A Repeated Mapping Scheme of Task Modules with Minimum Communication Cost in Hypercube Multicomputers

Joo-Man Kim and Cheol-Hoon Lee

CONTENTS

ABSTRACT

This paper deals with the problem of one-to-one mapping of 2^n task modules of a parallel program to an n-dimensional hypercube multicomputer so as to minimize the total communication cost during the execution of the task. The problem of finding an optimal mapping has been proven to be NP-complete. First we show that the mapping problem in a hypercube multicomputer can be transformed into the problem of finding a set of maximum cutsets on a given task graph using a graph modification technique. Then we propose a repeated mapping scheme, using an existing graph bipartitioning algorithm, for the effective mapping of task modules onto the processors of a hypercube multicomputer. The repeated mapping scheme is shown to be highly effective on a number of test task graphs; it increasingly outperforms the greedy and recursive mapping algorithms as the number of processors increases. Our repeated mapping scheme is shown to be very effective for regular graphs, such as hypercube-isomorphic or 'almost' isomorphic graphs and meshes; it finds optimal mappings on almost all the regular task graphs considered.

I. INTRODUCTION

The rapid progress of VLSI and computer networking technologies has made it possible to develop a variety of multicomputers. A hypercube is one of them and has received considerable attention in recent years due to its regularity for ease of construction and its potential for high degree of parallelism. An n-dimensional hypercube, also known as a binary n-cube, is a highly concurrent loosely-coupled multicomputer with 2^n processors (nodes). Each node is located on one of the 2^n vertices of the binary n-cube. A node with its own processor and memory is connected to its n neighboring nodes. Hypercube design, performance evaluation, and other related issues have been addressed by numerous researchers [1]-[3]. A few machines based on the hypercube topology have been built and experimented, such as the Intel iPSC [4], NCUBE [5], Caltech/JPL [6], and the Connection machine [7].

In order to run a parallel program on a hypercube multicomputer, each of its task modules must be placed at one of the processors. The problem of placing the task modules on the node processors has been termed as the *mapping problem* [8]. The mapping problem is different from the *processor allocation* problem, which attempts to allocate processing power in an efficient way without focusing on the communication structure of the task modules. The processor allocation problem is similar to the conventional memory allocation problem, but the objective is to maximize processor utilization. Several processor allocation strategies have been proposed, such as the Gray-code strategy [9], which is designed for hypercube multicomputers, and wave scheduling [10], which is for multicomputers that can be linked together in the form of a tree. After processor allocation, a mapping procedure should be applied to handle the structural mismatch between the task and the hypercube so as to minimize the interprocessor communication cost and the execution time.

Many parallel programs are communication rather than processing limited. Some fine-grain parallel programs execute as few as ten instructions in response to a message. Also, in the task level—where a computation task is decomposed into a set of communicating modules—intermodule traffic, and hence interprocessor communication when the modules are assigned to different processors, tends to be bursty. To execute such programs efficiently, the communication network must be able to handle heavy concurrent traffic. In this paper, we consider the problem of mapping a set of communicating task modules composing a parallel program onto a hypercube multicomputer so as to minimize the total interprocessor communication traffic. This problem, however, has been proven to be NP-complete [11]. Hence fast heuristic algorithms are used to find good mappings instead of the optimal ones. A greedy algorithm for the mapping problem in hypercubes has been proposed in [12], which

utilizes a graph-oriented strategy to map a communication graph to the hypercube. This greedy algorithm performs reasonably well on certain types of graphs, such as linear arrays, hypercubes-isomorphic, and nearly-isomorphic communication graphs, but not on others. Other researchers [13] suggested a recursive divide-and-conquer strategy which performs repeated recursive bipartitioning of a task graph, based on the Kernighan-Lin's mincut bisection heuristic [14]. In this paper, based on the graph bipartitioning heuristic in [15], we propose an efficient *repeated mapping scheme* for the effective mapping of task modules of a parallel program onto a hypercube multicomputer. The effectiveness of the proposed algorithm is evaluated by comparing the quality of mappings obtained with those derived using the greedy and the recursive mapping algorithms on the same sample problems.

The approach proposed in this paper uses the following topological property of hypercube graphs: by partitioning along any k dimensions, $1 \leq k \leq n$, an n-cube is partitioned into 2^k $(n-k)$-subcubes. A set of 2^n modules are repeatedly bipartitioned, thus the k-th bipartition determining the k-th bit of each module's processor mapping. Each bipartition is obtained by applying a graph partitioning algorithm based on the heuristic in [15], which has been shown to be more efficient than Kernighan-Lin's heuristic algorithm [14]. After the n-th bipartitioning, the full binary address for each module's processor mapping is determined. Each k-th bipartition determines the communication cost imposed on the k-th dimensional communication links under the resulting mapping. Our repeated mapping strategy differs from the recursive mapping strategy [13] in that the former performs each bipartitioning on an appropriately modified graph while the latter performs bipartitioning recursively on subgraphs. The shortcoming of the recursive mapping strategy is that even if each subgraph is optimally bipartitioned, the mapping problem is not optimally solved. On the other hand, our repeated mapping strategy finds an optimal mapping with the minimum total communication cost by optimally bipartitioning at each iteration, i.e., minimizing each dimensional communication cost. The key idea is a graph modification technique which makes the resulting mapping between modules and nodes one-to-one. Extensive simulation results show that the repeated mapping strategy proposed in this paper outperforms both the greedy and recursive mapping strategies.

The paper is organized as follows. In Section II, we present the system model and the assumptions used. Our problem is also formally stated there. Some simple topological properties of the hypercube are drived first in Section III. We then present the cutset formulation which serves as a framework for the design of efficient heuristics. In Section IV, we describe an efficient graph partitioning algorithm based on the heuristic in [15]. We then propose a graph modification technique which enables the graph partitioning algorithm to be applied

successfully to the mapping problem in hypercube multicomputers. Experimental results on a variety of task graphs are shown and compared with those of the greedy algorithm and the recursive mapping algorithm in Section V. The paper concludes with Section VI.

II. PROBLEM STATEMENT

In this section, we formalize the mapping problem under consideration and develop the cost function to be minimized. An n-dimensional hypercube is constructed as follows. First, the 2^n nodes/processors are labeled by the 2^n numbers ranging from 0 to 2^n-1. Then there is a direct communication link between two nodes if and only if their binary addresses differ by one and only one bit. Therefore, n-dimensional hypercube multicomputer is composed of 2^n processors $\{p_k|0 \le k \le 2^n-1\}$ each with its own memory, and $n2^{n-1}$ bidirectional communication links since each processor is connected with their n neighbor processors. The system is assumed to be homogeneous, with all processors being equally powerful and all communication links having the same bandwidth. As in most existing hypercube multicomputers, intermodule communications are assumed to be accomplished via message passing. Each message is further decomposed into smaller packets of fixed length. By using a packet as the unit of communication, the communication volume between each pair of modules can be expressed as the number of packets to be exchanged between them during execution of the task modules.

