

Design and Experiment of a Communication-Aware Parallel Quicksort with Weighted Partition of Processors

Sangman Moh¹, Chansu Yu², and Dongsoo Han³

¹ Dept. of Internet Eng., Chosun Univ.
375 Seoseok-dong, Dong-gu, Gwangju, 501-759 KOREA
smmoh@chosun.ac.kr

² Dept. of Electrical and Computer Eng., Cleveland State Univ.
Cleveland, OH 44115, USA
c.yu91@csuohio.edu

³ School of Eng., Information and Communications Univ.
58-4 Hwaam-dong, Yuseong-gu, Daejeon, 305-348 KOREA
dshan@icu.ac.kr

Abstract. In most parallel algorithms, inter-processor *communication cost* is much more than *computing cost* within a processor. So, it is very important to reduce the amount of inter-processor communication. This paper presents the design and experiment of a new communication-aware parallel quicksort scheme for distributed-memory multiprocessor systems. The key idea of the proposed scheme is the *weighted partition* of processors, which enables not only less inter-processor communication but also better load balancing among the participating processors during the quicksort. The proposed scheme was designed and experimented on the Cray T3E parallel computer. According to the comparative performance measurement, for up to 64 processors, the proposed scheme results in about 40 ~ 60 percent shorter run time compared to the conventional parallel quicksort. That is mainly due to the small amount of inter-processor communication that results from the weighted partition and allocation of processors. The performance improvement is more substantial as the number of processors, the input size, and the input item size increases.

1 Introduction

Sorting is a fundamental operation that appears in many computing applications; it rearranges a list of input numbers in non-decreasing (or non-increasing) order. In any sequential sorting, the best performance is bounded to $O(n \log n)$ which is achieved by two well-known algorithms: mergesort and quicksort [1]. *Quicksort* [2-3] is often the best practical choice for sorting because it is remarkably efficient on the average and the constant factors hidden in the $O(n \log n)$ notation are quite small. The best-case performance of quicksort is $O(n \log n)$ and it is proven to be the same as the average performance while the worst-case performance of quicksort is $O(n^2)$ [1].

Since $O(n \log n)$ is optimal for any sequential sorting algorithm that does not use any special properties for the input patterns, the best parallel time complexity we can expect for a sequential algorithm using n processors is $O(n \log n) / n = O(\log n)$. Leighton [4] demonstrated an $O(\log n)$ sorting algorithm with n processors based on an algorithm by Ajtai, Komlos, and Szemerédi [5], but the constant hidden in the order notation was extremely large. Bitton et al. [6] published an extensive survey paper on parallel sorting. Akl [7] wrote a book devoted entirely to parallel sorting algorithms, which describes 20 different parallel sorting algorithms. The outcome of all this investigation is that a realistic $O(\log n)$ algorithm with n processors is a goal that will not be easy to achieve [8].

For parallel computer systems, some parallel sorting algorithms have been newly developed. On the other hand, the parallelized version of the sequential sorting algorithms has been also researched and used more actively than the newly developed parallel sorting algorithms [8]. We also focus on the *parallelized algorithm* of sequential sorting. In particular, our work concentrates on quicksort, which is popular and effectively used in many computing areas. It has been implemented on several well-known architectures such as hypercubes [9-10]. Jelenkovic and Omecen-Ceko [11] presented some experiments with multithreading in parallel quicksort. In order to speed up the computation-intensive tasks of sorting, a dedicated hardware solution was researched [12].

In most parallel algorithms, inter-processor *communication cost* is much more than *computing cost* within a processor. So, it is very important to reduce the amount of inter-processor communication. This paper proposes a *communication-aware parallel quicksort* scheme that is suitable for distributed-memory multiprocessor systems. The key idea of the proposed scheme is the *weighted partition* of processors, which enables not only less inter-processor communication but also better load balancing among the participating processors during the quicksort. We implemented the proposed scheme in C language using *MPI* APIs and ran it on the Cray T3E parallel computer. We then measured the performance of the proposed scheme and compared it with that of the conventional parallel quicksort [8]. According to our extensive performance measurement, for up to 64 processors, the proposed scheme results in about 40 ~ 60 percent shorter run time than the conventional scheme. This improvement is primarily due to the small amount of inter-processor communication that results from the weighted partition and allocation of processors, compared to the conventional approach. The performance improvement is more substantial as the number of processors, the input size, and the input item size increases. In addition, a more balanced partition of input numbers to participating processors is achieved.

