International Journal of Foundations of Computer Science
Vol. 22, No. 5 (2011) 1057–1071
© World Scientific Publishing Company
DOI: 10.1142/S0129054111008568

AN EFFICIENT PARALLEL SORTING COMPATIBLE WITH THE STANDARD QSORT

DUHU MAN
*Department of Information Engineering, Hiroshima University,
1-4-1 Kagamiyama, Higashihiroshima, Hiroshima 739-8527, Japan
manduhu@cs.hiroshima-u.ac.jp*

YASUAKI ITO
*Department of Information Engineering, Hiroshima University,
1-4-1 Kagamiyama, Higashihiroshima, Hiroshima 739-8527, Japan
yasuaki@cs.hiroshima-u.ac.jp*

KOJI NAKANO
*Department of Information Engineering, Hiroshima University,
1-4-1 Kagamiyama, Higashihiroshima, Hiroshima 739-8527, Japan
nakano@cs.hiroshima-u.ac.jp*

Received 2 March 2010
Accepted 20 June 2010
Communicated by Akihiro Fujiwara

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 open source 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 four Intel hexad-core processors (i.e. twenty four processor cores). The experimental results show that our "psort" is approximately 11 times faster than standard "qsort" using 24 processors.

Keywords: Sorting; multicore processor; C standard library; loading balance.

1. 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 and parallel sorting algorithms have been studied in the past [1, 2].

To speedup the sorting, multiprocessors are employed for parallel sorting. In [4, 5], bitonic sort has been presented. Bitonic sort is one of the earliest parallel sorting algorithms and is executed using a comparator-based sorting network. Parallel radix sort [6, 8, 9], parallel Merge sort [3, 21], parallel quick sort [10], and column sort [7] have been devised. These parallel sorting algorithms are based on existing sequential ones. Helman and JáJá have presented generalized parallel sorting for SMPs (Symmetric Multiple Processor) [11]. They estimated memory access and showed the complexity of the algorithm using a computational model. However, since it is a theoretical model, the model was not practical for the modern computer architecture.

Many investigations show that modern shared-memory computer architectures with multiple cores can perform high performance on sorting. The first parallel sort algorithm for shared memory MIMD (multiple-instruction-multiple-data) multiprocessors that has a near linear speedup is exhibited in [12]. An efficient multithreaded implementation of MergeSort on current multi-core architecture is shown in [13]. The loading balance is an important issue on the parallel computing. Therefore a load-balanced scheme for parallelizing quicksort using the hyperthreading technology is proposed in [14].

Lately, emerging GPUs (Graphic Processing Unit) and Cell processors have been used to achieve an efficient acceleration of sorting works. Several implementations of bitonic sort on GPU have been proposed in [15, 16]. In [17], a GPU-based parallel sorting algorithm that is a hybrid method of bucket sort merge sort has been devised. On the other hand, parallel bitonic sort has also been implemented with SIMD (Single Instruction Multiple Data) operations on the Cell processor [18]. Another Cell-processor-based parallel sorting algorithm, AA-Sort, has been exhibited in [19]. AA-Sort is implemented by combining combsort and mergesort.

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 "psort" 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 [20]. This parallel sorting algorithm implemented on the multicore processors. The experimental results have shown that for random 64-bit unsigned integer numbers, this parallel sorting algorithm is approximately 11 times faster than sequential sorting using 24 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 four Intel hexad-core processors. The results have shown that our parallel sorting algorithm is 11 times faster than sequential sorting. From the experimental results, we discuss how many samples are appropriate for efficient multicore sorting.

The paper is organized as follows. In Section 2, we present an idea of our parallel sorting algorithm for multicore processors. Section 3 shows an implementation of parallel sorting for multicore processors. Section 4 shows an improved parallel sorting algorithm. In Section 5, we reports experimental results performed on multicore processors. We conclude in the last section.

