Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

An Optimal In-Place Parallel Quicksort

Kaijung Chang, Yan Huat Tan, Yaakov Varol, Jiang-Hsing Chu
Department of Computer Science
Southern Illinois University at Carbondale
Carbondale, IL 62901, USA

Abstract

We give a complete analysis of the parallel quicksort algorithm which we call partswap. Unlike many other parallel quicksort algorithms, the partswap algorithm is in-place, that is, no auxiliary memory is needed. We will also show that partswap achieves maximal speed-up on the average. Average-case analyses are also given to justify the choice of the load distribution strategy over a straightforward and seemingly equally efficient load distribution strategy.

1 Introduction

Sorting is one of the most extensively studied subjects in computer science. Many sorting algorithms [7] have been developed. It is well known that the lower bound on sequential comparison-based sorting algorithms is $O(n \log n)$, where n is the size of the input array. Several sorting algorithms achieve this lower bound, among them quicksort[4] is one of the more commonly used. As the lower bound on sorting has been achieved, many researchers have turned their attention to parallel sorting algorithms [1, 5, 10].

A parallel quicksort algorithm which assigns a processor to each item in the array, was developed by Martel and Gusfield [9]. It runs in $O(\log n)$ time, using an expected $O(n^3)$ space on CRCW (concurrent read concurrent write) PRAM (parallel random access machine) with $O(n)$ processors. A more practical, quicksort based, algorithm called *parallel sorting by regular sampling*, PSRS, was proposed by Shi and Schaeffer [11]. PSRS has a time complexity of $O((n/p)\log n + p^2 \log p + (n/p)\log p)$, where p is the number of processors. The parallel quicksort for the EREW (exclusive read exclusive write) PRAM model presented by Zhang and Rao [14] has an expected time complexity of $O((n/p+\log p)\log n)$ and requires $O(n)$ space. These and other parallel quicksort algorithms [8] seek to engage all available processors in useful and non-overlapping work from start to finish. One way to accomplish this is to partition and distribute the data among the processors. A well known hypercube formulation of this approach called *hyperquicksort* was given by Wagar [12]. Another formulation called *parasort* proposed by Wheat and Evans [13] uses two way merging of sorted partitions and is geared to shared memory multiprocessors.

Brown and Xiong [2] presented *pquicksort*, which initially assigns items evenly among the processors in an interleaving manner. A pivot is chosen and broadcast to every processor. Each processor partitions its own items and then swaps are performed so that the whole array is partitioned around the pivot. Recursively apply this process to all partitions until the entire array is sorted. Three procedures for swapping were proposed and discussed by the authors. Two of the swapping procedures require $O(n)$ auxiliary space and the third one, although it requires only a constant amount of additional space, is less efficient because it uses producer-consumer style procedures.

We proposed a new parallel quicksort algorithm [3] which we call *partswap* for the EREW PRAM model. It is a variation of pquicksort, but runs efficiently without the need for $O(n)$ additional space. Both partswap and pquicksort assign items to processors in an interleaving manner. The main difference between them is the way that items are swapped in the partitioning process. The partswap swaps items in iterations, and maintains the interleaving structure throughout the iterations. The partswap algorithm is similar to hyperquicksort in principle, but is designed for shared memory multiprocessors. We will show that the partswap algorithm achieves maximal speed-up on the average. The efficiency of partswap is due to its load distribution strategy. We will give a complete analysis to prove why the chosen load distribution strategy outperforms a straightforward and seemingly equally efficient load distribution strategy.

2 Partswap — A New Parallel Quicksort

In this section we present a new parallel quicksort algorithm, partswap, which achieves maximal speed-up on the average and does not require additional memory space. Recall that the process of quicksort is simply a series of partitionings. Generally, there are two approaches to parallelizing the quicksort algorithm. One approach lets processors work on different partitionings in parallel; the other lets all processors work on the same partitioning. Our algorithm is a combination of both approaches. At the beginning, all processors work together to partition the input array. After the input array is partitioned into two subarrays, half of the processors work on one subarray and the other half of the processors work on the other subarray. Eventually, each processor works on its own subarray.

