sequence thus obtained. In other words, $b_{i,0} < b_{i,1} < \cdots < b_{i,\frac{n}{p}-1}$ holds. For an arbitrary integer $k > 0$, we further partition each sorted block B_i $(0 \leq i \leq p-1)$ into pk sub-blocks $B_{i,0}, B_{i,1}, \ldots, B_{i,pk-1}$ such that $B_{i,j} = \langle b_{i,j\cdot\frac{n}{p^2k}}, b_{i,j\cdot\frac{n}{p^2k}+1}, \ldots, b_{i,(j+1)\cdot\frac{n}{p^2k}-1}\rangle$. Clearly, each $B_{i,j}$ has $\frac{n}{p^2k}$ keys. Let C_i denote the sequence of keys obtained by picking the minimum key from each of the sub-blocks $B_{i,0}, B_{i,1}, \ldots, B_{i,pk-1}$. In other words, $C_i = \langle b_{i,0\cdot\frac{n}{p^2k}}, b_{i,1\cdot\frac{n}{p^2k}}, \ldots, b_{i,(pk-1)\cdot\frac{n}{p^2k}}\rangle$. Let C denote the combined sequence of $C_0, C_1, \ldots, C_{p-1}$. Since each C_i has pk keys, C has p^2k keys. Let $\langle c_0, c_1, \ldots, c_{p^2k-1}\rangle$ denote the sorted sequence of C. In other words, $c_0 < c_1 < \cdots < c_{p^2k-1}$ holds. We pick every pk keys from sorted sequence C. Let $D = \langle d_0, d_1, \ldots, d_{p-1}\rangle$ be the sequence thus obtained. In other words, $d_i = c_{i\cdot pk}$ $(0 \leq i \leq p-1)$ holds. We use keys in D as threshold values to partition elements in A. Let A_i $(0 \leq i \leq p-1)$ denote a set of values such that $A_i = \{x \in A \mid d_i \leq x < d_{i+1}\}$, where $d_p = +\infty$. By sorting keys in each A_i independently, we can obtain the sorted sequence of A.

Quite surprising, we can prove that the number of keys in A_i is well balanced as follows. Let $D_i = \{x \in C \mid d_i \leq x < d_{i+1}\} = \{c_{i\cdot pk}, c_{i\cdot pk+1}, \ldots, c_{(i+1)\cdot pk-1}\}$. Clearly, each D_i has pk keys. Further, let $D_{i,j} = B_i \cap D_j$ and $A_{i,j} = B_i \cap A_j$. For example, in Figure 1, $|D_{0,1}| = 0$, $|D_{1,1}| = 4$, $|D_{2,1}| = 2$, and $|D_{3,1}| = 2$. From the figure, it is easy to see that if $|D_{i,j}| = 0$ then $|A_{i,j}| \leq \frac{n}{p^2k} - 1$ holds. For example, in the figure, since $|D_{0,1}| = 0$, we can guarantee that $|A_{0,1}| \leq \frac{n}{p^2k} - 1$. Similarly, if $|D_{i,j}| = 1$ then $|A_{i,j}| \leq 2\frac{n}{p^2k} - 1$ holds. In general, we have

$$|A_{i,j}| \leq (|D_{i,j}| + 1)\frac{n}{p^2k} - 1.$$

Thus, we can compute the upper bound of the number of keys in A_j as follows:

$$\begin{aligned} |A_j| &= \sum_{i=0}^{p-1} |A_{i,j}| \\ &\leq \sum_{i=0}^{p-1}((|D_{i,j}| + 1)\frac{n}{p^2k} - 1) \\ &= (pk + p)\frac{n}{p^2k} - p \\ &= \frac{n}{p} + \frac{n}{pk} - p. \end{aligned}$$

Thus, we have the following important lemma.

Lemma 1: Each A_j $(0 \leq j \leq p-1)$ has no more than $\frac{n}{p} + \frac{n}{pk} - p$ keys.

Note that, if A is equally partitioned into p groups, each of them has $\frac{n}{p}$ keys. It follows that, A_j may have at most $\frac{n}{pk} - p$ additional keys, and the number of additional keys decreases as k increases.