2. Sorting by Sampling

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

Let $A = \langle a_0, a_1, \dots, a_{n-1} \rangle$ 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 $d_0, d_1, \dots, d_{p-1}$ such that d_0 is the minimum key in A.

Step 2 Partition A into p groups $A_0, A_1, \dots, A_{p-1}$ 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.

Let $A = \langle a_0, a_1, \dots, a_{n-1} \rangle$ be a sequence of keys stored in a memory. We first assume that every a_i is distinct. Later, we consider the case that some of the keys are the same. We partition A into p blocks B_i $(0 \leq i \leq p-1)$ of the same size such that $B_i = \langle a_{i \cdot \frac{n}{p}}, a_{i \cdot \frac{n}{p}+1}, \dots, a_{(i+1) \cdot \frac{n}{p}-1} \rangle$. Suppose that each block B_i $(0 \leq i \leq p-1)$ is sorted independently, and $B_i = \langle b_{i,0}, b_{i,1}, \dots, b_{i,\frac{n}{p}-1} \rangle$ 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 keys 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.

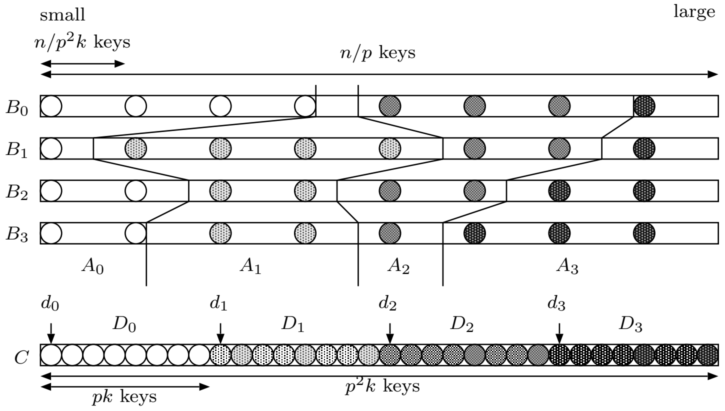


Fig. 1. Illustrating the sorting algorithm using threshold values.

Quite surprising, we can prove that the number of keys in A_i is well balanced if all the keys are distinct 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 $A_{i,j} = B_i \cap A_j$. Here, $D_{i,j}$ denotes a set of keys (we can call them samples) which are larger than or equal to d_j and smaller than d_{j+1}. Those keys also exist in block B_i. Differently, $A_{i,j}$ denotes a collection of input values which are larger than or equal to d_j and smaller than d_{j+1}, they also exist in block B_i. In other words, $D_{i,j}$ denotes a set of samples, and $A_{i,j}$ denotes a set of input values.

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:

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

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 if all the input keys are distinct.*

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.

Let us consider the case that all the keys may not be distinct. For example, suppose that the input is binary, that is, $n/2$ keys are 0 and the remaining $n/2$ keys are 1. Clearly, k threshold values are also either 0 or 1. Thus, one of the groups has all 0-keys and another group has all 1-keys. All the other groups are empty. Hence, Lemma 1 does not hold. To partition the input keys into equal-sized groups even if all the keys are not distinct, we use lexicographical order comparison to compare two keys. We define lexicographical order $<_l$ between two keys a_i and a_j in A as follows:

$$a_i <_l a_j \text{ if } a_i < a_j \text{ or } (a_i = a_j \text{ and } i < j).$$

It should be clear that for any two keys a_i and a_j, exactly one of $a_i <_l a_j$ and $a_j <_l a_i$ holds. We use the relation $<_l$ instead of $<$ when we partition the input keys into groups. Then, since all the keys are distinct in terms of the relation $<_l$, we have,

Lemma 2. *Each A_j $(0 \leq j \leq p-1)$ has no more than $\frac{n}{p} + \frac{n}{pk} - p$ keys even if input keys are not distinct.*

3. 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. We use the relation $<_l$ to sort 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. We use the relation $<_l$ to for the binary search.