Since our new algorithm can be made to work with any load distribution strategies, we will discuss our load distribution strategy separately. The following steps are the outline of the partswap algorithm.

Load Distribution: assign items in the subarray to the processors for partitioning.

Initial Partitioning: choose a pivot and broadcast it to the processors, each processor then employs a sequential partition algorithm to partition items assigned to it around the pivot. After this step, each group of items assigned to the same processor is partitioned, but the subarray as a whole is not partitioned. Swaps are needed in order for the subarray to become partitioned.

Swapping: processors swap items so that the subarray is partitioned around the pivot. This is done by repeatedly combining two groups of partitioned items into a larger group of partitioned items until there is only one group left. Since pairs of groups can be combined in parallel, after each iteration, the number of groups is halved, while the number of processors working on each group is doubled. If there are p processors to partition a subarray, then it takes $\log_2 p$ iterations of swaps to partition the subarray into two smaller subarrays.

Recursion: divide processors into two groups, one group of processors continue working on one of the subarrays while the other group of processors continue working on the other subarray. When there is only one processor left, this processor will be responsible for sorting the entire subarray all by itself.

The choice of a pivot could be made elaborate and if chosen properly could improve load balancing and thereby complexity. However, pivot selection is not the focus of this paper and we could simply take it to be the first item in the subarray. Pivot broadcasting can become the bottleneck of this algorithm when the number of processors is large. In such a case, we can let all processors select a pivot using the same scheme and thus avoid the cost of broadcasting. Therefore, in our complexity analysis, the cost for broadcasting the pivot will be ignored.

As mentioned earlier, we did not describe the load distribution strategy in the Load Distribution step because our algorithm can work with any load distribution strategies. Naturally, the way processors cooperate with each other in the Swapping step would depend on the load distribution strategy. We will now discuss the load distribution strategy that is used in partswap.

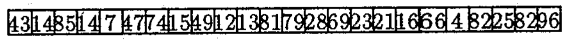
2.1 Interleaving Load Distribution Strategy

Suppose there are p processors P_0, P_1, $\ldots$, P_{p-1}, where $p = 2^k$ for some integer k, and the subarray $A[low, high]$ is to be partitioned. A load distribution strategy, which we call *interleaving* strategy, assigns the first and every other p items to the first processor, the second and every other p items to the next processor, and so on. In other words, we assign $A[low]$, $A[low+p]$, $A[low+2p]$, $\ldots$ to P_0, and $A[low+1]$, $A[low+p+1]$, $A[low+2p+1]$, $\ldots$ to P_1, and so on.

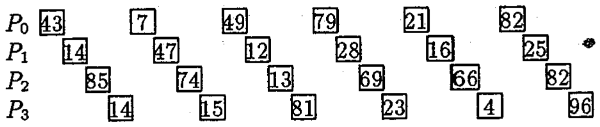
In the Swapping step, processors will work together as if they were grouped in a hypercube formulation, from the highest dimension to the lowest dimension. In the first iteration of the Swapping step, processors P_0 and $P_{p/2}$ work together, processors P_1 and $P_{p/2+1}$ work together, and so on. In the second iteration, processors P_0, $P_{p/4}$, $P_{p/2}$, and $P_{3p/4}$ work together, processors P_1, $P_{p/4+1}$, $P_{p/2+1}$, and $P_{3p/4+1}$ work together, and so on. This may seem complex at the first glance, but it is really easy if the indices are viewed as binary numbers. In iteration i, all processors whose indices (represented in binary) have the same rightmost $k-i$ bits are in the same group.

