Fault-tolerant parallel $k$ selection algorithm in $n$-cube networks

Jang-Ping Sheu

Department of Electrical Engineering, National Central University, Chungli 32054, Taiwan, ROC

Communicated by K. Ikeda
Received 26 October 1990
Revised 5 February 1991

Abstract

In this paper, a parallel algorithm to select the first $k$ largest numbered processes in $n$-cube networks is proposed. The proposed algorithm can tolerate at most $n - 1$ faults. The time complexity of this algorithm is $O(\max(kn, n^2))$.

Keywords: Parallel algorithms, $k$ selection, fault-tolerant, $n$-cube networks

1. Introduction

Election is the problem of choosing a unique processor as the leader of a network of processors. The election problem was first discussed by LeLann [2] in a ring connection network to elect a new leader as responsible for regenerating a new control token after the previous token is lost in the ring. Selecting the first $k$ largest process numbers in a network is called the $k$ selection problem. The objective of the $k$ selection problem is that the largest process number is the leader and the other process numbers are used as standby. If the leader fails, the next largest can immediately take control without restarting the election algorithm.

Sheu and Tang [4] propose a parallel $k$ selection algorithm in the $n$-cube networks with time complexity $O(\max(k, n^2))$ which is optimal when $k \geq n^2$. Sheu, Wu, and Chen [5] present a fault-tolerant parallel $k$ selection algorithm in the $n$-cube networks. Their algorithm can tolerate at most two faulty nodes and the time complexity of the algorithm is $O(k(n - \log k) + k)$.

In this paper, we shall propose a fault-tolerant parallel algorithm to determine the first $k$ largest process numbers in the $n$-cube networks. The proposed algorithm can tolerate at most $n - 1$ node/link faults. The time complexity of the algorithm is $O(\max(kn, n^2))$.

2. Fault-tolerant parallel $k$ selection algorithm

In this section, we propose a fault-tolerant parallel algorithm for the $k$ selection problem in an $n$-cube network. The $n$-cube network is a hypercube of dimension $n$. It consists of $N = 2^n$ identical nodes so that every node is a general-purpose processor with its own local memory. Every node is numbered from 0 to $2^n - 1$ by an $n$-bit binary number $(a_n a_{n-1} \ldots a_1)$. If two nodes

* This work was supported by the National Science Council under Grant NSC 79-0408-E-008-01.
have node numbers different in $a_j$ only, then they are called opposite ones in the $j$th direction [3]. Every node has a direct link to the opposite node. So each node has $n$-neighbor nodes with direct links.

In order to study the efficiency of different algorithms, we use not only the computation time but also the communication time as the measure in any execution of these algorithms. We measure the communication steps because they reflect the communication overhead on the communication system of the $n$-cube networks. The following assumptions are used to analyze the algorithm complexity:

1. Moving a fixed-sized data packet from one node to a neighbor one figures as a message complexity and takes a unit time;
2. Moving the same data packet from one node to any one of its $n$ neighbors takes the same time and message complexity;
3. The data communication is bidirectional and each node has a data list of size $O(k)$.

In general, we might consider the nodes and links faults in the $n$-cube networks. Node and link failures complicate the selection problem. Once a node or link fails, it never sends any other message. The maximum number of faulty nodes/links that our algorithm can tolerate is $n - 1$. The idea of our algorithm is simply described as follows.

First, we consider the case of $k > n$. Initially, every node sends its residing process number to $n$-outgoing channels and marks the sent process number in its data list. Every node of the $n$-cube will receive $n$ process numbers from its $n$-neighbors.

Algorithm FPKS (Fault-tolerant parallel $k$ selection algorithm).

Step 1: /* Initial step */
1. Initially, each node sends its residing process number to $n$-outgoing channels and marks the sent process number in its data list. Every node receives $n$ process numbers from its $n$-neighbors.
2. If $k < n$, then let $l = n + k - 1$, else $l = 2n - 1$.