The rest of the paper is organized as follows: Conventional parallel quicksort is reviewed in the following section. Section 3 presents the proposed communication-aware parallel quicksort scheme with examples. Experiment and performance results are discussed in Section 4. Finally, conclusion is covered in Section 5.

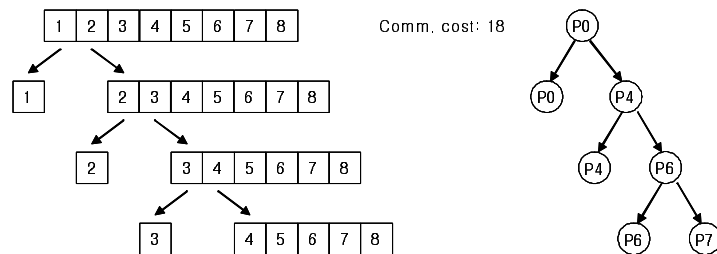
2 Related Work

Quicksort divides a list of input numbers into two sublists by choosing a *pivot* and moving the numbers smaller than the pivot into one list and the larger numbers into the other list. The algorithm then recursively sorts the sublists by choosing a new pivot and subdividing each of the sublists. If an input number is smaller than the pivot, it is placed in the left sublist. Otherwise, it is placed in the right sublist. The pivot could be any input number in the list, but often the first number in the list is chosen.

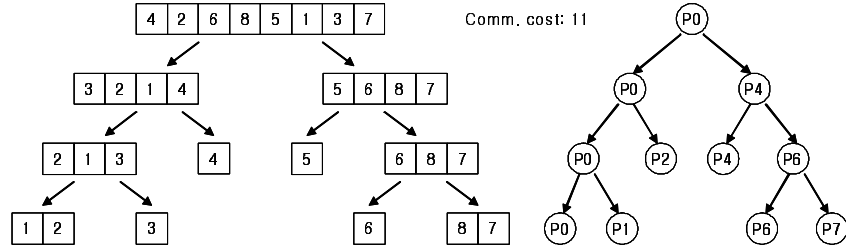
Quicksort is based on the *divide-and-conquer* concept, which consists of partitioning and merging. The partitioning is the major time-consuming part of quicksort, whereas the merging phase is very simple. The procedure is repeated on the partitioned sublists recursively. By repeating the procedure recursively, we are left with sublists of one number each. With proper merging (combining) of the sublists, a sorted list is obtained.

The code of quicksort can be formed as follows:

```
quicksort(list, start, end)
{
  if (start < end) {
    partition(list, start, end, pivot);
    quicksort(list, start, pivot-1);
    quicksort(list, pivot+1, end);
  }
}
```



(a) For a worst-case input pattern



(b) For a highly balanced input pattern

Fig. 1. Examples of the conventional parallel quicksort.

The function `partition()` moves numbers in the list between `start` to `end` so that those less than the pivot are before the pivot and those equal to or greater than the pivot are after the pivot.

One obvious way to parallelize quicksort is to start with one processor and pass on one of the recursive calls to another processor while keeping the other recursive call to perform. In the tree structure of parallel quicksort, the pivot is carried with the left list until the final sorting action. The conventional parallel quicksort algorithm is well described in [8], and two examples of this algorithm are shown in Fig. 1.

As Fig. 1 reveals, in general, the tree structure in quicksort may not be perfectly balanced. The sort tree becomes unbalanced if the pivots do not divide the lists into equal sublists. When we choose the first number in a sublist as the pivot, the original ordering of the numbers being sorted is the determining factor in the speed of the quicksort.

