1990

Mapping Finite Element Graphs on Hypercubes

Yeh-Ching Chung

Sanjay Ranka
Syracuse University

SCHOOL OF COMPUTER AND INFORMATION SCIENCE

Syracuse University

School of Computer and Information Science
4–116, CST Center for Science and Technology
Syracuse University
Syracuse, NY 13244–4100
(315) 443–4457

Suite 4-116
Center for Science and Technology
Syracuse, New York 13244-4100

Mapping Finite Element Graphs on Hypercubes

Yeh-Ching Chung and *Sanjay Ranka*

School of Computer and Information Science
4-116, CST Center for Science and Technology
Syracuse University
Syracuse, NY 13244-4100
(315) 443-4457

Abstract - The *2-way stripes partition mapping* and the *greedy assignment mapping* are proposed to map *finite element graphs* (FEGs) onto hypercubes. They can be used to map both 2-D and 3-D FEGs on hypercubes. The 2-way stripes partition mapping is a two phase mapping approach. In the first phase, a 2-way stripes partition approach is used to achieve low communication cost. In the second phase, the load transfer heuristic is used to balance the computational load among processors. The greedy assignment mapping tries to minimize the communication cost and balance the computational load of processors simultaneously.

1. INTRODUCTION

In parallel computing, it is important to map a parallel program onto a parallel computer such that the total execution time of a parallel program is minimized. In general, a parallel program and a parallel computer can be represented by a *task graph* (TG) and a *processor graph* (PG), respectively. For a TG, nodes represent tasks of a parallel program and edges denote the data communication needed between tasks. The weights associated with nodes and edges represent the computational load and communication cost, respectively. For a PG, nodes and edges denote processors and communication channels, respectively. By using the graph model, the mapping problem becomes a task allocation problem.

In the task allocation problem, we try to distribute the computational load of a parallel program to the processors of a parallel computer as evenly as possible (the load balance criterion (LBC)) and minimize the communication cost of processors (the minimum communication cost criterion (MCCC)). The optimal assignment of tasks to processors in order to minimize the total execution time is known to be NP-complete [GaJo79]. This means that the optimal solution is intractable. Therefore, satisfactory suboptimal solutions are generally sought.

In this paper, we will discuss how to map *finite element graphs* (FEGs) onto *hypercubes*. Our schemes are general and are applicable to a wide variety of PGs. The finite element method (FEM) is a widely used method for the structural modeling of physical system [LaPi83]. Due to the properties of compute-intensiveness and compute-locality, it is very attractive to implement this method on parallel computers [BeBo87] [Bokh81] [Jord78] [SaEr87]. The number of nodes in a FEG is usually greater than the number of processors in a parallel computer. It is important to partition a FEG into M modules such that the computational load of modules are equal and the communication cost among modules are minimized, where M is the number of processors of a parallel computer.

In [BeBo87], a *binary decomposition* approach was used to partition a nonuniform mesh graph (a kind of FEG) into modules such that each module has the same computational load. These modules were then mapped onto meshes, trees, and hypercubes. This method does not try to minimize the communication cost. [SaEr87] proposed the *nearest-neighbor mapping*

approach to map planar FEGs onto meshes. It used the *stripes partition* (*stripes mapping*) strategy to minimize the communication cost among processors and then used the *boundary refinement* heuristic to balance the computational load among processors. All of the FEGs used by those mapping approaches are two dimensional graphs. They cannot be trivially extended to three dimensional FEGs. In a structural modeling system, most of the cases encountered are three dimensional FEGs. Therefore, it is important to show that a mapping approach can be applied to all kinds of FEGs.

We propose two mapping approaches, the *2–way stripes partition mapping* and the *greedy assignment mapping*, which can be applied to all kinds of FEGs. The 2–way stripes partition mapping tries to minimize the communication cost by assigning a node and its *neighbor nodes* of a FEG to the same processor or *neighbor processors* of a hypercube (the definitions of neighbor node and neighbor processor will be defined latter). Since the computational load may not be equally assigned to each processor by using this approach, the *load transfer heuristic* is used to balance the computational load among processors. The greedy assignment mapping tries to minimize the communication cost and balance the computational load simultaneously. It assigns one node of a FEG to a particular processor of a hypercube at a time according to the current status of the neighbor nodes of that node.

In our analysis, we assume that the number of edges (E), the number of finite elements (F), and the number of nodes (N) differ from each other by a multiplicative constant, i.e., $E = c_1F = c_2N$, for some constants c_1 and c_2. These assumptions are true for most of the FEGs. The computational complexities of the 2–way stripes partition mapping and the greedy assignment mapping are $O(MN^2\log M)$ and $O(N\log^2 M + N\log N)$, respectively, where M is the number of processors of a hypercube and N is the number of nodes of a FEG. Our simulation results show that the speedups for the 2–way stripes partition mapping are better than those for the greedy assignment mapping when the LBC is achieved in both approaches. However, the greedy approach gives good performance at a much lower cost.

This paper is organized as follows. Section 2 introduces the definitions and notations used in this paper. The cost models of mapping a FEG onto a hypercube are also described in this section. The 2–way stripes partition mapping and the greedy assignment mapping are

addressed in Sections 3 and 4, respectively. In Section 5, we compare the mapping results of these two approaches.

2. PRELIMINARIES

2.1. Hypercubes

Hypercubes or n-cubes are highly concurrent loosely coupled multiprocessors based on the binary n-cube network and are referred to by different names (such as *cosmic cube* [Seit85], n-cube [HaMu86], *binary n-cube* [BhAg84], etc.).

Definition 1 : An n-dimensional hypercube Q_n, for $n > 1$, can be recursively defined in terms of the graph product $\times$ as follows [Hara69]:

$$Q_n = K_2 \times Q_{n-1}, \tag{1}$$

where $K_2 = Q_1$ is the complete 2-node graph. ■

From Definition 1, we know that an n-dimensional hypercube consists of 2^n processors. The address of each processor can be represented by an n-bit binary number ranging from 0 to 2^n-1.

Definition 2: In an n-cube, two processors p_x and p_y are *adjacent processors* if the address of p_x differs from that of p_y by one bit. ■

In Figure 1, n-dimensional hypercubes are shown, for $n = 1$, 2, and 3. We use symbol M to denote the total number of processors of a hypercube throughout this paper.

2.2. Finite Element Graphs (FEGs)

The finite element method (FEM) is a widely used technique to solve the *partial differential equations* (PDEs) by using iterative approach. In the finite element model, an object can be viewed as a FEG. A FEG is a connected and undirected graph which consists of a number of rectilinear 4-node finite elements (FEs).

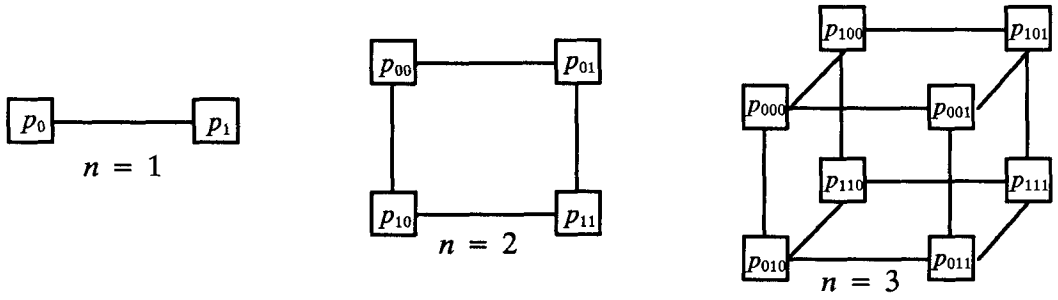


Figure 1 : An example of n-cubes, for $n = 1$, 2, and 3

Definition 3 : A FEG is a 2-D FEG if it is a planar graph. ■

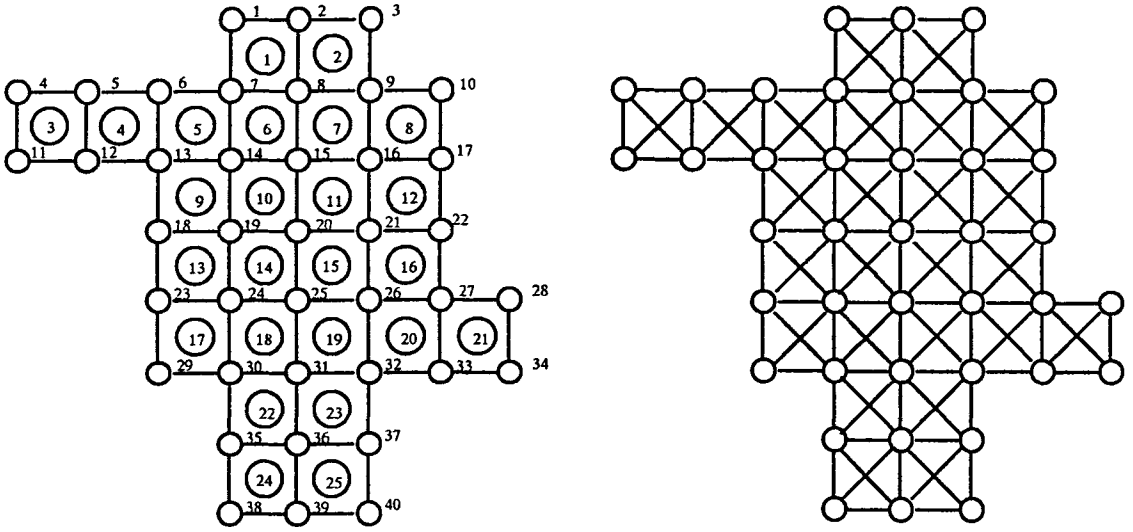
Definition 4 : In a FEG, two nodes *node*(x) and *node*(y) are *adjacent nodes* if $<$*node*(x), node(y)$>$ is an edge of the FEG. ■

