ELSEVIER

Information Processing Letters 83 (2002) 21–26

Information Processing Letters

www.elsevier.com/locate/ipl

A time-optimal distributed sorting algorithm on a line network

Atsushi Sasaki

NTT Communication Science Laboratories, NTT Corporation, Kyoto, Japan

Received 28 February 2001; received in revised form 11 October 2001
Communicated by K. Iwama

Abstract

We have achieved a strict lower time bound of $n-1$ for distributed sorting on a line network, where n is the number of processes. The lower time bound has traditionally been considered to be n because it is proved based on the number of disjoint comparison-exchange operations in parallel sorting on a linear array. Our result has overthrown the traditional common belief.

Keywords: Distributed sorting; Algorithms; Computational complexity; Distributed computing; Line network

1. Introduction

We have achieved a strict lower time bound of $n-1$ for distributed sorting on a line network, where n is the number of processes. The lower time bound has traditionally been considered to be n rounds because, in parallel sorting on a linear array, n steps has been proved based simply on the number of disjoint comparison-exchange operations [4]. We achieved the strict lower time bound of $n-1$ by creating copies of elements. Our result has overthrown the traditional common belief. This makes our result meaningful for both distributed and parallel algorithms.

Sorting is one of the most fundamental problems in computer science. Accordingly, it has also been studied in distributed contexts. There are various types of sorting problem. These problems include a static problem with a reliable network [2,5,10,11]; a dynamic problem [10]; a static problem on special network models such as a local area network [6], a broadcast network [8]; and a problem with an unreliable network [1,9]. We decided to tackle a static problem with a reliable network since this problem is the basis of all sorting algorithms.

Many algorithms have been designed and discussed for static distributed sorting with a reliable network [2, 5,10,11]. The main burden of these discussions has involved finding a way to reduce the amount of communication. In particular, Gerstel et al. [2] proved the lower bound of the bit complexity for distributed sorting on a rooted tree.

By contrast, Hofstee et al. [3] designed a time-optimal algorithm for a simple problem—sorting on a line network—with a restricted local memory. However, their algorithm has a restriction in that each process has to have at least two elements, i.e., their algorithm cannot solve a case where each process has exactly one element. Therefore, we aim to design an algorithm with the minimum time complexity for a simple problem with a restricted local memory when each process has exactly one element.

E-mail address: atsushi@cslab.kecl.ntt.co.jp (A. Sasaki).

0020-0190/01/$ – see front matter
PII: S0020-0190(01)00307-6

2. The model and the problem

A *line network* is defined as a linear collection of n processes $P_1, P_2, \ldots, P_n$ where $P_i, 1 < i < n$, is bi-directionally connected to P_{i-1} and P_{i+1}. Without loss of generality, we assume that the network is laid horizontally such that P_1 is at the endpoint on the left. Furthermore, we assume that each process knows its neighbors only by the local names of "*left*" and "*right*", with the orientation consistent along the line. An endpoint process knows that it is at an endpoint by the fact that one of the corresponding local names is *null*. Moreover, we assume that process P_i has no knowledge of i or n. Each process is equipped with a local memory, but it simultaneously has at most a constant number of elements.

Each process communicates with both neighbors. The process communications mainly involve the *synchronous* and *asynchronous* models [7]. This paper is mainly concerned with algorithms designed in accordance with the synchronous model because a synchronous algorithm can easily be extended to an asynchronous one. Two complexity measures are usually considered in the synchronous model: *time complexity* and *communication complexity*. The former is measured in terms of the number of rounds, while the latter is measured in terms of the total number of messages.

We define the distributed sorting problem adopted in this paper as follows. At the initial state, process P_i has a numerical element u_i for sorting. Then, the position of each element is arranged to satisfy the condition $\forall i,\ 1 \leqslant i < n,\ u_i \leqslant u_{i+1}$ at the final state.

