Heuristic Algorithms for Task Assignment in Distributed Systems*

Virginia Mary Lo

CIS-TR-86-13
April 15, 1987

Department of Computer and Information Science
University of Oregon

*This work was supported in part by the U.S. Office of Naval Research under contract N00014-79-C09775 and by the U.S. Department of Energy under contract DE-AC02-76ER02383.A003.

Abstract

In this paper we investigate the problem of task assignment in distributed computing systems, i.e., given a set of k communicating tasks to be executed on a distributed system of n processors, to which processor should each task be assigned? We propose a family of heuristic algorithms for Stone's classic model of communicating tasks which considers the execution cost of each task on each processor and the communication costs between tasks. We augment this model to include *interference costs* which reflect the degree of incompatibility between two tasks. Whereas high communication costs serve as a force of attraction between tasks, causing them to be assigned to the same processor, interference costs serve as a force of repulsion between tasks, causing them to be distributed over many processors. The inclusion of interference costs in the model yields assignments with greater concurrency thus overcoming the tendency of Stone's model to assign all tasks to one or a few processors. Simulation results show that our algorithms perform well and in particular, that the highly efficient Simple Greedy Algorithm performs almost as well as more complex heuristic algorithms. When we consider task assignment to be an *initial* placement of tasks subject to subsequent refinement by dynamic task migration algorithms, efficient algorithms such as Simple Greedy are attractive candidates for practical distributed systems. If task assignment modules make a *permanent* assignment of tasks to processors, the increased overhead of a more complex heuristic is justified for the improvement in the assignment.

1. Introduction

In the distributed computing environment, a job to be executed on the distributed system consists of a set of communicating tasks which we shall refer to as a *task force*. We define a *distributed system* as any configuration of two or more processors each with private memory. A system-wide operating system provides a message-passing mechanism among the processors, and it is assumed that the cost of transporting messages between processors is non-negligible. During the lifetime of the task force in the distributed system, task management modules guide the task force through several clearly identifiable phases:

- *task definition* - the specification of the identity and characteristics of the task force by the user, the compiler, and based on monitoring of the task force during execution.
- *task assignment* - the initial placement of tasks on processors
- *task scheduling* - local CPU scheduling of the individual tasks in the task force with consideration of the overall progress of the task force as a whole
- *task migration* - dynamic reassignment of tasks to processors in response to changing loads on the processors and communication network.

In this paper we focus on the problem of *task assignment*. We use this term to describe an initial assignment of tasks to processors which neither requires nor precludes subsequent dynamic migration of tasks. In particular, we are concerned with centralized task assignment algorithms that have global knowledge of the characteristics of the task force and of the distributed system. These task assignment algorithms seek to assign tasks to processors in order to achieve goals such as minimization of interprocess communication costs (IPC), good load balancing among the processors, quick turnaround time for the task force, a high degree of parallelism, and efficient utilization of system resources in general.

Our work is an extension of the graph theoretic approach to the task assignment problem begun by Stone [22] in which the definition of the task force is limited to (1) the execution cost of each task on each of the (heterogeneous) processors and (2) communication costs (IPC) incurred between tasks when they are assigned to different processors. In Stone's work a Max Flow/Min Cut algorithm can be utilized to find assignments which minimize total execution and communication costs. In this paper, we use Stone's model to develop a heuristic algorithm which combines recursive invocation of Max Flow/Min Cut algorithms with a greedy-type algorithm to find suboptimal assignments of tasks to processors. We present simulation results that show the performance of this heuristic to be very good.

We also discuss a serious deficiency in Stone's model in that it makes no direct effort to achieve concurrency, yielding assignments which utilize only one or a few of the processors. We therefore propose an extension of Stone's model to include an additional factor called *interference costs* which are incurred when tasks are assigned to the same processor. Interference costs reflect the degree of incompatibility between two tasks. For example, a pair of tasks that are both highly CPU-bound would have greater interference costs than a pair in which one task is CPU-bound and the other is I/O bound. Similarly, if two tasks were involved in pipelining it would be undesirable that they be assigned to the same processor; this incompatibility would be expressed in a high interference cost for that pair of tasks. Simulations show that addition of interference costs as a factor greatly improves the degree of concurrency in task assignments. We also show that network flow algorithms can be successfully applied to the extended model to find task assignments which minimize total execution, communication, and interference costs in certain restricted cases and near minimal cost assignments in more general cases.