Definition 5 : In a FEG, two nodes *node*(x) and *node*(y) are *neighbor nodes* if *node*(x) and node(y) are in the same FE. ■

In Figure 2(a), for example, a 40-node FEG which consists of 25 FEs is shown (The circled and uncircled numbers denote the FE numbers and node numbers, respectively.). Let $FE(x)$ denote the set of nodes which form FE x, $ADJ(node(y))$ denote the set of adjacent nodes of *node*(y), $NB(node(y))$ denote the set of neighbor nodes of *node*(y), and $\#(NB(node(y)))$ denote the cardinality of $NB(node(y))$, i.e., the number of nodes in $NB(node(y))$. We have $FE(6)$ = {*node*(7), *node*(8), *node*(14), *node*(15)}, $ADJ(node(14))$ = {*node*(7), *node*(13), *node*(15), *node*(19)}, $NB(node(14))$ = {*node*(6), *node*(7), *node*(8), *node*(13), *node*(15), *node*(18), *node*(19), *node*(20)}, and $\#(NB(node(14)))$ = 8. It is clear that $ADJ(node(y))$ is a subset of $NB(node(y))$, i.e., $ADJ(node(y)) \subseteq NB(node(y))$. In this paper, we assume that the number of edges (E), the number of finite elements (F), and the number of nodes (N) differ from each other by a multiplicative constant, i.e., $E = c_1F = c_2N$, for some constants c_1 and c_2. These assumptions are true for most of the FEGs. We also assume that the degree of every node in a FEG is upper

bounded by a constant, i.e. $\#(ADJ(node(y)))$ is a constant. This assumption implies that $\#(NB(node(y)))$ is also a constant.

In a FEG, a node represents a particular amount of computation. Each node has the same computational load and can be executed independently. Each node has to send data to its neighbor nodes after completing its computation and all the nodes have to finish their communication before they can commence next iteration. The communication needed between nodes in the FEG of Figure 2(a) are shown in Figure 2(b). We use symbol N to denote the number of nodes of a FEG throughout this paper.



(a) : A 40-node FEG with 25 FEs. (b) The communication needed between nodes.

Figure 2 : An example of a 40-node FEG and the communication needed between nodes.

2.3. The Cost Models of Mapping FEGs onto Hypercubes

From the parallel processing point of view, a FEG can be characterized as a *task interaction graph* (TIG) [SaEr87]. In a TIG, nodes represent tasks and edges denote the

communication needed between tasks. All the tasks can be executed independently and simultaneously, i.e., the temporal dependencies of tasks are not represented explicitly.

To map an N–node FEG onto an M–processor hypercube, we need to assign the nodes of a FEG to the processors of a hypercube. There are M^N mapping ways. The total execution time of a FEG on a hypercube under a particular mapping MAP_i is defined as follows:

$$T_{par}(MAP_i) = \max\{load_i(p_j)\} \times T_{task} + C_i(P), \quad (2)$$

where $T_{par}(MAP_i)$, $load_i(p_j)$, T_{task}, and $C_i(P)$ represent the total execution time, the computational load assigned to processor p_j, the time to execute a task on a processor, and the communication cost of processors under mapping MAP_i, respectively, where $i = 1, ..., M^N$ and $j = 0, ..., M-1$.

The computational load assigned to each processor of a hypercube is equal to the nodes of a FEG assigned to it. Since the processor with the maximal computational load determines the computational cost of a mapping, Equation 2 employs the synchronous communication mode implicitly, i.e., the communication between processors cannot be started until all the processors have completed their computations.

If we assign the four nodes of a FE to different processors, there exists at least one pair of nodes in a FE such that the communication distance of this pair of nodes in a hypercube is greater than or equal to 2. In this paper, we consider only mappings such that the communication distance between neighbor nodes of a FEG in a hypercube is less than or equal to 2.

Definition 6 : In an n–cube, any two processors whose addresses differ by at most two bits are *neighbor processors.* ■

Definition 7 : A mapping is a *neighbor mapping* if any two neighbor nodes (nodes corresponding to a FE) of a FEG are assigned to the same processor or two neighbor processors of a hypercube. ■

From Definitions 6 and 7, we have the following lemma.

Lemma 1 : To map a FEG onto a 2–cube, any mapping approach is a neighbor mapping.■

In our communication models, we assume that every processor can communicate with all its adjacent processors in one step. Since we use the synchronous communication mode, $C_i(P)$ is defined as follows:

$$C_i(P) = \sum_{j=1}^{S} (T_{setup} + \max_j\{c_{kl}\} \times T_c), \tag{3}$$

where S is the number of steps to finish the data communication among processors, T_{setup} is the setup time of the I/O channel, $\max_j\{c_{kl}\}$ is the maximal amount of data sent from p_k to p_l in step j, and T_c is the data transmission time of the I/O channel per word. An I/O channel between two adjacent processors, p_i and p_j, of a hypercube is a *bidirectional channel* if p_i and p_j can send data to each other simultaneously; otherwise, it is a *unidirectional channel.*

If the I/O channel used in a hypercube is bidirectional (the bidirectional communication model), algorithm *bidirectional_comm_cost* is used to compute the value of $C_i(P)$.

---

algorithm bidirectional_comm_cost(X)

/* X is the intermediate processor matrix. $\forall x_{ij} \in X$, if $p_i = a_{n-1}\ldots a_{k+1}a_k a_{k-1}\ldots a_0$ and $p_j = b_{n-1}\ldots b_{k+1}\overline{a}_k a_{k-1}\ldots a_0$, then $x_{ij} = a_{n-1}\ldots a_{k+1}\overline{a}_k a_{k-1}\ldots a_0$ */

1. Compute the communication cost matrix C according to a particular mapping;
2. $C_i(P) = 0$;

/* For the neighbor mapping, this loop is executed at most twice */

3. **while** ($(\exists\, c_{ab} > 0)$ **and** (p_a and p_b are neighbor processors)) **do**
4. { $\forall\, c_{ij} > 0, 0 \le i, j \le M-1$, send data c_{ij} from p_i to x_{ij}; Update C and $C_i(P)$; }
5. **return**($C_i(P)$);

end_of_bidirectional_comm_cost

The initialization of the communication cost matrix requires $O(M\log^2 M)$ time*. Line 1 requires $O(\sum_{i=1}^{N} \#(NB(node(i)))) = O(N)$ time; line 2 requires c_1 time; line 3 requires c_2 time; line 4 requires $O(M\log^2 M)$ time, and line 5 requires c_3 time, where c_1, c_2, and c_3 are constants. Lines 3 and 4 form a loop and this loop is executed at most twice. The computational complexity of this algorithm is equal to $O(N + c_1 + 2 \times (c_2 + M\log^2 M) + c_3) = O(N + M\log^2 M)$. The communication behavior of algorithm *bidirectional_comm_cost* is shown in Figure 3(a). In Figure 3(a), S is equal to 2, $\max_1\{c_{kl}\} = c_{01} + c_{03} = c_{10} + c_{12} = c_{21} + c_{23} = c_{30} + c_{32} = 2$, and $\max_2\{c_{kl}\} = c_{02} = c_{13} = c_{20} = c_{31} = 1$. We can derive that $C_i(P) = 2 \times T_{setup} + (2 + 1) \times T_c = 2 \times T_{setup} + 3 \times T_c$.

If the I/O channel used in a hypercube is unidirectional (the unidirectional communication model), algorithm *unidirectional_comm_cost* is used to compute the value of $C_i(P)$.