Let the *address* of a processor or node p_k in the hypercube multicomputer be the binary representation, $k^n k^{n-1} \cdots k^1$, of k. Let $d_{k,l}$ be the Hamming *distance* between any two processors p_k and p_l, i.e., the minimum number of communication links to be traversed between p_k and p_l. The hypercube multicomputer under consideration is assumed to possess the e-cube routing scheme under which a packet is routed from the source to the destination by modifying different bits between the address of the source and that of the destination in a fixed order, e.g., from the least-significant bit to the most-significant bit. Thus, each packet is routed through a fixed shortest path from the source to the destination.

A task to be executed on an n-cube is composed of M interacting modules m_1, m_2, $\cdots$, m_M and is represented with an undirected task graph $G(V,E)$ whose vertices $V=\{m_i\}$ represent the task modules, and edges $E=\{e_{ij}\}$ correspond to the data communication between the task modules. The weight of edge e_{ij} between m_i and m_j, denoted by $W_{i,j}$, represents the communication volume (i.e., the number of packets) required between the two task modules. If the exact values of $W_{i,j}$'s are not available, one can use their estimates, e.g., average or worst-case values.

When mapping a set of task modules into an n-dimensional hypercube, one can, without loss of generality, assume the task to consist of 2^n modules. For a task with

M modules such that $2^{n-1} < M < 2^n$ for some integer n, one can add *dummy* modules to make the task consist of 2^n modules. In case of $M > 2^n$, a workload partitioning scheme can be applied to cluster modules into groups so that the number of groups may match the size of the hypercube taking into account the communication structure between modules. So, we will henceforce assume $M = 2^n$ where n is the hypercube dimension, and thus, the mapping of task modules into hypercube nodes is one-to-one. This assumption is not unrealistic since most parallel programs written for hypercubes follow this rule, and most existing hypercube multicomputers do not support a per-node multi-programming environment [16].

The module-to-processor mapping is a one-to-one function $X : m_i \rightarrow p_k$. $X(i)$ represents the processor onto which task m_i is mapped. Suppose $comp(m_i)$ is the computation cost to be incurred for a node to execute m_i. Since a hypercube is homogeneous, any mapping will have the same total computation cost of $\sum_{i=1}^{M} comp(m_i)$. However, a different mapping will lead to a different communication cost. We define the communication cost in executing a set of task modules as the sum of time units each communication link is used during the task execution. In other words, the communication cost is a measure of the link resources used by an instance of task execution expressed in time units.

Suppose $c(h)$ is the number of time units needed to send a packet over a path of h hops, and the time a link is kept busy for purposes other than packet transmission—such as establishing the communication path—is assumed to be negligible. For hypercubes with packet switching, we have $c(h) = h \cdot c(1)$. This relation may be less accurate in case of circuit switching. However, if the "call request" signal to hunt for a free path occupies each link only for a very short time, then this expression would be a good approximation even for circuit-switched hypercubes [16]. Since we are considering the case with heavy concurrent traffic, as discussed in [17], cut-through switching degenerates into packet switching and hence, will not be discussed here.

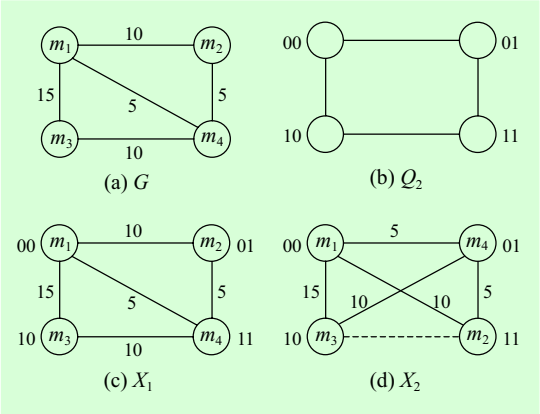

Fig. 1. An example task with two mappings to a Q_2.

Without loss of generality, we define $c(1)$ as the unit of communication cost which is the link usage by one packet traversing one link. Then the communication cost resulting from executing a task under mapping X becomes $COST(X) = \sum_{i<j} W_{i,j} \cdot d_{X(i),X(j)}$, since each packet between m_i and m_j should traverse $d_{X(i),X(j)}$ links under

mapping X. For example, in Fig. 1, there are two different mappings X_1 and X_2 of a four-module task to a 2-cube. The total communication costs of X_1 and X_2 are 50 and 65, respectively. Since the total computation cost is constant irrespective of mappings, only the cost of interprocessor communication needs to be considered for mapping tasks. This point differentiates our work from others' related to task assignment in multicomputers, such as those in [18]–[20]. However, since the hypercube is also symmetric, any two mappings X_a and X_b will lead to the same communication cost if $X_a(i) \oplus x = X_b(i)$, $1 \le i \le M$, for any n-bit binary number x. We call such two mappings to be *equivalent*. The mapping problem considered in this paper is the problem of finding a one-to-one mapping X_o of the task modules into the processors with the minimum total communication cost, *i.e.*,

$$\begin{aligned} COST(X_o) &= \min_X COST(X) \\ &= \min_X \Big(\sum_{i<j} W_{i,j} \cdot d_{X(i),X(j)} \Big). \end{aligned}$$

III. CUTSET FORMULATION

One can construct an n-cube as follows. First, its 2^n nodes are labeled with the 2^n numbers from 0 to $2^n - 1$. Then a link between two nodes is drawn if and only if their binary numbers differ by one and only one bit.

Definition 1: An n-cube, Q_n, is an undirected graph whose node set V_n consists of 2^n n-dimensional Boolean vectors, i.e., vectors with binary coordinates 0 or 1, where two nodes are adjacent whenever they differ in exactly one coordinate.

Using the definition of an n-cube, one can partition the n-cube into two $(n-1)$-cubes by removing each edge between two nodes whose i-th coordinates are different. This will be referred to as partitioning along the i-th direction. Since there are n bits in each node address of an n-cube, there are n different ways of partitioning an n-cube into two $(n-1)$-cubes. More generally, by repeatedly partitioning along any k different directions, $1 \le k \le n$, an n-cube is partitioned into 2^k $(n-k)$-subcubes.

Definition 2: A cutset C_i of a task graph $G(V,E)$ with $|V| = 2^n$ is a set of edges that separates V_i from $\overline{V}_i$ such that $V_i \cap \overline{V}_i = \emptyset$, $V_i \cup \overline{V}_i = V$. If $|V_i| = |\overline{V}_i| = |V|/2$ $(= 2^{n-1})$, then C_i is called a *balanced* cutset. The weight of C_i, $W(C_i)$, is the total weight of the edges in the cutset.

Let $\mathbf{C}$ be a set of n balanced cutsets, $\{C_i | 1 \le i \le n\}$, of a task graph $G(V,E)$ with $|V| = 2^n$. Then $\mathbf{C}$ is said to be *admissible* if, by k cutsets $C_1, C_2, \cdots, C_k$ in $\mathbf{C}$, $1 \le k \le n$, the set V is partitioned into 2^k subsets each of which contains exactly 2^{n-k} task modules. Each admissible set $\mathbf{C} = \{C_i | 1 \le i \le n\}$ of a task graph $G(V,E)$ with $|V| = 2^n$ one-to-one corresponds to a mapping of the task modules into the processors of an n-cube. The weight of $\mathbf{C}$ is the total weight of the cutsets in $\mathbf{C}$, i.e., $W(\mathbf{C}) = \sum_i W(C_i)$.