3. Odd–even transposition sort [4]

The distributed sorting problem defined in the previous section is nearly the same as a parallel sorting problem on a linear array, which can be solved by using the *odd–even transposition sort*. If the odd–even transposition sort can be executed on our synchronous model, it will take the same number of rounds as the odd–even transposition sort. Accordingly, we try to design an algorithm based on the odd–even transposition sort. We first explain how the odd–even transposition sort works.

The odd–even transposition sort operates as follows: At an odd-numbered step, a process P_i whose suffix i is odd exchanges its element v_i with P_{i+1}'s element v_{i+1} if v_i is larger than v_{i+1}. At an even-numbered step, P_i whose suffix i is odd exchanges its element v_i with P_{i-1}'s element v_{i-1} if v_i is not larger than v_{i-1}. Each process iterates the above operations in n steps. An example of the execution of the odd–even transposition sort where $n = 5$ is shown in Fig. 1. In this figure, a double-headed arrow denotes a comparison-exchange operation of elements between two neighboring processes.

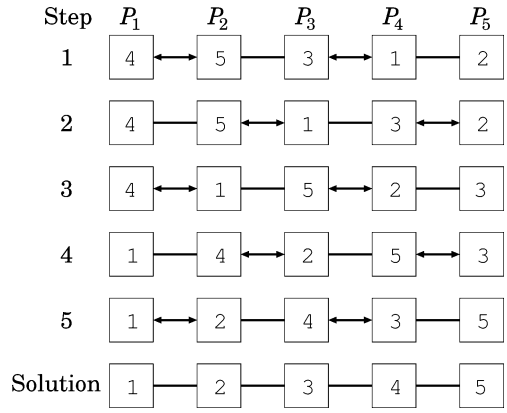

Fig. 1. Example: odd–even transposition sort.

4. A basic idea and a simple algorithm

We first consider whether the odd–even transposition sort can be applied to the distributed sorting problem. To apply the odd–even transposition sort as it is, P_i has to know its global position, i.e., i, because of the difference between the operation at an odd-numbered step and that at an even-numbered step depending on i. Although an operation can be executed to learn the global position in $n - 1$ rounds, these rounds would be regarded as overhead rounds. To avoid such an overhead, we adopt a strategy that does not require information on the global position, i.e., a strategy where each process communicates with both neighbors simultaneously as in Hofstee et al.'s algorithm [3]. However, this strategy requires at least two elements for each process to communicate with both neighbors despite each process having only one element. To achieve such a means of communication, we introduce the following new idea:

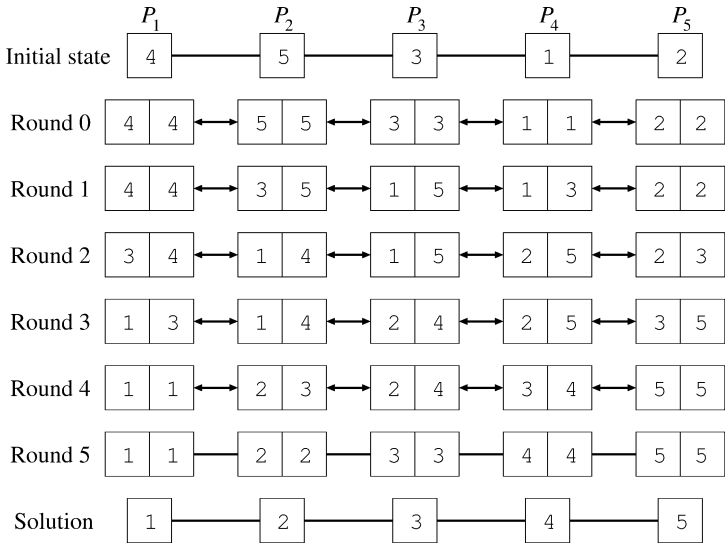

Fig. 2. Example: the n-round algorithm.