---

algorithm unidirectional_comm_cost(X)

/* X is the intermediate processor matrix. $\forall x_{ij} \in X$, if $p_i = a_{n-1}\ldots a_{k+1}a_k a_{k-1}\ldots a_0$ and $p_j = b_{n-1}\ldots b_{k+1}\overline{a}_k a_{k-1}\ldots a_0$, then $x_{ij} = p_l = a_{n-1}\ldots a_{k+1}\overline{a}_k a_{k-1}\ldots a_0$ */

1. Compute the communication cost matrix C according to a particular mapping;
2. *toggle* = 0; $C_i(P) = 0$;
   /* For the neighbor mapping, this loop is executed at most four times */
3. **while** ($(\exists\ c_{ab} > 0)$ **and** (p_a and p_b are adjacent processors)) **do**
   /* Set the communicating direction of $channel_{il}$ from p_i to p_l if $p_i = a_{n-1}\ldots a_{k+1}a_k a_{k-1}\ldots a_0$, $p_l = a_{n-1}\ldots a_{k+1}\overline{a}_k a_{k-1}\ldots a_0$, $p_j = b_{n-1}\ldots b_{k+1}\overline{a}_k a_{k-1}\ldots a_0$, a_k = *toggle*, and $c_{ij} > 0$ */
4. $\{\ \forall c_{ij} > 0, 0 \le i, j \le M\text{-}1, p_i = a_{n-1}\ldots a_{k+1}a_k a_{k-1}\ldots a_0$, $p_j = b_{n-1}\ldots b_{k+1}\overline{a}_k a_{k-1}\ldots a_0$, $x_{ij} = p_l = a_{n-1}\ldots a_{k+1}\overline{a}_k a_{k-1}\ldots a_0$, and a_k = *toggle*,

---

* Note that each processor can only have $O(\log^2 M)$ neighbor processors. The other values of the matrix are useless. For ease of presentation, $\forall c_{ij} > 0$ in the algorithm refers to only c_{ij} in which i and j are neighbor processors. Thus the complexity of this operation is $O(M\log^2 M)$ as compared to obvious $O(M^2)$. This assumption is true for the rest of the presentation.

5. **if** ($channel_{il}$ is available **or** $channel_{il} = p_i \rightarrow p_l$) **then**
{ $channel_{il} = p_i \rightarrow p_l$; /*The communicating direction of $channel_{il}$ is set from p_i to p_l */
Send data c_{ij} from p_i to p_l; Update C and $C_i(P)$; }
/* If there are some channels $channel_{jl}$ are still available after steps 4–5 are executed, set the communicating direction of $channel_{jl}$ from p_j to p_l if $p_j = b_{n-1}\ldots b_{k+1}\overline{a}_k a_{k-1}\ldots a_0$, $p_l = b_{n-1}\ldots b_{k+1}a_k a_{k-1}\ldots a_0$, $p_i = a_{n-1}\ldots a_{k+1}a_k a_{k-1}\ldots a_0$, $a_k = toggle$, and $c_{ji} > 0$. */
6. $\forall\, c_{ji} > 0,\ 0 \le i, j \le M\text{–}1, p_i = a_{n-1}\ldots a_{k+1}a_k a_{k-1}\ldots a_0$, $p_j = b_{n-1}\ldots b_{k+1}\overline{a}_k a_{k-1}\ldots a_0$, and $x_{ji} = p_l = b_{n-1}\ldots b_{k+1}a_k a_{k-1}\ldots a_0$, and $a_k = toggle$,
7. **if** ($channel_{jl}$ is available **or** $channel_{jl} = p_j \rightarrow p_l$) **then**
{ $channel_{jl} = p_j \rightarrow p_l$; Send data c_{ji} from p_j to p_l; Update C and $C_i(P)$; }
8 $toggle = (toggle + 1)$ **mod** 2;
9. }
10. **return**($C_i(P)$);

end_of_compute_comm_cost

In algorithm *unidirectional_comm_cost*, line 1 requires $O(\sum_{i=1}^{N} \#(NB(node(i)))) = O(N)$ time; line 2 requires c_1 time; line 3 requires c_2 time; lines 5 and 7 require c_3 time; line 8 requires c_4 time; and line 10 requires c_5 time, where c_1, c_2, c_3, c_4, and c_5 are constants. Lines 3 to 9, 4 to 5, and 6 to 7 form loops and these loops have $O(c)$, $O(M\log^2 M)$, and $O(M\log^2 M)$ iterations, respectively, where c is a constant. The computational complexity of this algorithm is equal to $O(N + c_1 + c \times (c_2 + M\log^2 M \times c_3 + M\log^2 M \times c_3 + c_4) + c_5) = O(N + M\log^2 M)$. An example of the communication behavior of algorithm *compute_comm_cost* is shown in Figure 3(b). In Figure 3(b), S is equal to 4, $\max_1\{c_{kl}\} = c_{01} + c_{03} = c_{21} + c_{23} = 2$, $\max_2\{c_{kl}\} = c_{10} + c_{12} = c_{30} + c_{32} = c_{31} = 2$, $\max_3\{c_{kl}\} = c_{02} = c_{13} = 1$, and $\max_4\{c_{kl}\} = c_{20} = 1$. We can derive that $C_i(P) = 4 \times T_{setup} + (2 + 2 + 1 + 1) \times T_c = 4 \times T_{setup} + 6 \times T_c$.

Let T_{seq} denote the total execution time of a FEG on a 0–cube which contains only one processor. The speedup of a mapping MAP_i is defined as follows:

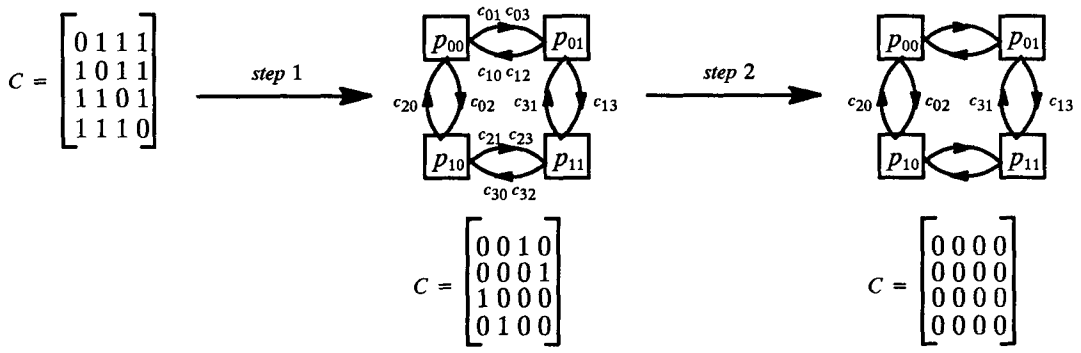


Figure 3(a) : The communication behavior of algorithm *bidirectional_comm_cost.*

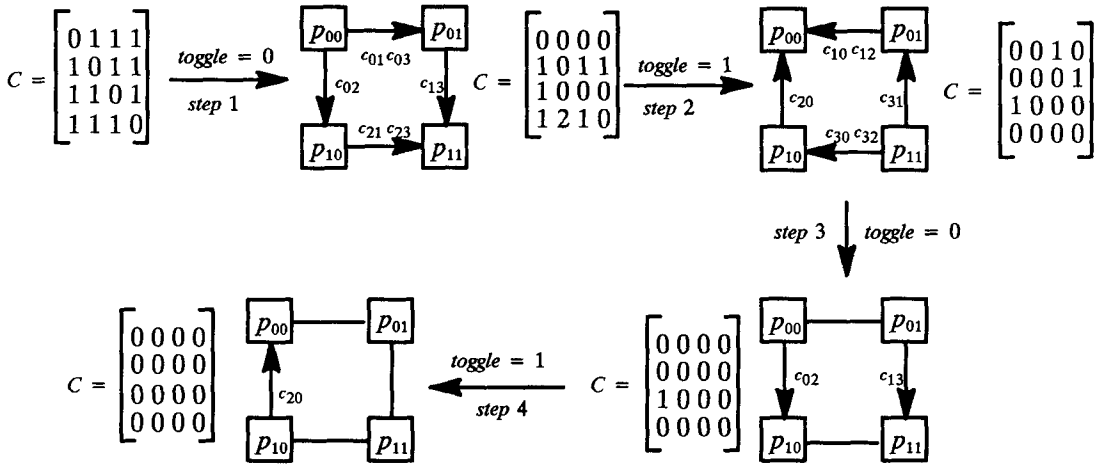


Figure 3(b) : The communication behavior of algorithm *unidirectional_comm_cost.*

$$SpeedUp(MAP_i) = \frac{T_{seq}}{T_{par}(MAP_i)} \tag{4}$$

The objective of mapping a FEG onto a hypercube is to minimize the total execution time, i.e., $min\{T_{par}(MAP_i)\}$, or maximize the speedup, i.e., $max\{SpeedUp(MAP_i)\}$, where i = 1, 2,

..., M^N. From Equation 2, we know that the processor with the maximal computational load; and the communication cost of processors determine the total execution time of a FEG on a hypercube under a particular mapping. Since our main objective is to minimize these quantities, there are three ways to achieve the objective of a mapping. (1) First minimize communication cost, then balance the computational load. (2) First balance the computational load, then minimize the communication cost. (3) Minimize the communication cost and balance the computational load simultaneously. The 2-way stripes partition mapping and the greedy assignment mapping adopt approaches (1) and (3), respectively.

3. THE 2-WAY STRIPES PARTITION MAPPING

The 2-way stripes partition mapping is a two phase mapping approach. In the first phase (partition and allocation phase), it uses the 2-way stripes partition heuristic and stripes merge to partition an N-node FEG into M modules and each module contains m tasks, where $0 \leq m \leq N$. These modules are assigned to processors by using the *binary reflected Gray code* (BRGC). Since the computational load may not be equally assigned to each processor in this phase, we will try to balance the computational load among processors by using the *load transfer heuristic* in the second phase (the load balancing phase).