Lemma 1: Let an admissible set **C** correspond to a mapping X. Then the weight of **C** is equal to the total communication cost of X.

Proof: The total communication cost of the mapping X is

$$\begin{aligned} COST(X) &= \sum_{a<b} W_{a,b} \cdot d_{X(a),X(b)} \\ &= \sum_{a<b} W_{a,b} \cdot \left(\sum_{i=1}^{n} X(a)^i \oplus X(b)^i \right) \\ &= \sum_{i=1}^{n} \left(\sum_{a<b} W_{a,b} \cdot \left(X(a)^i \oplus X(b)^i \right) \right) \\ &= \sum_{i=1}^{n} W(C_i) \\ &= W(\mathbf{C}). \end{aligned}$$

This proves the lemma. □

For example, given a task graph with 2^3 nodes as in Fig. 2(a), the set $\mathbf{C} = \{C_1, C_2, C_3\}$ shown in Fig. 2(b) is admissible. The corresponding mapping X is shown in Fig. 2(c) with the total communication cost of 105 which is equal to the weight of **C**, i.e., $COST(X) = W(C_1) + W(C_2) + W(C_3) = 25 + 40 + 40 = 105$. Note that the weight, $W(C_i)$, of each cutset C_i is equal to the communication cost on the i-th directional communication links, i.e., the links connecting processors whose i-th address bit is 0 and those whose i-th address bit is 1, under the resulting mapping X. The above lemma states that the mapping problem is equivalent to the problem of finding a minimum-weight admissible set **C**.

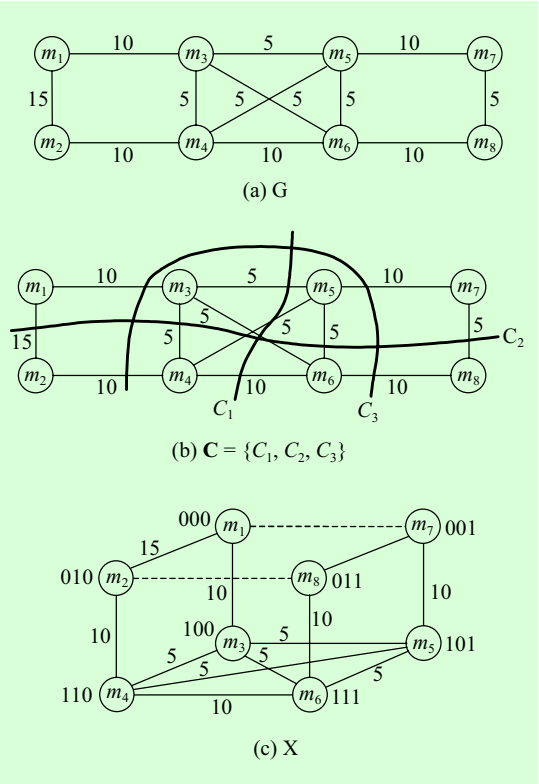

Fig. 2. An admissible set **C** and its corresponding mapping X.

IV. MAPPING BY REPEATED BIPARTITIONING

Lee *et al.* [15] proposed an effective algorithm for graph partitioning, with an empirically determined time complexity of $O(N^2)$ where N is the number of nodes in a graph. They showed that the graph partitioning problem with size constraints is transformed into the *max-cut* problem without any size constraint using a graph modification technique. A maximum weight cutset in the modified graph corresponds to a minimum weight balanced cutset in the origi-

nal graph. They also presented an efficient algorithm for the max-cut problem. Their algorithm has been shown to be highly efficient even as compared to Kernighan and Lin's algorithm [14]. Moreover, it guarantees exactly balanced-partitions when the nodes of a graph are all of the same size such as the task graph considered here. Note that, in this paper, only one-to-one mappings are considered irrespective of the size (i.e., execution cost) of each task module. The basic bipartitioning algorithm used here is similar to the algorithm in [15].

We can apply the bipartitioning procedure repeatedly to obtain a balanced N-way partition of a graph, where $N = 2^n$, by first creating two balanced partitions, then modifying the graph so as to enable the next partition to decompose each of these partitions into two balanced subpartitions each, and so on, until N balanced partitions are created. The basic idea is to make partial mapping of the task modules to the processors during these repeated bipartitioning steps. At level k in this process, for each module, the k-th bit of the address of its mapped processor is determined. Equivalently, the bisections at level k may be viewed as successively refining the subcube to which a module is to be assigned. Initially, prior to any partitioning, the entire hypercube is the single subcube under consideration and each module clearly is to be assigned within this subcube. The first bipartitioning of the task graph separates the modules into two groups, each to be assigned to a distinct subcube of size $N/2$. In other words, the first bit of the processor to be assigned to is uniquely determined. The n-th level bipartitioning determines the full address of each module's mapped processor. The bipartitioning algorithm presented in this paper guarantees exactly-balanced partitions by virtue of a graph modification technique (to be described later). Moreover, the resulting n bipartitions are admissible since, after each k-th level bipartitioning, exactly 2^{n-k} task modules are assigned to each of 2^k $(n-k)$-cubes. Therefore, after the final n-th level bipartitioning, each module is assigned a unique address for its mapped processor. This means that the resulting mapping is one-to-one.

Let each k-th level bipartition correspond to a cutset C_k. Also let the set of modules of a task graph be partitioned into 2^k subsets P_k^1, P_k^2, $\cdots$, $P_k^{2^k}$ by the k cutsets C_1, C_2, $\cdots$, C_k such that $P_{k-1}^i = P_k^{2i-1} \cup P_k^{2i}$, i.e., each of the 2^{k-1} subsets P_{k-1}^i's partitioned by the $k-1$ cutsets C_1, C_2, $\cdots$, C_{k-1} is further partitioned into two subsets P_k^{2i-1} and P_k^{2i} by the next cutset C_k. Let all the modules be initially in P_0^1 before any bipartitioning, i.e., $P_0^1 = V$. Then, by definition, a set $\mathbf{C} = \{C_1, C_2, \cdots, C_n\}$ is admissible if $|P_k^1| = |P_k^2| = \cdots = |P_k^{2^k}|$, for all $1 \le k \le n$. At each k-th level, before bipartitioning, the task graph $G(V,E)$ is transformed into G_k^* so as to make the resulting cutsets admissible as follows. For any two nodes m_a and m_b of G,

1. if they are not separated by the $k-1$ cutsets C_1, C_2, $\cdots$, C_{k-1}, (i.e., $m_a \in$

P^i_{k-1} and $m_b \in P^i_{k-1}$ for some i, $1 \le i \le 2^{k-1}$), then modify the weight of the edge between them to be $R - W_{a,b}$, where the augmenting factor R is set to an appropriate positive value such that $R > \sum_{a<b} W_{a,b}$ (if there is no edge between them in the original graph G, then make a new edge with the weight of R),

2. otherwise, modify the weight of the edge between them to be $-W_{a,b}$.