Finally, we look at several versions of our algorithm which vary in their degree of complexity. We show that the more efficient algorithms perform almost as well as more complex versions. Thus, if initial task assignment is followed by later dynamic task migration, it would be more cost effective to use a simpler task assignment algorithm. This choice is also justified by the imprecise nature of task definition which can only make approximations of task characteristics such as IPC, interference cost, and execution costs. However, if task assignment modules make a *permanent* assignment of tasks to processors, the increased overhead of a more complex algorithm is justified for the incremental improvement in the assignment.

Section 2 gives background information on the task assignment problem. Sections 3 and 4 discuss the two models for this problem, heuristic algorithms for task assignment, and simulation results. Section 5 compares the efficiency of these algorithms and Section 6 discusses conclusions and directions for further research in this area.

2. Background on the Task Assignment Problem

The task assignment problem has received quite a lot of attention in the past decade. One approach to this problem has been through the development of centralized algorithms whose purpose is minimization/maximization of a clearly defined objective function that reflects the goals mentioned in Section 1. Stone and Bokhari [1], [2], [3], [22], [23], [24] conducted numerous studies of the task assignment problem for non-precedence constrained task systems with the objective of minimizing total execution and communication costs. Other researchers have looked at task assignment to minimize inter-process communication costs (IPC) with constraints on the degree to which the processors' loads are balanced [6]; minimization of the number of tasks per processor [14]; minimization of completion time [7], [14], [15], [19]. There is general agreement about the desire to minimize IPC as well as to achieve load balancing and to maximize parallelism, and the fact that these goals often come into conflict with each other.

In these many formulations of the task assignment problem, the problem of finding an optimal assignment of tasks to processors is found to be NP-hard [11], [14] in all but very restricted cases. Thus research has focused on the development of heuristic algorithms to find suboptimal assignments. Many of these heuristic algorithms use a graphical representation of the task-processor system such that a Max Flow/Min Cut Algorithm [10] can be utilized to find assignments of tasks to processors which minimize total execution and communication costs [14], [22], [23], [22]. This is the approach we shall take in this paper.

Before proceeding, we briefly mention some additional approaches to the task assignment problem. One such approach that is in contrast to the use of centralized algorithms involves de-centralized negotiation between the individual processors and a manager working on behalf of the task force. In [20], a contract bidding protocol is used in a hierarchically structured processor system to establish the assignment of tasks to processors. In the MICROS operating system for MICRONET [25], a scheme called *wave scheduling* is used to assign tasks on a distributed system that has an underlying hierarchical virtual machine which reflects the management structure of the system. In this scheme a task force manager requests a set of processors for its tasks and that request is transmitted in waves to lower levels of the management hierarchy. This technique is used to achieve simultaneous, decentralized task assignment for several task forces at the same time.

Some problems that occur in both these negotiation methods are that they do not adequately address the problem of minimization of IPC; and they incur significant overhead during the negotiation process. In addition, there is no concept of optimal assignment associated with this approach and thus it is difficult to evaluate a particular assignment over other possible assignments. However, the negotiation approach takes into account many more factors than the theoretically oriented algorithms described earlier.

Other techniques that have been used to study the task assignment problem include 0-1 quadratic programming [5], [18], clustering analysis [12], and queueing theory [4], [16]. A good overview of the task assignment problem can be found in [5].