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 $|A_j|$ keys from α_j-th key 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 2, 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 2, 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 3. *Sorting of n keys can be done in $O(\frac{n}{p}(\log \frac{n}{p} + \log p) + p^2 k \log p + p \log \frac{n}{p})$ 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.

4. 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 keys 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 keys 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 the 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 [22]. 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 keys 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 four hexad-core processors, that is, we have used twenty four processor cores, where t is the number of used processor cores and k is a parameter described in Sections 2 and 3. 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 keys 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 keys. 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 $n > 10,0000$, 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 1 shows the parameter t and k, which seem to be optimal. For example, when the number of input keys is 200,000 and the number of available cores is 4, from the table, the optimal parameters t and k are 4 and 8, respectively.

Table 1. Optimal parameters.

The number of available cores	1		2		4		8		16		24	
	t	k	t	k	t	k	t	k	t	k	t	k
$n < 50,000$	1	–	1	–	1	–	1	–	1	–	1	–
$50,000 \leq n < 500,000$	1	–	2	32	4	8	4	8	4	8	4	8
$500,000 \leq n$	1	–	2	1	4	1	8	1	16	1	24	1

5. Experimental Results

We have implemented and evaluated the performance of our parallel sorting algorithm in a Linux server (CentOS 5.4) with four hexad-core processors (Intel Xeon X7460 2.66GHz [23]), that is, we have used twenty four processor cores. Each multicore processor contains its own three-level caches. The capacity of each level cache is 64KB, 3MB and 6MB, respectively. The size of the main memory is 128GB. Figure 3 shows the architecture of our experimental system. The program is compiled by gcc 4.1.2 with -O2 option.

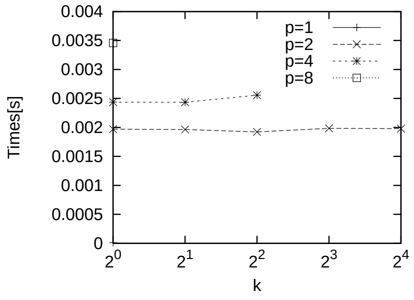
5.1. *Performance evaluation*

We evaluate the two versions of our parallel sorting "psort1" and "psort2" as follows:

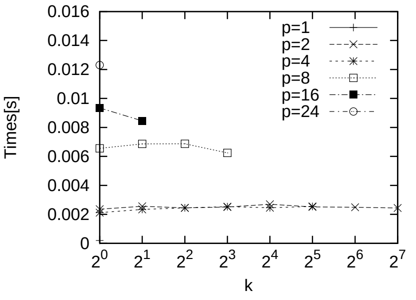
psort1 We use regular relation "$<$" in Steps 1.3 and 2.1. Thus, if the input keys are the same, they may not be partitioned into equal-sized groups.

psort2 We use relation "$<_l$" in Steps 1.3 and 2.1. The input keys are partitioned almost equal-size, but the comparison of two keys takes more time.

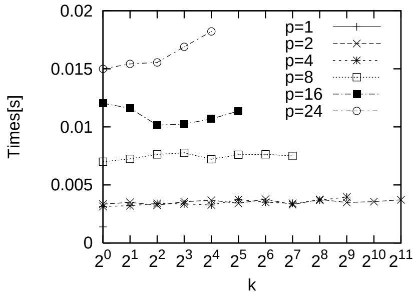
Figure 4 shows the number of keys assigned to each processor in Step 2 and Step 3 for 10^8 keys with values either 0 or 1. The number of keys assigned to each processor is almost the same for "psort2", while in "psort1" only two processors $P(0)$



