JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **32,** 223–231 (1996)
ARTICLE NO. 0016

Effective Utilization of Hypercubes in the Presence of Faults[1]

GUANGHUA LIN AND NIAN-FENG TZENG[2]

Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, Louisiana 70504

To effectively utilize a faulty hypercube, it is often necessary to reconfigure the hypercube in such a way as to retain as many fault-free nodes as possible. This inspires us to identify maximal *incomplete subcubes* in a faulty hypercube, as the subcube so reconfigured is often much larger than any complete subcube obtainable and is likely to retain performance better. An efficient algorithm is first presented to find incomplete subcubes in a faulty hypercube. Three applications are then implemented on both an incomplete system and a complete one to measure their actual performance differences. Each application is mapped onto incomplete hypercubes, following the techniques developed for complete hypercubes (possibly with some modifications). Among the three applications, *Gaussian elimination* takes exactly the same mapping scheme on an incomplete system as on a complete one, while *FFT* requires some efforts to adapt it to the incomplete topology. The measured results of the three applications indicate that reconfiguring a faulty hypercube into an incomplete subcube is beneficial in practice.

1. INTRODUCTION

For a large hypercube system, the probability of fault occurrence can be high and it is therefore necessary to consider effective utilization of a system after faults occur. In this paper, we consider reconfiguring a faulty hypercube with an effort to keep as many fault-free nodes as possible, called *maximum reconfiguration,* which often leads to an *incomplete hypercube* (to be defined formally in Section 2), and then validate the benefit of our maximum reconfiguration through various experiments on the Intel *iPSC*/860 by mapping applications onto incomplete hypercubes. Our reconfiguration begins with finding all maximum fault-free complete subcubes in a faulty hypercube using an efficient procedure, followed by identifying maximum collections of complete subcubes of distinct sizes such that elements in each collection together form an incomplete cube.

A reconfigured maximum incomplete subcube is often much larger than a reconfigured maximum complete subcube because the former always involves a copy of the latter *plus* some smaller subsystem(s). Simple and deadlock-free algorithms for routing and broadcasting messages in an incomplete hypercube have been developed [4]. A recent study on the incomplete hypercube system suggests that no congestion point exists in such a system [8]. In this work, three applications are considered to measure the degree of benefit with maximum reconfiguration. They are *Gaussian elimination, mixed sort* (a mixture of the shell sort and an odd–even transposition sort), and *FFT.* For *Gaussian elimination* and *mixed sort,* our implementations follow a ring topology on both complete and incomplete systems, thus employing the same mapping scheme on an incomplete system as on a complete one. For *FFT,* however, some efforts are required to adapt it to the incomplete topology, leading to a slightly different mapping scheme on an incomplete system than on a complete one. Our experimental results show that for *Gaussian elimination,* an incomplete system of size 12 may outperform its complete counterpart (i.e., a complete system of size 8) by as much as 49%. This is expected as *Gaussian elimination* is computation-intensive. For *mixed sort,* an incomplete system of size 12 may outperform its complete counterpart by 30%. At the other extreme, *FFT* is communication-intensive following the butterfly communication pattern, and an incomplete system then achieves a less impressive performance gain.

Note that if a system attempts to maintain all the fault-free nodes without assuring the incomplete topology, it is impossible to have simple, deterministic routing and broadcasting algorithms. In fact, unlike an incomplete hypercube, this often requires complicated node-to-node routing and broadcasting, which do not guarantee deadlock-freeness. Moreover, traffic density over a link is not bounded by a small constant in this situation, possibly causing congestion points to occur. It is thus more advantageous to reconfigure a faulty hypercube into a maximum incomplete subcube than to maintain all the healthy nodes without reconfiguration.

2. BACKGROUND AND DEFINITIONS

An n-dimensional complete hypercube, H_n, comprises 2^n nodes, each with n links connected directly to n other nodes, called neighbors. In H_n, a node is labeled by an n-

[1] This material is based upon work supported in part by the NSF under Grants MIP-9201308 and CCR-9300075 and by the State of Louisiana under Contract LEQSF(1994-96)-RD-A-39. A preliminary version of this article was presented at the 9th International Parallel Processing Symposium, April 1995.

[2] E-mail: tzeng@cacs.usl.edu.

0743-7315/96 $18.00

bit binary string $b_{n-1}b_{n-2}\cdots b_1b_0$, and a link is labeled by an n-tuple $\{0, 1, *\}^n$, which contains exactly one *don't care* "*" at a coordinate position. Two end nodes of the link can be uniquely determined by assigning "0" and "1" to the * coordinate (in other words, a link connects two nodes whose labels differ in exactly one bit position). A k-dimensional subcube in H_n, S_k, is labeled by an n-tuple $\{0, 1, *\}^n$ such that there are exactly k ($k < n$) * positions in its label.

Incomplete hypercubes, introduced in [4], retain many interesting properties of complete hypercubes, and at the same time, can be of arbitrary size. An n-dimensional incomplete hypercube of size $\mathscr{M}$, $I_n^{\mathscr{M}}$, is defined recursively as follows: (1) $I_n^{\mathscr{M}} = H_{n-1} + I_k^{\mathscr{M}-2^{n-1}}(k < n)$; (2) a link exists between a node in H_{n-1} and another node in $I_k^{\mathscr{M}-2^{n-1}}$ if and only if their labels differ in exactly one bit; (3) I_0^1 (the base case) is a 0-dimensional incomplete hypercube of size 1.