3. Task Assignment to Minimize Total Execution and Communication Costs

We begin with the following model of task-processor systems and look at task assignment to minimize total execution and communication costs. Formally, we define a task force as a set of k tasks $T = \{t_1, t_2, ..., t_k\}$. In a distributed system containing n processors $P = \{p_1, p_2, ..., p_n\}$, x_{iq} denotes the execution cost of task t_i when it is assigned to and executed on processor p_q, $1 \leq i \leq k$, $1 \leq q \leq n$. The execution cost of task t_i on processor p_q depends on the work to be performed by that task and on the attributes of the processor, such as its clock rate, instruction set, existence of floating point hardware, cache memory, etc. Let c_{ij} denote the communication cost between two tasks t_i and t_j if they are assigned to different processors. Throughout our discussion, we will assume that the communication cost between two tasks executed on the same processor is negligible. These execution and communication costs are derived from an appraisal of the characteristics of the task force and of the distributed system. They may be specified explicitly by the programmer, deduced automatically by the compiler, queried from the operating system, or refined by dynamic monitoring of previous executions of the task force. For this study, we presume that the data about the execution and communication costs is somehow available and that these costs are expressible in some common unit of measurement. For tractibility we ignore other attributes of the task-processor system such as memory requirements, deadlines, precedence constraints; and we assume that communication costs are independent of the communication link upon which they occur.

An assignment of tasks to processors can formally be described by a function from the set of tasks to the set of processors, $f : T \rightarrow P$. In a system of k tasks and n processors there are

n^k possible assignments of tasks to processors. An optimal assignment is defined as one which minimizes the total sum of execution and communication costs incurred by that assignment. For example, consider the task processor system depicted in Figure 1. This system is made up of 4 tasks and 3 processors. The execution costs x_{iq} and the communication costs c_{ij} are represented in tabular form. Figure 1a shows the total execution and communication costs incurred by the arbitrary assignment, and Figure 1b shows the cost incurred by an optimal assignment (one which minimizes total execution and communication costs).

The problem of finding an assignment of tasks to processors which minimizes total execution and communication costs was elegantly analyzed using a network flow model and network flow algorithms by Stone [22], [23] and by a number of other researchers [9], [14], [15], [17], [24], [26]. Using this approach a system of k tasks and n processors is modeled as a network in which each processor is a distinguished node and each task is an ordinary node. An edge is drawn between each pair of task nodes t_i and t_j and is given the weight c_{ij}, the communication costs between the two tasks. There is an edge from each task node t_i to each processor node p_q with the weight

$$w_{iq} = \frac{1}{n-1} \sum_{r \neq q} x_{ir} - \frac{n-2}{n-1} x_{iq} . \tag{1}$$

An n-way cut in such a network is defined to be a set of edges which partitions the nodes of the network into n disjoint subsets with exactly one processor node in each subset and thus corresponds naturally to an assignment of tasks to processors. The cost of an n-way cut is defined to be the sum of the weights on the edges in the cut. Because of the judcious choice of weights according to Eq. (1), the cost of the n-way cut is exactly equal to the total sum of execution and communication costs incurred by the corresponding assignment. This construction is illustrated in Figure 2. In a 2-processor system, an optimal assignment can be found in polynomial time utilizing Max Flow/Min Cut Algorithms [10]. However, for arbitrary

n, the problem is known to be NP-hard [11]. Thus it is necessary to turn to heuristic algorithms which are computationally efficient but which may yield suboptimal assignments.

3.1. Algorithm A

Our algorithm, which we shall refer to as Algorithm A consists of three parts: (I) Grab, (II) Lump, and (III) Greedy. We first describe Algorithm A informally and then give a formal treatment of each portion of the algorithm.

The first part of Algorithm A, Grab, produces a partial (possibly complete) assignment of tasks to processors by having each processor "grab" those tasks that are strongly attracted to it (i.e., the weight is large on the edges connecting those tasks to this processor). This process is accomplished as follow:

(1) For a given processor p_i we convert the n-processor network described above into a 2-processor network consisting of p_i and a supernode $\overline{p}_i$ which represents the other $n-1$ processors. (The details of this construction are described later.)

