COMMUNICATION–ORIENTED ASSIGNMENT OF TASK MODULES IN HYPERCUBE MULTICOMPUTERS

Bing–rung Tsai and Kang G. Shin

Real–Time Computing Laboratory
Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109–2122

ABSTRACT

We formulate and solve the problem of assigning a task consisting of multiple communicating modules to a hypercube multicomputer such that the total communication traffic during the execution of the task is minimized. Exclusion of the timing details from the objective function used allows us to formulate the task assignment problem as a combinatorial optimization problem. Since finding an optimal solution for this problem is NP–hard, a standard state–space search algorithm as well as other heuristic algorithms are used to find optimal/suboptimal solutions. The relative performances of various algorithms are evaluated using simulations. The assignments obtained from these algorithms are also evaluated using an event–driven simulator to learn how they perform under real–world execution environments.

1 Introduction

Hypercube multicomputers have been drawing considerable attention due to their regularity for ease of construction and their potential for the high degree of parallelism and fault–tolerance. A hypercube multicomputer system often consists of a host and a number of nodes connected in binary cube configuration. When a user writes a parallel program for a hypercube, separate program modules must be written for the host and each of the nodes. In most situations, each processor node is responsible for executing at most a single task module and the mapping between modules and nodes is one–to–one. We will henceforth use both the terms "assign" and "map" interchangeably in this paper.

As we will show in Section 2, different assignments of task modules to processor nodes do not affect the cost of computation, but they affect the cost associated with interprocessor communications. The *communication cost* of executing a set of task modules is defined to be the total link resources (measured in number of time units) used during the execution of these modules.

We propose and solve the problem of mapping a set of interacting task modules into a hypercube so as to minimize an objective function, called *communication traffic*, which is closely related to the communication cost. Since the actual timing details are not included explicitly in the formulation, the problem can be formulated as a combinatorial optimization problem for which several algorithms are then developed. It is shown via simulations that the assignments found by optimizing this objective perform significantly better than random assignments in most real–world situations where the actual timing is figured in.

In [1], it is pointed out that there are two types of cost in assigning task modules to processors: the cost of executing a module on a processor and the cost of interprocessor communication. Note that the cost of executing a module on a processor is invariant among different assignments, since at most one module is assigned to each processor node. Thus, only the cost of interprocessor communication needs to be considered for task assignment. This point differentiates the work presented in this paper from others related to task assignment, such as [2, 3, 4, 5, 6, 7, 8].

The paper is organized as follows. Section 2 deals with the system model and the assumptions used. Our problem is also formally stated there. In Section 3, the NP–hardness of the problem is proved first. We then propose a heuristic function that can be used for standard state–space search algorithms. The performance of the proposed heuristic function is evaluated using simulations. Other fast heuristic algorithms are also considered as means to find good suboptimal (instead of optimal) solutions for large hypercubes. Event–driven simulations are carried out and described in Section 4 to determine how well these assignments perform in real–world execution situations. The paper concludes with Section 5.

2 Preliminaries

As in virtually all existing hypercube multicomputers, inter-module communications are accomplished via mes-

The work reported in this paper was supported in part by the Office of Naval Research under Grants No. N00014–85–K–0122 and N00014–92–J–1080.

0-8186-2865-0 /92 $3.00 © 1992 IEEE

sage 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. In case the communication volume cannot be determined precisely, one can use the maximum number of packets that may possibly be communicated between each pair of interacting modules.

The hypercube multicomputer under consideration is assumed to possess the e–cube routing scheme [9] 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 between the source to the destination. It is also assumed that each packet is routed through the chosen path to its destination by message switching or circuit switching.

A task is composed of a set of modules, each of which is to be executed on a node, and at most one module is assigned to a node. This is not unrealistic since most parallel programs written for hypercubes follow this rule, and most existing hypercube systems do not support per–node multi-programming environments. Besides, the number of processors can be used by a task is not a limiting factor since the number of processors is usually quite large in a distributed–memory system like hypercubes. Therefore, the number of nodes in a subcube allocated for executing a task must be greater than, or equal to, the number of modules of the task.

When embedding a group of task modules into an n–dimensional hypercube or n–cube, 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 some "dummy" modules and make the task consist of 2^n modules. The scheme for allocating subcubes of necessary size has been discussed in detail in [10]. So, we will henceforth assume $M = 2^n$ where n is the dimension of the subcube allocated by the host to execute the task, and thus, the mapping of modules into nodes is now one–to–one and onto.