The incomplete hypercubes under consideration are reconfigured from a complete hypercube with faulty nodes and/or faulty links. To simplify subsequent explanation, a subcube is represented by its label, and the term "subcube" refers to a *complete* subcube unless stated otherwise. Two subcubes of the same dimension are said to be *lateral* with parameter δ if the * positions in their labels are exactly the same and the remaining bits differ in δ and only δ positions. A subcube involving fault(s) is said to be a co-subcube (*COS* for short) of a fault-free subcube if they are lateral with $\delta = 1$. A *COS* contains at least one fault (either a faulty link or a faulty node). A fault-free subcube may have multiple *COS*s, or may have no *COS* at all (e.g., when a faulty hypercube contains only two fault-free subcubes, and all links between them are faulty).

An incomplete hypercube is essentially a *collection* of fault-free subcubes of distinct dimensions (i.e., no two constituent subcubes have the same size). A collection corresponding to an n-dimensional incomplete hypercube, C_n, is defined recursively as follows: (1) $C_n = H_{n-1} \cup C_k$ ($k < n$); (2) C_k is a collection *resided* in a *COS* of H_{n-1}. Therefore, a collection, C_n, is actually derived from a pair of fault-free subcube H_{n-1} and its *COS*. The problem of identifying maximum incomplete subcubes in a faulty H_n is equivalent to finding maximum collections of fault-free subcubes with distinct dimensions.

3. IDENTIFYING MAXIMUM INCOMPLETE SUBCUBES

Since a collection of a faulty H_n, C_p (with $p \le n$), comes from a pair of H_{p-1} and its *COS,* a maximum collection must be derived from such a pair that involves a maximum (fault-free) complete subcube (*MCS* for short) in H_n and a *COS* of the *MCS*. Such a pair, (*MCS, COS*), is of interest to us and is referred to as an *M–C pair.* Our objective is to examine *M–C* pairs and find the maximum collections. A maximum collection is called a *collection* for simplicity.

Let d be the dimension of the *MCS* in a faulty H_n; then an *M–C* pair in H_n actually resides *within* a $(d + 1)$-dimensional subcube. A $(d + 1)$-dimensional subcube may involve multiple *M–C* pairs. It can be shown that the total number of *M–C* pairs involved in a $(d + 1)$-dimensional subcube in H_n is no more than $(d + 1)$ (see Lemma 1 in [9] for a proof). The sizes of the collections corresponding to the respective *M–C* pairs involved in a $(d + 1)$-dimensional subcube are of the same size, as otherwise, it violates the condition that the collections are of the maximum size. Therefore, examining *only one* of these *M–C* pairs for a collection suffices. Several *MCS*s in a faulty hypercube may share a common *COS*. In general, the number of *MCS*s that share a common *COS* can be up to $(n - d)$, as described by Lemma 2 of [9]. One need *not* examine all of the *M–C* pairs in a faulty H_n, but selects only *a portion* of them for generating collections, resulting in a considerably reduced search space. The maximum number of the selected *M–C* pairs in H_n can be proved to be

$$\frac{1}{n-d}\binom{n}{d+1}2^{n-d-1}.$$

Finding Complete Subcubes

For a general n-tuple $\{0, 1, *\}^n$, there are exactly $\binom{n}{n-k}$ different $(n - k)$-bit position combinations, each having 2^{n-k} combinations of $\{0, 1\}^{n-k}$. Thus, by assigning "*" to the remaining k positions, we have a total of $\binom{n}{n-k}2^{n-k}$ different labels, each corresponding to a k-dimensional subcube in H_n. An efficient algorithm for finding all fault-free complete subcubes in a hypercube involving *only faulty nodes* can be developed based on the observation [9] that if a set of faulty nodes, F', covers (or contains) a combination of $\{0, 1\}^{n-k}$ at $(n - k)$ given bit positions, the subcube whose label consists of the combination at those $(n - k)$ given bit positions and * at the remaining k positions involves at least one faulty node. It is clear that if all the 2^{n-k} combinations at each set of the $(n - k)$ bit positions in faulty H_n are exhaustively covered by F', then every S_k involves at least one faulty node, and there is no fault-free subcube with dimension k or larger. However, if a combination at any $(n - k)$ bit positions is absent, i.e., not covered by the set of faulty nodes, a fault-free subcube can be identified by assigning the combination to those particular $(n - k)$ bit positions and * to the remaining k positions. Therefore, the problem of identifying all *MCS*s in the presence of a given set of faulty nodes is equivalent to finding all the combinations absent from each set of $(n - k)$ bit positions, starting with $k = n - 1$.

A faulty link which does not reside *within* a subcube has no damaging impact on the subcube. This suggests a modification to the preceding algorithm to deal with the case where fault set F involves both faulty nodes and faulty links, as follows. In the search for combination(s) absent from a set of bit positions, faulty links which contain * at that set of bit positions are ignored. Moreover, if a combination is covered by F at a given $(n - k)$ bit positions, the subcube whose label consists of the combination at those given $(n - k)$ bit positions and * at the remaining