Lemma 2: Given a graph G and its modified graph G^*_k, let cutset C^*_k (C_k) partition the nodes of G^*_k (G) into two subsets A_k and B_k. Then,

$$W(C^*_k) = \sum_{i=1}^{2^{k-1}} |A_k \cap P^i_{k-1}| \cdot |B_k \cap P^i_{k-1}| \cdot R - W(C_k).$$

Proof: Let each of the 2^{k-1} subsets P^i_{k-1}'s, $1 \le i \le 2^{k-1}$, be partitioned by the cutset C_k such that $P^{2i-1}_k = P^i_{k-1} \cap A_k$ and $P^{2i}_k = P^i_{k-1} \cap B_k$. Then,

$$\begin{aligned} W(C^*_k) &= \sum_{i=1}^{2^{k-1}} \sum_{m_a \in P^{2i-1}_k} \sum_{m_b \in P^{2i}_k} R - \sum_{\substack{m_c \in A_k \\ m_d \in B_k}} W_{c,d} \\ &= \sum_{i=1}^{2^{k-1}} |P^{2i-1}_k| \cdot |P^{2i}_k| \cdot R - W(C_k). \quad \square \end{aligned}$$

Thus, the weight of each cutset C^*_k on a modified graph G^*_k has the information on both the size of the associated subsets and the weight of the corresponding cutset on the original graph G. If each cutset C^*_k is a maximum-weight cutset on its associated graph G^*_k, then the corresponding cutset C_k on G is a minimum-weight admissible cutset since all the subsets P^j_k's, $1 \le j \le 2^k$ are of the same size, i.e., $\sum_i |P^{2i-1}_k| \cdot |P^{2i}_k| = 2^{2n-k-1} =$ constant. This is shown in the following theorem.

Theorem 1: For a given task graph $G(V,E)$ with $|V| = 2^n$, a set $\mathbf{C}^* = \{C^*_1, C^*_2, \cdots, C^*_n\}$ is admissible if each cutset C^*_k is a maximum-weight cutset on its associated graphs G^*_k, for $1 \le k \le n$.

Proof: Let each cutset C^*_k correspond to C_k on G. We prove the admissibility of the set $\mathbf{C}^*$ by showing that $|P^1_k| = |P^2_k| = \cdots = |P^{2^k}_k|$ for all $1 \le k \le n$. We prove it by induction on k.

(1) For $k = 1$, the set of task modules P^1_0 is partitioned into P^1_1 and P^2_1 by cutset C^*_1. Suppose that the partition (P^1_1, P^2_1) is not balanced, i.e., $|P^1_1| \ne |P^2_1|$. Then there exists another balanced cutset $C^{*'}_1$ (C'_1) on $G^*_1(G)$ which partitions P^1_0 into $P^{1'}_1$ and $P^{2'}_1$ such that $|P^{1'}_1| = |P^{2'}_1|$. Then, $|P^1_1| \cdot |P^2_1| + 1 \le |P^{1'}_1| \cdot |P^{2'}_1|$, since $|P^1_1| + |P^2_1| = |P^{1'}_1| + |P^{2'}_1|$ = constant. Thus,

$$\begin{aligned} &W(C^{*'}_1) - W(C^*_1) \\ &= |P^{1'}_1| \cdot |P^{2'}_1| \cdot R \\ &\quad - W(C'_1) - \Big(|P^1_1| \cdot |P^2_1| \cdot R - W(C_1)\Big) \\ &\ge R - \Big(W(C'_1) - W(C_1)\Big) \ge 0. \end{aligned}$$

The last inequality comes from the inequality $R > \sum_{a<b} W_{a,b}$. Therefore, $W(C^{*'}_1) > W(C^*_1)$. This is contradictory to the fact that C^*_1 is a maximum cutset. Therefore, $|P^1_1| = |P^2_1|$.

(2) Suppose it holds for $k = i - 1$. Then, $|P^1_{i-1}| = |P^2_{i-1}| = \cdots = |P^{2^{i-1}}_{i-1}|$. Let the set of task modules be partitioned into

A_i and B_i by C_i^*. Suppose it does not hold for $k = i$. Then there exists another cutset $C_i^{*'}(C_i')$ on $G_i^*(G)$ which partitions the task modules into A_i' and B_i' such that $|A_i' \cap P_{i-1}^j| = |B_i' \cap P_{i-1}^j| = 2^{n-i}$ for all $1 \le j \le 2^{i-1}$. Then, $\sum_{j=1}^{2^{i-1}} |A_i \cap P_{i-1}^j| \cdot |B_i \cap P_{i-1}^j| + 1 \le \sum_{j=1}^{2^{i-1}} |A_i' \cap P_{i-1}^j| \cdot |B_i' \cap P_{i-1}^j|$, since $\left(\sum_{j=1}^{2^{i-1}} |A_i \cap P_{i-1}^j| + |B_i \cap P_{i-1}^j|\right) = \left(\sum_{j=1}^{2^{i-1}} |A_i' \cap P_{i-1}^j| + |B_i' \cap P_{i-1}^j|\right) =$ constant. Then,

$$
\begin{aligned}
&W(C_i^{*'}) - W(C_i^*) \\
&= \sum_{j=1}^{2^{i-1}} |A_i' \cap P_{i-1}^j| \cdot |B_i' \cap P_{i-1}^j| \cdot R - W(C_i') \\
&\quad - \left(\sum_{j=1}^{2^{i-1}} |A_i \cap P_{i-1}^j| \cdot |B_i \cap P_{i-1}^j| \cdot R - W(C_i)\right) \\
&\ge R - \Big(W(C_i') - W(C_i)\Big) \ge 0.
\end{aligned}
$$

Therefore, $W(C_i^{*'}) > W(C_i^*)$. This is contradictory to the fact that C_i^* is a maximum cutset. Therefore, it holds for $k = i$. □

Theorem 1 states that the problem of finding an admissible set of minimum-weight cutsets can be transformed into the problem of finding a set of maximum-weight cutsets, which is called the *max-cut* problem, by using the graph modification technique proposed in this paper. In other words, if we transform the mapping problem into the max-cut problem, we need not try to keep the resulting partition admissible. Consequently, it is easier to devise a heuristic algorithm for the max-cut problem than for the original admissible min-cut problem since the former has only one objective whereas the latter has two objectives to optimize. For example, for the task graph in Fig. 2(a), each modified graph G_k^*'s with its maximum-weight cutset C_k^* is shown in Fig. 3. Note that the set $\mathbf{C}^* = \{C_1^*, C_2^*, C_3^*\}$ is admissible, and each corresponding cutset C_k in Fig. 2(b) is a minimum-weight cutset on its associated graph.