3.1. Phase I : The 2-way Stripes Partition and Stripes Allocation

The basic approach used in the 2-way stripes partition to partition a FEG into modules is the *stripes partition* approach. The stripes partition approach starts at an arbitrary node *node*(x) of a FEG and labels it as 0. Next, the neighbor nodes of *node*(x), *NB*(*node*(x)), are labeled as 1. This process continues till each node in a FEG is assigned a label. Our approach is more general than the stripes partition approach of [SaEr87]. The approach proposed in [SaEr87] can only be used to partition 2-D FEGs and has some restrictions. Our approach removes the restrictions in [SaEr87] and can be used to partition any kind of FEGs. The 2-way

stripes partition uses the stripes partition method twice. The partitioning starts at *node*(1) and *node*($\lfloor \frac{N}{2} \rfloor + 1$), respectively. By using this method, the labels assigned to each node can be denoted by a 2-tuple (l_1, l_2), where l_1 and l_2 denote the labels assigned to a node by the first and second stripes partition, respectively.

The next step is to assign these nodes to processors according to their labels. By using the 2-way stripes partition, the 2-tuple labels assigned to nodes imply the following lemma.

Lemma 2: For any two neighbor nodes *node*(i) and *node*(j) with labels (l_{i_1}, l_{i_2}) and (l_{j_1}, l_{j_2}), respectively, we have $|l_{i_1} - l_{j_1}| \leq 1$ and $|l_{i_2} - l_{j_2}| \leq 1$. ■

To assign nodes to processors according to their labels, we need to flatten an n-cube into a two dimensional form. For any two neighbor processors *processor*(i_1, j_1) and *processor*(i_2, j_2) in a mesh, we have $|i_1 - i_2| \leq 1$ and $|j_1 - j_2| \leq 1$. To map a FEG onto a mesh, the neighbor mapping can be easily achieved by assigning *node*(i) with labels (l_{i_1}, l_{i_2}) to *processor*(l_{i_1}, l_{i_2}). Since an n-cube can emulate 1×2^n, $2 \times 2^{n-1}$, ..., $2^n \times 1$ meshes, we will try all cases. A binary reflected Gray code (BRGC) [ChSa86] is defined as follows:

$$N_k = \begin{cases} (0, 1) & \text{if } k = 1 \\ 0N_{k-1} + 1N_{k-1}^* & \text{if } k > 1 \end{cases} \tag{5}$$

where + and * denote *sequence concatenation* and *sequence reversal operations*, respectively. From Equation 5, we know that $N_1 = (0, 1)$, $N_1^* = (0, 1)^* = (1, 0)$, $N_2 = 0N_1 + 1N_1^* = 0(0, 1) + 1(1, 0) = (00, 01) + (11, 10) = (00, 01, 11, 10)$, $N_3 = (000, 001, 011, 010, 110, 111, 101, 100)$, $N_3(0) = 000$, and $N_3(3) = 010$. Note that $N_k(r)$ denotes the $(r+1)$th element of N_k, where $r = 0, ..., 2^k-1$. To embed a $2^x \times 2^y$ mesh in a $(x+y)$-cube, we assign *processor*(i, j) in a mesh to the processor in the $(x+y)$-cube according to the following equation:

$$addr : processor(i, j) \rightarrow N_x(i) \hat{} N_y(j), \tag{6}$$

where $0 \leq i \leq 2^x-1$, $0 \leq j \leq 2^y-1$, and $\hat{}$ is the *binary string concatenation operation.*

An example of embedding a 2×4 mesh in a 3-cube by using Equation 6 is shown in Figure 4(e). In Figure 4(e), the addresses of *processor*(0, 2) and *processor*(1, 0) are $N_1(0) \hat{} N_2(2) = 011$ and $N_1(1) \hat{} N_2(0) = 100$, respectively. By using the BRGCs, the addresses of any two adjacent processors and any two neighbor processors of a mesh differ by one and two bits, respectively, when the mesh is embedded in a hypercube.

Let L_a^b represent the number of nodes whose labels are equal to b in the ath stripes partition, where $a = 1$ or 2. Let L_1 and L_2 represent the largest label numbers of the first and the second stripes partition, respectively. Assume that a $2^x \times 2^y$ mesh is embedded in a $(x+y)$-cube by using Equation 6. If $2^x-1 < L_1$ ($2^y-1 < L_2$), we will merge the two adjacent stripes m and $m+1$ (n and $n+1$) which minimize $L_1^m + L_1^{m+1}$ ($L_2^n + L_2^{n+1}$), for all $m = 0$, ..., $L_1 - 1$ (for all $n = 0, ..., L_2 - 1$). This merge processing continues till $L_1 = 2^x-1$ ($L_2 = 2^y-1$). The computational complexity of this merge process is equal to $O(N^2)$. After this merge processing, every node in a FEG is assigned a new 2-tuple labels (l_1', l_2'), where $0 \le l_1' < 2^x-1$ and $0 \le l_2' < 2^y-1$. Then, we assign nodes with new labels to processors of a $(x+y)$-cube according to the following equation:

$$allc : node(i) \rightarrow N_x(l_1') \hat{} N_y(l_2'), \tag{7}$$

where the 2-tuple labels (l_1', l_2') are the new labels assigned to *node*(i), $0 \le l_1' < 2^x-1$ and $0 \le l_2' < 2^y-1$. An example of partitioning a FEG into stripes and assigning stripes to a 3-cube is shown in Figure 4.

The algorithm of the 2-way stripes partition and allocation is given as follows.

---

algorithm 2_way_stripes_partition_allocation(row, col)
/* *row* and *col* denote the length and width of a mesh, respectively. */
1. Calculate the adjacent and neighbor nodes of each node in a FEG.
2. The first stripes partition.
3. The second stripes partition

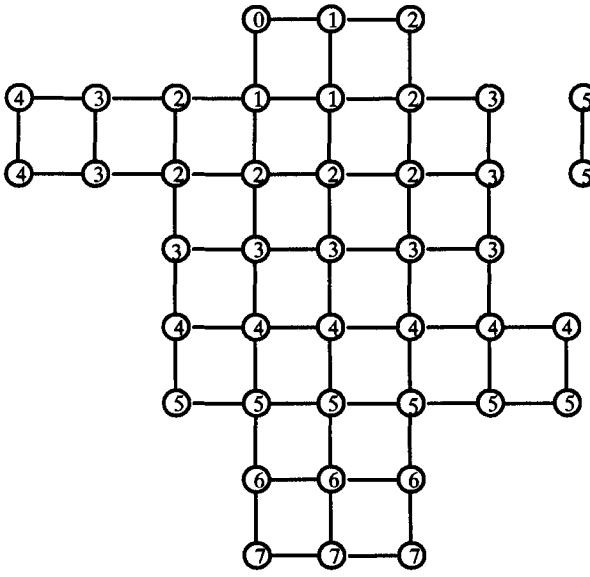


Figure 4(a) : The labels assigned to nodes by the first stripes partition.

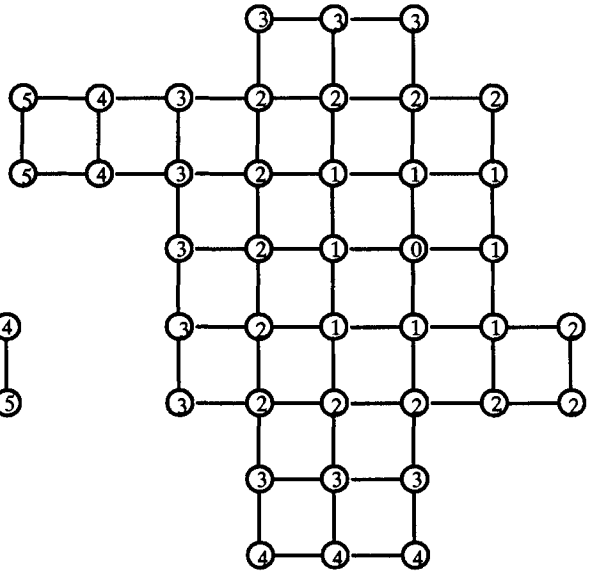


Figure 4(b) : The labels assigned to nodes by the second stripes partition.

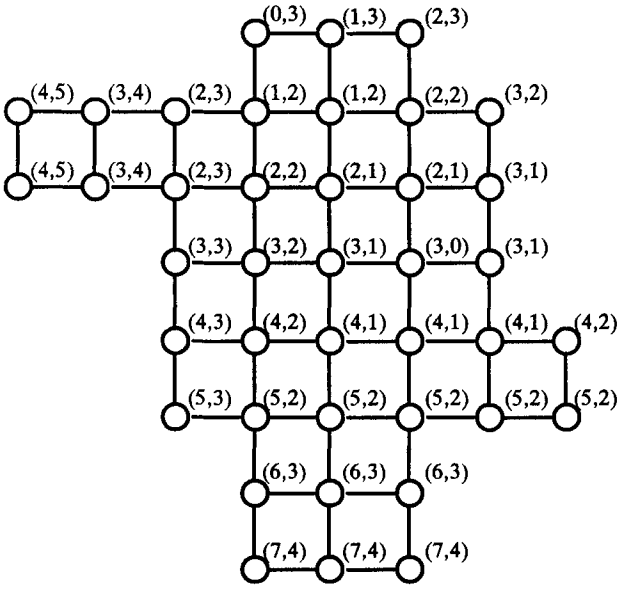


Figure 4(c) : The 2-tuple labels assigned to nodes.