Suppose $cost_{cpu}(m_i)$ is the computation cost to be incurred by a node to execute module m_i. Since a hypercube is homogeneous and symmetric, any assignment will have the same total computation cost, i.e., $\sum_{0 \le i \le M-1} cost_{cpu}(m_i)$. However, a different assignment 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 execution. In other words, the communication cost is a measure of the link resources used by an instance of 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

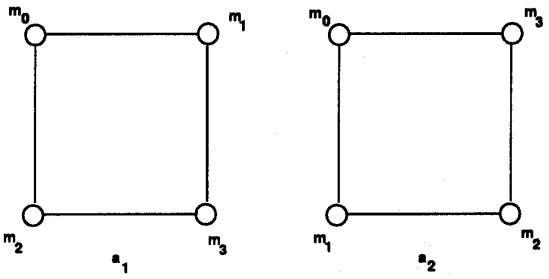

Figure 1: An example task with two assignments to a Q_2.

busy for purposes other than packet transmission — such as establishing the communication path — is assumed to be negligible.

Suppose $\mathbf{U}$ is the $M \times M$ communication volume matrix, where $U_{ij} = U_{ji}$ is the communication volume between m_i and m_j expressed in number of packets, and $M = 2^n$ is the number of modules. In Fig. 1, let

$$\mathbf{U} = \begin{bmatrix} 0 & 30 & 10 & 80 \\ 30 & 0 & 70 & 20 \\ 10 & 70 & 0 & 40 \\ 80 & 20 & 40 & 0 \end{bmatrix}.$$

Then for the assignments ($\mathbf{a}_1$ and $\mathbf{a}_2$) shown, $cost_{com}(\mathbf{a}_1)$ and $cost_{com}(\mathbf{a}_2)$ — which denote the total communication cost of $\mathbf{a}_1$ and $\mathbf{a}_2$, respectively — are:

$$cost_{com}(\mathbf{a}_1) = 150c(2) + 100c(1)$$

and

$$cost_{com}(\mathbf{a}_2) = 30c(2) + 220c(1).$$

Since the actual timing is not considered explicitly in the cost, the above equations are most accurate if the communication between each pair of module is done in "one shot," i.e., all packets are combined and sent as a single message. Thus, the above equations are the lower bounds of the actual costs.

For hypercubes with message switching, it is obvious that we have $c(x) = xc(1)$. This relation may be less accurate in case of circuit switching. However, if we assume that the "call request" signal — which is sent out to search for a free path — occupies each link for a very short time, then this expression will be a good approximation even for circuit switching [11]. As a result, in either case, $cost_{com}(\mathbf{a}_1) = 400c(1)$, which is substantially greater than $cost_{com}(\mathbf{a}_2) = 280c(1)$.

By defining a unit of *communication traffic* as $c(1)$, the amount of communication traffic generated from executing a task under assignment $\mathbf{a}$ becomes:

$$k(\mathbf{a}) = \sum_{1 \le i \le n} iv_i,$$

where v_i is the number of packets traversing i links. Thus, we have

$$cost_{com}(\mathbf{a}) \propto k(\mathbf{a}).$$

The following notation will be used throughout the paper:

- n : the dimension of a subcube available for executing the task under consideration.
- $\mathbf{a}$: a $1 \times M$ assignment vector, the i-th component of which represents the fact that m_i is assigned to a node whose address is a_i, $0 \leq i \leq M-1$.
- $H(n_i, n_j)$: the Hamming distance between node n_i and node n_j.
- $h(m_i, m_j, \mathbf{a})$: the Hamming distance between the nodes to which m_i and m_j are assigned under assignment $\mathbf{a}$.

The communication traffic under assignment $\mathbf{a}$ can be rewritten as:

$$k(\mathbf{a}) = \sum_{0 \leq i < j \leq M-1} h(m_i, m_j, \mathbf{a}) * U_{ij}$$

When introducing the notion of communication cost and communication traffic, we deliberately avoid the low–level timing details. For example, we do not account for how packets are grouped into messages or when these messages are going to be sent. We only consider the total number of packets to be sent/received between a pair of task modules during the whole mission time, thus allowing for a simple function that can be translated into a simple combinatorial optimization problem. Our simulation results (in Section 4) show that optimizing this function yields optimal assignments for communication–bound tasks with respect to an actual low–level communication performance measure.