(1) provide two variables v_L and v_R for each process,
(2) $v_L := u_i$ and $v_R := u_i$ in each process, and
(3) execute the odd–even transposition sort using the two values of v_L and v_R.

By adopting this idea, each process always has two elements of the same value ($v_L = v_R$) at the final state. Therefore, by executing $u_i := v_L$ (or $u_i := v_R$) in each process at the final state, the sorting is completed correctly.

Fig. 2 shows an example of this algorithm's execution, where the initial state is the same as that in Fig. 1. This figure shows the values of u_i for process P_i at the initial state and as a solution and also the values of v_L and v_R from round 0 to round 5. In each process of each round in the figure, the left and right values denote v_L and v_R, respectively.

Then, we evaluate the complexities of the algorithm. This algorithm performs two steps of the odd–even transposition sort in one round because the even-numbered steps are executed within each process. Consequently, the time complexity is n rounds since the sort is executed with $2n$ elements.

5. A time-optimal algorithm

We designed an n-round algorithm for the distributed sorting problem in the previous section. In this section, we improve it to achieve $n-1$ rounds, which is an explicit lower bound of the time complexity.

In the n-round algorithm, the leftmost and rightmost elements are never sent to their neighboring processes. Therefore, the algorithm can be improved by cancelling the creation of copies in the leftmost and rightmost processes, i.e., each P_1 and P_n has only one element in every round. This implies that the number of elements used for sorting is $2n-2$. Accordingly, the time complexity of $n-1$ rounds can be achieved by using the odd–even transposition sort in the same way as in the n-round algorithm. However, this improved algorithm does not ensure that $v_L = v_R$ in each process in the final round. As a result, it cannot select one of these two elements in all processes in the same way. Furthermore, it is forbidden to take additional rounds to determine what can be selected.

In the following, we consider how to decide which element is selected in each process. We first observe the situations that result when the above improved algorithm is executed as it is. We then obtain a decision based on these observations. We finally describe practical operations necessary for the decision.

For the observation below, we make the following assumptions and definitions. First, we denote the ith minimum element among all elements as w_i, i.e., $\forall i,\ u_i = w_i$ should be held in the final state. Without

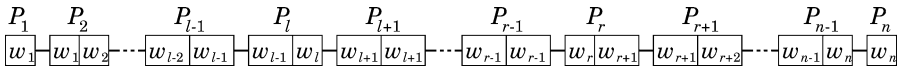

Fig. 3. The system state at the final round immediately before selecting the solution.

loss of generality, we assume that, in the initial state, $u_1 = w_\ell$, $u_n = w_r$, and each process has a distinct element, i.e., $\forall i,\ j \neq i,\ u_i \neq u_j$. We call a set of values of v_L and v_R at each round the system state.

In executing the improved algorithm as it is, for any distribution of initial elements where $w_\ell \leqslant w_r$, we obtain the system state in the final round without selecting elements in each process as shown in Fig. 3, in which the left and right elements in each process are stored in v_L and v_R, respectively. If $w_\ell > w_r$, the system state becomes one such that ℓ and r are exchanged from one where $w_\ell \leqslant w_r$. Therefore, we only argue the system state where $w_\ell \leqslant w_r$.

We regard the sequence of processes in the final round as being composed of three types of process:
(1) $P_1, P_2, \ldots, P_\ell$,
(2) $P_{\ell+1}, P_{\ell+2}, \ldots, P_{r-1}$, and
(3) $P_r, P_{r+1}, \ldots, P_n$.
Since P_i's solution has to be w_i, according to the system state shown in Fig. 3, the solution at a process in each type should be as follows:
(1) v_R has to be the solution.
(2) Either v_L or v_R can be the solution because $v_L = v_R$, but we decide that the process selects v_L as the solution for the uniform description.
(3) v_L has to be the solution.
Since the system state shown in Fig. 3 is always obtained for any distribution of elements in the initial state, the above selection rules for the solution are always useful.