III. Parallel Sorting Algorithm

This section shows an implementation of parallel sorting for multicore processors. Let $P(i)$ $(0 \leq i \leq p-1)$ denote a processor i. We assume that the input n keys are stored in array A, and the parallel sorting algorithm stores the sorted n keys in array R. The details of the parallel sorting algorithm are spelled out as follows:

- **Step 1.1** Partition A into p groups B_0, B_1, …, B_{p-1} and sort each group B_i $(0 \leq i \leq p-1)$ using $P(i)$.
- **Step 1.2** We use an array of size p^2k to store C. Each $P(i)$ picks every $\frac{n}{p^2k}$ keys in B_i and copy them to the array for C in an obvious way.
- **Step 1.3** $P(0)$ sorts keys in C. Since keys in each C_0, C_1, …, C_{p-1} are sorted, this can be done by merge sort. Pick every pk keys in C.

It should be clear that, the picked keys are threshold values d_0, d_1, …, d_{p-1}.

- **Step 2.1** Let $s_{i,j}$ $(0 \leq i, j \leq p-1)$ be the minimum index of a key in B_i satisfying $b_{i,s_{i,j}} \geq d_j$. Clearly, $A_{i,j} = \{b_{i,s_{i,j}}, b_{i,s_{i,j}+1}, \ldots, b_{i,s_{i,j+1}-1}\}$ holds, where $s_{i,p} = \frac{n}{p}$. Each $P(i)$ $(0 \leq i \leq p-1)$ computes the values of $s_{i,0}, s_{i,1}, \ldots, s_{i,p-1}$ using the obvious binary search.

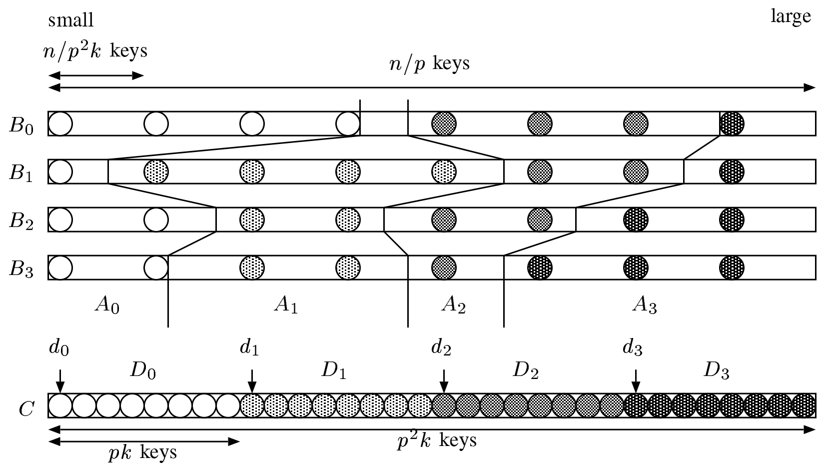


Figure 1. Illustrating the sorting algorithm using threshold values

- **Step 2.2** Clearly, $A_{i,j}$ has $s_{i,j+1} - s_{i,j}$ keys. Each $P(j)$ $(0 \leq j \leq p-1)$ computes $|A_{0,j}|+|A_{1,j}|+\cdots+|A_{p-1,j}|$, which is equal to $|A_j|$. After that, $P(0)$ computes the prefix sums $\alpha_j = |A_0| + |A_1| + \cdots + |A_j|$ for each j $(0 \leq j \leq n-1)$.
- **Step 2.3** Let R_j be a subset of array R such that R_j consists of $|A_j|$ elements from α_j-th element of R. Each $P(j)$ $(0 \leq j \leq p-1)$ copies keys in A_j to R_j.

Finally, we sort each R_j as follows:

- **Step 3** Each $P(j)$ $(0 \leq j \leq p-1)$ sort sub-array R_j independently. Note that, R_j consists of $A_{0,j}$, $A_{1,j}$, ..., $A_{p-1,j}$. Also, each $A_{i,j}$ is sorted. Hence, the sorting of R_j can be done by merging $A_{0,j}$, $A_{1,j}$, ..., $A_{p-1,j}$.