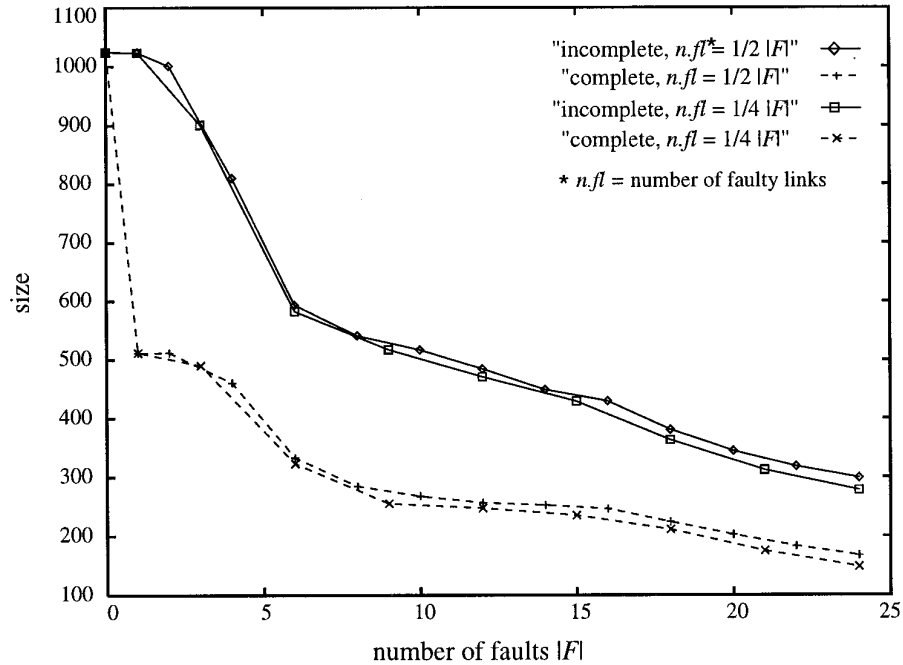

FIG. 1. Simulated size vs number of faults in a 10-dimensional hypercube.

k positions contains at least one fault, either a faulty link or a faulty node. On the other hand, if a combination at any $(n - k)$ bit positions is absent, a fault-free subcube has been found.

Maximum Incomplete Subcube Identification

The algorithm for identifying maximum incomplete subcubes builds on the above algorithm for finding complete subcubes, as described below. First, all *MCS*s present in H_n are determined by using the above algorithm. Then, for each *MCS* determined, we search for its *COS*s. If no *COS* is found (recall that an *MCS* may have no *COS*), the algorithm terminates and all the *MCS*s determined are the maximum incomplete subcubes. Otherwise, we identify all possible *M–C* pairs (recall that an *MCS* may have more than one *COS*), but select only a subset of the *M–C* pairs for examination. In the next step, for each one of the selected *M–C* pairs, the *MCS* itself is assigned to the collection currently in search, with the *COS* treated as a faulty subcube. In addition, for each faulty link with one of its two end nodes involved in the *MCS* of the selected *M–C* pair, and the other in the *COS* of the pair, the end node in the *COS* is regarded as a faulty node (to reflect the effect of the faulty link) in order to prevent such a faulty link from being included in any candidate incomplete subcube. The above algorithm is then employed to identify *MCS*s *residing* in the *COS*. All possible *M–C* pairs *residing* in the *COS* (called *residual M–C* pairs) are recognized but only a subset of them are selected for further examination. For each chosen residual *M–C* pair, the *MCS* itself again is appended to the collection, with its *COS* examined to determine residual *M–C* pairs as before. This recursive search process continues until no further subcube can be added to the collection. When the current search terminates, the size of the generated collection is compared to the size of the maximum collections generated so far, and the bigger one is recorded as the current maximum size. If the newly generated collection has the same size, it is kept along with the previously generated maximum collections. As a result, the search carried out for an *M–C* pair in H_n gives rise to one or multiple collections. The examination of other *M–C* pairs in H_n is carried out one by one until all candidate *M–C* pairs are exhausted. After completion of the entire search, the collections obtained are the maximum incomplete subcubes.

Reconfiguration Results

Extensive simulations were carried out to examine our maximum reconfiguration results of a 10-dimensional hypercube. For each given number of faults $|F|$, the following two types of fault sets are considered: (1) the number of faulty nodes is equal to the number of faulty links in F; (2) the number of faulty links in F is one quarter of $|F|$. Figure 1 plots the mean size of maximum incomplete subcubes as a function of $|F|$, where the average size of maximum complete subcubes is also given for comparison. The sizes in Fig. 1 are averages over 3000 repetitions. From the simulated results, it can be seen that for a given fault pattern, a maximum incomplete subcube is considerably larger than its maximum complete counterpart. In fact, the former always have an average size roughly two times as large as that of the latter. It is also observed that, as the number of faulty nodes increases (or equivalently, the number of faulty links decreases), reconfigured system

sizes decrease slightly under both complete and incomplete reconfigurations.

As system performance tends to be proportional to the computational power, a reconfigured incomplete system with more nodes is likely to retain performance better. It should be noted, however, that a larger communication overhead may be present in the incomplete system, due to additional communication overhead caused by (1) the finer granularity incurred by solving the same sized problem on more nodes and (2) mapping an application onto what is now a potentially different topology. It is thus interesting to investigate the actual performance levels on the incomplete hypercubes, taking into account both increased computational power and possibly higher communication overhead.

4. EXPERIMENTAL STUDIES

To study the actual performance levels on incomplete hypercubes, we map applications running on complete hypercubes onto incomplete hypercubes, contrasting performance differences on an incomplete hypercube and its complete counterpart. The following application are considered: *Gaussian elimination, mixed sort* (a mixture of the shell sort and an odd–even transposition sort), and *FFT*. These applications are chosen because (1) they are well known and (2) they involve different computational complexities and communication patterns. Although extra effort may be required in some cases to adopt the applications to the incomplete topology, it is our hope that such efforts are minimal or even can be avoided for most of the applications (and thus, an application developed for complete hypercubes may be extended to reconfigured incomplete subcubes with minor modifications (if any), after faults arise). The experiments for each application are conducted on an Intel *iPSC*/860 with 32 nodes.

The following notations are employed to facilitate subsequent explanation.