3 Design of a Communication-Aware Parallel Quicksort

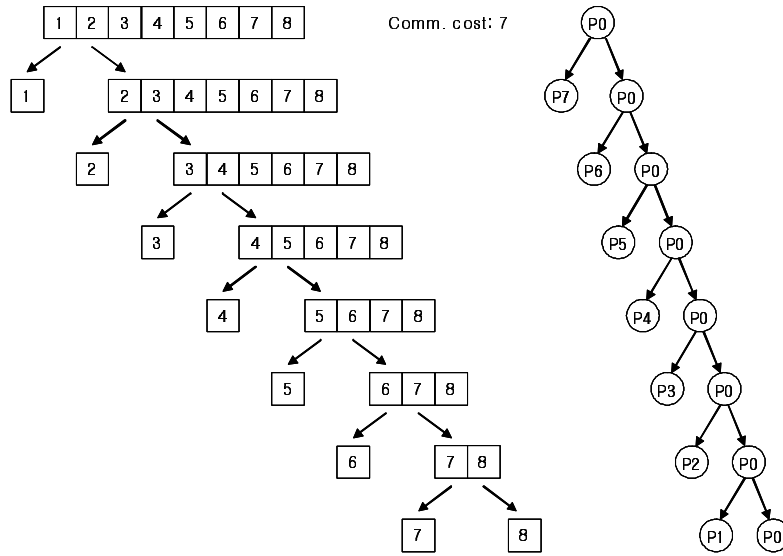
As mentioned earlier, the key idea of the proposed scheme is the weighted partition and allocation of processors, which enables not only less inter-processor communication but also better load balancing among the participating processors during the quicksort. Initially, the master processor takes the input list. By default, the master processor has the lowest processor identifier (*i.e.*, P_0) among the participating processors.

Let the number of input items that are less than the pivot and the number of input items that are greater than the pivot be N_L and N_R , respectively. Let the partition composed of input items that are less than the pivot and the partition composed of input items that are greater than the pivot be P_L and P_R , respectively. Then, at each level of *recursive partitioning* (tree operation), the proposed parallel quicksort operates as follows:

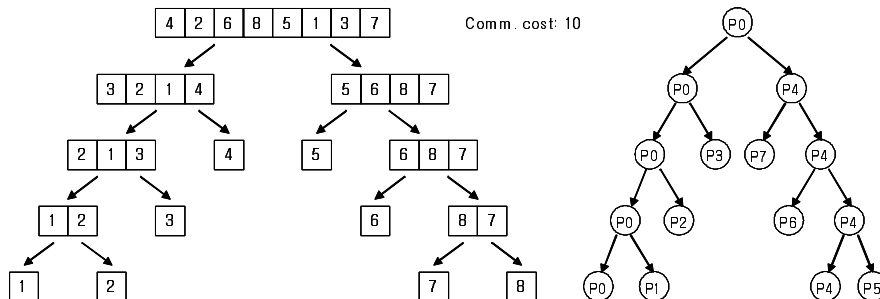
- (1) Partition the processors into two subpartitions by the ratio of N_L to N_R in the non-decreasing order of processor identifiers;
- (2) Send the smaller of P_L and P_R to the first processor in the other subpartition without the current processor.

The rest parts of the proposed scheme are the same as the conventional parallel quicksort [8], that is, the proposed scheme partitions the participating processors into two groups by the ratio of the size of two sublists and assigns (sends) the smaller of the two sublists to the other group without the current processor. We implemented the proposed scheme in C language using MPI APIs on the Cray T3E parallel computer, and the experiment results are discussed in Section 4.

Given an input list and processors, the proposed scheme minimizes the amount of inter-processor communication. As Fig. 2(a) shows, in the worst case, this scheme remarkably reduces the number of messages transferred between processors. Moreover, due to the weighted partition and allocation of processors, the communication cost is reduced and the parallelized computation is more balanced among participating processors. It results in the shorter run time of the proposed parallel quicksort compared to the conventional one.



(a) For a worst-case input pattern



(b) For a highly balanced input pattern

Fig. 2. Examples of the proposed communication-aware parallel quicksort.

Fig. 2 reveals the following: (i) for the worst-case input patterns, the proposed scheme outperforms the conventional one, (ii) for the best-case input patterns, the proposed scheme has the same performance as the conventional one, and (iii) for most of general input patterns, the proposed scheme also outperforms the conventional one. Thus, we can conclude that our approach is better than the conventional parallel quicksort.