Our task assignment approach is more suitable for on-line, automatic compile–and–run systems, especially for the code–loading phase after compilation. If the timing details of task–execution behaviors are known *a priori*, the indirect optimization can also be viewed as the first phase of *task allocation* [12] which consists of task assignment and scheduling. After applying our traffic–minimization algorithm, one can apply a message scheduling algorithm to further optimize the utilization of communication links during an instance of task execution.

3 Task Assignment Algorithms

A graph mapping problem is intrinsically hard, as stated in the following theorem.

Theorem 1: Given an $M \times M$ task communication volume matrix $\mathbf{U}$, where $M = 2^n$, n is the dimension of the hypercube, and given an integer L, the problem of deciding the existence of a mapping $\mathbf{a}$ with $k(\mathbf{a}) \leq L$ is NP–hard.

Proof: To show it is NP–hard, we restrict it to the hypercube embedding problem discussed in [13]. Construct a graph $G = (V, E)$ representing the interaction between modules where $V = \{m_i \mid 0 \leq i \leq M-1\}$ and $E = \{(m_i, m_j) \mid U_{ij} > 0\}$. If m_i communicates with m_j, then there is a link between m_i and m_j. Let $L = \sum_{0 \leq i < j \leq M-1} U_{ij}$, then there is a mapping $\mathbf{a}$ such that $k(\mathbf{a}) = L$ if and only if G can be embedded into a Q_n. □

Though the above problem is formulated as a *decision* problem, it can be shown that the corresponding optimization problem is also NP–hard [14]. Thus, there are no known polynomial–time algorithms to find an optimal mapping. This in turn calls for heuristic approaches.

The use of state–space search with heuristics for improving the search performance in NP–hard task mapping problems has been investigated by Shen and Tsai[8]. It was shown to be effective in finding optimal solutions for this type of problems. Following this approach, we will develop a heuristic function to be used for finding a task mapping with minimum communication traffic. With this heuristic function, the state–space search algorithm, or so called the A^* algorithm [15], is then implemented.

The module mapping problem is formulated as a state–space search problem as follows:

1. *State Description*: A state is described by a set of ordered pair(s), $S = \{(m_i, x) \mid x \in B(Q_n)\}$, where m_i is a module and $B(Q_n)$ is the set of binary addresses of nodes in a Q_n, and (m_i, x) means that module m_i is assigned to node x in the Q_n, or equivalently, m_i is labeled with the node address x. Each state S denotes a *partial* mapping.
2. *Initial State*: $S_0 = \{(m_0, 0^n)\}$, where m_0 is an arbitrary module. Since a Q_n is symmetric, without loss of generality, one can start with assigning any module to 0^n.
3. *Operator*: An operator adds a new ordered pair to a state S and generates at least one new state. The procedure works as follows. Let R be the set of labels in $B(Q_n)$ that have not yet been used for mapping, and m_i be the last module assigned. For each $b \in R$, construct a new ordered pair (m_{i+1}, b) and generate a new state $S \cup \{(m_{i+1}, b)\}$. The state–generating operator should not be applied to a goal state which is defined below.
4. *Goal State*: Any state which includes an ordered pair (m_{2^n-1}, x) is a goal state, i.e., the search stops when all modules are assigned.

For a standard A^* search algorithm, the heuristic function $f(S)$ of state S consists of two parts, $g(S)$ and $h(S)$, where $g(S)$ is the actual cost of the path to the current state, and $h(S)$ is an estimated cost of the path from the current state to a goal state. To guarantee the algorithm to lead to a goal state, $h(S) \leq h^*(S)$ must hold for all S

where $h^*(S)$ is the actual cost of the path from the current state S to the corresponding goal state. This is termed the *admissibility* of an search algorithm[15].

For a state S, $g(S)$ is calculated as follows:

$g(S) = 0$;
if $|S| = 1$ **then exit**
else
  for each pair of (m_i, x) and (m_j, y) in S
    $g(S) = g(S) + H(x, y) * U_{ij}$.