T.exec. System execution time, determined by the slowest node in the system, i.e., $T.exec = \max(t.exec(i), 0 \le i \le \mathcal{M} - 1)$, where $t.exec(i)$ is the execution time on the ith node of the system and $\mathcal{M}$ is the total number of nodes in the system.

T.comu. System average communication time, determined as $T.comu = (1/\mathcal{M}) \sum_{i=0}^{\mathcal{M}-1} t.comu(i)$, where $t.comu(i)$ is the total communication (*send & recv*) time on the ith node of the system. Note that $t.comu(i)$ may include some waiting time if the message to be received has not yet arrived when needed.

In the following, we differentiate the size of an incomplete system $\mathcal{M}^+$ from the size of a complete system $\mathcal{M}$, if such differentiation is necessary; otherwise, $\mathcal{M}$ is used to indicate the size of a hypercube system, whether complete or incomplete.

4.1. Gaussian Elimination

Consider a nonsingular $N \times N$ matrix $A \in \Re$. Gaussian elimination (without pivoting[3]) is achieved by repeatedly applying the following basic matrix computation to A,

$$\mathrm{a}_{ij} = a_{ij} - a_{ik} \times \frac{a_{kj}}{a_{kk}}, i = k, \cdots, N-1; \quad j = k+1, \cdots, N-1, \tag{1}$$

starting with $k = 0$, where $0 \le k \le N - 1$. Particularly, if $i = k$, (1) becomes

$$a_{ij} = 0, i = k; \quad j = k+1, \cdots, N-1. \tag{2}$$

We refer to the operations defined in (1) and (2) as *update* and *zerout,* respectively.

Parallel Implementations

The parallel implementations of Gaussian elimination on a complete cube and an incomplete cube are exactly the same, as follows. A ring topology is used for the implementations of Gaussian elimination on both complete and incomplete systems (a ring with an even number of nodes can be easily embedded in a complete hypercube [1] or an incomplete hypercube with an even number of nodes [6]). To achieve good load balance, matrix A is distributed across the nodes on the ring in an *interleaved* manner; i.e., row i $(0 \le i \le N - 1)$ is assigned to node P_j if $j = i \bmod \mathcal{M}$.

Initially, each node has $(N/\mathcal{M})$ rows to be processed. At the first repetition, P_0 (sender) broadcasts[4] a copy of its first resident row to all the other nodes (receivers) on the ring, and then zerouts the first row and updates all the rows below it. Among the receivers, P_1 receives the message from P_0 and realizes that it will be the next sender itself. So it first updates the first row using the message received, and then *immediately*[5] forwards a copy of this updated row to all the other nodes on the ring (including P_0), with the updates of the remaining rows followed. On receiving the message, all the other receivers update their respective resident rows individually, using the messages received. In the next repetition, P_1 just zerouts its first row and updates all the rows below it, involving no communication with any other node. For the rest of nodes, P_2 follows the same process as P_1 just went through in the last repeti-

[3] The computational requirements of Gaussian elimination with pivoting are similar, and the parallel implementation described here can be easily extended to that with pivoting.

[4] On a complete system, broadcasting is done in a binomial tree manner which requires $\log \mathcal{M}$ steps of message transmission [7]. For an incomplete system, broadcasting can be achieved in the same way as that in a complete system, because the spanning tree corresponding to an incomplete cube is nothing but the original binomial spanning tree with certain branches removed.

[5] This is to make sure that the message arrives at the corresponding nodes as early as possible, in order to eliminate or reduce waiting time at each node.

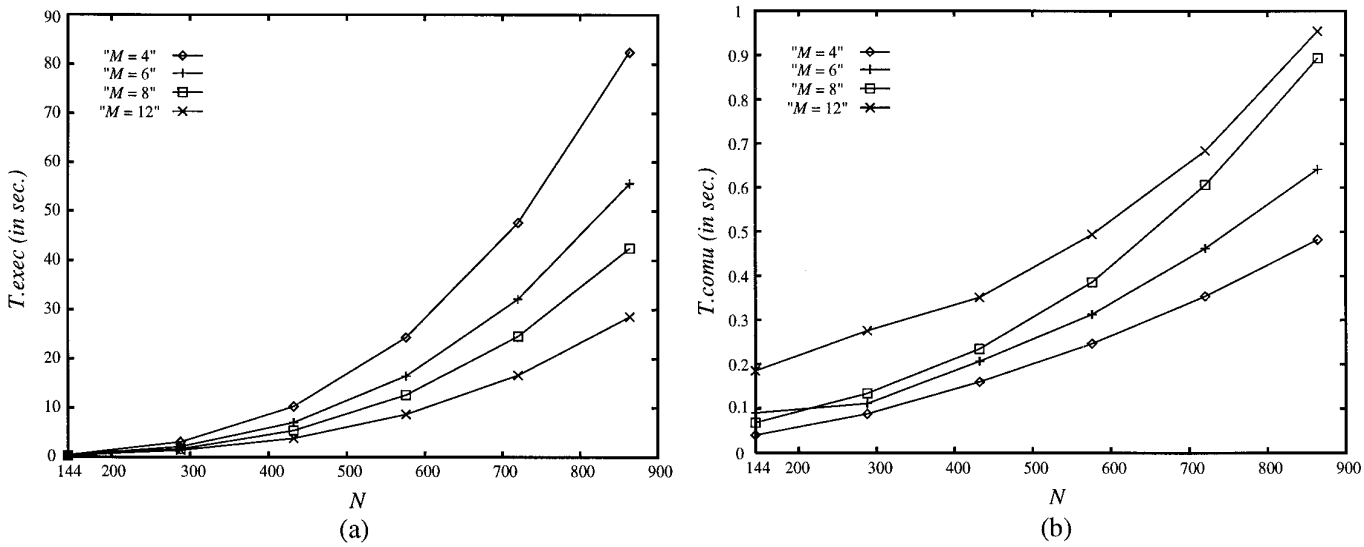

FIG. 2. Results for *Gaussian elimination* carried out on various sized hypercubes.