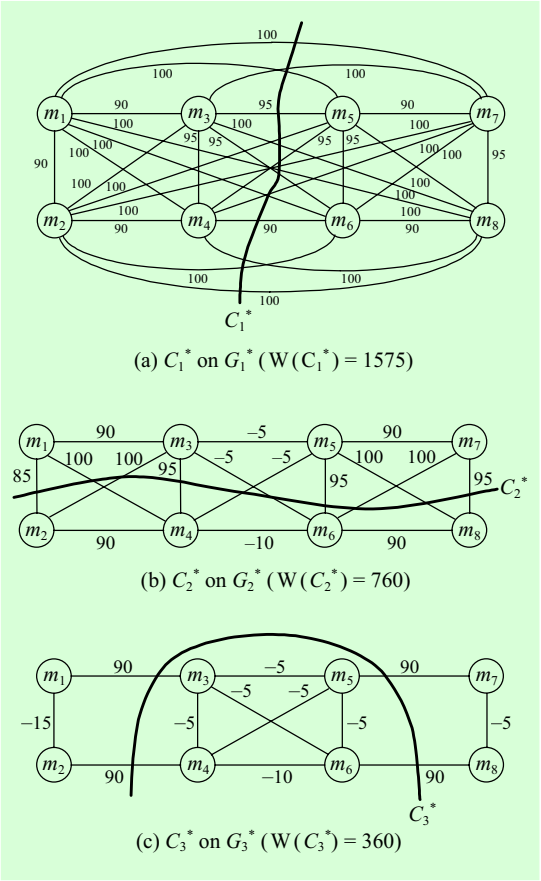

Fig. 3. An admissible set $\mathbf{C}^*$ on the modified graphs ($R = 100$).

We are now ready to present a heuristic max-cut algorithm. Let V be the set of

```
Input: a modified graph G_k*
Output: bipartition(A_k, B_k)                    //initially empty//
Variables
   Part[1..N]: integer;                          //Part[i] = partition block number of m_i//
   Gain[1..N]: real;                             //Gain[i] = g(m_i)//
   State[1..N]: boolean;                         //0 = unused, 1 = used//
   History[1..N]: integer;                       //information on previous move operation//
   Temp[1..N]: integer;                          //temporary gain for previous move//

1)    construct an initial partition P;          //Part[i] = 1 for all i//
2)    for i := 1 to N do
        Gain[i] :=g(m_i);
        State[i] :=0;                            //"unused"//
3)    for i :=1 to N do
3.1)    select m_d such that g(m_d)=max_j(Gain[j]) and State[d]=0;
3.2)    State[d] :=1                             //m_d is a candidate element, set 'used'//
3.3)    Part[d] := (Part[d] + 1) mod 2;          //one-move//
3.4)    History[i] :=d;                          //save information on one-move//
3.5)    Temp[i] :=Gain[d];                       //save gain of m_d//
3.6)    for j :=1 to N do                        //recalculate gains of all unused elements//
          if State[j] = 1 then continue;
          if Part[j] = Part[d]                   //if m_j and m_d are in the same block,//
            then Gain[j] :=Gain[j] + 2C_dj;
          else Gain[j] :=gain[j] – 2C_dj;
4)    choose t to maximize T = ∑_{j=1}^{t} Temp[j]; //select the best partition P'//
5)    if T > 0 then                              //if an improvement is obtained, then//
        for j :=t + 1 to N do                    //remove elements to make P'//
          h :=History[j];                        //m_h is to be removed//
          Part[h] := (Part[h] + 1) mod 2;        //remove//
        goto 2);                                 //start next pass with the partition P'//
6)    else do                                    //if no improvement, then P is locally optimal//
        for j :=1 to N do                        //remove elements to make P//
          h :=History[j];                        //m_h is to be removed//
          Part[h] := (Part[h] + 1) mod 2;        //remove//
          if Part[h] :=0 then A_k: = A_k ∪ {m_h};
          else B_k := B_k ∪ {m_h};
end.                                             //exit with the resulting partition(A_k, B_k)//
```

Fig. 4. The algorithm MAXCUT.

nodes of the graph G_k^* obtained from a given graph G, with an associated cost matrix $C = (c_{ij})$ where c_{ij} is the cost of the edge between m_i and m_j. In G_k^*, the problem is to partition V into two balanced subsets A_k and B_k such that the weight of the associated cutset C_k^*, $W(C_k^*) = \sum_{m_i \in A_k, m_j \in B_k} c_{ij}$, is maximized, whereas the original problem in G is to find a minimum-weight admissible cutset.

The basic approach is to start with an arbitrary partition and to improve it by iteratively choosing one element (task module) from one set and moving it to the other set. This is called *one-move* operation. The element to be moved is called a *candidate* element and is chosen so as to obtain a maximum increase in cutset weight (or minimum decrease if no increase is possible). The algorithm consists of a series of passes: in

each pass, one element is moved in turn until all $|V|$ elements have been moved. At each iteration, the elements to be moved are chosen from among the ones that have not yet been moved during the pass. The $|V|$ partitions produced during the pass are examined and the one with the maximum cutset is chosen as the starting partition for the next pass. These passes will continue until no increase in cutset weight can be obtained. This optimization technique is a simple modification of Kernighan and Lin's [14].

Using the *one-move* operation instead of *pairwise-exchange* used in [14] as the basic mechanism of improving an existing partition is more suitable for the max-cut problem because we can consider the balance constraint in the weight of a cutset.

Definition 3: In some partition A_k and B_k, a *gain* is defined for each element as follows:

$$g(m_a) = \sum_{m_i \in A_k} c_{ai} - \sum_{m_j \in B_k} c_{aj}, \quad \forall\, m_a \in A_k,$$

$$g(m_b) = \sum_{m_j \in B_k} c_{bj} - \sum_{m_i \in A_k} c_{bi}, \quad \forall\, m_b \in B_k.$$

An element with the largest gain is selected as a candidate element. After moving the element, the gains of all the elements that have not yet been moved during the pass are recalculated. For example, if m_i is moved from A_k to B_k,

$$g'(m_a) = g(m_a) - 2c_{ai}, \quad \text{for all } m_a \in A_k,$$

$$g'(m_b) = g(m_b) + 2c_{bi}, \quad \text{for all } m_b \in B_k.$$

The algorithm called MAXCUT for the max-cut problem is described in Fig. 4. Especially, the N-tuple implemented in the form of an array $Part[1..N]$ describes the current partition. $History[1..N]$ is used to save the history information on one-move operations.

For time complexity analysis, we define a *pass* to be the operation involved in making one cycle from step 2 to step 5 of the algorithm MAXCUT. The computing time needed for step 2 is $O(N^2)$ since we need $O(N)$ time to compute the gain of each element. Each iteration of step 3 needs $O(N)$ computing time because of step 3.6. Thus, the total time needed for step 3 is $O(N^2)$. The computing time of $O(N)$ is sufficient for step 4 and 5. Therefore, the total computing time for a pass is $O(N^2)$.

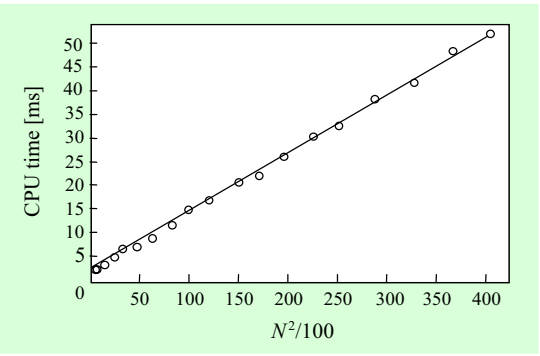

Fig. 5. Running time of the algorithm MAXCUT.