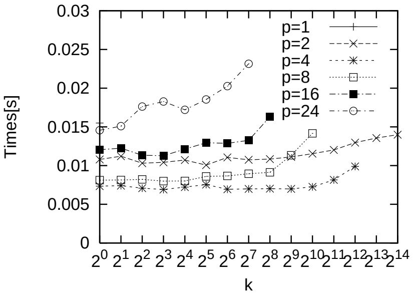
(a) $n = 100$



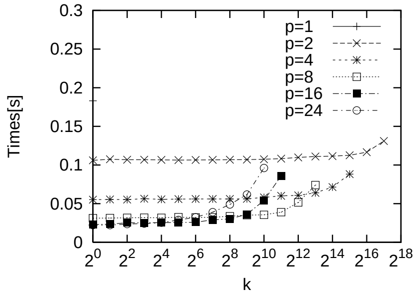
(b) $n = 1,000$



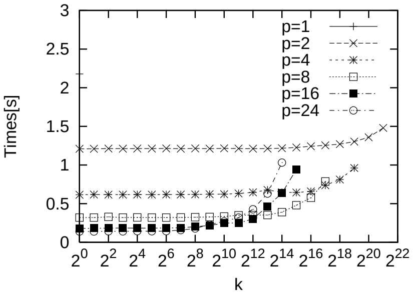
(c) $n = 10,000$



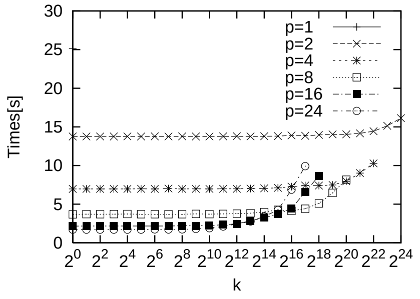
(d) $n = 100,000$



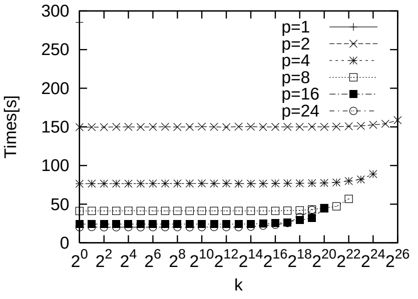
(e) $n = 1,000,000$



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



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



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

Fig. 2. Computing time for our parallel sort.

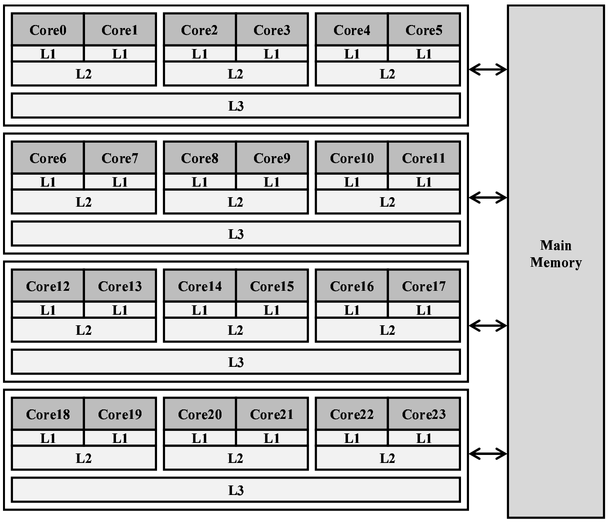


Fig. 3. Architecture of experimental system.

and $P(4)$ have keys and the other processor have no key. Table 2 shows sorting time of our "psort1" and "psort2". The performance evaluation has been carried out for a value of n and different values of p. Recall that n represents the number of the input keys and p is the number of using processors. The experimental results show that, when the input keys are 64-bit random numbers, the two psort algorithms can sort the input keys at the same efficiency. However, if the input keys are 1-bit random numbers, then the "psort2" can sort the input keys more efficient than the "psort1". Because the "psort2" can achieve a good loading balance even if some of the input keys are the same and it can save time significantly in Step 3.