2.2 An Example

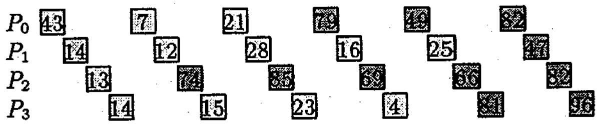
We now illustrate how our algorithm works by showing the first partitioning step by step. Assume we have 4 processors, P_0 through P_3, to partition 24 items with indices from 0 to 23 shown in the following array.



In the Load Distribution step, we assign items with indices 0, 4, 8, $\ldots$, 20 to P_0, items with indices 1, 5, 9, $\ldots$, 21 to P_1, items with indices 2, 6, 10, $\ldots$, 22 to P_2, and items with indices 3, 7, 11, $\ldots$, 23 to P_3.



In the Initial Partitioning step, the pivot, 43, is chosen as the pivot and each processor partitions the items assigned to it around the pivot. In the figures, the items which are less than or equal to the pivot are lightly shaded while the items which are greater than the pivot are heavily shaded.



In the Swapping step, we perform iterations of swapping. In the first iteration, P_0 works with P_2 while P_1 work with P_3. Note that in this iteration, only the items 21 74 have to be swapped. After swapping, we have

In the next iteration, which is the last iteration for this example, the items 74, 85, and 79 have to be swapped with the items 16, 4, and 25. After swapping, we have

$P_{0..3}$	43	14	13	14	7	12	21	15	16	28	4	23	25	74	69	85	49	79	75	66	63	18	24	78	96

The Recursion step brings us to next stage, where processors P_0 and P_1 will work together to partition the first subarray, while processors P_2 and P_3 will work together to partition the second subarray.

2.3 Complexity Analysis

We now analyze the partswap algorithm. We will first study the time steps without considering the recursive calls. Later we will sum over all levels of recursion to obtain the total number of time steps. With this approach, only Initial Partitioning and Swapping contribute time steps.

Let us begin with the number of time steps required in the Swapping step. First we consider the expected number of swaps when combining two subarrays of items, each of size $m/2$. Let l_1 and l_2 be the number of the items that are less than or equal to the pivot in the two subarrays to be combined. Given l_1 and l_2, the number of swaps needed is $|\lceil (l_2 - l_1)/2 \rceil|$.

Given that the pivot is chosen randomly, The probability

$$Pr(l_1 = i, l_2 = j) = \frac{\binom{i+j}{i}\binom{m-i-j}{m/2-i}}{m\binom{m}{m/2}}.$$

It is obvious that the probabilities $Pr(l_1 = i, l_2 = j)$ and $Pr(l_1 = j, l_2 = i)$ are equal due to symmetry. Therefore, when computing the expected number of swaps, the contribution from the cases $(l_1 = i, l_2 = j)$ and $(l_1 = j, l_2 = i)$ can be considered altogether. Without loss of gererality, let us assume that $l_2 \geq l_1$. Then we have

$$|\lceil (l_2 - l_1)/2 \rceil| + |\lceil (l_1 - l_2)/2 \rceil| = l_2 - l_1.$$

Therefore, the expected number of swaps is

$$\sum_{j=1}^{m/2} \sum_{i=0}^{j-1} (j-i) Pr(l_1 = i, l_2 = j)$$

which is equal to

$$\sum_{j=1}^{m/2} \sum_{i=0}^{j-1} (j-i) \frac{\binom{i+j}{i}\binom{m-i-j}{m/2-i}}{m\binom{m}{m/2}}.$$

The following summation

$$\sum_{0 \leq i \leq j} (j-i) \binom{i+j}{i} \binom{2n-i-j}{n-j} = n 2^{2n-2}$$

related to the analysis of shellsort can be found in [7, p599], from which we obtain

$$\sum_{j=1}^{m/2} \sum_{i=0}^{j-1} (j-i) \binom{i+j}{i} \binom{m-i-j}{m/2-i} = m 2^{m-3}.$$