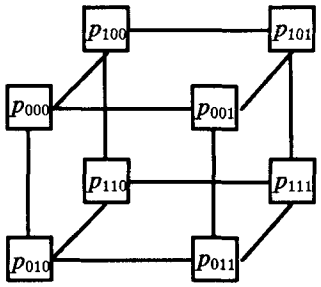


Figure 4(d) : A 3-cube.

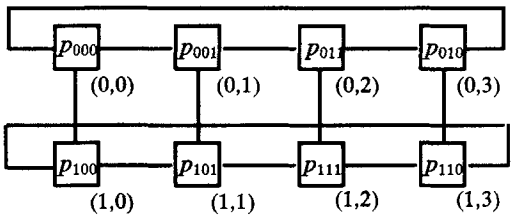


Figure 4(e) : Emulate a 2×4 mesh.

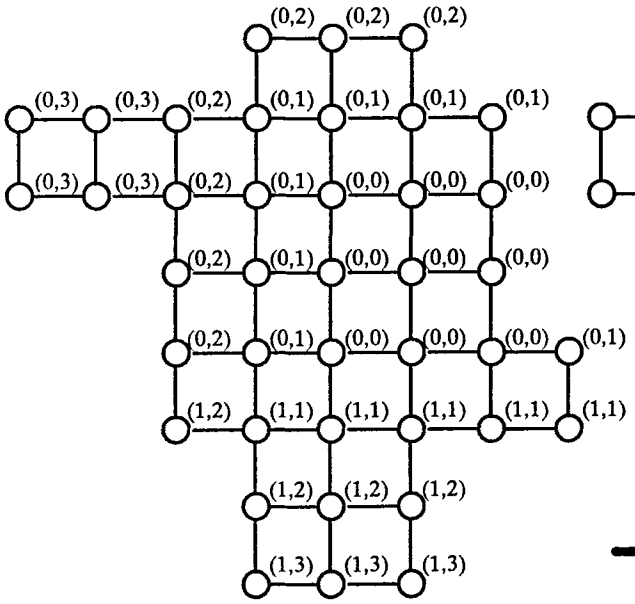


Figure 4(f) : The new labels of nodes after merging stripes.

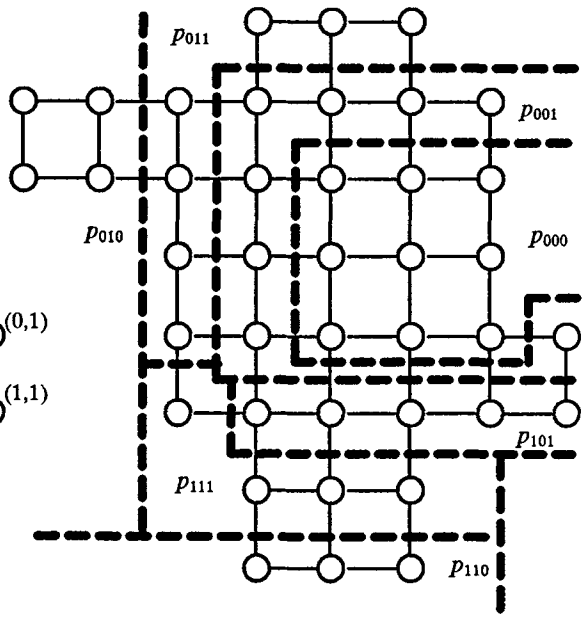


Figure 4(g) : Allocate nodes to processors by using Equation 7.

4. Merge stripes produced by the first and second stripes partition if necessary.
5. Assign nodes to processors according their new labels by using Equation 7.

end_of_2_way_stripes_partition_allocation

In algorithm *2_way_stripes_partition_allocation*, line 1 requires O(the number of FEs of a FEG) = $O(N)$ time; both lines 2 and 3 require $O(\sum_{i=1}^{N} \#(NB(node(i)))) = O(N)$ time; line 4 requires $O(N^2)$ time; and line 5 requires $O(N)$ time. The computational complexity of this algorithm is equal to $O(N + N + N + N^2 + N) = O(N^2)$.

3.2. Phase II : The Load Balance Phase

The objective of this phase is to balance the computational load assigned to processors in the first phase while preserving the neighbor mapping property. It consists of two steps. In the first step, an $M \times M$ *load transfer matrix* A is computed. Element a_{ij} in A denotes the number of nodes p_i needs to transfer to p_j. If a_{ij} is negative, $|a_{ij}|$ denotes the number of nodes p_i needs to receive from p_j. Since a $2^x \times 2^y$ mesh is embedded in a $(x+y)$-cube, we can start with computing the balanced load for processor $N_x(0) \wedge N_y(0)$, i.e., the number of nodes $N_x(0) \wedge N_y(0)$ needs to transfer to or receive from its neighbor processors. Next, we compute the balanced load for processor $N_x(0) \wedge N_y(1)$. This process continues till the balanced load for processor $N_x(2^x-1) \wedge N_y(2^y-1)$ have been computed.

Let $load(N_x(i) \wedge N_y(j))$ denote the number of nodes assigned to $N_x(i) \wedge N_y(j)$, $right(N_x(i) \wedge N_y(j))$ denote the right adjacent processor of $N_x(i) \wedge N_y(j)$, i.e., $N_x(i) \wedge N_y(j+1)$, and $down(N_x(i) \wedge N_y(j))$ denote the down adjacent processor of $N_x(i) \wedge N_y(j)$, i.e., $N_x(i+1) \wedge N_y(j)$. Note that processors $N_x(i) \wedge N_y(2^y-1)$ and $N_x(2^x-1) \wedge N_y(j)$ do not have right and down adjacent processors, respectively. To balance the computational load among processors, every processor should be assigned $\frac{N}{M}$ nodes. For simplicity, we assume N is divisible by M. The number of nodes needed to be transferred to or received from the right or down adjacent processor of $N_x(i) \wedge N_y(j)$ is determined by the following rules. This scheme is similar to that of [SaEr87].

Rule 1: $load(N_x(i) \wedge N_y(j)) > \frac{N}{M}$. If $load(down(N_x(i) \wedge N_y(j))) < load(right(N_x(i) \wedge N_y(j)))$, then $N_x(i) \wedge N_y(j)$ needs to transfer one node to $down(N_x(i) \wedge N_y(j))$; otherwise, $N_x(i) \wedge N_y(j)$ needs to transfer one node to $right(N_x(i) \wedge N_y(j))$. We update the load of processors and continue to apply Rule 1 till $load(N_x(i) \wedge N_y(j)) = \frac{N}{M}$. For those processors do not have right or down processors, the load of their right or down processors are equal to ∞.

Rule 2: $load(N_x(i) \wedge N_y(j)) < \frac{N}{M}$. If $load(down(N_x(i) \wedge N_y(j))) > load(right(N_x(i) \wedge N_y(j)))$, then $N_x(i) \wedge N_y(j)$ needs to receive one node from $down(N_x(i) \wedge N_y(j))$; otherwise, $N_x(i) \wedge N_y(j)$ needs to receive one node from $right(N_x(i) \wedge N_y(j))$. We update the load of processors and continue to apply Rule 2 till $load(N_x(i) \wedge N_y(j)) = \frac{N}{M}$. For those processors do not have right or down processors, the load of their right or down processors are equal to $-\infty$.

Rule 3: If $load(N_x(i) \wedge N_y(j)) = \frac{N}{M}$, then the load of this processor is balanced.

The time required to compute the load transfer matrix is equal to $O(MN)$.

In the second step, we perform the load transfer from one processor to another according to the load transfer matrix A. The algorithm proceeds iteratively, in an incremental manner, and is similar to that of [SaEr87].