For comparing with other implementation of parallel sorting, we have used C Multithread Library (beta release 0.1) [24]. The library features two interface-compatible sorting functions for "qsort" and "mergesort" from C Standard Library. Here, we call them as "qsort_mt" and "mergesort_mt". These two parallel sorting functions are implemented from original function in C Standard Library using POSIX Threads. In implementation of "qsort_mt", a pivot is selected and the input sequence is partitioned into two blocks using one processor such that all keys in the first block are smaller than the threshold, and the other keys are larger than the pivot. After that, two processors are used one for each block. And each block is partitioned in the same way. The same procedure is recursively executed until the number of blocks is equal the number of processors. Finally, each processor sorts the block sequentially. It is clear that the loading balance of "qsort_mt" is affected by the selection of pivot keys. However, in "qsort_mt", pivots are selected randomly, therefore "qsort_mt" cannot achieve a good loading balance in each executing step. On the other hand, "qsort_mt" also cannot utilize all the available processors in each

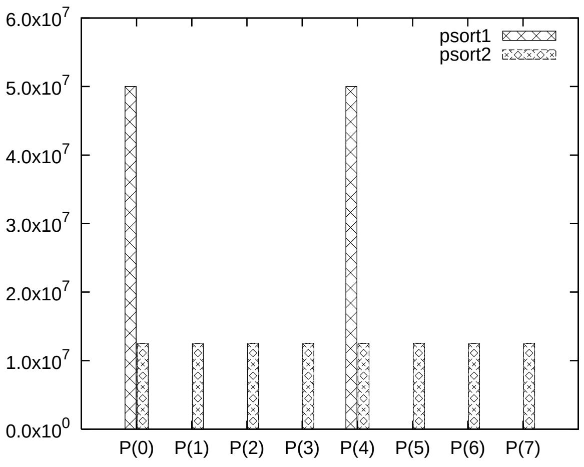


Fig. 4. The number of keys assigned to each processor in Step 2 and Step 3.

Table 2. Sorting time of psort1 and psort2 ($n = 1,000,000,000$).

(a) All keys are random 64-bit unsigned integers from 0 to $2^{64} - 1$

	psort1				psort2			
p	Step1[s]	Step2[ms]	Step3[s]	Total[s]	Step1[s]	Step2[ms]	Step3[s]	Total[s]
2	159.654	0.038	14.867	174.521	159.769	0.035	14.677	174.445
4	78.803	0.038	9.724	88.527	78.493	0.034	9.815	88.308
8	46.281	0.062	6.902	53.184	46.382	0.062	6.857	53.238
16	33.469	0.185	5.623	39.092	33.733	0.168	5.766	39.499
24	32.060	0.395	4.953	37.013	32.174	0.385	4.957	37.131

(b) All keys are random 64-bit unsigned integers taking values either 0 or 1

	psort1				psort2			
p	Step1[s]	Step2[ms]	Step3[s]	Total[s]	Step1[s]	Step2[ms]	Step3[s]	Total[s]
2	80.220	0.026	9.237	89.456	80.144	0.021	9.700	89.844
4	40.151	0.028	9.199	49.350	40.330	0.030	5.665	45.995
8	22.147	0.034	10.266	32.413	21.862	0.036	3.481	25.342
16	13.357	0.048	11.565	24.921	13.366	0.052	2.466	15.831
24	11.699	0.089	12.312	24.011	11.562	0.108	2.801	14.362

parallel step. For implementing "mergesort_mt", the input sequence is partitioned into p blocks. Each processor is arranged to a block and sorts it independently. After that a pair of two blocks is merged using a processor. A pairwise-merging is repeated until all blocks are merged into one. It is clear that, in each merging step, the number of used processors is reduced by half. So the "mergesort_mt" can not utilize all the available processors in each parallel step. However, our implementation can achieve a good loading balance for both uniform input data and non-uniform input data. Moreover, our method can utilize all the available processors in each parallel step. Therefore our implementation is faster than other two methods significantly.

Table 3 shows the performance of our implementation for random 64-bit unsigned integers. In the table, "qsort", "psort2","qsort_mt" and "mergesort_mt" denote qsort in C Standard Library, proposed multicore sorting methods, 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.