It follows that the expected number of swaps needed to combine two size $m/2$ subarrays is

$$\frac{2^{m-3}}{\binom{m}{m/2}}.$$

From [6], we know that

$$\binom{m}{m/2} \approx \frac{2^m}{\sqrt{\frac{\pi m}{2}}}$$

and therefore,

$$\frac{2^{m-3}}{\binom{m}{m/2}} \approx \frac{2^{m-3}}{\frac{2^m}{\sqrt{\frac{\pi m}{2}}}} = \sqrt{\frac{\pi m}{128}}$$

Now we can use the above formula on expected number of swaps to compute the number of time steps. Assume there are q processors to partition a subarray of size m. In the first iteration, processors work in parallel as groups of two on combining two size m/q subarrays. From what we derived above, we know there are $\sqrt{\pi m / 64q}$ swaps (which can be shared by two processors without overhead) to be done within each group. In the second iteration, each group has 4 processors to work on combining two size $2m/q$ subarrays, and so on. Table 1 summarizes the number of time steps required in each iteration.

iteration	p	size	swaps	time steps
1	2	m/q	$\sqrt{\pi m/64q}$.	$\frac{1}{16}\sqrt{\pi m/q}$
2	4	$2m/q$	$\sqrt{\pi m/32q}$	$\frac{1}{16}\frac{1}{\sqrt{2}}\sqrt{\pi m/q}$
3	8	$4m/q$	$\sqrt{\pi m/16q}$	$\frac{1}{16}\frac{1}{2}\sqrt{\pi m/q}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\log q$	q	$m/2$	$\sqrt{\pi m/128}$	$\frac{1}{16}\frac{\sqrt{2}}{\sqrt{q}}\sqrt{\pi m/q}$

Table 1: Number of time steps required by iterations (interleaving)

The total number of time steps is

$$\frac{1}{16}\sqrt{\frac{\pi m}{q}} \left(1 + \frac{1}{\sqrt{2}} + \ldots + \frac{\sqrt{2}}{\sqrt{q}} \right),$$

which is $O\left(\sqrt{m/q}\right)$.

In our analysis, the number of time steps decreases in a factor of $1/\sqrt{2}$ as the algorithm proceeds to next iternation. Note that the expected number of steps we obtained is smaller than what it should be because we assume that the swaps occur evenly among the subarrays. In reality, some subarrays will have more swaps than the others. Nevertheless, the number of time steps still decreases in a factor less than 1. Consequently, our analysis is correct in terms of Big-O.

Recall that the above result is only the expected number of time steps in the Swapping step, assuming q processors are partitioning two size $m/2$ subarrays. We need to compute the number of time steps required in the Initial Partitioning step too. The time steps needed in the Initial Partitioning step is obviously proportional to the number of items that are assigned to each processor. Assume there are p processors to sort an array of n items and the items are equally partitioned in each level. With this assumption, the number of time steps required in the Initial Partition step is always $O(n/p)$ in every level because the number of items assigned to each processor remains a constant. Because the ratio of the number of items in a subarray to the number of processors working on the subarray is fixed, the number of time steps required in the Swapping step also remains constant until when there is only one processor left to partition a subarray, i.e., levels deeper than $\log p$. Table 2 gives the number of time steps needed by the partswap algorithm. They are listed by levels of recursion.

Levels	Partitioning	Swapping
1	$O(n/p)$	$O(\sqrt{n/p})$
2	$O(n/p)$	$O(\sqrt{n/p})$
$\vdots$	$\vdots$	$\vdots$
$\log p$	$O(n/p)$	$O(\sqrt{n/p})$
$\log p + 1$	$O(n/p)$	0
$\vdots$	$\vdots$	$\vdots$
$\log n$	$O(n/p)$	0

Table 2: Number of time steps by levels

Summing over all levels of recursion, we have the expected complexity of the partswap algorithm, which is

$$O((n/p)\log n + \sqrt{n/p}\log p).$$

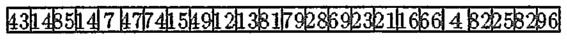
We note that partswap has the performance comparable to other parallel quicksort algorithms, and the advantage of not requiring auxiliary memory. The partswap algorithm achieves maximal speed-up when p is smaller than or comparable to n.