tion, with all the remaining nodes updating their respective resident rows as follow: P_0 updates *only* those rows below its first row and P_i $(i > 2)$ updates its *entire* resident rows, upon receiving the message (sent by P_1 in the last repetition). In general, during the rth $(r > 0)$ repetition, node P_i, where $i = r \bmod \mathscr{M}$, zerouts its $\lceil r/\mathscr{M} \rceil$th row and updates all the rows below it, while the rest receive the message (sent in the $(r-1)$th repetition) from P_i, and then operate as follows: (1) P_{i+1} first updates its $\lceil r/\mathscr{M} \rceil$th row, and then immediately broadcasts a copy of this updated row to all the other nodes on the ring, followed by the updates of all the rows below the $\lceil r/\mathscr{M} \rceil$th row; (2) P_j $(j < i)$ updates its $(\lceil r/\mathscr{M} \rceil + 1)$th row and all the rows below it; (3) P_k $(k > i + 1)$ updates its $\lceil r/\mathscr{M} \rceil$th row and all the rows below it. After $(N - \mathscr{M})$ repetitions, each node has only one row left to process. P_0 sends a copy of its last row to all the *subsequent* nodes in the ring and becomes idle immediately after it zerouts the last row. Each subsequent node receives the message and updates its last row individually. In the next repetition, P_1 acts in the same way as P_0 just did, and every subsequent node (excluding P_0) gets the message and updates its last row. This process continues, and more and more nodes in the ring become idle. The entire process ends as soon as $P_{\mathscr{M}-1}$ receives a message from $P_{\mathscr{M}-2}$ and finishes updating its last row.

Measured Results and Discussion[6]

The nonsingularity of matrix A is guaranteed by making each diagonal element in A sufficiently large such that the sum of absolute values of all the nondiagonal elements in any row never exceeds the value of the diagonal element in that row. The times in Fig. 2 are averages over 100 repetitions.

[6] All the programs used in our experiments were written in C and were compiled with the -O2 optimization flag.

As can be seen from Fig. 2b, the incomplete systems always involve higher communication overhead than their respective complete counterparts do. This is partially due to the fact that an incomplete system involves more nodes in the execution, and the amount of the communication in this application grows as the system size increases. Nevertheless, the incomplete systems still outperform their respective complete counterparts and the performance gain increases as the problem size grows, as indicated in Fig. 2a. Particularly, when $N = 864$, an incomplete system of size 12 outperforms its complete counterpart by as much as 49%, which is almost the theoretical upper bound of gain (i.e., 50%). This substantial performance gain may be attributed to the following causes: (1) when a problem of fixed size is divided among nodes in a system, the larger the system, the smaller memory space used in each node, resulting in a higher cache efficiency; and (2) for a sufficiently large N, the impact of larger communication overhead in an incomplete system becomes negligible.

4.2. Mixed Sort

A mixture of shell sort and odd–even transposition sort [5] (dubbed the mixed sort for short) is selected for the investigation of sorting, because it appears to be very efficient when implemented on Caltech/JPL's Hypercube machine [3] and on the Symult Series S2010 [5].

For simplicity, we consider the incomplete hypercube composed of two subcubes, a q-dimensional subcube and a $(q-1)$-dimensional subcube. The parallel sorting algorithm given in [5] is generalized to be applicable to both complete systems and incomplete systems (with an even number of nodes). As in [5], nodes in a system (either complete or incomplete) are arranged in a ring topology. The generalized algorithm is shown in Algorithm 1, where *mynode* is a node ID.

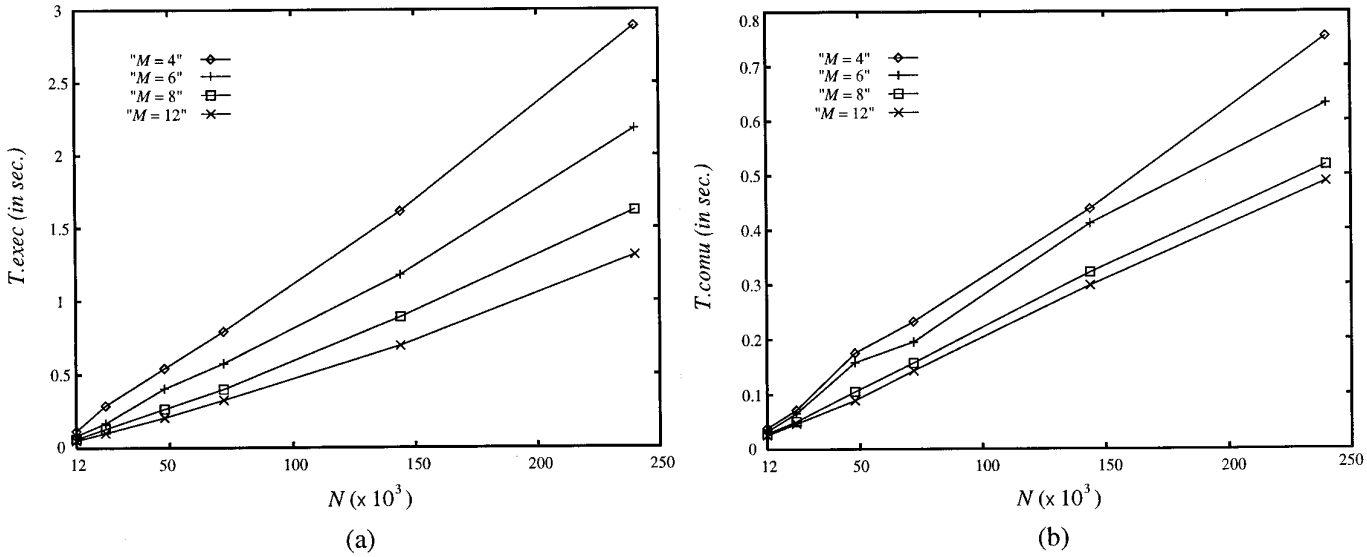

FIG. 3. Results for *mixed sort* carried out on various sized hypercubes.