Let us evaluate the computing time necessary to perform each step. In Step 1.1, each processor performs the sorting of $\frac{n}{p}$ keys. This can be done in $O(\frac{n}{p}\log\frac{n}{p})$ time using the heap sort, and in expected $O(\frac{n}{p}\log\frac{n}{p})$ time using the quick sort. In Step 1.2, each processor performs the copy of pk keys, and thus, it takes $O(pk)$ time. In Step 1.3, $P(0)$ performs the merging of p sorted sequences of pk keys each, which can be done in $O(p^2k\log p)$ time. Therefore, Step 1 can be done in $O(\frac{n}{p}\log\frac{n}{p} + p^2k\log p)$ time.

In Step 2.1, each processor performs p binary searches on $\frac{n}{p}$ keys. Hence, Step 2.1 can be done in $O(p\log\frac{n}{p})$ time. In Step 2.2, the sum and the prefix sums of p integers are computed, which takes $O(p)$ time. In Step 2.3, $P(j)$ performs the copy operation of $|A_j|$ keys, which takes $O(|A_j|)$ time. From Lemma 1, we can guarantee that Step 2.3 can be done in $O(\frac{n}{p} + \frac{n}{pk} - p)$ time. Therefore Step 2 can be done in $O(\frac{n}{p} + p\log\frac{n}{p})$ time.

In Step 3, each $P(j)$ performs merge sort of p sorted sequences of totally $|A_j|$ keys, which can be done in $O(|A_j|\log p)$ time. From Lemma 1, we can guarantee that the computing time is no more than $O(\frac{n\log p}{p} + \frac{n\log p}{pk})$ time.

Finally we have

Theorem 2: Sorting of n keys can be done in $O(\frac{n}{p}(\log\frac{n}{p} + \log p) + p^2k\log p + p\log\frac{n}{p})$ time using p processors.

Note that, if $p \ll n$ and $k \ll n$, then the computing time is $O(\frac{n\log n}{p})$. Since the sequential sorting takes $O(n\log n)$ time, our algorithm achieves the speed up of factor p using p processors. Therefore, our parallel sorting algorithm is optimal.

IV. Multicore sorting compatible with qsort

The main purpose of this section is to show an idea of our multicore sorting compatible with "qsort". Standard qsort function is an implementation of the quick sort algorithm provided in C Standard Library. The contents of the array are sorted in ascending order according to a user-supplied comparison function. The interface of "qsort" is shown, as follows.

```
void qsort(void *base, size_t nmemb, size_t size, int(*compar)(const void *, const void *));
```

The interface of "qsort" consists of four arguments:

- ***base** : a pointer to the first entry in array to be sorted.
- **nmemb** : the number of elements in the array to be sorted.
- **size** : the size, which is in bytes, of each entry in the array.
- ***compar()** : the name of the comparison function which is called with two arguments that point to the elements being compared.

Since "qsort" operates on void pointers, it can sort arrays of any size, containing any kind of object and using any kind of comparison predicate. If the objects are not the same in size, pointers have to be used. To satisfy the above property of "qsort", we have developed our parallel sorting such that its interface is same as that of "qsort".

Our method has implemented in C language with OpenMP 2.0 (Open Multi-Processing). The OpenMP is an application programming interface that supports shared memory environment [11]. It consists of a set of compiler directives and library routines. By using OpenMP, it is relatively easy to create parallel applications in FORTRAN, C, and C++. However, there is considerable overhead due to parallel processing when the number of elements in a sorted array is small.

Therefore, to obtain the optimal parameters t and k, we have implemented and evaluated the performance of our parallel sorting in a Linux server with two quad-core processors, that is, we have used eight processor cores, where t is the number of used processor cores and k is a parameter described in Sections II and III. Figure 2 shows the computing time of our implementation when random 32-bit unsigned integers are sorted for general purpose. The evaluation has been carried out for different values of k, n and p (recall that n represents the number of the input data and p represents the number of using processing cores). Note that in Step 1, each processor does local sort using "qsort", and for $p = 1$, that is single process, the implementation performs "qsort" for the whole input data. Therefore, the computing time for $p = 1$ is independent of k.