3 Segmentation

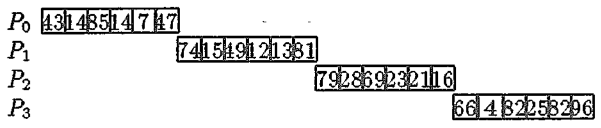
One obvious question is why one would use the interleaving strategy. A natural and simpler load distribution strategy, called *segmentation*, simply assigns the first segment of n/p items to the first processor, next segment of items to the next processor, and so on. That is, assign $A[low]$, $A[low+1]$, …, $A[low+p-1]$ to P_0, and $A[low+p]$, $A[low+p+1]$, …, $A[low+2p-1]$ to P_1, and so on.

The pairing of the processors is easier than that of the interleaving strategy. In the first iteration of the Swapping step, processors P_0 and P_1 work together, processors P_2 and P_3 work together, and so on. In the second iteration, processors P_0, P_1, P_2, and P_3 work together, processors P_4, P_5, P_6, and P_7 work together, and so on. In general, in the kth iteration, the first 2^k processors work together, the next 2^k processors work together, and so on.

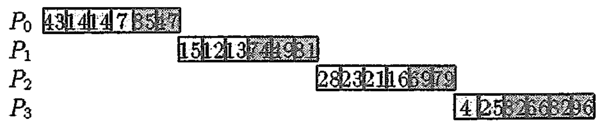
We now illustrate how our algorithm works by showing the first partitioning, step by step. Assume we have 4 processors, P_0 through P_3, to partition 24 items with indices from 0 to 23. Suppose we want to partition the given array:



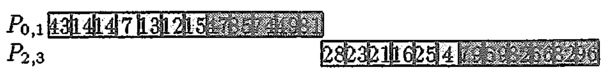
Since we are using the segmentation method, in the Load Partitioning step, items with indices 0, 1, …, 5 are distributed to processor 0, items with indices 6, 7, …, 11 are distributed to processor 1, items with indices 12, 13, …, 17 are distributed to processor 2, and items with indices 18, 19, …, 23 are distributed to processor 3.



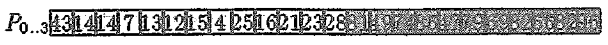
In the Initial Partitioning step, the pivot, 43, is broadcast to all processors and each processor partitions its part of the array around the pivot.



In the Swapping step, we perform iterations of swapping. In the first iteration, processors P_0 and P_1 are working together, processors P_2 and P_3 are working together. Note that in this iteration, 4 swaps are needed. After swapping, we have



In the next iteration, which is the last iteration in this example, all processors are working together, and 5 swaps are needed. We have



The Recursion step brings us to next stage, where processors P_0 and P_1 will work together to partition the first subarray, while processors P_2 and P_3 will work together to partition the second subarray.

It would seem that the segmentation strategy is as efficient as the interleaving strategy. However, the following analysis will show otherwise.

3.1 Analysis of Segmentation Strategy

Note that the change of load distribution strategy will only affect the number of time steps required in the Swapping steps. In order to compute the number of time steps required in the Swapping steps, we will consider the expected number of swaps when combining two subarrays of items, each of size $m/2$. Let l_1 and l_2 be the number of the items that are less than or equal to the pivot in the two subarrays to be combined. Given l_1 and l_2, the number of swaps needed is the minimum of l_2 and $m/2 - l_1$. In the following table, we show the number of swaps needed for different values of l_1 and l_2.