The number of passes required for the algorithm MAXCUT to terminate is small. From the experiments on all graphs with up to 200 vertices, only 2 to 6 passes were required. From these experiments, we can see that the number of passes does not strongly

```
Input: a task graph G = (V, E) with |V| = 2^n   //N = 2^n//
Output: one-to-one mapping X
Variables
    X[1..N][1..n]: boolean;                     //X[i][k] = the k-th bit assignment of m_i//
    P[1..N][0..n]: integer;                     //m_i is in P_k^{P[i][k]}//

1)  for i := 1 to N do                          //construct P_0^1//
        P[i][0] := 1;
        for j := 1 to n do
            X[i][j] := 0;                       //initialization//
2)  for k := 1 to n do
2.1)    for i := 1 to N do                      //make a graph G_k* from G//
            for j := 1 to N do
                if P[i][k-1] = P[j][k-1]        //if in the same subset//
                    then C_{i,j} := R - W_{i,j};//modify the edge weight//
2.2)    (A_k, B_k) := MAXCUT(G_k*);             //find a maxcut on G_k*//
2.3)    for i := 1 to 2^{k-1} do                //construct 2^{k-1} subsets//
            if m_i ∈ A_k then P[i][k] := 2 × P[i][k-1] - 1;
            else P[i][k] := 2 × P[i][k-1];
2.4)    for i := 1 to N do                      //k-th bit assignment//
            if m_i ∈ A_k then X[i][k] := 0;
            else X[i][k] := 1;
end.                                            //exit with the mapping X//
```

Fig. 6. The algorithm MRM.

depend on the value of N. Running times are plotted in Fig. 5, which are reasonably close to N^2.

The mapping algorithm, termed *Mapping by Repeated Maxcut (MRM)*, is shown in Fig. 6. Repeated bipartitioning is performed, using procedure MAXCUT. After each bipartitioning step, one bit is set for each modules's mapped processor, according to the outcome of the partition. Algorithm MRM comprises three phases. The first phase (step 2.1) modifies the graph G to G_k^* so as to make the next partition (A_k, B_k) admissible. The second phase (step 2.2) performs the bipartitioning using the algorithm MAXCUT, which is the most time-consuming procedure. The last phase (step 2.3 and 2.4) constructs 2^k subsets, and assigns each task module the k-th address bit for its processor mapping according to the partition obtained during phase 2. The time complexity of MRM is $O(nN^2)$, because of n iterations of the procedure MAXCUT.

Theorem 2: For a given task graph $G(V, E)$ with $|V| = 2^n$, the set $\mathbf{C}^* = \{C_k^* | 1 \le k \le n\}$ found by Algorithm MRM on the modified graphs G_k^*'s is admissible, i.e., the resulting mapping X is one-to-one.

Proof: We prove the admissibility of the set $\mathbf{C}^*$ by showing that $|P_k^1| = |P_k^2| = \cdots = |P_k^{2^k}|$ for all $1 \le k \le n$. We prove it by induction on k.

(1) For $k = 1$, the set of task modules is partitioned into P_1^1 and P_1^2 by the cutset C_1^* found by Algorithm MAXCUT. Then,

the gain of each element under the partition (P_1^1, P_1^2) must be nonpositive, because if there is any element with a positive gain, the partition (P_1^1, P_1^2) must be transformed into another partition. Without loss of generality, we can assume that the partition (P_1^1, P_1^2) is not balanced, i.e., $|P_1^1| \neq |P_1^2|$. Then $||P_1^1| - |P_1^2|| \geq 2$ because $|P_1^1| + |P_1^2| = 2^n$. Assume $|P_1^1| \geq |P_1^2| + 2$. Let $W_{i,j}$ be the original edge cost between any two modules m_i and m_j in G. Then, for each element m_a in P_1^1, the gain $g(m_a)$ is

$$\begin{aligned} g(m_a) &= \sum_{m_i \in P_1^1} c_{ai} - \sum_{m_j \in P_1^2} c_{aj} \\ &= \Big((|P_1^1| - 1) \cdot R - \sum_{m_i \in P_1^1} W_{a,i} \Big) \\ &\quad - \Big(|P_1^2| \cdot R - \sum_{m_j \in P_1^2} W_{a,j} \Big) \\ &\geq R - \Big(\sum_{m_i \in P_1^1} W_{a,i} - \sum_{m_j \in P_1^2} W_{a,j} \Big) \geq 0. \end{aligned}$$

In other words, when the partition (P_1^1, P_1^2) is not balanced and $|P_1^1| > |P_1^2|$, then all the elements in P_1^1 have a positive gain. This is a contradiction. Therefore, (P_1^1, P_1^2) is a balanced partition, i.e., $|P_1^1| = |P_1^2|$.

(2) Suppose that it holds for $k = i - 1$. Then, $|P_{i-1}^1| = |P_{i-1}^2| = \cdots = |P_{i-1}^{2^{i-1}}|$. Let the set of task modules is partitioned into A_i and B_i by C_i^* such that $A_i \cap P_{i-1}^j = P_i^{2j-1}$ and $B_i \cap P_{i-1}^j = P_i^{2j}$, for $1 \leq j \leq 2^{i-1}$. For the same reason in (1), the gain of each element must be nonnegative. Suppose that it does not hold for $k = i$. This means that there exists at least one j, $1 \leq j \leq 2^{i-1}$, such that $|P_i^{2j-1}| \neq |P_i^{2j}|$. Then $||P_i^{2j-1}| - |P_i^{2j}|| \geq 2$ because $|P_i^{2j-1}| + |P_i^{2j}| = |P_{i-1}^j| = 2^{i-1}$. Assume $|P_i^{2j-1}| \geq |P_i^{2j}| + 2$. Then, for each element m_a in P_i^{2j-1}, the gain $g(m_a)$ is

$$\begin{aligned} g(m_a) &= \sum_{m_b \in A_i} c_{ab} - \sum_{m_c \in B_i} c_{ac} \\ &= \Big((|P_i^{2j-1}| - 1) \cdot R - \sum_{m_b \in A_i} W_{a,b} \Big) \\ &\quad - \Big(|P_i^{2j}| \cdot R - \sum_{m_c \in B_i} W_{a,c} \Big) \\ &\geq R - \Big(\sum_{m_b \in A_i} W_{a,b} - \sum_{m_c \in B_i} W_{a,c} \Big) \\ &\geq 0. \end{aligned}$$

This is a contradiction. Therefore, the condition for admissibility is satisfied for $k = i$. Thus, it holds for all k by the principle of induction. □

V. EXPERIMENTAL RESULTS

To evaluate the performance of Algorithm MRM, two simulation experiments were performed on a variety of task graphs, random and regular graphs, and the results of MRM were compared with those of the greedy and the recursive mapping algorithms. The first experiment was performed for three types of random graphs: sparse, normal, and dense graphs. Sparse, normal, and dense random graphs are defined as graphs with the number of edges $2N(N-1)/14$, $3N(N-1)/14$, and $4N(N-1)/14$, respectively. For each type, the weights of edges were determined randomly within its range, i.e., the weights of edges for range-k graphs were picked randomly from the interval 1 to k. For each case, 100 runs were performed. The average cost of