---

algorithm load_transfer(A)

/* $ND(p_i)$ denote the set of nodes assigned to processor p_i and A is the load transfer matrix. */

1. Q = The set of processors that need to transfer nodes to other processors and mark them as *active*;
2. Make a heap $H(P)$ for all the processors in a hypercube according to their load;
3. **repeat**
4. { **repeat** /* Consider transferring $node(x) \in ND(p_i)$ to p_j such that $node(x) \in NB(node(y))$ and $node(y) \in ND(p_j)$).*/
5. { p_i = the *active* processor with the largest computational load in Q;
6. $max\_load = load(root(H(P)))$; /* The maximal load assigned to processors */
7. **if** ($\exists\, j$, $node(x)$ such that $a_{ij} > 0$, $load(p_j) < max\_load$, $node(x) \in ND(p_i)$, $ND(p_j) \cap NB(node(x))$ is not empty, and transfer of $node(x)$ from p_i to p_j preserves the neighbor mapping) **then**
8. { Assign $node(x)$ to p_j; $load(p_i) = load(p_i) - 1$; $load(p_j) = load(p_j) + 1$; $a_{ij} = a_{ij} - 1$; Update $H(P)$
9. **if** ($\forall\, k = 0, ..., M-1, a_{ik} = 0$) **then** $Q = Q - \{p_i\}$;
10. **if** ($\forall\, p_l \in Q$, p_l is *inactive*, and $NB(node(x)) \cap ND(p_l) \neq \emptyset$) **then** mark p_l as *active*;}
11. **else** mark p_i as *inactive*;
12. } **until** (all processors in Q are *inactive*);
13. Mark all the processor in Q as *active*;

14. **repeat** /* Consider transferring any node in $ND(p_i)$. */
15. { p_i = the *active* processor with the largest computational load in Q;
16. $max\_load = load(root(H(P)))$;
17. **if** ($\exists\, j$, $node(x)$ such that $a_{ij} > 0$, $load(p_j) < max\_load$, $node(x) \in ND(p_i)$,
and transfer of $node(x)$ from p_i to p_j preserves the neighbor mapping) **then**
18. { Assign $node(x)$ to p_j; $load(p_i) = load(p_i) - 1$; $load(p_j) = load(p_j) + 1$;
$a_{ij} = a_{ij} - 1$; Update $H(P)$;
19. **if** ($\forall\, k = 0, ..., M\text{-}1, a_{ik} = 0$) **then** $Q = Q - \{p_i\}$;
20. **if** ($\forall\, p_l \in Q$, p_l is *inactive*, and $NB(node(x)) \cap ND(p_l) \neq \emptyset$) **then** mark p_l as *active*;}
21. **else** mark p_i as *inactive*;
22. } **until** (all processors in Q are *inactive*);
23. } **until** (load is balanced or further balancing is impossible);
end_of_load_transfer

In algorithm *load_transfer*, lines 1 and 2 require $O(M)$ time; lines 5 and 15 require $O(M)$ time; lines 6 and 16 require $O(c_1)$ time; line 7 requires $O(2 \times \#(ND(p_i)) \times (\#(NB(node(x))) + \#(NB(node(x))))) = O(N)$; lines 8 and 18 require $O(\log M)$ time; lines 9, 13, and 19 require $O(M)$ time; lines 10 and 20 require $O(M \times \#(NB(node(x))) = O(M)$ time; lines 11 and 21 require $O(c_2)$; lines 12 and 22 require $O(c_3)$ time; line 17 requires $O(2 \times \#(ND(p_i)) \times \#(NB(node(x)))) = O(N)$ time; line 23 requires $O(c_4)$ time, where c_1, c_2, c_3, and c_4 are constants. Lines 3 to 23, lines 4 to 12, and lines 14 to 22 form loops and these loops have $O(c)$, $O(MN)$, and $O(MN)$ iterations, respectively, where c is a constant. The computational complexity of this algorithm under this assumption is equal to $O(M + M + c \times (MN \times (M + c_1 + N + \log M + M + M + c_2 + c_3) + M + MN \times (M + c_1 + N + \log M + M + M + c_2 + c_3) + c_4) = O(M^2N + MN^2)$. We assume that N is usually greater than M. The worst case of the computational complexity of this algorithm is $O(M^2N + MN^2) \simeq O(MN^2)$.

Algorithm *load_transfer* does not guarantee to balance the computational load of processors. If the computational load of processors can be balanced by this algorithm, the values of all the elements in A are equal to zeros.

The 2-way stripes partition mapping algorithm is given as follows.

---

algorithm 2_way_stripes_partition_mapping(M, N, X)

/* X is the intermediate processor matrix. $\forall x_{ij} \in X$, if $p_i = a_{n-1}\ldots a_{k+1}a_k a_{k-1}\ldots a_0$ and $p_j = b_{n-1}\ldots b_{k+1}\overline{a}_k a_{k-1}\ldots a_0$, then $x_{ij} = p_l = a_{n-1}\ldots a_{k+1}\overline{a}_k a_{k-1}\ldots a_0$ */

1. *row* = 1; *col* = *M*; *best_bi* = 0; *best_uni* = 0;
2. **repeat**
3. { *2_way_stripes_partition_allocation(row, col)*;
4. Compute the load transfer matrix A;
5. *load_transfer(A)*;
6. **if** (*best_bi* < *bidirectional_comm_cost(X)*) **then** *best_bi* = *bidirectional_comm_cost(X)*;
7. **if** (*best_uni* < *unidirectional_comm_cost(X)*) **then** *best_uni* = *unidirectional_comm_cost(X)*;
8. *row* = *row* * 2; *col* = *col* / 2;
9. } **until** (*row* > *M*);

end_of_2_way_stripes_partition_mapping

In algorithm *2_way_stripes_partition_mapping*, line 1 requires $O(c_1)$ time; line 3 requires $O(N^2)$; line 4 requires $O(MN)$ time; line 5 requires $O(MN^2)$ time; lines 6 and 7 require $O(N + M\log^2 M)$ time; line 8 requires $O(c_2)$ time; and line 9 requires $O(c_3)$ time, where c_1, c_2, and c_3 are constants. Lines 2 to 9 form a loop and this loop has $\log M$ iterations. The computational complexity of this algorithm is equal to $O(c_1 + \log M \times (N^2 + MN + MN^2 + (N + M\log^2 M) + (N + M\log^2 M) + c_2 + c_3) = O(MN^2\log M)$.

Lemma 3 : The 2-way stripe partition mapping is a neighbor mapping. ■

4. THE GREEDY ASSIGNMENT MAPPING

The *greedy assignment mapping* is a heuristic approach. It assigns a node to a particular processor according to the current status of its neighbor nodes. Initially, it assigns *node*(a), which has the largest number of adjacent nodes in a FEG, to processor 0 and the adjacent nodes of *node*(a) are put into a queue Q. The node *node*(i) in Q which has the largest number of adjacent nodes is selected as the next node to be assigned. Let $P(NB(node(i)))$ denote the set of processors which the neighbor nodes of *node*(i) are assigned and $P(POS(node(i)))$ denote the set of processors whose addresses differ from the address of each processor in $P(NB(node(i)))$ by at most two bits. If $P(POS(node(i)))$ is empty, it implies that the neighbor mapping is impossible for this approach; otherwise, for all p_x, $p_y \in P(POS(node(i)))$ and $load(p_x) \leq load(p_y)$, it assigns *node*(i) to p_x. Then, the adjacent nodes of *node*(i) are inserted in Q. This process continues till all the nodes are assigned or the neighbor mapping is impossible. The algorithm is given as follows.

---

algorithm greedy_assignment_mapping(X)

/* X is the intermediate processor matrix. $\forall x_{ij} \in X$, if $p_i = a_{n-1}\ldots a_{k+1}a_k a_{k-1}\ldots a_0$ and $p_j = b_{n-1}\ldots b_{k+1}\overline{a}_k a_{k-1}\ldots a_0$, then $x_{ij} = p_l = a_{n-1}\ldots a_{k+1}\overline{a}_k a_{k-1}\ldots a_0$ */

1. Calculate the adjacent and neighbor nodes of each node in a FEG;
2. $Q = \emptyset$;
3. *node*(a) = The node with the largest number of adjacent nodes in a FEG;
4. Assign *node*(a) to processor 0 and $Q = Q \cup ADJ(node(a))$;
5. Make a heap $H(Q)$ for the nodes in Q according to the number of their adjacent nodes ;
6. **while** (Q is not empty) **do**
7. { *node*(i) = *root*($H(Q)$); /* the node with the largest number of adjacent nodes in Q */
8. Compute $P(POS(\text{node}(i)))$.
9. **if** ($P(POS(\text{node}(i)))$ is empty) **then** stop ("The neighbor mapping is impossible");
10. p_x = the processor with the smallest load in $P(POS(\text{node}(i)))$;
11. $load(p_x) = load(p_x) + 1$; $Q = Q - \{node(i)\}$;
    $Q = Q \cup$ {those nodes in $ADJ(node(i))$ which have not been assigned} ; Update $H(Q)$;
12. }
13. *best_bi* = *bidirectional_comm_cost*(X);
14. *best_uni* = *unidirectional_comm_cost*(X);

end_of_greedy_assignment_mapping

In algorithm *greedy_assignment_mapping*, line 1 requires O(the number of FEs of FEG) $= O(N)$ time; line 2 requires $O(c_1)$ time; line 3 requires $O(N)$ time; line 4 requires $O(c_2)$ time; line 5 requires $O(c_3)$ time; line 7 requires $O(c_4)$ time; line 8 requires $O(\#(NB(node(i))) \times \log^2 M) = O(\log^2 M)$ time; line 9 requires $O(c_5)$ time; line 10 requires $O(\log^2 M)$ time; line 11 requires $O(\log N)$ time; and lines 13 and 14 require $O(N + M\log^2 M)$, where c_1, c_2, c_3, c_4, and c_5 are constants. Lines 6 to 12 form a loop. This loop has N iterations. The computational complexity of this algorithm is equal to $O(N + c_1 + N + c_2 + c_3 + N \times (c_4 + \log^2 M + c_5 + \log^2 M + \log N) + (N + M\log^2 M) + (N + M\log^2 M)) = O(N\log^2 M + N\log N)$. An example of mapping a FEG onto a hypercube by using algorithm *greedy_assignment_mapping* is shown in Figure 5.