For a worst-case input pattern, in the conventional parallel quicksort in Fig.1, the inter-processor communication cost is 18 and four out of eight processors are effectively used during the sorting, where the inter-processor communication cost represents the normalized amount of data transferred among processors. On the other hand, in the proposed parallel quicksort, the inter-processor communication cost is 7 and all the eight processors are effectively used, resulting in better performance.

4 Experiment and Performance Evaluation

In order to evaluate the performance of the proposed scheme and compare it with that of the conventional scheme, we implemented and ran both schemes on the Cray T3E parallel computer system in C language using MPI APIs. We then measured the run time of the two parallel quicksort schemes.

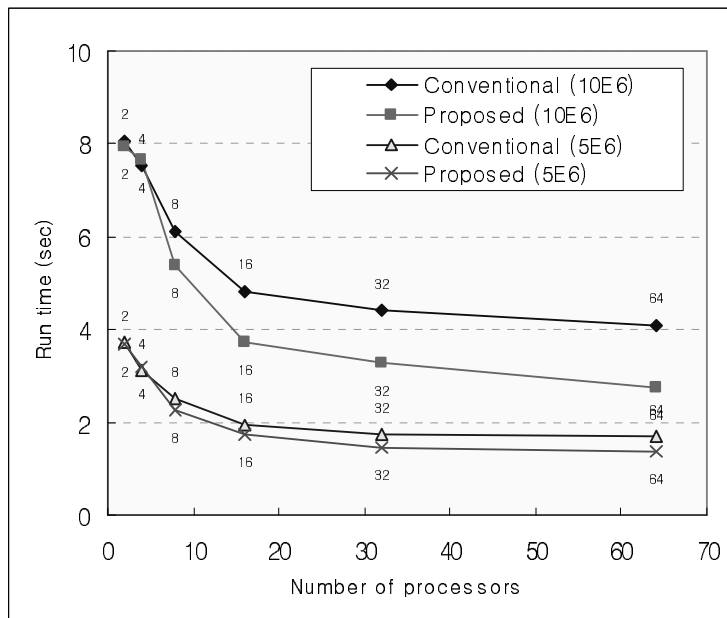


Fig. 3. Performance of parallel quicksort schemes (input item size = 4 bytes).

In our practical measurement, the input patterns were randomly generated and then the execution time was measured by an in-line timing check function inserted into the quicksort programs. Since both quicksort schemes used randomized input patterns, we can conclude that a reasonable average performance was obtained in our measurements. Note here that, during the run time, the inter-processor communication cost could not be measured separately from run time; however, it is inherently included in the run time.

Fig. 3 shows the execution time of the two parallel quicksort schemes, which were measured for input sizes of 5,000,000 and 10,000,000, where each input item is 4 bytes long. As the figure shows, the proposed parallel quicksort sorts the same size problem in shorter time than the conventional parallel quicksort. For instance, for 64 processors, the proposed scheme is faster than the conventional scheme by factors of 1.35 and 1.50 for the input sizes of 5,000,000 and 10,000,000, respectively. When few processors (up to 4) are used, the performance gain is small, because the difference between the two schemes is negligible for a small number of processors.

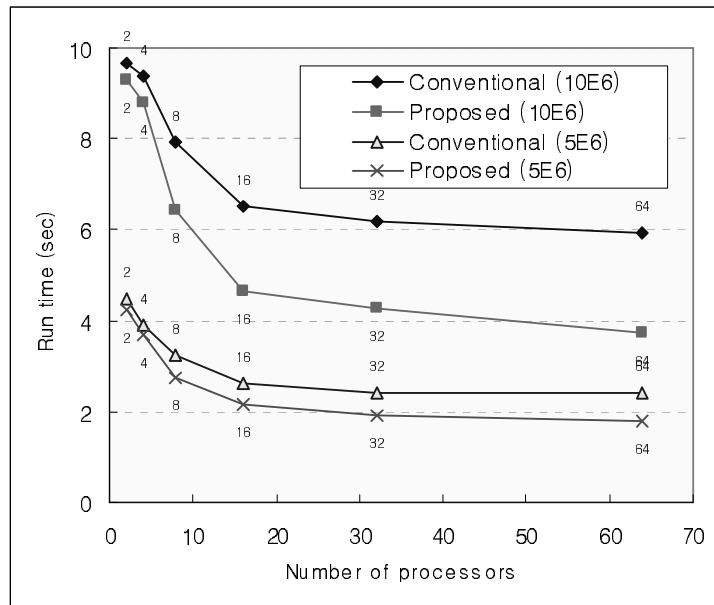


Fig. 4. Performance of parallel quicksort schemes (input item size = 8 bytes).