The figure shows that for $n \leq 10,000$, the execution time of the single process, that is $p = 1$, is short because in comparison with the total execution time, there is considerable overhead due to parallel processing. On the other hand, for $n > 10,000$, the execution time of the single process is quite long. For $n \geq 1,000,000$, when k is not large, the execution time is independents of k. Based on the results, Table I shows the parameter t and k, which seem to be optimal. For example, when the number of input data is 200,000 and the number of available cores is 4, from the table, the optimal parameters t and k are 4 and 2, respectively.

V. Experimental Results

We have implemented and evaluated the performance of our parallel sorting algorithm in a Linux server (CentOS 5.1) with two quad-core processors (Intel Xeon X5355 2.66GHz [12]), that is, we have used eight processor cores. The program is compiled by gcc 4.1.2 with -O2 option.

For comparing with other implementation of parallel sorting, we have used C Multithread Library (beta release 0.1) [13]. The library features two interface-compatible sorting functions for "qsort" and "mergesort" from C Standard Library. These two parallel sorting functions are implemented from original function in C Standard Library using POSIX Treads.

Table II shows the performance of our implementation when random 32-bit unsigned integers, 64-bit unsigned integers and 64-bit double precision floating-point numbers are sorted. In the table, "qsort", "psort", "qsort_mt" and "mergesort_mt" denote qsort in C Standard Library, our proposed multicore sorting, parallel qsort in C Multithread Library and parallel mergesort in C Multithread Library, respectively. The performance evaluation has been carried out for different values of n (recall that n represents the number of the input data).

The table shows that for $n \leq 10,000$, our method is almost the same as "qsort". On the other hand, computing time of qsort_mt and mergesort_mt is longer. For $n \geq 100,000$, our method sorted faster than the others. For $n \geq 10,000,000$, our method sorted at least 6 times faster than qsort in C Standard Library and approximately 2 times faster than qsort and mergesort in C Multithread Library. Since the speed up factor cannot be more than p if we use p cores, our algorithm is close to optimal.

VI. Concluding remarks

We have presented an efficient multicore sorting compatible with qsort. Our multicore sorting is implemented such that its interface is compatible with qsort in C Standard Library and can sort arrays of any size, containing any kind of object and using any kind of comparison predicate. By replacing calls to qsort with our multicore sorting, the sequential sorting is replaced with our parallel sorting easily.

Also, we have implemented and evaluated our algorithm in a Linux server with two Intel quad-core processors (Intel Xeon X5355 2.66GHz). The experimental results have shown that our multicore sorting is 6 times faster than original qsort. Since the speed up factor cannot be more than 8 if we use 8 cores, our algorithm is close to optimal.

References

[1] D. E. Knuth, *The Art of Computer Programming. Vol.3: Sorting and Searching*, 1973.

[2] M. Jeon and D. Kim, *Parallel Merge Sort with Load Balancing*, International Journal of Parallel Programming, pp. 21–33, Vol.31, No.1, February, 2003.