(2) We then apply a Max Flow/Min Cut Algorithm to this 2-processor network to find those tasks that would be assigned to p_i in the 2-processor network. This construction and application of the Max Flow/Min Cut Algorithm is repeated for each processor, yielding a partial assignment of tasks to processors.

(3) The n-processor network is then reconfigured by eliminating the tasks assigned in steps (1) and (2) and by recalculating the edge weights to reflect the partial assignment. Then Grab is applied recursively to the reconfigured network.

(4) Grab halts when no further assignment of tasks to processors occurs.

If the assignment is complete, it is optimal. However, it is possible that some tasks may remain unassigned. In that case, Lump tries to find a quick and dirty assignment by assigning all

remaining tasks to one processor if it can be done "cheaply enough". The precise meaning of "cheaply enough" is a tunable parameter and is described later. Finally, if Lump cannot complete the assignment, Greedy is invoked. Greedy identifies clusters of tasks between which communication costs are "large". Greedy merges such clusters of tasks and assigns all tasks in the same cluster to the cheapest processor for that cluster. The resultant assignment may be suboptimal.

3.1.1. Details of Grab

In this section we describe Grab in detail and prove that if Grab produces a total assignment of tasks to processors, that assignment is optimal.

Definition: Let $\Pi = \{S_1, S_2, ..., S_n, S^r\}$ be a partition of the tasks T in an n-processor system into $n+1$ disjoint subsets, with S^r possibly empty. Let f_Π be the (partial) assignment of tasks to processors in which tasks in S_i are assigned to processor p_i, $i = 1,...,n$ and tasks in S^r are not assigned to any processor. The (partial) assignment f_Π is said to be a *prefix* of an optimal assignment if there exists an optimal assignment for which tasks in S_i are assigned to processor p_i, $i = 1,...,n$.

Theorem 3.1: Let p_j be an arbitrary processor in an n-processor system. Consider the network G_j obtained from the network G in Figure 2 by the following construction: the set of processor nodes $P - \{p_j\}$ are merged into a single super node $\overline{p}_j$. For each task node t_i, $i = 1,...,k$, the edges from node t_i to processor nodes p_q in the set $P - \{p_j\}$ are replaced with one edge with weight equal to the combined sum of the weights on the original edges (see Figure 3). The minimum cut in the network G_j with p_j and $\overline{p}_j$ as source and sink, respectively, induces a partition of nodes in G_j into two disjoint subsets, A_j containing p_j and $\overline{A}_j$ containing $\overline{p}_j$. The

(partial) assignment in which tasks in A_j are assigned to processor p_j is a prefix of all optimal assignments for G. A similar theorem was proved by Stone [22]. Our proof can be found in [14].[1]

Lemma 3.1: Let G be a network as described in Theorem 3.1 and let p_{j_1} and p_{j_2} be two distinct processor nodes in G. Let A_{j_1} be the set of tasks assigned to processor p_{j_1} and let A_{j_2} be the set of tasks assigned to processor p_{j_2} by the application of Theorem 3.1 to G. Then $A_{j_1} \bigcap A_{j_2} = \phi$. In other words, no two processors will try to grab the same task.

Proof: By Theorem 3.1 the partial assignment in which tasks in A_{j_1} are assigned to p_{j_1} is a prefix of all optimal assignments for G and the partial assignment in which tasks in A_{j_2} are assigned to p_{j_2} is also a prefix of all optimal assignments for G. If $A_{j_1} \bigcap A_{j_2} \neq \phi$ then there exists a task which is assigned to both p_{j_1} and to p_{j_2} in every optimal assignment. This result is impossible. Q.E.D.