Fig. 4 shows the same performance metric as depicted in Fig. 3 except that each input item is 8 bytes long. In this case, for 64 processors, the proposed scheme is faster than the conventional one by factors of 1.43 and 1.61 for the input sizes of 5,000,000 and 10,000,000, respectively. From Fig. 3 and 4, it is clear that the performance is better as the input item size increases. This is mainly due to the fact that the communication cost increases as the input item size increases but the proposed scheme is more communication-efficient than the conventional one. Conclusively, the performance improvement is more substantial (i) as the number of processors increases, (ii) as the input size increases, and (iii) the input item size increases.

5 Conclusion

In this paper, a new communication-aware parallel quicksort scheme has been presented and discussed, which was implemented on the Cray T3E parallel computer in C language using MPI APIs. The key idea of the proposed scheme is the *weighted partition* of processors, which enables not only less inter-processor communication but also better load balancing among the participating processors during the quicksort. According to the extensive experiment results, the proposed scheme reduces the sorting time by 40 ~ 60 percent for up to 64 processors and is more communication efficient than the conventional scheme. The performance improvement is more substantial as the number of processors, the input size, and the input item size increases. This effect is mainly due to the weighted partition and allocation of processors. In addition, a more balanced partition of input numbers to participating processors is achieved.

In the near future, the proposed scheme will be implemented on a Linux-based PC cluster system using the MPI interface. In cluster systems, since most interconnection networks (*i.e.*, clustering networks) are much slower than the dedicated proprietary interconnection networks used in massively parallel multicomputers, it can be easily inferred that the performance gain is more improved.

References

1. Cormen, T.H., Leiserson, C.E., and Rivest, R.L.: Introduction to Algorithms, MIT Press, Cambridge, Massachusetts (1994)
2. Hoare, C.A.R.: Quicksort, *Computer Journal*, Vol. 5 (1962) 10-15
3. Wainwright, R.L.: A Class of Sorting Algorithms Based on Quicksort, *Comm. of ACM*, Vol. 28 (1985) 396-402
4. Leighton, F.T.: Tight Bounds on the Complexity of Parallel Sorting, *Proc. 16th Annual ACM Symp. on Theory of Computing*, New York (1984) 71-80
5. Ajtai, M., Komlos, J., and Szemerédi, E.: An $O(n \log n)$ Sorting Network, *Proc. 15th Annual ACM Symp. on Theory of Computing*, Boston, Massachusetts (1983) 1-9
6. Bitton, D., DeWitt, D.J., Hsiao, D.K., and Menon, J.: A Taxonomy of Parallel Sorting, *Computing Surveys*, Vol. 16 (1984) 287-318

7. Akl, S.: Parallel Sorting Algorithms, Academic Press, New York (1985)
8. Wilkinson B. and Allen, M.: Sorting Algorithms, Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers, Prentice-Hall, Upper Saddle River, New Jersey (1999)267-297
9. Fox, G.C., Williams, R.D., and Messina, P.C.: Parallel Computing Works, Morgan Kaufmann, San Francisco, California (1994)
10. Quinn, M.J.: Parallel Computing: Theory and Practice, 2nd Ed., McGraw-Hill, New York (1994)
11. Jelenkovic L. and Omecen-Ceko, G.: Experiments with Multithreading in Parallel Computing, Proc. 19 Int. Conf. on Information Technology Intercafes, Pula, Croatia (1997) 357-362
12. Beyer, D.A.: Memory Optimization for a Parallel Sorting Hardware Architecture, Thesis of MS, Electrical and Computer Engineering, Oregon State University (1998)