The table shows that for $n \leq 10,000$, our method is almost same as "qsort". In Section 4, we showed that for a small number of input keys, the sequential sorting using single processor is faster than parallel sorting using multicore processors due to miscellaneous overhead. For example, some constant overhead of time c of parallel overhead such as invoking processors is always needed in parallel computation. If n is so small that the sequential sorting of n keys can be done in less than c time, parallel sorting cannot be faster than the sequential sorting. Therefore, for $n \leq 10,000$, our method has directly used standard qsort (in C Standard Library) to perform the whole sorting work using optimal parameters shown in Table 1. Consequently, for $n \leq 10,000$, our method is almost same as "qsort". On the other hand, computing time of qsort_mt and mergesort_mt is longer. When the input keys are random 64-bit unsigned integers and $n \geq 10,000,000$, our method sorted at most 11 times faster than qsort in C Standard Library and approximately 3 times faster than qsort and mergesort in C Multithread Library. When the input keys are either 0 or 1, our method also sorted faster than qsort at most 13 times. Since the speed up factor cannot be more than p if we use p cores, our algorithm is efficient.

5.2. *Performance scalability analysis*

As shown in Figure 3, our system has one main memory and every processor core shares it. Therefore, two or more processor cores cannot access the memory simultaneously. If the number of simultaneous memory access is large, the memory access time cannot be ignored.

Let us evaluate the computing time with the memory access necessary to perform each step. Given the memory access, if $p \ll n$ and $k \ll n$, in Step 1, the number of memory access is $O(n \log n)$. From Section 3, Step 1 can be done in $O(\frac{n \log n}{p})$. Since memory access cannot be done simultaneously, the total execution time of Step 1 is no more than $C_1 \cdot n \log n + D_1 \cdot \frac{n \log n}{p}$, where C_1 and D_1 are constant values. Similarly, Step 2 and Step 3 takes no more than $C_2 \cdot (n + p^2 \log n) + D_2 \cdot (\frac{n}{p} + p \log n)$ and $C_3 \cdot (n \log p + \frac{n \log p}{k}) + D_3 \cdot (\frac{n \log p}{p} + \frac{n \log p}{pk})$, respectively. Also, C_2, C_3, D_2 and D_3 are constant values. Note that the first terms $C_1 \cdot n \log n$, $C_2 \cdot (n + p^2 \log n)$, and $C_3 \cdot (n \log p + \frac{n \log p}{k})$ correspond to the time for the shared memory access and the second terms $D_1 \cdot \frac{n \log n}{p}$, $D_2 \cdot (\frac{n}{p} + p \log n)$, and $D_3 \cdot (\frac{n \log p}{p} + \frac{n \log p}{pk})$ correspond to the local computing time. Since $C_i \ll D_i (i = 1, 2, 3)$, the second terms are dominant for practically small p and k.

Table 3. Performance of parallel sorting.

(a) All keys are random 64-bit unsigned integers from 0 to $2^{64} - 1$

	qsort	psort2		qsort_mt		mergesort_mt	
n	Time[s]	Time[s]	Speed up	Time[s]	Speed up	Time[s]	Speed up
100	0.000011	0.000011	1.00	0.001149	0.01	0.001009	0.01
1,000	0.000174	0.000172	1.01	0.001438	0.12	0.001534	0.11
10,000	0.001374	0.001381	0.99	0.003874	0.35	0.003181	0.43
100,000	0.016122	0.007734	2.08	0.024758	0.65	0.016122	1.00
1,000,000	0.194148	0.028720	6.89	0.270884	0.72	0.077644	2.50
10,000,000	2.416300	0.212756	11.34	2.196236	1.10	0.644443	3.75
100,000,000	28.525679	2.712458	10.47	22.66808	1.25	6.415592	4.44