Algorithm Grab: The first pass of Grab begins with the network $G^1 = G$ as defined above. In each pass, the Max Flow/Min Cut Algorithm is applied for $j = 1,2,...,n$ with processor node p_j as source and the set $P - \{p_j\}$ as the sink (as described in Theorem 3.1) to determine the subset of tasks assigned to p_j. Note that by Lemma 3.1 no task will be assigned to more than one processor by this procedure. The resultant assignment may be partial in that there may be

[1] Theorem 3.1 as stated above was shown to be incorrect by Abraham and Davidson in *Task Assignment Using Network Flow Methods for Minimizing Communication in n-processor Systems,* Center for Supercomputing Research and Development Technical Report No.598, September 1986. The last sentence of the theorem should be reworded to read "The (partial) assignment which corresponds to *a minimum cut with minimum node cardinality* in which tasks in A_j are assigned to processor p_j is a prefix of all optimal assignments for G." Fortunately, the error in Theorem 3.1 above does not affect the operation of Algorithm A if a Max Flow/Min Cut Algorithm which finds the minimum cut of minimum cardinality (such as the Ford-Fulkerson Algorithm) is utilized.

tasks which remain unassigned. Let T^m denote the set of tasks which remain unassigned after m passes. We construct a network G^{m+1} from the network G^m used in the mth pass by deleting from G^m all task nodes not in T^m and by redefining the execution cost for t_i in T^m on processor p_q as

$$x_{iq}^{m+1} \triangleq x_{iq} + \sum_{r \neq q} \sum_{t_j \in S_r^m} c_{ij} \tag{2}$$

where S_r^m is the set of tasks assigned to processor p_r by the first m passes of Grab. In other words, x_{iq}^{m+1} is equal to the original execution cost x_{iq} plus the sum of communication costs between t_i and all tasks already assigned to processors other than p_q. The weight on the edge from t_i to p_q is recalculated according to Eq. (1) with the new values of execution cost for all tasks in T^m. The process of applying the Max Flow/Min Cut Algorithm in the network G^{m+1} with p_j and P - $\{p_j\}$ as source and sink, respectively, is repeated for $1 \leq j \leq n$. The iteration process halts when either all tasks are assigned or when no tasks are assigned in the last iteration. In the latter case, Part II of Algorithm A is invoked.

Theorem 3.2: An assignment produced by Algorithm Grab is a prefix of all optimal assignments for G.

Proof: The proof is by induction on the number of passes m in Algorithm Grab and can be found in [14].

Lemma 3.2: If the assignment produced by Grab is complete, that assignment is optimal.

Proof: By Theorem 3.2, the assignment f produced by Grab is a prefix of all optimal assignments for G^1. Since f is a prefix of itself, it is therefore an optimal assignment. Q.E.D.

3.1.2. Details of Lump

If Grab halts with unassigned tasks remaining, Part II of Algorithm A, Lump, tests the possibility of assigning all the remaining tasks to one processor. Lump is applied to a reduced network containing the subset of tasks T^m not assigned by Grab. In the reduced network, the processor nodes are eliminated and we only look at the task nodes with edges between communicating tasks labeled with weight c_{ij}. Lump computes a lower bound L on the cost of an optimal n-way cut for the reduced network under the constraint that more than one processor be utilized in the corresponding assignment. We defined the lower bound to be

$$L = \sum_{t_i \in T^m} \min_p (x_{ip}) + \min_{i \neq r} c(t_r, t_i)$$

where $c(t_r, t_i)$ is the cost of the minimum cut for some arbitrarily chosen task t_r and task t_i.

L then is the sum of two quantities. The first term is a lower bound on the *execution* costs in the optimal n-way cut. This term is simply the execution costs incurred if each task in T^m is assigned to its cheapest processor. The second term is a lower bound on the *communication* costs incurred in an optimal n-way cut. This lower bound is computed by arbitrarily choosing some task t_r and computing all the minimum cuts between t_r and the other tasks in T^m. We then find the minimum of these mincuts and this quantity serves as a lower bound on the communication costs incurred in an optimal n-way cut because in such a cut, t_r must be separated from some other task. Based on this lower bound, the algorithm then checks to see if it would be cheaper to assign all remaining tasks to one processor. If so, the tasks in T^m are all assigned to the one processor yielding minimum total execution cost for those tasks. In this case, the resultant assignment in combination with the assignment from Part I is optimal. Otherwise, Part III is invoked to complete the assignment.