These selection rules for the solution require knowing only where w_ℓ (P_ℓ) is. In order for each process to know where it is, the following operations are applied: First, at the initial state, P_1 and P_n mark their initial elements with *unique*, which always moves with the element, and each process is provided with a variable *area* that initializes to 0. Then, in process P_i, $\mathit{area} := \mathit{area} - 1$ if an element marked with *unique* is moved from P_{i-1} to P_i; $\mathit{area} := \mathit{area} + 1$ if an element marked with *unique* is moved from P_i to P_{i-1}. In particular, we consider P_1 to have received its initial element from *left* and $\mathit{area} := \mathit{area} - 1$ in round 0.

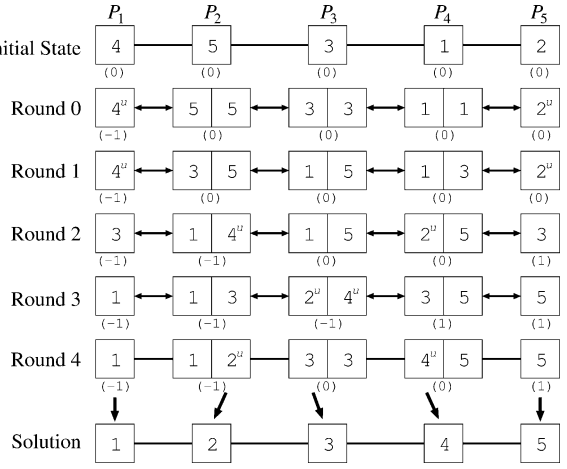

Fig. 4. Example: the $(n-1)$-round algorithm ("u" denotes the mark of *unique*, each number in parentheses denotes the value of *area* in each process).

According to these operations, in the final round, a process recognizes that it is in $\{P_1, P_2, \ldots, P_\ell\}$ if $\mathit{area} = -1$, and in $\{P_{\ell+1}, P_{\ell+2}, \ldots, P_n\}$ if $\mathit{area} = 0$ or 1. The recognition is correct for all processes because w_ℓ and w_r only move to *right* and *left*, respectively, which can be easily proved; we omit the proof because of space limitations. Concluding the argument, we obtain the following rules for process P_i to select an element as the solution:
(1) if $\mathit{area} = -1$, P_i selects v_R,
(2) if $\mathit{area} = 0$ or 1, P_i selects v_L.
In this way, we have been able to design a time-optimal algorithm whose time complexity is $n - 1$.

Fig. 4 shows an example of this algorithm's execution. The initial state is the same as that in Figs. 1 and 2, and the meaning of each character is the same as in Fig. 2.

Finally, we show the algorithm in the following. Each process has to know n for the termination detection, which is initially unknown. Therefore, the actual algorithm is more complex than the following because operations to compute n should be simultaneously executed with the following.

The $(n-1)$-round algorithm:

1. Definitions of the primitive internal operations for P_i:
   receive(x, P): receives a message from P, then $x :=$ the message's element with its mark.
   send(x, P): sends element x with its mark to process P.
   swap(a, b): swaps the elements of variables a and b.
   sleep: finishes the execution of this algorithm.
2. Local variables for P_i:
   u: an initial element or a sorted element, initially the initial element.
   v_L, v_R: variables for storing elements with their marks, initially undefined.
   v'_L, v'_R: variables for received elements with their marks, initially undefined.
   t: time (round), initially 0.
   area: a variable to decide whether v_L or v_R is to be selected, initially 0.
   n: the number of processes in the network.
3. Operations in round 0 for P_i:
   if *left* = *null* then (at P_1)
      $v_L := -\infty$
      $v_R := u$ marked with *unique*
      area := *area* $- 1$
   else if *right* = *null* then (at P_n)
      $v_L := u$ marked with *unique*
      $v_R := \infty$
   else
      $v_L := u$
      $v_R := u$
   send$(v_L, left)$
   send$(v_R, right)$
   $t := t + 1$
4. Operations after round 0 for P_i:
   if *left* $\neq$ *null* then
      receive$(v'_L, left)$
      if $v'_L > v_L$ then
         if v'_L is marked with *unique* then
            area := *area* $- 1$
         if v_L is marked with *unique* then
            area := *area* $+ 1$
         $v_L := v'_L$
   if *right* $\neq$ *null* then
      receive$(v'_R, right)$
      if $v_R > v'_R$ then
         $v_R := v'_R$
   if $v_L > v_R$ then
      swap(v_L, v_R)
   if $t < n - 1$ then
      send$(v_L, left)$
      send$(v_R, right)$
      $t := t + 1$
   else (selection of the solution)
      if *area* $= -1$ then (corresponds to rule 1)
         $u := v_R$
      if *area* $= 0$ or *area* $= 1$ then
         (corresponds to rule 2)
         $u := v_L$
      sleep

6. Conclusion

We have achieved a strict lower time bound of $n-1$ for distributed sorting on a line network by employing a new idea. Despite this idea of creating nearly double the number of elements, the algorithm is faster than the odd–even transposition sort. This is a very promising characteristic. Our algorithms can be executed with both synchronous and asynchronous models by simply coping with wakeup. The $(n-1)$-round algorithm can also have a great impact in the field of parallel algorithms.

Acknowledgements

The author is grateful to Professor Nancy A. Lynch of MIT for giving him the motivation to carry out this study. He thanks Professor Shigeru Masuyama of Toyohashi University of Technology, Dr. Shin-ichi Nakayama of Tokushima University, Dr. Kiyoshi Kogure, Dr. Yoshifumi Manabe, Mr. Ken Mano, Dr. Kenichiro Ishii, Mr. Yoshifumi Ooyama and Dr. Kiyoshi Shirayanagi of NTT Communication Science Laboratories for their discussions, suggestions and support of this study. Finally, he thanks the editor and the reviewers for their useful comments.

References

[1] G. Alari, J. Beauquier, J. Chacko, A.K. Datta, S. Tixeuil, A fault-tolerant distributed sorting algorithm in tree networks,

in: Proc. 1998 IEEE Internat. Performance, Computing and Communications Conf., 1998, pp. 37–43.
[2] O. Gerstel, S. Zaks, The bit complexity of distributed sorting, Algorithmica 18 (3) (1997) 405–416.
[3] H.P. Hofstee, A.J. Martin, J.L.A. van de Snepscheut, Distributed sorting, Sci. Comput. Programming 15 (2–3) (1990) 119–133.
[4] F.T. Leighton, Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, Morgan Kaufmann, San Mateo, CA, 1992.
[5] M.C. Loui, The complexity of sorting on distributed systems, Inform. and Control 60 (1–3) (1984) 70–85.
[6] W.S. Luk, F. Ling, An analytic/empirical study of distributed sorting on a local area network, IEEE Trans. Software Engrg. 15 (5) (1989) 575–586.
[7] N.A. Lynch, Distributed Algorithms, Morgan Kaufmann, San Mateo, CA, 1996.
[8] J.M. Marberg, E. Gafni, Distributed sorting algorithms for multi-channel broadcast networks, Theoret. Comput. Sci. 52 (3) (1987) 193–203.
[9] B.M. McMillin, L.M. Ni, Reliable distributed sorting through the application-oriented fault tolerance paradigm, IEEE Trans. Parallel Distrib. Systems 3 (4) (1992) 411–420.
[10] D. Rotem, N. Santoro, J.B. Sidney, Distributed sorting, IEEE Trans. Comput. C-34 (4) (1985) 372–376.
[11] S. Zaks, Optimal distributed algorithms for sorting and ranking, IEEE Trans. Comput. C-34 (4) (1985) 376–379.