ALGORITHM 1. Parallel Algorithm for Mixed Sort.

(1) Sort $(N/\mathscr{M})$ elements locally, yielding a sorted list called A list.
(2) $d := \mathscr{M}$;
repeat
$d := d/2$;
if $\lfloor mynode/d \rfloor$ is even, *partner* := *mynode* + *d*, *mask* := 0;
otherwise, *partner* := *mynode* − *d*, *mask* := 1;
exchange my A list with the list of my *partner;*
compare my A list with the list received.
if they are not in order, do the following operations:
copy the first $(N/\mathscr{M})$ small elements into my A list if *mask* = 0;
copy the last $(N/\mathscr{M})$ large elements into my A list, otherwise.
until d is not divisible by 2
(3) Do exchange–compare operations between pairs of adjacent nodes on the ring until all the A lists are globally in order.

The second part of Algorithm 1 is actually a presorting procedure. For a complete system, it takes $\log \mathscr{M}$ exchange–compare steps in total. For an incomplete system, however, the total exchange–compare steps taking place in the second part is $\lfloor \log \mathscr{M}^+ \rfloor - 1 = \log \mathscr{M} - 1$ (recall that $\mathscr{M}^+ = \frac{3}{2}\mathscr{M}$). Thus, an incomplete system actually takes *one less* step than its complete counterpart does, in the second part.

Parallel Implementation

The implementation of this algorithm is the same on an incomplete system as on a complete one. Initially, N random elements are evenly distributed across $\mathscr{M}$ nodes on a ring. Each node sorts its local $(N/\mathscr{M})$ random elements using the UNIX quick sort routine, *qsort*( ). The final part of the algorithm requires an early termination detection in order to terminate the odd-even transposition sort as soon as all the A lists are globally in order. Observe that node 0 and node $(\mathscr{M} - 1)$ on the ring are idle at even steps of the odd–even transposition sort. Therefore, either node 0 or node $(\mathscr{M} - 1)$ can be assigned to perform the early termination detection, making use of these idle cycles. An efficient implementation of the early termination detection is provided in [6]. Although the early termination detection introduces communication overhead as well as synchronization overhead, it tends to save a lot of compare–exchange steps during the odd–even transposition sort.

Measured Results and Discussion

The random elements are generated by using UNIX *drand48*( ) routine. The times in Fig. 3 are averages over 500 repetitions. It is interesting to observe from Fig. 3b that the incomplete systems actually involve less communication overhead than their respective complete counterparts. This may be explained as follows: (1) for small system sizes, an incomplete system does not necessarily have more *total* sorting steps (presorting steps plus odd–even transposition sorting steps) than its complete counterpart (recall that the former involves one less presorting step than the latter); (2) a larger system spends less time to perform exchange–compare operations at each sorting step, possibly resulting in a smaller synchronization overhead. As $\mathscr{M}$ grows, an incomplete system tends to have more total sorting steps. This can be seen in Fig. 3b, where *T.comu*'s of incomplete systems catch up with those of complete systems, as $\mathscr{M}$ grows. In any case of our study,

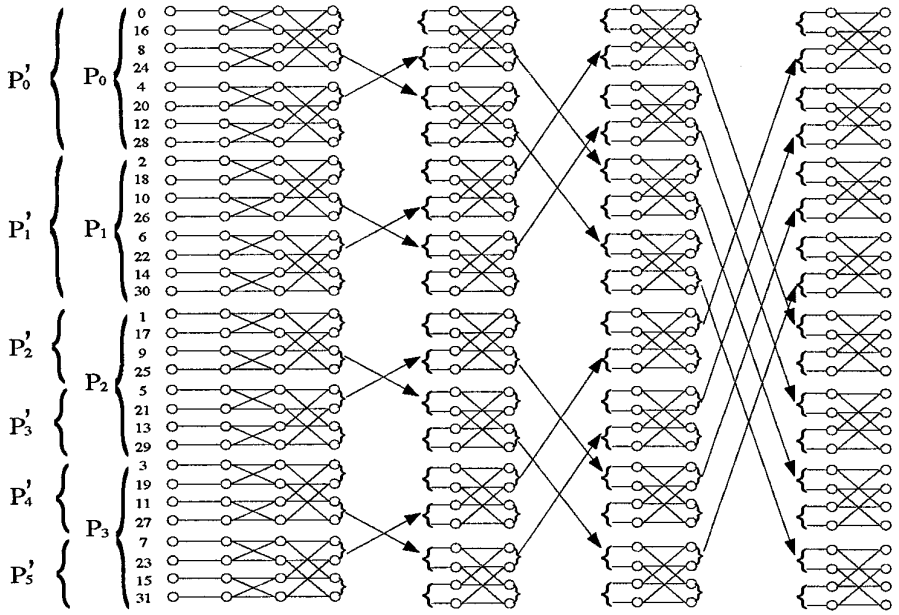

FIG. 4. The mapping of 32-pt *FFT* onto a 4-node complete hypercube (P0–P3) and a 6-node incomplete hypercube (P0′–P5′).

the incomplete system always exhibits a significant performance gain over its complete counterpart, as shown in Fig. 3a.