3.1.3. Details of Greedy

Part III, Algorithm Simple Greedy, locates clusters of tasks between which communication costs are "large". Tasks in a cluster are then assigned to the same processor, and the resultant assignment may be suboptimal. Let $T^m = \{t_1, t_2, ..., t_7\}$ be the set of unassigned tasks remaining after Lump. Let G be a graph in which each task t_i is represented by a node and in which there is an edge between each pair of communicating tasks with weight c_{ij}. Greedy uses two tunable parameters: C, a cutoff value for communication costs, and X, a cutoff value for execution costs. For this implementation of Simple Greedy we defined

C = the average communication costs over all pairs of tasks

$$= \sum_{1 \le i \le j \le k} \frac{c_{ij}}{\binom{k}{2}}$$

and

$X = \infty$ (i.e., clusters are always merged)

Algorithm Simple Greedy

- Initially, each task in T^m is in a task group by itself.
- Compute the average communication cost C as defined above.
- Mark all edges between tasks $t_1, t_2, ..., t_7$ for which $c_{ij} \le C$.
- While there are unmarked edges remaining
    - Find an unmarked edge $e = (t_i, t_j)$. Mark it.
      G_i is the task group containing t_i.
      G_j is the task group containing t_j.
    - If there is some processor p_q for which $\sum_{t_l \in G_i \bigcup G_j} x_{lq} < X = \infty$ then
        - Merge the two groups: $G = G_i \bigcup G_j$

- Mark all the edges between tasks in G_i and tasks in G_j

- Else do not merge G_i and G_j.

- Assign each task group to a processor which minimizes the total execution cost of the group.

3.2. Simulations

In order to evaluate the performance of Algorithm A in finding suboptimal task assignments which minimize total execution and communication costs, simulation runs were performed on a variety of typical task forces. Altogether, 536 task forces were simulated with the number of tasks ranging from 4 to 35 and the number of processors ranging from 3 to 6. The simulations were performed under the UNIX operating system running on a VAX 11/780. Optimal assignments were computed using a branch and bound backtracking algorithm.

The data used in the simulations are organized into four categories. Dataset 1 (*Clustered*) consists of randomly generated task-processor systems in which tasks form clusters. Communication costs between tasks within the same cluster are on the average larger than communication costs between tasks in different clusters. Dataset 2 (*Sparse*) consists of randomly generated task-processor configurations in which the communication matrix is sparse. In particular, the communication costs are nonzero for only $\frac{1}{6}$ of the $\binom{k}{2}$ possible pairs of tasks. Dataset 3 (*Actual*) consists of data representing actual task forces derived from numerical algorithms, operating systems programs, and general applications programs. In this dataset, specific information about the number of tasks and/or about which pairs of tasks communicate with each other was available in the literature. Estimates of execution and communication costs were made from information such as the number and type of messages passed between tasks,

from the function of the tasks, and from raw data on these costs. Dataset 4 (*Structured*) consists of task forces whose task graphs have the structure of a ring, a pipe, a tree, or a lattice. Details about these datasets can be found in [14].

The results of these simulations show Algorithm A to be very successful in finding suboptimal assignments. Figure 4 summarizes this information by showing the distribution of the ratio $\frac{T_A}{T_O}$, where T_A is the total sum of execution and communication costs for assignments produced by Algorithm A, and T_O is the total sum of execution and communication costs for an optimal assignment. For all datasets combined, Algorithm A found an optimal assignment in 34.8% of the cases and for dataset 2 Algorithm A found an optimal assignment in 59.6% of the cases. In 94.6% of the cases, the cost of the assignment produced by Algorithm A was less than 1.5 times greater than the cost of an optimal assignment. Ratios greater than 2.0 were found in only 3 cases of the 536 cases. The worst ratio $\frac{T_A}{T_O}$ was 2.7. Algorithm A did not perform as well on the *Actual* and *Structured* datasets because Greedy presumes some clustering of tasks while the these datasets did not exhibit this feature.