Table 1. Results for random task graphs on 3-cubes ($N = 8$).

graph type	range	random		greedy		recursive		MRM		optimal	
		avg.	std.	avg.	std.	avg.	std.	avg.	std.	avg.	std.
sparse	1	13.36	4.69	10.30	4.02	9.12	3.48	8.82	3.34	8.66	3.26
	5	43.44	15.94	33.06	14.06	29.40	12.09	29.06	11.71	28.30	11.49
	10	82.39	30.55	56.82	25.78	51.86	22.93	51.28	23.06	49.76	22.29
normal	1	25.08	5.23	22.22	5.71	20.12	5.20	19.56	4.95	19.26	4.86
	5	70.95	18.24	58.33	16.52	53.92	13.62	53.12	13.34	51.90	13.14
	10	124.65	35.48	106.06	34.08	94.74	29.00	93.90	28.44	91.64	28.45
dense	1	33.28	4.88	31.12	5.31	28.76	4.54	28.40	4.43	28.22	4.55
	5	101.14	18.58	88.12	14.78	82.24	13.70	81.30	13.39	79.82	12.91
	10	178.33	32.90	163.08	32.03	153.86	29.84	151.72	29.02	149.70	29.61

the mappings and the standard deviation of the costs are both recorded for each test case. For each case, the cost of randomly-generated mappings is also recorded, to provide a basis for comparing the mappings obtained by the greedy, the recursive mapping, and the MRM algorithms.

Table 1 summarizes the results obtained for 3-dimensional random task graphs, where optimal solutions were obtained using a simple exhaustive search algorithm. All of the three methods generated mappings that were significantly superior to randomly-generated mappings. The mappings generated by the recursive mapping algorithm were generally slightly better than those by the greedy algorithm. Among them, the mappings generated by the MRM algorithm were the best for all cases. The MRM algorithm generated nearly-optimal mappings for 3-dimensional task graphs. For n-dimensional task graphs, the results are shown in Tables 2−3, for $n = 6$, and 10, respectively. We do not know the optimal mapping for $n > 3$, since it is impratical to search for an optimal mapping exhaustively. (Even a 4-dimensional task graph already has 16!, about 2×10^{13}, ways to map.) From these results, one can see that the MRM algorithm increasingly outperforms the greedy and the recursive mapping algorithms as the value n increases.

The second experiment was performed for some regular 10-dimensional graphs by setting the weight of each edge to one in order to evaluate the performance of each algorithm with clarity. First, we establish

Table 2. Results for random graphs on 6-cubes ($N = 64$).

graph type	range	random		greedy		recursive		MRM	
		avg.	std.	avg.	std.	avg.	std.	avg.	std.
sparse	1	1952	63.64	1794	66.80	1516	58.98	1449	56.86
	5	5866	240.72	5128	240.18	4397	205.10	4260	199.23
	10	11154	446.02	9335	449.17	7997	382.48	7733	371.81
normal	1	3176	69.96	3040	72.22	2710	64.96	2552	66.81
	5	9381	252.27	8850	284.58	7875	240.50	7593	243.69
	10	17594	507.10	16179	503.35	14458	474.63	13982	463.13
dense	1	4353	66.75	4277	66.92	3968	68.38	3889	67.21
	5	13211	289.84	12615	301.65	11581	276.18	10982	272.01
	10	24101	554.89	23051	562.27	21141	539.34	20167	535.88

Table 3. Results for random graphs on 10-cubes ($N = 1024$). All data are downscaled to 1/1000.

graph type	range	random		greedy		recursive		MRM	
		avg.	std.	avg.	std.	avg.	std.	avg.	std.
sparse	1	786	1.60	765	1.67	745	1.41	737	1.52
	5	2357	5.22	2328	5.20	2218	5.18	2192	4.83
	10	4719	14.21	4701	15.34	4439	13.17	4387	13.14
normal	1	1310	2.15	1299	2.22	1267	2.38	1258	2.33
	5	3933	6.69	3924	6.37	3773	5.85	3742	5.79
	10	7177	16.15	6785	17.18	5943	16.34	5830	16.32
dense	1	1867	1.67	1802	1.71	1675	1.63	1637	1.62
	5	5583	6.29	5282	6.42	4860	6.98	4735	6.60
	10	10179	17.65	9692	19.20	8719	18.58	8594	18.25

a task graph G_H which is exactly isomorphic to a hypercube. Obviously, the optimal mapping cost C_o for G_H is equal to the number of edges of the hypercube, i.e., $C_o = (N \log N)/2$. Then we add a random edge to G_H and calculate the distance d when this edge is mapped to a hypercube. Since an edge is added, the new cost C_o is equal to the old $C_o + d$. We may keep adding edges to G_H and update C_o. When a very few edges are added, C_o is the optimal cost. But, when a large number of edges are added, the C_o-based solution may not always be optimal. However, C_o is not going to be too bad as compared to the real optimal cost. To evaluate the mapping performance of the algorithm with graphs which are subgraph-isomorphic to a hyper-

Table 4. Results for regular graphs on 10-cubes ($N = 1024$).

graph type	random		greedy		recursive		MRM		optimal	
	avg.	std.	avg.	std.	avg.	std.	avg.	std.	avg.	std.
hypercube	25652	110.92	17116	0.00	5120	0.00	5120	0.00	5120	0.00
SUB1	25645	135.50	17112	0.97	5119	0.00	5119	0.00	5119	0.00
SUB2	25653	116.61	17109	191.36	5118	0.00	5118	0.00	5118	0.00
ADD1	25647	138.79	17121	1.55	5125	1.64	5125	1.64	5125	1.64
ADD2	25589	105.92	17150	105.12	5133	3.63	5133	3.63	5133	3.63
mesh	9911	51.35	4562	0.00	2712	0.00	1984	0.00	1984	0.00

cube, we subtract edges from G_H and calculate the total distance d of the subtracted edges. In this case, $C_o - d$ is guaranteed to be optimal.

We simulated adding or subtracting exactly one or two edges to hypercube-isomorphic graphs and recorded the mapping results. We also performed simulations on regular meshes. A $2^{\lfloor \frac{n}{2} \rfloor} \times 2^{\lceil \frac{n}{2} \rceil}$ mesh was used as an n-dimensional mesh, which is subgraph-isomorphic to an n-cube. Obviously, the optimal cost for a $2^{\lfloor \frac{n}{2} \rfloor} \times 2^{\lceil \frac{n}{2} \rceil}$ mesh is $(2^{\lfloor \frac{n}{2} \rfloor} - 1)2^{\lceil \frac{n}{2} \rceil} + 2^{\lfloor \frac{n}{2} \rfloor}(2^{\lceil \frac{n}{2} \rceil} - 1)$. The results are summarized in Tables 6−10. As can be seen in the table, the mappings generated by the recursive mapping or MRM algorithms were much better than those by the greedy algorithm for ADD1, ADD2, or meshes. The recursive mapping algorithm found optimal solutions for all the regular graphs except meshes. On the other hand, the mappings generated by the MRM algorithm were optimal for all the regular graphs tested.