(b) All keys are random 64-bit unsigned integers that are either 0 or 1

	qsort	psort2		qsort_mt		mergesort_mt	
n	Time[s]	Time[s]	Speed up	Time[s]	Speed up	Time[s]	Speed up
100	0.000008	0.000009	0.90	0.000848	0.01	0.001015	0.01
1,000	0.000152	0.000160	0.95	0.000899	0.17	0.001475	0.10
10,000	0.000917	0.000919	0.99	0.000955	0.96	0.002760	0.33
100,000	0.009676	0.004746	2.03	0.002463	3.92	0.009834	0.98
1,000,000	0.108484	0.020650	5.25	0.023172	4.68	0.055484	1.95
10,000,000	1.265774	0.099879	12.67	0.249060	5.08	0.342431	3.69
100,000,000	14.707196	1.117982	13.15	2.456557	5.98	3.300517	4.45

Table 4. Performance of our implementation with different number of cores ($n = 100$ M).

Number of Cores	1	2	4	8	16	24
Time	28.525 [s]	15.791 [s]	8.121 [s]	4.762 [s]	3.221 [s]	2.712 [s]
Speed-up	1.00	1.80	3.51	6.00	8.86	10.47

Table 4 shows the scaling performance for 100M random 64-bit unsigned integers with different number of cores. From the table, when the number of cores is small, the speed up factor is close to the number of cores. However, when the number of cores is large, the speed up factor is approximately half of the number of cores. The result shows that memory access time cannot be ignored for large p.

6. 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 four Intel hexad-core processors (Intel Xeon X7460 2.66GHz). The experimental results have shown that our multicore sorting is 11 times faster than original qsort.

Since the speed up factor cannot be more than 24 if we use 24 cores, our algorithm is efficient.

References

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

[2] S. G. Akl, *Parallel Sorting Algorithms*, Academic Press Inc., 1990.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] P. Tsigas and Y. Zhang. *A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000*, in Proceedings of Euromicro Conference on Parallel, Distributed, and Network-Based Processing, pp. 372–381, 2003.

[11] D. R. Helman and J. JáJá, *Designing Practical Efficient Algorithms for Symmetric Multiprocessors*, In Algorithm Engineering and Experimentation (ALENEX'99), Vol. 1619 of Lecture Notes in Computer Science, pp. 37–56, Baltimore, MD, January 1999. Springer-Verlag.

[12] R. Francis and I. Mathieson, *A Benchmark Parallel Sort for Shared Memory Multiprocessors*, IEEE Transactions on Computers, Vol. 37, pp. 1619-1626, 1988.

[13] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, P. Dubey, *Efficient Implementation of Sorting on Multi-Core SIMD CPU Architecture*, in Proceedings of the 34th International Conference on Very Large Data Bases, Vol.1, No.2, August 2008.

[14] R. Parikh, *Accelerating QuickSort on the Intel Pentium 4 Processor with Hyper-Threading Technology*, http://softwarecommunity.intel.com/articles/eng/2422.htm, 2008.

[15] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. *GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 325-336, 2006.

[16] A. Gress and G. Zachmann. *GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures*, in Proceedings of 20th International Parallel and Distributed Processing Symposium, April 2006.

[17] 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.

[18] B. Gedik, R. Bordawekar, and P. S. Philip, *CellSort: high performance sorting on the cell processor*, in Proceedings of the 33rd international conference on Very large data bases, pp. 1286–1297, September 2007.

[19] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani. *AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors*, in Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques , pp. 189-198, 2007.

[20] K. Nishihata, D. Man, Y. Ito and K. Nakano. *Parallel sampling sorting on the multicore processors*, in Proceedings of the 8th International Conference on Applications and Principles of Information Science, pp. 233–236, January 2009.

[21] T. Hayashi, K. Nakano, and S. Olariu, *Work-Time Optimal k-merge Algorithms on the PRAM*, IEEE Trans. on Parallel and Distributed Systems, Vol. 9, No.3, pp. 275–282, March, 1998.

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

[23] Intel Corporation, *Intel Xeon Processor 7000 Sequence*,
http://www.intel.com/products/processor/xeon7000/

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