4. Task Assignment with Interference Costs

A major flaw in the use of total execution and communication costs as the performance criteria to be optimized is that no explicit advantage is given to concurrency. In other words, no explicit effort is made to utilize many processors in order to reduce the completion time of the set of tasks. Some degree of parallelism is introduced into task assignments as a by-product of the goal of avoiding high total costs, but concurrency is not sought as a goal itself. Thus, the use of total execution and communication costs as the performance measure often yields assignments which utilize only a few of the available processors.

For example, in the two processor task system shown in Figure 5, an assignment which minimizes total execution and communication costs is shown with solid lines (task t_1 assigned to processor p_1 and tasks t_2 through t_{10} assigned to processor p_2). The assignment shown with dotted lines (t_1 through t_5 on p_1 and t_6 through t_{10} on p_2) has the same cost. We note that the latter assignment yields a higher degree of parallelism. Use of total execution and communication costs as the performance criterion fails to discriminate between these two assignments, and the Max Flow/Min Cut Algorithm will select the former assignment. In systems with n identical processors, the use of total execution and communication costs as the criteria for optimality is even more undesirable since an optimal assignment always assigns all tasks to one processor (thereby eliminating all communication costs).

For this reason, we present the concept of interference costs which are incurred when two tasks are assigned to the same processor. Interference costs reflect the degree of incompatibility between two tasks based on characteristics of the two tasks and the processors to which they may be mutually assigned. For example, a pair of tasks that are both highly CPU-bound could have greater interference cost than a pair in which one task is CPU-bound and the other performs alot of I/O. Interference costs serve as forces of repulsion between tasks to counterbalance the forces of attraction due to (high) communication costs. We assume interference costs are derived somehow from user specifications, compiler analyses, and dynamic monitoring of the task force; and that a common unit of measure can be found for execution, communication, and interference costs.

In particular, let $T = \{t_1, \cdots t_k\}$ be the set of tasks, $P = \{p_1, \cdots p_n\}$ be the set of processors, and let x_{ij}, $1 \leq i \leq k, 1 \leq j \leq n$ and $c_{ij}, 1 \leq i, j \leq k$ be the execution costs and communication costs, respectively, as defined before. Let $I_q(i,j), 1 \leq i, j \leq k, 1 \leq q \leq n$ be the

interference cost incurred if tasks t_i and t_j are assigned to the same processor p_q. We assume that $I_q(i,j) = I_q(j,i)$. We define an optimal assignment as one which minimizes the total sum of execution, communication, and interference costs.

Interference costs can be attributed to two main factors. The first factor affects every pair of tasks that are assigned to the same processor and involves contention between tasks for the resources of the processor to which the tasks are both assigned. In particular, when several tasks execute on the same processor, they incur overhead due to process switching in a multiprogrammed environment and overhead due to synchronization for access to shared resources such as memory, I/O devices, CPU time, etc. We shall refer to the portion of interference costs attributable to contention for resources as *processor-based interference costs.* The second factor which contributes to interference cost involves only those tasks which communicate with each other. When two communicating tasks are assigned to the same processor, they may utilize the interprocess communication services provided by that processor in order to send and receive messages. Thus communicating tasks incur an interference cost due to contention for message buffers and synchronization for message-passing. We shall refer to the portion of interference costs attributable to contention for these latter resources as *communication-based interference costs.* We note that the communication-based interference costs which are incurred when two communicating tasks are assigned to the same processor are always smaller in magnitude than the communication costs incurred when the two tasks are assigned to different processors. In both cases, the communicating tasks incur costs because they utilize the interprocess communication facilities. However, if the tasks reside on different processors, communication costs include, in addition, transit delay incurred by sending messages through the communication subnetwork.