4.3. FFT Computation

The *discrete Fourier transform* (*DFT*) of an input sequence $x(k)$ of length N is defined as $Y(j) = \sum_{k=0}^{N-1} x(k)W_N^{jk}, j = 0, 1, \cdots, N-1$, where W_N^{jk} is a twiddle factor, and Y is transformed data. If we assume $N = 2^n$, *DFT* computation can be done in n stages, yielding the *Cooley–Tukey FFT* algorithm [2]. Each stage of the algorithm requires $N/2$ butterfly computations, which are given by $Y(j + N/2) = Y' - temp$, $Y(j) = Y' + temp$, and $j = 0, 1, \cdots, N/2 - 1$, where $temp = W_N^j Y'(j + N/2)$ and Y' is the data computed in the previous stage.

Mapping the *FFT* algorithm onto an incomplete hypercube is slightly different from mapping it onto a complete hypercube, as described separately below.

Parallel Implementation on the Complete Hypercube

The mapping scheme is to assign each node with $(N/\mathcal{M})$ *FFT* points, where $\mathcal{M} = 2^m$. Figure 4 shows a mapping of a 32-point *FFT* onto a 4-node hypercube. With this scheme, no communication takes place during the first $(n - m)$ stages since each node holds all information needed to update its local points at such stages. For the remaining m stages, however, each node has to exchange its $(N/\mathcal{M}/2)$ points with the $(N/\mathcal{M}/2)$ points of its neighboring node before it carries out updates.

Parallel Implementation on the Incomplete Hypercube

For simplicity, the incomplete hypercube under investigation consists of two subcubes, a q-dimensional subcube and a $(q - 1)$-dimensional subcube. Cube nodes are divided into two parts, with the first part involving the nodes in the $(q - 1)$-dimensional subcube and the second part involving all the rest. The mapping scheme is achieved by evenly distributing $(N/2)$ *FFT* points to the nodes in the first part, and the remaining $(N/2)$ points to the nodes in the second part. Figure 4 shows a mapping of 32-pt *FFT* onto a 6-node incomplete hypercube. With this mapping scheme, the number of points assigned to each node in the first part is twice as many as that to a node in the second part, and thus nodes in the first part involve one less communication stage. Prior to the last stage, nodes in the first part participate in the same way as if they were within an $\mathcal{M}$-node complete cube (refer to Fig. 4), while nodes in the second part operate as if they were within a $(2\mathcal{M})$-node complete cube (recall that $\mathcal{M}^+ = \frac{3}{2}\mathcal{M}$). At the last stage, each node in the first part needs to exchange one half of its local points, i.e., $(\frac{3}{4}N/\mathcal{M}^+)$ points, with points from two nodes in the second part, each responsible for $(\frac{3}{8}N/\mathcal{M}^+)$ points.

It is clear that whether or not an incomplete hypercube can outperform its complete counterpart depends primarily on the communication overhead involved at the last stage in the first part of the incomplete system. To minimize this portion of communication overhead, each node in the first part issues two nonblocking *recv*'s at the very beginning, in anticipation of two messages to arrive (from nodes in the second part), which are used in the last stage. This is mainly to avoid message buffering which may occur at the last stage. Also, at the second last stage, each node in the first part updates one half of its points first, and then issues two nonblocking *send*'s to initiate the transmission of updated points to the corresponding destinations, with the updates of the remaining points followed. Thus, these

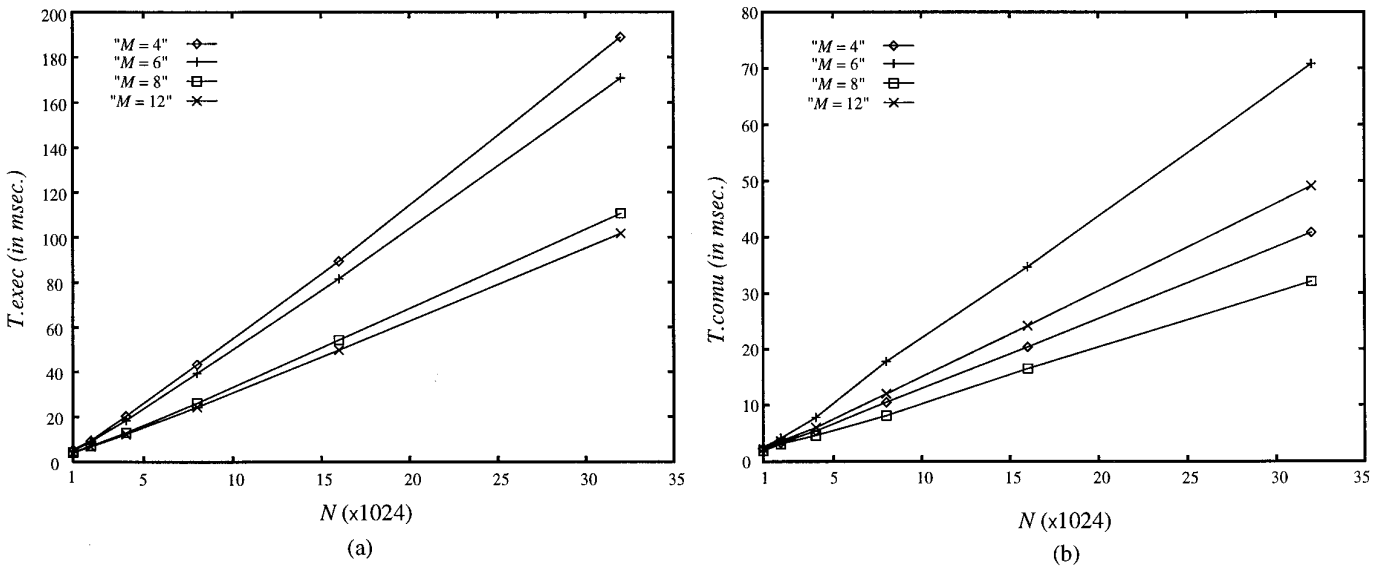

FIG. 5. Results for *FFT* carried out on various sized hypercubes.