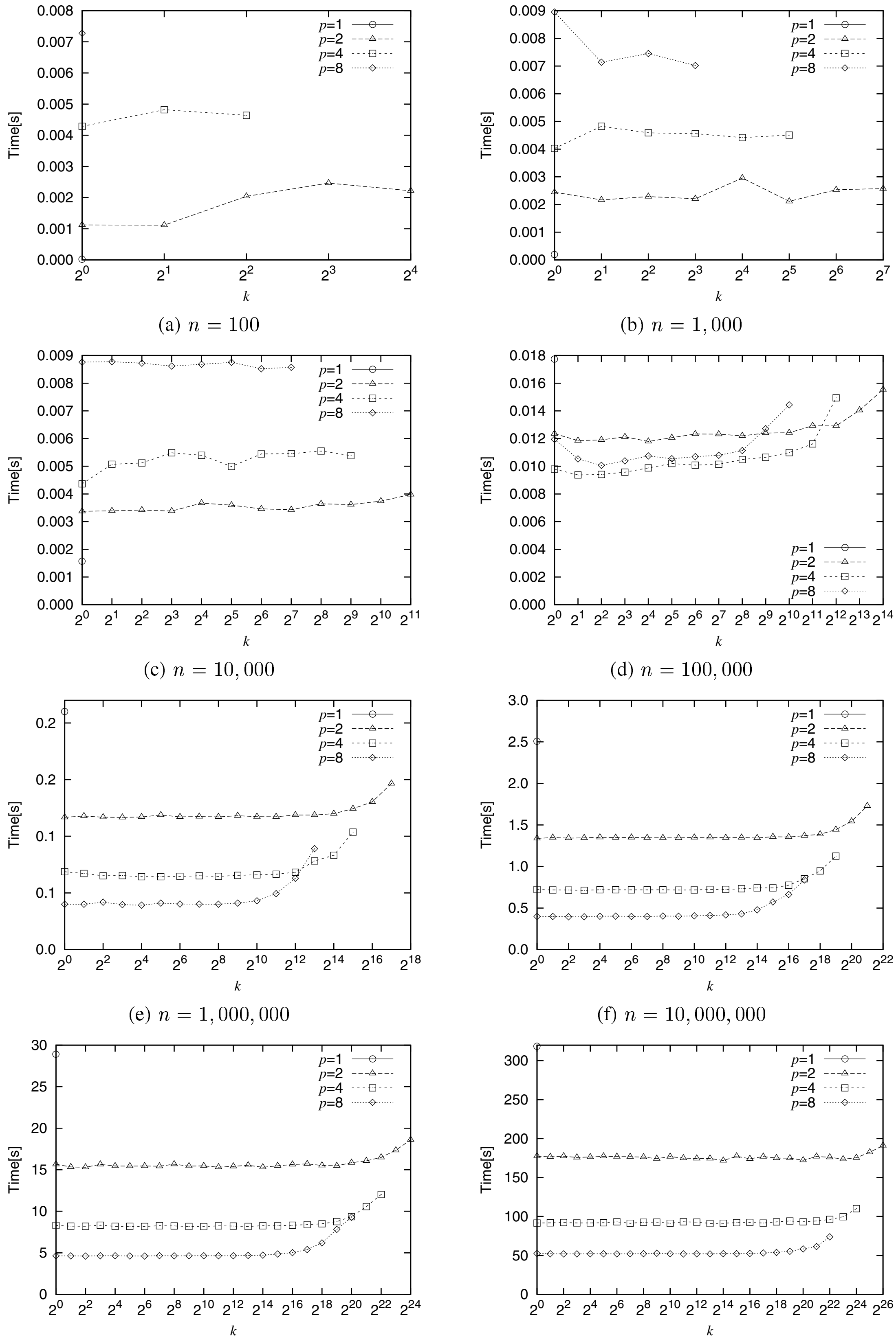
[3] K. Batcher, *Sorting Networks and Their Applications*, Proceedings of the AFIPS Spring Joint Computer Conference 32, Reston, VA, pp. 307–314, 1968.

[4] M. F. Ionescu and K. E. Schauser, *Optimizing Parallel Bitonic Sort*, in Proceedings of the 11th International Symposium on Parallel Processing, pp. 303–309, Geneva, Switzerland, April, 1997.

[5] D. R. Helman, D. A. Bader and J. JáJá, *A randomized parallel sorting algorithm with an experimental study*, Journal of Parallel and Distributed Computing, Vol. 52, pp. 1–23, July, 1998.

[6] A. C. Dusseau, D. E. Culler, K. E. Schauser, and R. P. Martin, *Fast Parallel Sorting under Log P: Experience with the CM-5*, IEEE Transactions on Parallel and Distributed Systems, Vol. 7, pp. 791–805, August, 1996.

[7] A. Sohn and Y. Kodama, *Load Balanced Parallel Radix Sort*, in Proceedings of the 12th ACM International Conference on Supercomputing, July, 1998.