The function $h(S)$ is calculated as:

$h(S) = 0$;
R = the set of nodes available for assignment;
V = the set of modules not yet been mapped;
for each $(m_j, y) \in S$
  for each $x \in R$, compute $H(x, y)$ and put each value in vector h;
  for each $m_i \in V$, put U_{ij} into vector u;
$\mathbf{h} = sort^a(\mathbf{h})$;
$\mathbf{u} = sort^d(\mathbf{u})$;
$h(S) = \mathbf{h} \cdot \mathbf{u}$.

We will now prove the admissibility of the search algorithm, i.e., $h(S) \leq h^*(S)$ for all S, and thus, the algorithm is guaranteed to find an optimal solution when terminated.

Theorem 2: For any complete mapping **a** containing S as a partial mapping,

$$h^*(S) = k(\mathbf{a}) - g(S) \geq h(S).$$

Proof: We have

$$k(\mathbf{a}) - g(S) = \sum_{m_i \in W, m_j \in W'} h(m_i, m_j, \mathbf{a}) * U_{ij},$$

where W is the set of modules already assigned in S and W' is the set of modules not yet assigned. Without loss of generality, we can assume that for each $m_j \in W'$ and for all m_i's in W, the corresponding U_{ij}'s are arranged in descending order into vector $\mathbf{u}_j$. Also, the values of $h(m_i, m_j, \mathbf{a})$ are put into a vector $\mathbf{h}^j$ in the corresponding order. Then the above expression can be rewritten as $\sum_{m_j \in W'} \mathbf{h}^j \cdot \mathbf{u}_j$. But from the algorithm of calculating $h(S)$, $h(S) = \sum_{m_j \in W'} \mathbf{h} \cdot \mathbf{u}_j$, in which h is the sorted (in ascending order) vector of $H(x, y)$ where y is the node assigned module m_j and x is some node not yet assigned in S. Since each $H(x, y) = h(m_i, m_j, \mathbf{a})$ for some m_i, $\mathbf{u} = sort^a(\mathbf{u}_j)$. Since $\mathbf{h} \cdot \mathbf{u}_j \leq \mathbf{h}^j \cdot \mathbf{u}_j$, $h(S) \leq k(\mathbf{a}) - g(S) = h^*(S)$ for any S. □

	avg. no. of states		
σ	$\mu = 10$	$\mu = 100$	$\mu = 1000$
$.10\mu$	877	914	905
$.30\mu$	878	909	906
$.50\mu$	890	873	871
$.70\mu$	872	863	863
$.90\mu$	842	843	837

Table 1: Number of states generated by the search algorithm.

Table 1 shows the number of states generated by the search algorithm with the proposed heuristic function. The inputs are the tasks consisting of eight modules with randomly generated values of U_{ij}'s. U_{ij}'s are generated by a normally–distributed random variable with mean μ and variance σ^2. When σ is relatively large and a negative number is generated, U_{ij} is set to 0. Each entry is obtained by averaging the results over 1000 different inputs. It is found that the number of states generated is not co-related to μ. However, the number of generated states tends to decrease as σ increases. This agrees with our intuition since if σ is small, U_{ij}'s have a lower variance, and thus, the differences among assignments are not significant. This in turn leads to more states generated before an optimal mapping is found.

Note that when mapping an 8–module task into a Q_3, the total number of states in the search tree is

$$1 + 7 + (7!/5!) + (7!/4!) + (7!/3!) + (7!/2!) + 7! = 13,700.$$

As can be seen from Table 1, the search algorithm generates less than 7% of the total number of states.

When mapping a 16–module task into a Q_4, the total number of states in the search tree is in the order of 10^{12}. For modules with sparse communications among them and thus a high σ/μ ratio, the search algorithm can still manage to find an optimal solution in a reasonable amount of time. For example, if U_{ij} is given by a normally–distributed random variable with $\sigma = 0.9\mu$, then an optimal mapping can be found after generating approximately 10^5 states. However, in general, even if only 0.1 % of the total number of states were generated using a computer capable of calculating 10^5 states per second, the total amount of time required still exceeds 10^4 seconds.

This fast–growing state space makes the computation time required for finding an optimal mapping unacceptably long even with the help of the proposed heuristic function. Thus, we also need some fast heuristic algorithm to find a good suboptimal solution for large problems.