two nonblocking *send*'s are overlapped with the updates. As a result, when N is sufficiently large, communication overhead at the last stage in the first part of an incomplete system may be entirely avoided, and in such a case, an incomplete hypercube outperforms its complete counterpart (refer to [6] for detailed discussion on *FFT*).

Measured Results and Discussion

The twiddle factors are precomputed and available in each node when needed. The times in Fig. 5 are averages over 500 repetitions. From Fig. 5a, it is observed that the performance gain of an incomplete system of size 6 over its complete counterpart is greater than that of an incomplete system of size 12 over the respective complete counterpart, for various problem sizes. This is somewhat expected, because as $\mathcal{M}$ grows, the number of communication stages increases, and thus the impact of communication overhead reduction at the last stage of an incomplete system decreases. The performance gain is less impressive in this application than the other two applications, because *FFT* is communication-intensive following the butterfly communication pattern, and is topology-dependent. An incomplete hypercube with more computational power cannot offer as much performance improvement as before, in this case.

It is worth noting that although communication overhead at the last stage in the first part of an incomplete system may be avoided when N is sufficiently large, considerable waiting time might be involved at each node in the second part during the last stage, yielding higher overall communication overhead in an incomplete system.

5. CONCLUSION

In this paper, we have considered identifying maximum incomplete subcubes in a faulty hypercube. It is demonstrated by simulation results that our reconfiguration can often give rise to a much larger reconfigured subsystem than an obtainable complete subsystem. To study the actual performance difference between an incomplete system and its complete counterpart, we have implemented three applications on the Intel *iPSC*/860. Our collected results show that it is indeed possible to achieve better performance on an incomplete system than on its complete counterpart (with fewer nodes). This investigation suggests that reconfiguring a faulty hypercube into an incomplete subcube so as to retain more fault-free nodes, is profitable practically and should be recommended.

ACKNOWLEDGMENTS

The authors thank Dr. Donna Reese at *ERC* for Computational Field Simulation, Mississippi State University, for allowing our access to the *iPSC*/860 machine there, on which these program execution studies were carried out.

REFERENCES

1. Bertsekas, D. P., and Tsitsiklis, J. N. *Parallel and Distributed Computation* (*Numerical Methods*). Prentice–Hall, Englewood Cliffs, NJ, 1989.
2. Cooly, J. W., and Turkey, J. W. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.* **19,** (Apr. 1965), 297–301.
3. Felten, E., Karlin, S., and Otto, S. Sorting on a hypercube. Unpublished Report Hm-244, Calif. Inst. of Technol., Pasadena, CA, 1986.
4. Katseff, H. P. Incomplete hypercubes, *IEEE Trans. Compt.* **37,** (May 1988), pp. 604–608.
5. Li, P. P., and Tung, Y.-W. Parallel sorting on Symult 2010. *Proc. 5th Distributed Memory Computing.* Apr. 1990, pp. 224–229.
6. Lin, G., and Tzeng, N.-F. Program execution behaviors on incomplete hypercubes, Tech. Rep. TR-94-8-5, Center for Advanced Computer Studies, Univ. of Southwestern Louisiana, 1993.
7. Tien, J.-Y., Ho, C.-T., and Yang, W.-P. Broadcasting on incomplete hypercubes. *IEEE Trans. Comput.* **42,** (Nov. 1993), 1393–1398.

8. Tzeng, N.-F. Empirical evaluation of incomplete hypercube systems. *Proc. 22nd Int'l Conf. Parallel Processing,* Aug. 1993, Vol. I, pp. 96–99.
9. Tzeng, N.-F., and Lin, G. Identifying maximal incomplete subcubes in a faulty hypercube. *Proc. 7th Int'l Conf. Parallel and Distributed Computing Systems.* Oct. 1994, pp. 186–193.

GUANGHUA LIN received his Ph.D. in computer engineering from the University of Southwestern Louisiana, Lafayette, in May 1995. He is a member of the IEEE Computer Society. His current research interests include parallel processing, fault-tolerant computing, high-performance computer architecture, and computer networking.

Received January 30, 1995; accepted May 5, 1995

NIAN-FENG TZENG received the Ph.D. in computer science from the University of Illinois at Urbana–Champaign in 1986. He is currently an associate professor in the Center for Advanced Computer Studies at the University of Southwestern Louisiana, Lafayette. From 1986 to 1987, he was a Member of Technical Staff, AT&T Bell Laboratories, Columbus, Ohio. Dr. Tzeng is the recipient of the outstanding paper award of the 10th International Conference on Distributed Computing Systems, May 1990. He is on the editorial board of the *IEEE Transactions on Computers,* has served on the program committees of several conferences, and is a Distinguished Visitor of the IEEE Computer Society. His research interests include parallel and distributed processing, high-performance computer systems, fault-tolerant computing, and high-speed networking.