Step 2: For $i = 1$ to $l$ do the following operations:
1. Each node receives at most $\max(n, n[k/n])$ process numbers from its $n$-neighbors. If any of these process numbers has not been received yet, then puts it into the rear of the data list; otherwise discards it.
2. Each node finds the $\min(k, s)$-th largest process number and keeps the first $\min(k, s)$ largest process numbers in its data list and discards the others, where $s$ is the current size of the data list.
3. Every node sends the first $\max(1, \lceil k/n \rceil)$ largest unmarked process numbers of the data list to its $n$-neighbors and marks these sent process numbers.

Step 3: Finally, each node sorts the $k$ marked process numbers of the data list in descending order.

In the following, we give an example to show how our algorithm works.
Fig. 1. Data exchanges and data lists for $k = 3$ selection in a 3-cube.
Example 2.1. We use a 3-cube to show how to get the first three largest process numbers.

**P**<sub>i</sub>: Represents the process number residing in node **i**.

**D**<sub>i</sub>: Represents a data list which will be used to store the first **k** largest process numbers at node **i**.

Initially, **P**<sub>i</sub> and **D**<sub>i</sub> of node **i** are shown in Fig. 1(a). After Step 1, each node receives three process numbers from its 3-neighbors. For example, node 0 receives process numbers 4, 5, and 2, and hence the data list **D**<sub>0</sub> = {8*, 5*, 4}. The mark "*" denotes that data have been sent from the data list and the mark "**" denotes that data will be sent in the next iteration. The other nodes also get their data lists. The result is shown in Fig. 1(b). After the second iteration, node 0 receives process number 8 from nodes 1, 2, and 4. Therefore, the process number in its data list remains unchanged. Figures 1(c)-(f) show the actions from iteration 2 to iteration 5. The final result is shown in Fig. 1(f).

The computation time is consumed in finding the first **k** largest process numbers and checking whether these process numbers have been received or not. In Step 2, each node receives at most \( \max(n, n[k/n]) \) process numbers from its **n**-neighbors. Therefore, the size of the new data list is no more than \( k + n \). In our algorithm, we must keep the first **k** largest process numbers in the data list. It is easy to solve it by the following steps. We first select the **k**th highest elements \( l_k \) in the data list. Next, we scan each element of the data list and keep the one in the data list if it is greater than \( l_k \). Finding the **k**th largest number from \( k + n \) elements takes \( O(\max(k, n)) \) time units [1]. Since each iteration needs \( O(\max(k, n)) \) computation time, the total computation time complexity is \( O(\max(kn, n^2)) \). Therefore, the time complexity of our algorithm is \( O(\max(kn, n^2)) \).

3. Correctness proof of the FPKS algorithm

In this section, we shall show our algorithm is correct and the algorithm can tolerate at most \( n - 1 \) faults in an **n**-cube network. For the sake of easy description, we define the following term. If two unmarked process numbers, \( u \) and \( v \), have been sent to a node at the same time, then \( u \) will be sent before \( v \) when \( u > v \) and we say that \( v \) is delayed one step by \( u \).

**Lemma 3.1.** Let \( n, n_1, \ldots, n_{i+d} \) denote a shortest path from node \( n_i \) to node \( n_{i+d} \) with distance \( d \). In the FPKS algorithm, if the **k**th largest process number reaches node \( n_i \), then it will be sent to node \( n_{i+d} \) at most delayed \( k - 1 \) steps by some of the first \( k - 1 \) largest process numbers when \( k \leq n \).

**Proof.** We shall prove it by induction as follows.

**Basic** \( k = 1 \): Clearly, it is correct.

**Induction hypothesis** \( k = q \): The **q**th largest process number that reaches node \( n_i \) can be sent to node \( n_{i+d} \) at most delayed \( q - 1 \) steps.

**Induction** \( k = q + 1 \):

**Case 1:** The \((q + 1)\)th largest process number is not delayed by the **q**th largest process number in the path from node \( n_i \) to node \( n_{i+d} \). Then the \((q + 1)\)th largest process number can be treated as the **q**th largest process number. By the induction hypothesis, it holds.