Greedy algorithms have been used to tackle some NP–hard problems and is shown to produce good results[14]. So, we will also investigate its performance on our task assignment problem.

In the greedy algorithm, it essentially tries to find a Hamiltonian cycle in the graph with the greatest total

weight using the greedy approach. As the algorithm traverses through the graph, each module is labeled with a distinct n–bit binary reflected Gray code. More heavily communicating module pairs will be placed in adjacent nodes, thus reducing the communication traffic between them. This process is done wherever a different module becomes the starting node of a Hamiltonian cycle while keeping the best solution with the least traffic obtained thus far. Since there are no data dependencies among the iterations of the algorithm, this greedy algorithm can be easily parallelized into an $O(M^2)$ algorithm executing on an M–processor system.

Obviously, the greedy approach does not find a Hamiltonian cycle with the minimum total weight since finding it is itself an NP–hard problem. Suppose there is an algorithm α which can find a Hamiltonian cycle in $\mathbf{U}$ with a total weight F, and $\mathbf{w}$ is a vector consisting of elements $U_{ij} \in \mathbf{U}$ which do *not* appear in the cycle and are sorted in ascending order. Also, let $\mathbf{v}$ denote the vector $[h^n[M], h^n[M+1], ..., h^n[Z-1]]$ where Z is the dimension of $\mathbf{h}^n$ as defined in Section 2, i.e., $Z = \binom{2^n}{2}$. Then, we have the following result which states the upper bound of communication traffic of assignments found with the results of applying α to $\mathbf{U}$.

Proposition 1: Given $\mathbf{U}$, the assignment $\mathbf{a}^\alpha$ which uses the Hamiltonian circuit found with algorithm α satisfies the following inequality:

$$k(\mathbf{a}^\alpha) \leq \mathbf{w} \cdot \mathbf{v} + F.$$

Our simulation results have indicated that the actual value of $k(\mathbf{a}^\alpha)$ is almost always within 20% of the upper bound. The simulation results also suggest that the Hamiltonian circuit with a minimized total weight can lead to a better solution. In the greedy approach, the Hamiltonian circuit found does not have the least weight. However, more complicated algorithms are much slower and do not produce solutions with traffic improvements large enough to justify the extra computational cost.

It is also found that the time required for the search algorithm to find the optimal solutions increases drastically when the cube size increases from 3 to 4. We will therefore find good suboptimal assignments efficiently for problems of sizes greater than $n = 3, M = 8$.

To assign a task consisting of $2^n, n \geq 4$, modules to a Q_n, the algorithm *Partition* divides the problem into 2^{n-3} sub-problems, each of which is a mapping problem of size $n = 3, M = 8$. Using the search algorithm with the proposed heuristic function, we find a local optimal assignment in a Q_3. These local optimal solutions are then combined into a solution for the original problem. The algorithm consists of three steps as follows:

1. Partitioning: the task graph corresponding to $\mathbf{U}$ is recursively partitioned down to K_8's so as to map subtasks into Q_3's. The edges in the task graph is first sorted according to U_{ij}'s. The graph is then partitioned into two equal–sized subgraphs by cutting through the least–weighted edges which are mutually disjoint, i.e., no two edges share a common vertex. This procedure is repeated until each subgraph becomes a K_8.
2. Application of the search algorithm to find local optimal solutions for mapping each K_8 into a Q_3.
3. Joining: Solutions for sub-problems are joined together to produce a solution for the original problem. The joining process follows the reverse of the cut procedure performed in the first step.

To study the performance of various algorithms, we simulated an n–cube to which 2^n task modules are to be assigned. The results for $n = 3, M = 8$ and $n = 4, M = 16$ are shown in Figs. 2 and 3, respectively. For larger problem sizes, the results are found to be consistent. In these plots, "A1" represents the greedy algorithm, "A2" denotes the state–space search, and "A3" represents the algorithm *Partition*. Also, the horizontal axis represents the σ/μ value while the vertical axis depicts the average communication traffic.

For each data point in these plots, the algorithm is executed for 1000 randomly–generated tasks where each U_{ij} is given by a normally–distributed random variable with mean $\mu = 100$ and variance σ^2, where σ ranges from 0.1μ to 0.8μ. Changing the absolute values of μ is found to have little effect on the relative performances of assignments found by different schemes as long as the ratio σ/μ remains constant. Therefore, only the results obtained for $\mu = 100$ are presented.