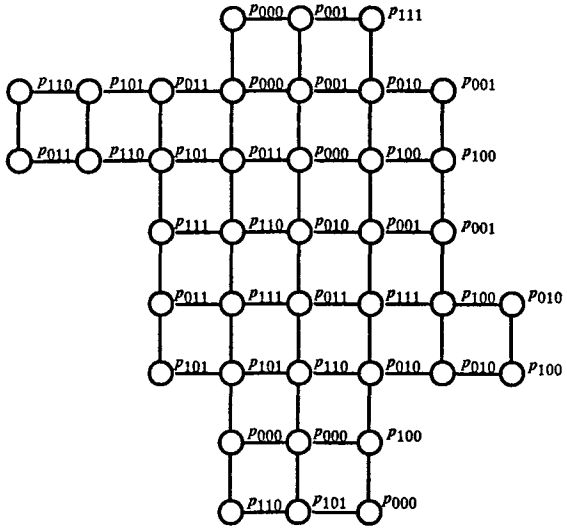


Figure 5 : Mapping a FEG onto a hypercube by using the greedy assignment mapping.

5. PERFORMANCE EVALUATION AND SIMULATION RESULTS

The samples of FEGs tested in this paper consist of four 2–D graphs and three 3–D graphs which are shown in Figures 7(a)–(d) and 7(e)–(g), respectively. The number of nodes of these FEGs are ranging from a few tens to a few hundreds. According to the communication models described in Section 2.3, we derive the *estimated lower bound speedup* (ELBS) and the *estimated upper bound speedup* (EUBS) for both of the bidirectional and unidirectional communication models to measure our mapping results. They are given as follows:

$$\begin{cases} EUBS_{bi} = \dfrac{N \times T_{task}}{\lceil \frac{N}{M} \rceil \times T_{task} + T_{setup} + 2 \times T_c} & (8.1) \\ ELBS_{bi} = \dfrac{N \times T_{task}}{\lceil \frac{N}{M} \rceil \times T_{task} + 2 \times T_{setup} + (2 \times \log M - 1) \times \lceil \frac{N}{M} \rceil \times T_c} & (8.2) \end{cases}$$

$$\begin{cases} EUBS_{uni} = \dfrac{N \times T_{task}}{\lceil \frac{N}{M} \rceil \times T_{task} + 2 \times (T_{setup} + 2 \times T_c)} & (9.1) \\ ELBS_{uni} = \dfrac{N \times T_{task}}{\lceil \frac{N}{M} \rceil \times T_{task} + 4 \times T_{setup} + (4 \times \log M - 2) \times \lceil \frac{N}{M} \rceil \times T_c} & (9.2) \end{cases}$$

where T_{task}, T_{setup}, and T_c denote the time required by a processor to execute the computation of a node, the setup time of the I/O channel, and the data transmission time of the I/O channel per word, respectively; $EUBS_{bi}$ and $ELBS_{bi}$ denote the EUBS and ELBS of the bidirectional communication model, respectively; $EUBS_{uni}$ and $ELBS_{uni}$ denote the EUBS and ELBS of the unidirectional communication model, respectively.

The EUBS and ELBS are obtained by assuming that both the LBC and the neighbor mapping are achieved. If the LBC is achieved by a mapping, the item $\max\{load_i(p_j)\}$ in Equation 2 is equal to $\lceil \frac{N}{M} \rceil$. If a mapping is a neighbor mapping, the best case of the communication cost is that any two neighbor nodes of a FEG are assigned to the same processor or two adjacent processors of a hypercube and every processor only need to send two nodes' data to each of its adjacent processors (see Figure 6). According to the communication models described in Section 2.3, we can derive Equations 8.1 and 9.1.

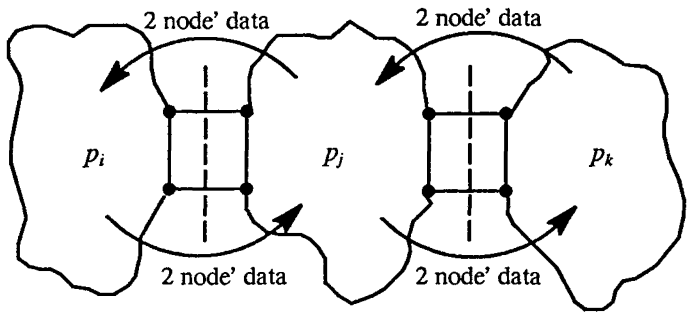


Figure 6 : The best case of the communication cost of a mapping.

If a mapping is a neighbor mapping, the worst case of the communication cost is that any two neighbor nodes of a FE are assigned to two processors whose addresses differ by two bits in a hypercube. For the bidirectional communication model, the maximal number of steps to finish the data communication among processors is equal to 2. In step 1, a processor receives data from its *adjacent* processors and sends data to its *neighbor* processors simultaneously. The maximal amount of data sent by processors is equal to $\log M \times \lceil \frac{N}{M} \rceil$. In step 2, the maximal amount of data sent from a processor to its *adjacent* processors is equal to $(\log M - 1) \times \lceil \frac{N}{M} \rceil$ (see Figure 3(a)). Therefore, we can derive Equation 8.2. For the unidirectional communication model, the maximal number of steps to finish the data communication among processors is equal to 4. A processor may receive (send) data from (to) its *adjacent* (*neighbor*) processors in step 1 and then send (receive) data to (from) its *neighbor* (*adjacent*) processors in step 2. The maximal amount of data sent by processors in steps 1 and 2 are both equal to $\log M \times \lceil \frac{N}{M} \rceil$. A processor may receive (send) data from (to) its *adjacent* (*adjacent*) processors in step 3 and then send (receive) data to (from) its *adjacent* (*adjacent*) processors in step 4. The maximal amount of data sent by processors in steps 3 and 4 are both equal to $\log M \times \lceil \frac{N}{M} \rceil - 1$. (see Figure 3(b)). Therefore, we can derive Equation 9.2.

We make the following assumptions about the capabilities of the processors of a hypercube [SaEr87]. T_{task} is equal to 1190 μs. T_{setup} is equal to 1150 μs. T_c is equal to 10 μs per word. By using the 2-way stripes partition mapping and the greedy assignment mapping, the speedups of all the test samples on n-cubes are shown in Tables 1, 2, and 3, for $n = 3$, 4, and 5, respectively. The following conclusions can be drawn from Tables 1, 2, and 3.

1) : Both mappings give excellent performance. The estimated speedups of these mappings are near optimal (given by EUBS) for most cases.

2) : The greedy assignment mapping, in general, can produce a good mapping at a low computation cost. This method is not restricted to hypercubes and can be applied to a wide variety of parallel architectures. It fails to preserve the neighbor mapping for sample 5 onto 4- and 5-cube. Every node in sample 5 has the same number of adjacent nodes. It is difficult for this algorithm to determine which node is the best node to be assigned next because the degree of all nodes are the same.

3) : For the cases where the LBC is achieved, the speedups for the 2-way stripes partition mapping are better than those for the greedy assignment mapping. The reason is that, by using algorithm *2_way_stripes_partition_mapping*, most of the nodes assigned to the same processor are connected. It produces a smaller communication cost than that of the greedy assignment mapping.

4) : In Table 3, although the LBC is not achieved in sample 6 by using the 2-way stripes partition mapping, the speedup for the 2-way stripes partition mapping is greater than the value of the ELBS when the unidirectional communication model is used. The is because that the number of steps, denoted by S, to finish the data communication among processors in the unidirectional communication model is greater than 1 and less than 5, i.e. $2 \leq S \leq 4$. If the maximal computational load assigned to processors by the 2-way stripes partition mapping is equal to $\lceil \frac{N}{M} \rceil + 1$ and S is equal to 3, according to Equation 9.2, it is possible that the speedup for the 2-way stripes partition mapping is greater than the value of the ELBS.

6. CONCLUSIONS

We proposed two mapping approaches, the 2-way stripes partition mapping and the greedy assignment mapping, to map FEGs onto hypercubes. The 2-way stripes partition mapping uses the stripes partition and BRGCs allocation to achieve the MCCC and uses the load transfer heuristic to achieve the LBC. The greedy assignment mapping uses greedy heuristic to achieve both MCCC and LBC. The cost models of mapping a FEG onto a hypercube are developed for the bidirectional communication model and the unidirectional communication model. Four 2-D and three 3-D FEGs are used as the test samples. To measure the mapping results, the EUBS and ELBS are derived for both of the communication models. The simulation results show that the speedups for the 2-way stripes partition mapping are better than those for the greedy assignment mapping when the LBC is achieved in both approaches. However, the greedy approach gives good performance at a much lower cost.