l_1 \ l_2	0	1	2	...	$m/2-1$	$m/2$
0	0	1	2	...	$m/2-1$	$m/2$
1	0	1	2	...	$m/2-1$	$m/2-1$
2	0	1	2	...	$m/2-2$	$m/2-2$
3	0	1	2	...	$m/2-3$	$m/2-3$
4	0	1	2	...	$m/2-4$	$m/2-4$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	...	$\vdots$	$\vdots$
$m/2-1$	0	1	1	...	1	1
$m/2$	0	0	0	...	0	0

Table 3: Number of swaps needed in the segmentation strategy

The following observations can be made about Table 3. First, if we trace the diagonals going from lower-left to upper-right, i.e., the entries with $l_1 + l_2 = X$ for some X, we have the series 0, 1, 2, etc. Second, for the diagonals in the upper half, these values coincide with the l_2 values. Third, the table is symmetric about the $l_1 + l_2 = m/2$ diagonal.

Let L be a random variable whose value is the number of items that are less than or equal to the pivot. Note that the probabilities $Pr(L = X) = 1/m$, for $1 \le X \le m$, since every item is equally likely to be picked as the pivot. It follows that for any $j \le m/2$,

$$Pr(l_2 = j|L = X) = \frac{\binom{X}{j}\binom{m-X}{m/2-j}}{\binom{m}{m/2}}.$$

Because $Pr(l_2 = j|L = X) = Pr(l_2 = j|L = m - X)$ and the symmetry property mentioned earlier, we only need to focus on the upper half of the table. For each such diagonal for some X, its contribution to the expected number of swaps is

$$\sum_{j=0}^{X} j \cdot Pr(l_2 = j|L = X) \cdot Pr(L = X),$$

which can be reduced to

$$\begin{aligned}
\sum_{j=0}^{X} \frac{j\binom{X}{j}\binom{m-X}{m/2-j}}{\binom{m}{m/2}\frac{1}{m}} &= \frac{1}{m}\sum_{j=1}^{X} \frac{j\binom{X}{j}\binom{m-X}{m/2-j}}{\binom{m}{m/2}} \\
&= \frac{1}{m}\sum_{j=1}^{X} \frac{X\binom{X-1}{j-1}\binom{m-X}{m/2-j}}{\frac{m}{m/2}\binom{m-1}{m/2-1}} \\
&= \frac{X}{2m}\sum_{j'=0}^{X'} \frac{\binom{X'}{j'}\binom{m-1-X'}{m/2-1-j'}}{\binom{m-1}{m/2-1}} \\
&= \frac{X}{2m}
\end{aligned}$$

The last step in the derivation was obtained because the summation of all probabilities of a distribution is 1. Thus, from the symmetry, the expected number of swaps is

$$\begin{aligned}
2\sum_{X=1}^{\frac{m}{2}} \frac{X}{2m} - \frac{\frac{m}{2}}{2m} &= \sum_{X=1}^{\frac{m}{2}} \frac{X}{m} - \frac{\frac{m}{2}}{2m} \\
&= \frac{1}{m}\left(\frac{\frac{m}{2}(\frac{m}{2}+1)}{2} - \frac{m}{4}\right) \\
&= \frac{m}{8}
\end{aligned}$$

Now we can use the above results on expected number of swaps to compute the number of time steps. Assume there are q processors to partition a subarray of size m. In the first iteration, processors work in parallel as groups of two on combining two size m/q subarrays. From what we derived above, we know that there are $2m/8q$ swaps to be done within each group, where work can be shared by two processors without overhead. In the second iteration, each group has 4 processors to work on combining two size $2m/q$ subarrays, and so on. Table 4 lists the number of time steps required in each iteration.

iteration	p	size	swaps	time steps
1	2	m/q	$2m/8q$	$m/8q$
2	4	$2m/q$	$4m/8q$	$m/8q$
3	8	$4m/q$	$8m/8q$	$m/8q$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\log q$	q	$m/2$	$m/8$	$m/8q$

Table 4: Number of time steps required by iterations (segmentation)

Thus, the expected number of time steps in the Swapping step is $(m/8q)\log q$, which is

$O((m/q)\log q)$. This result is definitely worse than $O(\sqrt{m/q})$, which is what we got for the interleaving strategy.

