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 sort-
ing 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) multi-
processors that has a near linear speedup is exhibited in [12]. An efficient multi-
threaded implementation of MergeSort on current multi-core architecture is shown
in [13]. The loading balance is an important issue on the parallel computing. There-
fore 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:

\[
\begin{align*}
\text{qsort(data, num, sizeof(int), comp);} \\
\downarrow \\
\text{psort(data, num, sizeof(int), comp);}
\end{align*}
\]
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 multi-core 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 hexadcore 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 \( A = \langle a_0, a_1, \ldots, a_{n-1} \rangle \) 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.

Let \( A = \langle a_0, a_1, \ldots, a_{n-1} \rangle \) 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 \( B_i = \langle a_{i \cdot \frac{p}{n}}, a_{(i+1) \cdot \frac{p}{n}}, \ldots, a_{(i+1) \cdot \frac{p}{n} - 1} \rangle \). Suppose that each block \( B_i \ (0 \leq i \leq p-1) \) is sorted independently, and \( B_i = \langle b_{i,0}, b_{i,1}, \ldots, b_{i, \frac{p}{n} - 1} \rangle \) denotes the sorted sequence
thus obtained. In other words, \(b_{i,0} < b_{i,1} < \cdots < b_{i,p-1}\) holds. For an arbitrary integer \(k > 0\), we further partition each sorted block \(B_i (0 \leq i \leq p-1)\) into \(p^k\) sub-blocks \(B_{i,0}, B_{i,1}, \ldots, B_{i,p^k-1}\) such that \(B_{i,j} = \langle b_{i,j\cdot p^k}, b_{i,j\cdot p^k+1}, \ldots, b_{i,(j+1)\cdot p^k-1} \rangle\). Clearly, each \(B_{i,j}\) has \(\frac{n}{p^k}\) keys. Let \(C_i\) denote the sequence of keys obtained by picking the minimum key from each of the sub-blocks \(B_{i,0}, B_{i,1}, \ldots, B_{i,p^k-1}\). In other words, \(C_i = \langle b_{i,0\cdot p^k}, b_{i,1\cdot p^k}, \ldots, b_{i,(p^k-1)\cdot p^k} \rangle\). Let \(D\) denote the combined sequence of \(C_0, C_1, \ldots, C_{p^k-1}\). Since each \(C_i\) has \(p^k\) keys, \(D\) 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 \(p^k\) 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, \(d_i = c_{i\cdot p^k} (0 \leq i \leq p-1)\) 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.](image)

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 \(D_i = \{ x \in C \mid d_i \leq x < d_{i+1} \} = \{c_{i\cdot p^k}, c_{i\cdot p^k+1}, \ldots, c_{(i+1)\cdot p^k-1} \}\). Clearly, each \(D_i\) has \(p^k\) 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 \(|A_{i,j}| \leq \frac{n}{p^k} - 1\) holds. For example, in the figure, since \(|D_{0,1}| = 0\), we can guarantee that \(|A_{0,1}| \leq \frac{n}{p^k} - 1\).
Similarly, if $|D_{i,j}| = 1$ then $|A_{i,j}| \leq 2 \frac{n}{p^2 k} - 1$ holds. In general, we have

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

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

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

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 \( B_0, B_1, \ldots, B_{p-1} \) 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 \( \frac{p\log n}{p^2k} \) 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 \( <_i \) to sort keys in \( C \). Since keys in each \( C_0, C_1, \ldots, C_{p-1} \) 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 \( d_0, d_1, \ldots, d_{p-1} \).

**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 \( b_{s_{i,j}} \geq d_j \). Clearly, \( A_{i,j} = \{b_{s_{i,j}}, b_{s_{i,j}+1}, \ldots, b_{s_{i,j}+1} \} \) holds, where \( s_{i,0} = \frac{p}{p} \). 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 \( <_i \) to sort keys in \( A_i \).

**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 \( \alpha_j \)-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 \( A_{0,j}, A_{1,j}, \ldots, A_{p-1,j} \). Also, each \( A_{i,j} \) is sorted. Hence, the sorting of \( R_j \) can be done by merging \( A_{0,j}, A_{1,j}, \ldots, A_{p-1,j} \).

Let us evaluate the computing time necessary to perform each step. In Step 1.1, each processor performs the sorting of \( \frac{p}{p} \) keys. This can be done in \( O(\frac{p}{p} \log \frac{p}{p}) \) time using the heap sort, and in expected \( O(\frac{p}{p} \log \frac{p}{p}) \) 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 \( O(\frac{p}{p} \log \frac{p}{p} + p^2k \log p) \) time.