Our simulation results show the superiority of all three algorithms to random assignments and the improvements becomes more pronounced as σ/μ increases. The communication traffic is reduced by ≈ 1 % when $\sigma/\mu = 0.1$, but increased to more than 16 % when $\sigma/\mu = 0.8$. The differences in traffic among the assignments found by all three algorithms are not very significant. For inputs with small values of σ/μ, the differences are less than 3 %. Overall, the *partition* algorithm still consistently finds better solutions than the greedy algorithm, especially when the variance among inputs becomes large. Since optimal assignments cannot do much better in this situation, the margin allowing improvements is actually quite small. This agrees with our intuition that if the variance among U_{ij}'s is small, all assignments tend to have almost the same communication cost, and none of the three assignment algorithms can make much difference.

4 Low–Level Performance Evaluation

The actual performance of an assignment during task execution time depends greatly on the actual system im-

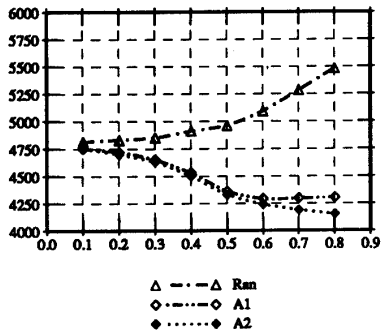

Figure 2: Performance comparison of algorithms, $n = 3, M = 8$.

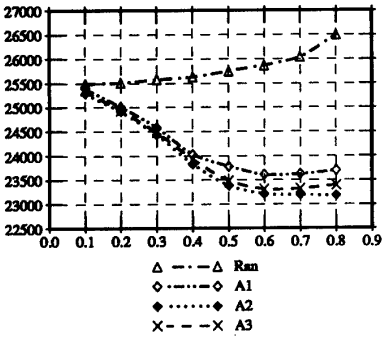

Figure 3: Performance comparison of algorithms, $n = 4, M = 16$.

plementation. In our simulations, we will again consider circuit switching and message switching.

We define a *communication event between modules*(CEBM) as an instance that a module needs to send a message — which usually consists of several packets — to another module, while defining a *communication event between nodes*(CEBN) as an instance of a node needing to send a message to some other node. In circuit switching, these two are indistinguishable. In message switching, however, a single CEBM can become several CEBNs. For example, when a pair of modules reside in two different nodes which are 2 hops apart, in circuit switching a CEBM from one module to the other is just a CEBN from one node to the other node. For message switching, however, this CEBM becomes two CEBNs: one from the source to the intermediate node, and the other from the intermediate node to the destination. Also, in circuit switching, messages can only be queued at the source node, but in message switching they may be queued at the source as well as at any intermediate nodes.

We say there is an *outstanding* CEBM or CEBN if a message needs to be sent by a node. An outstanding CEBM (or

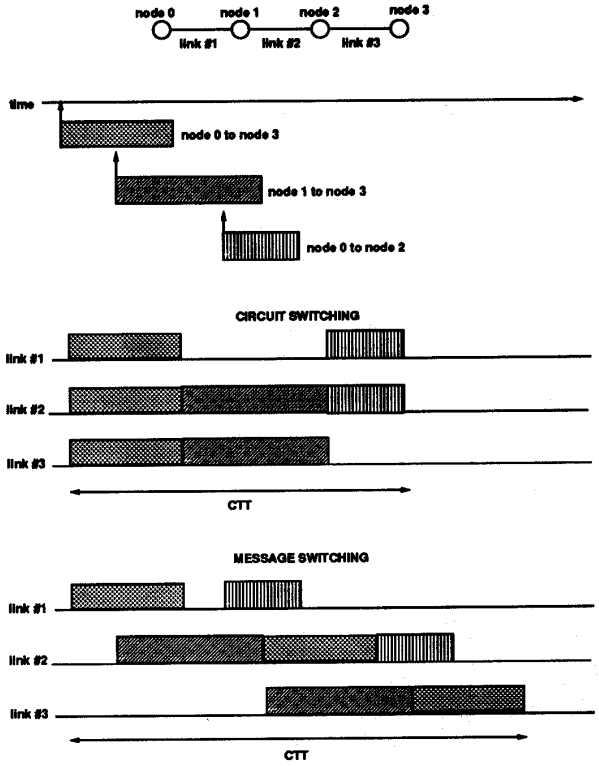

Figure 4: Example demonstrating the definition of CTT.