If the segmentation strategy were used, the time complexity of our algorithm would become

$$O((n/p)\log n + (n/p)(\log p)^2),$$

assuming p processors are used to sort an array of size n.

Obviously both strategies will have similar performance when $n \gg p$ since the number of time steps performed in the Swapping step in not the dominating factor in this case. However, the interleaving strategy outperforms the segmentation strategy when p is comparable to or smaller than n.

This can be explained as follows. Consider the best cases when combining two subarrays. If the segmentation strategy is used, the best cases are when $l_1 = n/2$ or $l_2 = 0$. In contrast, if the interleaving strategy is used, the best cases are when l_1 and l_2 are close. Given a random input, it is more likely that l_1 is close to l_2. The interleaving strategy takes advantage of this situation and that is why it outperforms the segmentation strategy on the average.

4 Conclusion

We developed a new in-place parallel quicksort algorithm called partswap. In partswap, we have all processors working on the same partitioning at first. After the partitioning is done, half of the processors will work on one partition and the other half of the processors will go on to work on the other partition. This repeats until all processors are working on different partitions at the same time. From then on, each processor is responsible for sorting its own partition all by itself. This algorithm has a good performance comparable to other well known parallel quicksort algorithms, plus the advantage of not needing auxiliary memory.

Note the performance of this algorithm is greatly dependent on how the array is actually partitioned. Eventually, each processor is given a subarray to sort. If all subarrays are of the same size, all processors will finish about at the same time and we will be able to achieve the best speedup. When the array is not partitioned evenly, the performance of our new algorithm suffers, because processors with smaller subarrays will finish early and become idle while processors with larger subarrays will have to work longer. Thus the speedup will not be good. It would be preferable to enhance our algorithm so that those processors which are given a small subarray can help other processors after they finish their own subarrays. Alternatively, at the early stage, when a subarray is not partitioned evenly, we could assign more processors to work on the larger partition and less processors to work on the smaller partition.

We use the interleaving load distribution strategy in partswap. Our analyses showed why the segmentation strategy does not work as well compared to the interleave strategy. Since the segmentation strategy is commonly used and is also related to the divide-and-conquer technique, it would be interesting to see if interleaving will prove to be useful in other algorithms.

References

[1] S. G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[2] T. Brown and R. Xiong. A parallel quicksort algorithm. *Journal of Parallel and Distributed Computing*, 19(10):83–89, Oct. 1993.

[3] K. Chang, Y. H. Tan, Y. Varol, and J.-H. Chu. Analysis of load partitioning: Segmentation vs. interleave. In *Proc. 7th IASTED/ISMM International Conference*, pages 202–206, Washington, D.C., Oct. 1995. IASTED.

[4] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5:10–15, 1962.

[5] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, MA., 1992.

[6] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, MA., 1973.

[7] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, MA., 1973.

[8] V. Kumar, A. Grama, A. Gupta, and G. Karyris. *Introduction to Parallel Computing.* Benjamin/Cummings, Redwood City, California, 1994.

[9] C. U. Martel and D. Gusfield. A fast parallel quicksort algorithm. *Information Processing Letters*, 30:97–102, Jan. 1989.

[10] M. J. Quinn. *Parallel Computing, Theory and Practice.* McGraw-Hill, New York, NY., 1994.

[11] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992.

[12] B. A. Wagar. Hyperquicksort: A fast sorting algorithms for hypercubes. In *Proceedings of the Second Conference on Hypercube Multiprocessors*, pages 292–299, 1987.

[13] M. Wheat and D. J. Evans. An efficient parallel sorting algorithm for shared memory multiprocessors. *Parallel Computing*, 18(1):91–102, Jan. 1992.

[14] W. Zhang and N. S. V. Rao. Optimal parallel quicksort on erew pram. *B.I.T.*, 31:69–74, 1991.