**Case 2:** The \((q + 1)\)th largest process number is delayed at least once by the **q**th largest process number and the last one delay is in node \( n_{i+j} \) \((j < d)\). Assume the **q**th largest process number reaches the node \( n_{i+d} \) \((m + 1)\) steps earlier than the \((q + 1)\)th largest process number. Thus, the \((q + 1)\)th largest process number is delayed \( m \) steps by one of the first \( q - 1 \) largest process numbers after the **q**th largest process number left the node \( n_{i+j} \). In addition, the **q**th largest process number cannot be delayed by any process number from node \( n_{i+j+1} \) to node \( n_{i+d} \). Once the **q**th largest
process number is delayed at node \( n_{i+p} \), \( j < p < d \), it will be caught up by the \( (q + 1) \)th largest process number in the node \( n_{i+p} \). This is in contradiction to the assumption of Case 1.

Moreover, in the path from \( n_i \), to \( n_{i+j} \), the \( q \)th largest process number is not delayed by any one of the process numbers which delay the \( (q + 1) \)th largest process number in the path from \( n_{i+j} \) to \( n_{i+d-1} \). If the \( q \)th process number is delayed by these process numbers in the path from \( n_i \) to \( n_{i+j} \), they cannot delay the \( (q + 1) \)th largest process number in the path from \( n_{i+j} \) to \( n_{i+d-1} \). Hence, the \( q \)th largest process number can be treated as the \( (q - m) \)th largest process number. By the induction hypothesis, it can only be delayed in \( q - m - 1 \) steps to send the \( q \)th largest process number from node \( n_i \) to node \( n_{i+d} \). Therefore, the \( (q + 1) \)th largest process number will be delayed at most \( q \) steps from node \( n_i \) to node \( n_{i+d} \). □

**Theorem 3.2.** The FPKS algorithm is correct if \( k \leq n \).

**Proof.** The length of the shortest path between any two nodes in an \( n \)-cube is at most \( n \). By Lemma 3.1, each node will get the first \( k \) largest process numbers after \( n + k - 1 \) steps. □

In the following, we shall prove that the FPKS algorithm is correct if \( k > n \). Without loss of generality, let \( k = \delta n \), where \( \delta \) is an integer. Then we can divide the first \( k \) largest process numbers into \( n \) groups, \( G_1, G_2, \ldots, G_n \), and each group \( G_i \) contains \( \delta \) process numbers. Let the process numbers in group \( G_i \) be larger than those in group \( G_{i+1} \), where \( 1 \leq i \leq n - 1 \).

**Lemma 3.3.** Let \( n_n, n_{n+1}, \ldots, n_{n+d} \) denote a shortest path from node \( n_i \) to node \( n_{i+d} \). In the FPKS algorithm, if the process numbers in group \( G_i \) reach node \( n_i \), they will be sent to node \( n_{i+d} \) at most delayed \( i - 1 \) steps by some of the process numbers in groups \( G_1, G_2, \ldots, G_{i-1} \), where \( 2 \leq i \leq n \).

**Proof.** When \( k = \delta n \), each node sends the first \( \delta \) unmarked process numbers instead of the maximal one. The proof of this Lemma is similar to that of Lemma 3.1. We omit the details. □

**Theorem 3.4.** The FPKS algorithm is correct if \( k > n \).

**Proof.** The length of the shortest path between any two nodes in an \( n \)-cube is at most \( n \). By Lemma 3.3, each node will get the first \( k \) largest process numbers after \( 2n - 1 \) steps. □

For an \( n \)-cube network, there is at least one shortest path of length no more than \( n + 1 \) from any node to any other node if the number of faults is no more than \( n - 1 \) [3]. Therefore, any one of the first \( k \) largest process numbers located in node \( n_i \) can reach to any node \( n_j \) of an \( n \)-cube at most in \( (n + 1) \) steps if there exists a shortest path between nodes \( n_i \) and \( n_j \). By Lemma 3.1 and Lemma 3.3, the \( k \)th largest process number is at most delayed by \( n - 1 \) steps. Hence, the \( k \) selection problem can be solved in \( 2n \) steps when there are \( n - 1 \) faults in the \( n \)-cube networks.

**4. Conclusions**

The advantage of selecting the first \( k \) largest process numbers is that every node knows where the first \( k \) largest process numbers are, and further, the ranks of them can be determined. The time complexity of the FPKS algorithm is \( \Omega(\max(\delta n, n^2)) \). The maximal number of faults that the algorithm can tolerate is \( n - 1 \).

**References**