CEBN) is said to be *processed* if it is sent through a certain link in the system. An outstanding CEBM or CEBN may not be processed immediately due to the limited link resources available. A CEBM or CEBN is said to be *blocked* if it is not processed immediately. When the message is sent to its destination, the corresponding CEBM or CEBN is said to be *completed*. For message switching, a CEBM is actually turned into several CEBNs during the execution of a task if the communicating modules are located more than one hop apart. Therefore, the number of CEBNs depends on how task modules are assigned.

The goodness of a task assignment during its execution is measured by the *communication turnaround time* (CTT), which is the time span from the first CEBN becoming outstanding to the completion of all CEBNs. As an illustrative example, let us consider a network of four nodes with three CEBMs during one instance of task execution, as shown in Fig. 4. Three CEBMs become outstanding at three different times. The status of each link during the processing of these CEBMs under circuit switching and message switching is each shown in this figure.

As we have shown in Section 2, the actual time needed for a node to execute a module is invariant among different assignments, since at most one module is assigned to each

node. Therefore, CTT is the main source of difference in the completion time of a task.

To evaluate the resulting CTTs of a task assignment, we conducted simulations. Our simulation model is described below.

1. **Timing:** A time unit is selected as the time required to send a packet over a single communication link.
2. **Routing algorithm and mechanism:**
   - Under message switching, the routing mechanism at an intermediate node on a path will take a certain amount of time to forward a message from one link to the next. We assume this time to be relatively small and absorbed into the length of the corresponding message.
   - Under circuit switching, the time needed for finding an available communication path is ignored.
   - The propagation delay on a communication path is assumed to be negligible.
3. **Task communication behavior:**
   - T, given for each task, denotes the time span between the arrivals time of the first and last outstanding CEBMs. The arrival times of outstanding CEBMs are uniformly distributed in $[0, T]$. Hence, for a task, a larger T represents the task being less communication–bound, while a smaller T represents it being more communication–bound.
   - L_{msg}: the maximum message length measured in number of packets. The communication volume between each pair of modules is randomly grouped into messages of lengths within $[1, L_{msg}]$.
4. **Message scheduling and queueing:** If a link is busy when it is to be used for transmitting an incoming message, the message is stored in a FIFO queue at the source end of the link. When more than one message requests the use of the same link at a time, one of them is randomly chosen to use the link. This selection procedure is repeated until all requests are honored.

The goal of our simulation is to comparatively evaluate the goodness of different assignments under the same execution environment, but not to compare the performance of different system implementations. So, the simulation results should not be used to determine the relative performance of different switching methods or routing algorithms.

The assignments found by all three algorithms, as well as random assignments, are fed into the event–driven simulation program to evaluate their performances in a close to real–world environment. The results are shown in Table 2(a), (b) for message switching and circuit switching, respectively. U_{ij}'s of the inputs have $\sigma/\mu = 0.4$. The effects of changing T under the same assignment for a given task are also demonstrated.

For the problems of size $n = 4, M = 16$ and larger, changing T in the range $[10, 300]$ does not have any significant impact on the relative performance of assignments found with different algorithms. The assignments found with all of the above algorithms have shown substantial improvements over random assignments for $\forall\ T \in [10, 300]$. This is because the network is saturated with messages when $T = 300$.

	$n = 3, M = 8$			$n = 4, M = 16$			
T	ran	a1	a2	ran	a1	a2	a3
10	220	176	173	831	665	642	657
25	220	176	173	831	666	640	656
50	221	177	175	833	670	643	659
100	222	182	176	835	673	647	663
200	308	306	302	835	675	650	665
300	311	308	303	837	678	657	666

Table 2(a): CTTs of assignments under message switching.

	$n = 3, M = 8$			$n = 4, M = 16$			
T	ran	a1	a2	ran	a1	a2	a3
10	216	172	171	816	645	631	637
25	215	172	172	818	646	632	640
50	217	173	173	822	646	633	643
100	218	177	176	824	653	641	646
200	237	214	213	828	665	645	650
300	307	305	304	832	668	650	657

Table 2(b): CTTs of assignments under circuit switching.