Sample	Unidirectional communication model				Bidirectional communication model			
	EUBS	ELBS	Greedy	2-way stripes partition	EUBS	ELBS	Greedy	2-way stripes partition
1 (64)	6.42	5.10	5.19*	6.36*	7.12	6.13	6.32*	7.08*
2 (40)	5.74	4.31	3.92	4.44*	6.69	5.47	4.94	5.71*
3 (160)	7.28	6.26	6.09	7.20*	7.63	6.97	6.78	7.57*
4 (505)	7.66	6.89	7.00*	7.61*	7.77	7.34	7.44*	7.75*
5 (160)	7.28	6.26	6.07	6.57*	7.63	6.97	6.77	7.20*
6 (340)	7.56	6.74	6.80*	7.18*	7.73	7.25	7.33*	7.52*
7 (198)	7.34	6.39	6.48*	7.23*	7.62	6.30	7.12*	7.29*

Table 1 : The speedups of mapping FEGs onto 3-cubes.

Sample	Unidirectional communication model				Bidirectional communication model			
	EUBS	ELBS	Greedy	2-way stripes partition	EUBS	ELBS	Greedy	2-way stripes partition
1 (64)	10.73	7.68	6.97	7.93*	12.84	10.10	9.00	10.61*
2 (40)	8.05	5.54	5.68*	5.70*	10.04	7.58	7.97*	7.96*
3 (160)	13.37	10.64	10.14	11.23*	14.57	12.61	12.01	13.17*
4 (505)	14.87	12.74	13.31*	14.74*	15.31	14.03	14.43*	15.24*
5 (160)	13.37	10.64	-	11.13*	14.57	12.61	-	13.06*
6 (340)	14.19	11.95	12.46*	12.79*	14.79	13.39	13.77*	13.99*
7 (198)	13.23	10.76	11.19*	11.47*	14.16	12.48	12.90*	13.06*

Table 2 : The speedups of mapping FEGs onto 4-cubes.

Sample	Unidirectional communication model				Bidirectional communication model			
	EUBS	ELBS	Greedy	2-way stripes partition	EUBS	ELBS	Greedy	2-way stripes partition
1 (64)	16.14	10.38	9.02	9.06	21.45	15.05	12.65	12.67
2 (40)	10.08	6.49	6.65*	6.66*	13.41	9.41	9.94*	9.96*
3 (160)	22.97	16.63	14.21	14.21	26.74	21.39	17.44	17.53
4 (505)	28.11	22.66	24.15*	24.75*	29.74	26.15	27.30*	27.64*
5 (160)	22.97	16.63	-	11.97	26.74	21.39	-	14.31
6 (340)	26.22	20.57	21.61*	20.65	28.37	24.40	25.43*	23.81
7 (198)	22.08	16.60	17.41*	17.57*	24.80	20.56	21.46*	21.60*

Table 3 : The speedups of mapping FEGs onto 5-cubes.

* : The LBC is achieved in this case.

- : The neighbor mapping cannot be achieved in this case.

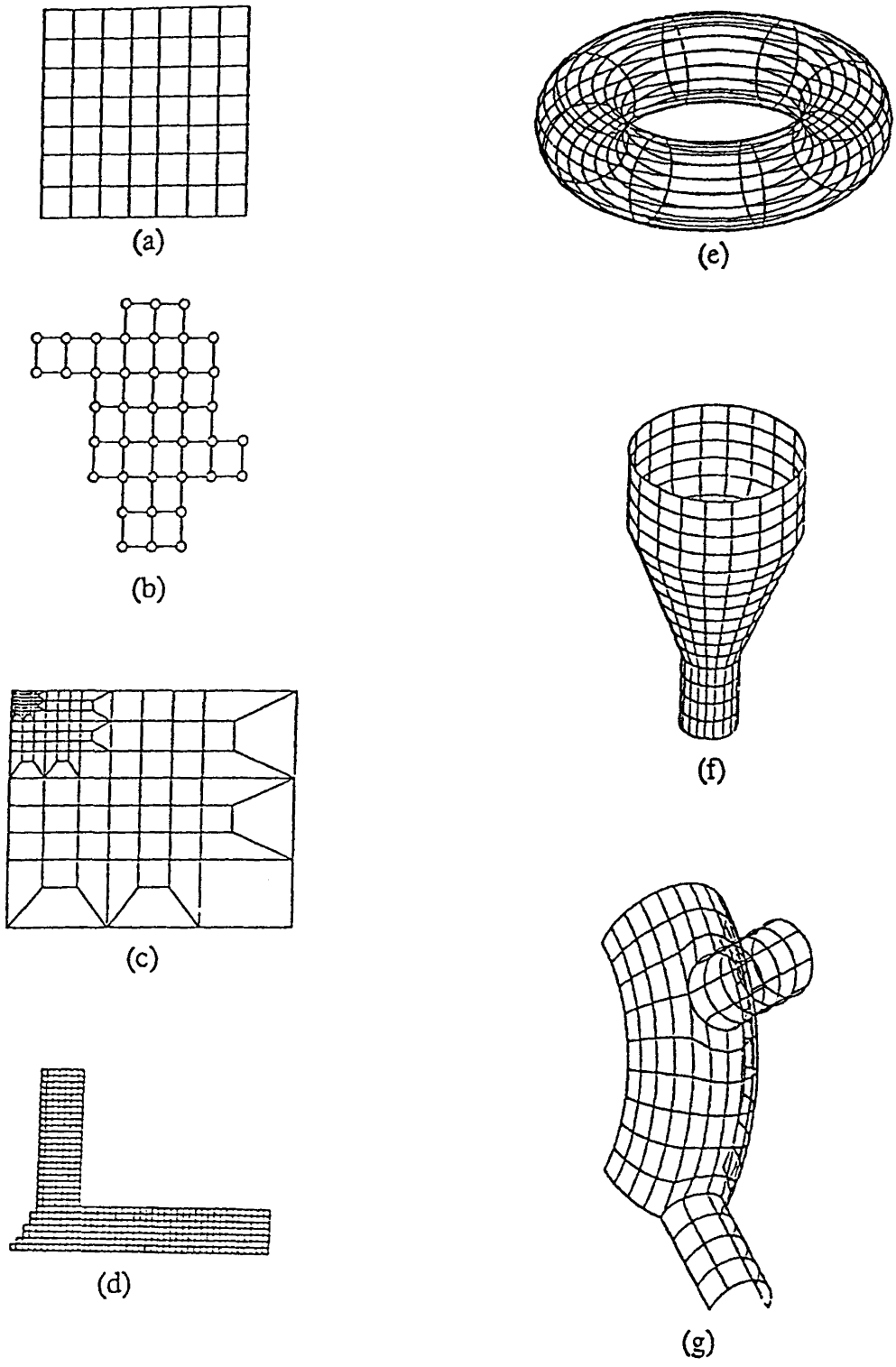


Figure 7 : The test samples.

References :

[BeBo87] M.J. Berger and S.H. Bokhari, "A Partitioning Strategy for Nonuniform Problems on Multiprocessors," *IEEE Trans. on Computers*, Vol. C-36, No. 5, pp. 570-580, 1987.

[BhAg84] L.N. Bhuyan and D.P. Agrawal, "Generalized Hypercube and Hyperchannel structures for a Computer Network," *IEEE Trans. on Computers*, Vol. C-33, pp. 323-333, 1984.

[Bokh81] S.H. Bokhari, "On the mapping problem," *IEEE Trans. on Computers*, Vol. C-30, pp. 207-214, 1981.

[ChSa86] T.F. Chan and Y. Saad, "Multigrid Algorithms on the Hypercube Multiprocessors," *IEEE Trans. on Computers*, Vol. C-35, pp. 969-977, 1986.

[GaJo79] M.R. Garey and D.S. Johnson, *Computers and Intractability, A Guide to Theory of NP-completeness.* San Francisco, CA: Freeman, 1979.

[HaMu86] J. Hayes and T. Mudge, "Architecture of a Hypercube Supercomputer," *Proc. of Int'l Conference on Parallel Processing*, pp. 653-660, 1986.

[Jord78] H. Jordan, "A special purpose architecture for finite element analysis," *Int'l Conference of Parallel Processing*, pp. 263-266, 1978.

[LaPi83] L. Lapidus and G.F. Pinder, *Numerical Solution of Partial Differential Equations in Science and Engineering.* New York: Wiley, 1983.

[Peas77] M.C. Pease, "The Indirect Binary n-cube Multiprocessor Array," *IEEE Trans. on Computers*, Vol. 26, pp. 458-473, 1977.

[SaEr87] P. Sadayappan and F. Ercal, "Nearest-Neighbor Mapping of Finite Element Graphs onto Processor meshes," *IEEE Trans. on Computers*, Vol. C-36 No. 12, pp. 1408-1424, 1987.

[Seit85] C.L. Seitz, "The Cosmic Cube," *Communications of ACM*, Vol. 28, pp. 22-33, 1985.