(a) $n = 100$ (b) $n = 1,000$

(c) $n = 10,000$ (d) $n = 100,000$

(e) $n = 1,000,000$ (f) $n = 10,000,000$

(g) $n = 100,000,000$ (h) $n = 1,000,000,000$

Figure 2. Computing time for our parallel sort

Table I
OPTIMAL PARAMETERS

The number of available cores	1		2		4		8	
	t	k	t	k	t	k	t	k
$n < 50,000$	1	–	1	–	1	–	1	–
$50,000 \leq n < 500,000$	1	–	2	16	4	2	4	2
$500,000 \leq n$	1	–	2	1	4	1	8	1

Table II
PERFORMANCE OF PARALLEL SORTING

(a) 32-bit unsigned integers

n	qsort	psort		qsort_mt		mergesort_mt	
	Time[s]	Time[s]	Speed up	Time[s]	Speed up	Time[s]	Speed up
100	0.0000148	0.0000145	1.02	0.0004480	0.03	0.0005950	0.02
1,000	0.0002306	0.0002286	1.01	0.0006281	0.37	0.0007240	0.32
10,000	0.0015593	0.0015594	1.00	0.0019760	0.79	0.0027144	0.57
100,000	0.0177477	0.0096842	1.83	0.0126510	1.40	0.0137161	1.29
1,000,000	0.2103429	0.0426486	4.93	0.1062432	1.98	0.0864644	2.43
10,000,000	2.4999987	0.4013662	6.23	0.9547787	2.62	0.8945034	2.79
100,000,000	28.8102803	4.6296428	6.22	9.1322992	3.15	9.2465315	3.12
1,000,000,000	318.0739780	51.8421750	6.14	91.5192440	3.48	100.5022250	3.16

(b) 64-bit unsigned integers

n	qsort	psort		qsort_mt		mergesort_mt	
	Time[s]	Time[s]	Speed up	Time[s]	Speed up	Time[s]	Speed up
100	0.0000153	0.0000151	1.01	0.0004584	0.03	0.0005992	0.03
1,000	0.0002227	0.0002213	1.01	0.0005805	0.38	0.0007268	0.31
10,000	0.0015831	0.0016246	0.97	0.0018301	0.87	0.0026526	0.60
100,000	0.0185624	0.0117281	1.58	0.0122585	1.51	0.0147567	1.26
1,000,000	0.2306025	0.0472184	4.88	0.1022577	2.26	0.0944008	2.44
10,000,000	2.7766366	0.5811376	4.78	0.9689922	2.87	0.9953102	2.79
100,000,000	32.4576996	7.0135219	4.63	9.6505263	3.36	10.9065100	2.98

(c) 64-bit double precision floating-point numbers

n	qsort	psort		qsort_mt		mergesort_mt	
	Time[s]	Time[s]	Speed up	Time[s]	Speed up	Time[s]	Speed up
100	0.0000149	0.0000154	0.97	0.0004594	0.03	0.0006109	0.02
1,000	0.0002401	0.0002305	1.04	0.0006145	0.39	0.0007513	0.32
10,000	0.0016692	0.0016998	0.98	0.0018899	0.88	0.0030235	0.55
100,000	0.0197145	0.0117318	1.68	0.0125129	1.58	0.0155153	1.27
1,000,000	0.2453897	0.0496379	4.94	0.1040864	2.36	0.1021421	2.40
10,000,000	2.9484040	0.5935436	4.97	0.9516576	3.10	1.0748308	2.74
100,000,000	34.4574052	7.0960471	4.86	9.5402221	3.61	11.5991955	2.97

[8] S. J. Lee, M. Jeon, D. Kim, and A. Sohn, *Partitioned Parallel Radix Sort*, Journal of Parallel and Distributed Computing, Vol. 62, pp. 656–668, 2002.

[9] E. Sintorn and U. Assarsson, *Fast parallel GPU-sorting using a hybrid algorithm* Journal of Parallel and Distributed Computing, Vol. 68, pp. 1381–1388, October, 2008.

[10] K. Nishihata, D. Man, Y. Ito and K. Nakano. *Parallel sampling sorting on the multicore procesors*, in Proceedings of the International Conference on Applications and Principles of Information Science, pp. 233–236, January, 2009.

[11] OpenMP Application Program Interface, http://www.openmp.org

[12] Intel Corporation, *Intel Xeon Processor 5000 Sequence*, http://www.intel.com/products/processor/xeon5000/

[13] Diomidis D. Spinellis and Markus Weissmann, C Multithread Library – libmt, http://libmt.sourceforge.net/