In Step 2.1, each processor performs \( p \) binary searches on \( \frac{p}{p} \) keys. Hence, Step 2.1 can be done in \( O(p \log \frac{p}{p}) \) time. In Step 2.2, the sum and the prefix sums of \( pk \) 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 \( O(\frac{p}{p} + \frac{p}{p} - p) \) time. Therefore Step 2 can be done in \( O(\frac{p}{p} + p \log \frac{p}{p}) \) 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.

```c
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 MultiProcessing). 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 $n > 10,000$, 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 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.

<table>
<thead>
<tr>
<th>The number of available cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n &lt; 50,000$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$50,000 \leq n &lt; 500,000$</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>$500,000 \leq n$</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>16</td>
</tr>
</tbody>
</table>

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 multi-core 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 “$<$” 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 $P(0)$
Fig. 2. Computing time for our parallel sort.
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).

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

<table>
<thead>
<tr>
<th>p</th>
<th>Step1[s]</th>
<th>Step2[ms]</th>
<th>Step3[ms]</th>
<th>Total[s]</th>
<th>Step1[s]</th>
<th>Step2[ms]</th>
<th>Step3[ms]</th>
<th>Total[s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>159.654</td>
<td>0.038</td>
<td>14.867</td>
<td>174.521</td>
<td>159.769</td>
<td>0.035</td>
<td>14.677</td>
<td>174.445</td>
</tr>
<tr>
<td>4</td>
<td>78.803</td>
<td>0.038</td>
<td>9.724</td>
<td>88.527</td>
<td>78.493</td>
<td>0.034</td>
<td>9.815</td>
<td>88.308</td>
</tr>
<tr>
<td>8</td>
<td>46.281</td>
<td>0.062</td>
<td>6.902</td>
<td>53.184</td>
<td>46.382</td>
<td>0.062</td>
<td>6.857</td>
<td>53.238</td>
</tr>
<tr>
<td>16</td>
<td>32.060</td>
<td>0.395</td>
<td>4.953</td>
<td>37.013</td>
<td>32.174</td>
<td>0.385</td>
<td>4.957</td>
<td>37.131</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>p</th>
<th>Step1[s]</th>
<th>Step2[ms]</th>
<th>Step3[ms]</th>
<th>Total[s]</th>
<th>Step1[s]</th>
<th>Step2[ms]</th>
<th>Step3[ms]</th>
<th>Total[s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>80.220</td>
<td>0.026</td>
<td>9.237</td>
<td>90.456</td>
<td>80.144</td>
<td>0.021</td>
<td>9.700</td>
<td>89.844</td>
</tr>
<tr>
<td>4</td>
<td>40.151</td>
<td>0.028</td>
<td>9.199</td>
<td>49.450</td>
<td>40.330</td>
<td>0.030</td>
<td>5.665</td>
<td>45.995</td>
</tr>
<tr>
<td>8</td>
<td>22.147</td>
<td>0.034</td>
<td>10.266</td>
<td>32.413</td>
<td>21.862</td>
<td>0.036</td>
<td>3.481</td>
<td>25.342</td>
</tr>
<tr>
<td>16</td>
<td>13.357</td>
<td>0.048</td>
<td>11.565</td>
<td>24.921</td>
<td>13.366</td>
<td>0.052</td>
<td>2.466</td>
<td>15.831</td>
</tr>
<tr>
<td>24</td>
<td>11.699</td>
<td>0.089</td>
<td>12.312</td>
<td>24.011</td>
<td>11.562</td>
<td>0.108</td>
<td>2.801</td>
<td>14.362</td>
</tr>
</tbody>
</table>

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 \( k \).
An Efficient Parallel Sorting Algorithm Compatible with the Standard qsort

Table 3. Performance of parallel sorting.

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

<table>
<thead>
<tr>
<th>n</th>
<th>qsort</th>
<th>qsort2</th>
<th>qsort_int</th>
<th>mergesort_int</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time[s]</td>
<td>Time[s]</td>
<td>Speed up</td>
<td>Time[s]</td>
</tr>
<tr>
<td>100</td>
<td>0.000011</td>
<td>0.000011</td>
<td>1.00</td>
<td>0.001149</td>
</tr>
<tr>
<td>1,000</td>
<td>0.000174</td>
<td>0.000172</td>
<td>1.01</td>
<td>0.001438</td>
</tr>
<tr>
<td>10,000</td>
<td>0.001374</td>
<td>0.001381</td>
<td>0.99</td>
<td>0.003874</td>
</tr>
<tr>
<td>100,000</td>
<td>0.016122</td>
<td>0.007734</td>
<td>2.08</td>
<td>0.024758</td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.194148</td>
<td>0.028720</td>
<td>6.89</td>
<td>0.270884</td>
</tr>
<tr>
<td>10,000,000</td>
<td>2.416300</td>
<td>0.212756</td>
<td>11.34</td>
<td>2.196236</td>
</tr>
<tr>
<td>100,000,000</td>
<td>28.525679</td>
<td>2.712458</td>
<td>10.47</td>
<td>22.66808</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>n</th>
<th>qsort</th>
<th>qsort2</th>
<th>qsort_int</th>
<th>mergesort_int</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time[s]</td>
<td>Time[s]</td>
<td>Speed up</td>
<td>Time[s]</td>
</tr>
<tr>
<td>100</td>
<td>0.000098</td>
<td>0.000099</td>
<td>0.98</td>
<td>0.000248</td>
</tr>
<tr>
<td>1,000</td>
<td>0.000152</td>
<td>0.000160</td>
<td>0.95</td>
<td>0.000399</td>
</tr>
<tr>
<td>10,000</td>
<td>0.000917</td>
<td>0.000919</td>
<td>0.99</td>
<td>0.000955</td>
</tr>
<tr>
<td>100,000</td>
<td>0.000676</td>
<td>0.000746</td>
<td>2.03</td>
<td>0.002463</td>
</tr>
<tr>
<td>1,000,000</td>
<td>0.108484</td>
<td>0.206560</td>
<td>5.25</td>
<td>0.023172</td>
</tr>
<tr>
<td>10,000,000</td>
<td>1.265774</td>
<td>0.099879</td>
<td>12.67</td>
<td>0.249060</td>
</tr>
<tr>
<td>100,000,000</td>
<td>14.707196</td>
<td>1.117982</td>
<td>13.15</td>
<td>2.456557</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>24</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>28.525[s]</td>
<td>15.791[s]</td>
<td>8.121[s]</td>
<td>4.762[s]</td>
<td>3.221[s]</td>
<td>2.712[s]</td>
</tr>
<tr>
<td>Speed-up</td>
<td>1.00</td>
<td>1.80</td>
<td>3.51</td>
<td>6.00</td>
<td>8.86</td>
<td>10.47</td>
</tr>
</tbody>
</table>

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

An Efficient Parallel Sorting Algorithm Compatible with the Standard Qsort