In case of $n = 3, M = 8$, the network becomes less congested at $T \approx 160$ and the differences of CTTs among different assignment algorithms and random assignments start to diminish. So, we can conclude that minimizing communication traffic yields a peak improvement when the task to be assigned is communication–bound and the communication network may become highly congested during the execution of this task. For $n = 4, M = 16$, the T value which results in small performance differences is approximately 750, while for $n = 5, M = 32$ it is found to be about 2250. However, when T is relatively large and the network is not near saturation, the difference in the length of node queues can be made smaller by using the assignments resulting from the minimization of communication traffic. Depending on system implementation, the performance of a node may also be influenced by the length of message queue it has to maintain.

The effects of changing L_{msg} are more subtle than changing T. Generally, shorter message lengths result

in better performances in circuit–switched hypercubes. For message–switched hypercubes, changing the message length does not affect system performance notably if the overall communication traffic is fixed.

Our simulation results have indicated that different switching techniques do not matter much to system performance for communication–bound tasks, i.e., the network is congested with messages during task execution. Circuit switching is shown to have only a slightly better performance than message switching for the same task assignments. However, as mentioned earlier, the actual performance will depend on system implementation, and thus, the simulation results should not be used to compare the effectiveness of the two switching methods.

5 Concluding Remarks

In this paper we have formulated and solved the problem of mapping a task which is composed of interacting modules into a hypercube multicomputer by minimizing an objective function called the communication traffic. This objective function allowed us to find module assignments with the usual straight–forward combinatorial optimization techniques. The problem of finding an assignment that minimizes the communication traffic was proven to be NP–hard. Several algorithms have been investigated for finding optimal or suboptimal solutions for this problem.

It is also verified via simulations that, by optimizing this relatively simple function, the actual task execution communication efficiency, measured in communication turnaround time (CTT), is also optimized for communication–bound tasks.

References

[1] G. S. Rao, H. S. Stone, and T. C. Hu, "Assignment of tasks in a distributed processor system with limited memory," *IEEE Trans. on Computers*, vol. C-28, no. 4, pp. 291–299, April 1979.

[2] C. E. Houstis, "Allocation of real-time applications to distributed systems," in *Proc. of the 1987 Int'l Conf. on Parallel Processing*, pp. 863–866, August 1987.

[3] V. M. Lo, "Task assignment to minimize completion time," in *Proc. of the Fifth Int'l Conf. on Distributed Computer Systems*, pp. 329–336, May 1985.

[4] V. M. Lo, "Temporal communication graphs: A new graph theoretic model for mapping and scheduling in distributed memory systems," in *Proc. 6–th Distributed Memory Computing Conference*, pp. 248–252, April 1991.

[5] D. T. Peng and K. G. Shin, "Static program assignment in circuit switched multiprocessors," in *Proc. 6–th Distributed Memory Computing Conference*, pp. 244–247, April 1991.

[6] W. W. Chu, L. J. Holloway, M. T. Lan, and K. Efe, "Task allocation in distributed data processing," *Computer*, pp. 57–69, November 1980.

[7] K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *Computer*, vol. 15, no. 6, pp. 50–56, June 1982.

[8] C. Shen and W. Tsai, "A graph matching approach to optimal task assignment in distributed computing system using a minimax criterion," *IEEE Trans. on Computers*, vol. c-34, no. 3, pp. 197–203, March 1985.

[9] C. Lang, "The extension of object-oriented languages to a homogeneous concurrent architecture," Technical Report 5012:TR:82, California Institute of Technology, Department of Computer Science, 1982.

[10] 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, pp. 1396–1407, December 1987.

[11] A. S. Tanenbaum, *Computer Networks*, Prentice Hall, 1988.

[12] D. T. Peng and K. G. Shin, "Static allocation of periodic tasks with presedence constraints in distributed real–time systems," in *Proc. 9–th Int'l Conf. on Distributed Comput. Syst.*, pp. 190–198, June 1989.

[13] D. W. Krumme, K. N. Venkataraman, and G. Cybenko, "Hypercube embedding is NP-complete," in *Proc. of the First Conf. on Hypercube Concurrent Computers and Applications*, pp. 148–157, August 1985.

[14] M. R. Garey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman and Co., 1979.

[15] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing, 1980.