VI. CONCLUDING REMARKS

In this paper, we have presented an efficient approach for the problem of one-to-one mapping of task modules of a parallel program into the processors of a hypercube with the objective of minimizing the total communication cost. The proposed MRM algorithm was based on a repeated bipartitioning strategy. We proposed a cutset formulation which transforms the hypercube mapping problem into the problem of finding a minimum-weight admissible set of cutsets on a given task graph. An efficient graph bipartitioning algorithm has been developed, which guarantees exactly-balanced partitions. This bipartitioning algorithm was applied successfully to the mapping

problem in hypercube multicomputers by using a graph modification technique.

To evaluate the performance of the MRM algorithm, we have performed a number of experiments for random and regular task graphs. Experimental results indicate that the MRM algorithm performs well on both random and regular graphs. The MRM algorithm is shown to increasingly outperform the greedy and recursive mapping algorithms as the number of processors increases. In real problems, task graphs are often regular or contain some known structures. The MRM algorithm is shown to be quite effective even for regular graphs, such as hypercube-isomorphic or 'almost' isomorphic graphs and meshes.

There are several related issues that warrant further investigation. The cost function presented here addresses only interprocessor communication. It needs to be changed so as to achieve other goals such as good load balancing, a small response time for the task, and efficient utilization of system resources in general. The simplified parallel execution model here does not take into account the precedence relationships between task modules, nor the time-dependent task behavior. It is interesting to extend our results by increasing the complexity of the model to include such factors.

REFERENCES

[1] M.-S. Chen and K.G. Shin, "Embedding of Interacting Task Modules into a Hypercube," *Proc. of the Second Conf. on Hypercube Concurrent Computers and Applications*," Oct. 1986, pp. 122–129.

[2] B. Becker and H.U. Simon, "How Robust is the n-cube," *Proc. of 27th Annual Symp. on Foundations Computer Sci.*, Oct. 1986, pp. 283–291.

[3] J. Kim, C.R. Das, and W. Lin, "A Processor Allocation Scheme for Hypercube Computers," *Proc. of the 1989 Conf. on Parallel Processing*, Aug. 1989, pp. 231–238.

[4] J. Rattner, "Concurrent Processing: A New Direction in Scientific Computing," *Proc. of AFIPS Conf.*, Vol. 54, 1985, pp. 157–166.

[5] J.P. Hayes, T.N. Mudge *et al.*, "Architecture of a Hypercube Supercomputer," *Proc. of the 1986 Conf. on Parallel Processing*, Aug. 1986, pp. 653–660.

[6] J.C. Peterson *et al.*, "The Mark III Hypercube Ensemble Concurrent Processor," *Proc. of the 1985 Conf. on Parallel Processing*, Aug. 1985, pp. 71–73.

[7] W.D. Hills, *The Connection Machine*, MIT Press, Cambridge, MA, 1985.

[8] S.H. Bokhari, "On the Mapping Problem," *IEEE Trans. on Computers*, Vol. C-30, No. 3, Mar. 1981, pp. 207–214.

[9] M.-S. Chen and K.G. Shin, "Processor Allocation in an n-cube Multiprocessor Using Gray Codes," *IEEE Trans. on Computers*, Vol. C-36, No. 12, Dec. 1987, pp. 1396–1407.

[10] A.M. van Tilborg and L.D. Wittie, "Wave Scheduling–Decentralized Scheduling of Task Forces in Multicomputers," *IEEE Trans. on Computers*, Vol. C-33, No. 9, Sep. 1984, pp. 835–844.

[11] D.W. Krumme, K.N. Venkataraman, and G. Cybenko, "Hypercube Embedding is NP-Complete," *Proc. of the First Conf. on Hypercube Concurrent Computers and Applications*, Aug. 1985, pp. 148–157.

[12] W.-K. Chen and E.F. Gehringer, "A Graph-oriented Mapping Strategy for a Hypercube," *Proc. of the Third Conf. on Hypercube Concurrent Computers and Applications*, Jan. 1988, pp. 200−209.

[13] F. Ercal, J. Ramanujan, and P. Sadayappan, "Task Allocation onto a Hypercube by Recursive Mincut Bipartitioning," *Proc. of the Third Conf. on Hypercube Concurrent Computers and Applications*, Jan. 1988, pp. 210−221.

[14] B.W. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Syst. Tech. J.*, Vol. 49, Feb. 1970, pp. 291−307.

[15] C.-H. Lee, C.-I. Park, and M. Kim, "Efficient Algorithm for Graph-Partitioning Problem Using a Problem Transformation Method," *Computer-Aided Design*, Vol. 21, No. 10, Dec. 1989, pp. 611−618.

[16] B.-R. Tsai and K.G. Shin, "Communication-oriented Assignment of Task Modules in Hypercube Multicomputers," *Proc. of 12-th Int'l Conf. on Distributed Comput. Syst.*, June 1992, pp. 38−45.

[17] P. Kermani and L. Kleinrock, "Virtual Cut-through: A New Computer Communication Switching Technique," *Comput. Networks*, Vol. 3, 1979, pp. 267−286.

[18] H.S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms," *IEEE Trans. on Software Engineering*, Vol. SE-3, No. 1, Jan. 1977, pp. 85−93.

[19] C.-H. Lee, D. Lee, and M. Kim, "Optimal Task Assignment in Linear Array Networks," *IEEE Trans. on Computers*, Vol. C-41, No. 7, July 1992, pp. 877−880.

[20] C.-H. Lee and Kang G. Shin, "Optimal Task Assignment in Homogeneous Systems," *IEEE Trans. on Parallel and Distributed Systems*, Vol. 8, No. 2, Feb. 1997, pp. 119−129.

Joo-Man Kim received the B.S. degree in computer science from Soongsil University, Seoul, Korea in 1984 and the M.S. degree in computer engineering from Chungnam National University, Taejon, Korea in 1998. He received the national qualification for professional engineer in information technology from the Korean Government in 1994. He joined ETRI in 1985, where he is currently working as principal member of research staff at the OS research team. His research interests include parallel processing, operating system, real-time kernel, diagnostic and fault tolerant computing.

Cheol-Hoon Lee received the B.S. degree in electronics engineering from Seoul National University, Seoul, Korea in 1983, and the M.S. and Ph.D. degrees in electrical engineering from Korea Advanced Institute of Science and Technology, Seoul, Korea in 1988 and 1992, respectively. From 1983 to 1994, he worked for Samsung Electronics Company in Seoul, Korea as a researcher. From 1994 to 1995, he was with the University of Michigan, Ann Arbor, as a research scientist at the Real-Time Computing Laboratory. Since 1995, he has been an assistant processor in the Department of Computer Engineering, Chungnam National University, Taejon, Korea. His research interests include parallel processing, operating system, real-time and fault-tolerant computing, and microprocessor design.