Thus, the interference cost between two tasks t_i and t_j which arises when they are both assigned to processor p_q can be expressed as the sum of two components:

$$I_q(i,j) = I_q^P(i,j) + I_q^C(i,j)$$

where $I_q^P(i,j)$ is the processor-based component of interference cost and $I_q^C(i,j)$ is the communication-based component. The communication-based component satisfies the inequality

$$I_q^C(i,j) \leq c_{ij}$$

In the next three sections we show that the network flow model can be successfully extended to several interesting cases which consider execution, communication, and interference costs. Simulation results show that the addition of interference cost to the model does indeed yield assignments with greater concurrency.

4.1. Interference Costs Which are Independent of Processor

In this section we consider task-processor systems for which interference cost is independent of the processor to which the two tasks t_i and t_j are assigned. That is, $I_p(i,j) = I_{ij}$. An n processor system can be modeled as a network in which an n-way cut corresponds to an assignment of tasks to processors. Let the edge from each task node t_i to each processor node p_q have the weight

$$w_{iq} = \frac{1}{n-1}\sum_{r \neq q} x_{ir} - \frac{n-2}{n-1}\, x_{iq} + \frac{1}{2(n-1)} \sum_{1 \leq l \leq k} I_{il}$$

Let the edge between two task nodes t_i and t_j have the weight

$$c'_{ij} = c_{ij} - I_{ij}.$$

This construction is illustrated in Figure 6.

Theorem 4.1: An n-way cut in such a network has cost equal to the total sum of execution, communication, and interference costs for the assignment corresponding to that cut. (Thus a minimum cut yields an assignment which minimizes the total sum of execution, communication,

and interference costs.) The proof of this theorem can be found in [14].

It is known that the problem of finding an optimal n-way cut in a network is NP-complete and thus the problem of finding an optimal assignment is also NP-complete. However, because an optimal assignment is equivalent to an n-way cut, Algorithm A of Section 3 can be applied to find suboptimal assignments with near minimal values for total execution, communication, and interference costs.

If we further assume that $I_{ij} \leq c_{ij}$, $1 \leq i,j \leq k$, then $c_{ij}' = I_{ij} - c_{ij} > 0$ and the Max Flow/Min Cut Algorithm can be applied to find optimal assignments for 2-processor systems. For n-processor systems, Algorithm A described above can be applied to find suboptimal assignments. For arbitrary I_{ij}, the Max Flow/Min Cut Algorithm cannot be invoked because there may be edges with negative weights in the network representation of the task-processor system. However, in this case, the Simple Greedy Algorithm of Algorithm A can be applied to find suboptimal assignments.

4.2. Simulations

Simulations were performed (1) to demonstrate that the use of interference costs does indeed yield assignments with greater parallelism and (2) to examine the performance of Algorithm A in finding assignments which minimize the total sum of execution, communication, and interference costs. The simulations used data representing typical task-processor configurations generated from the four datasets described in Section 3. For each configuration, interference costs were generated from the uniform distributions over the intervals $[1,2\overline{c}]$, $[1,\frac{3}{2}\overline{c}]$, $[1, \frac{\overline{c}}{2}]$, and $[1, \frac{\overline{c}}{10}]$, where $\overline{c}$ is the average communication cost for a particular task-processor system.

For each task-processor configuration, we measured optimal and suboptimal values of *total costs* for the configuration with execution, communication, and interference costs and also for the same configuration without interference costs (execution and communication costs only). In order to assess the degree of parallelism attained by assignments, we also measured optimal and suboptimal values of *completion time* for each task-processor configuration, both with and without interference costs. Our definition of *completion time* is a natural extension to the classical definition of *latest finishing time* used in deterministic scheduling theory for multiprocessor systems with execution costs only [6]. In the model with execution and communication costs, we define *completion time* as

$$\omega_f = \max_{1 \le q \le n} \left(\sum_{f(t_i) = p_q} x_i + \sum_{\substack{f(t_j) = p_q \\ f(t_i) \neq p_q}} c_{ij} \right).$$

i.e., the total execution and communication costs incurred on the processor for which these costs are maximal over all processors. Similarly, in the model with execution, communication, and interference costs, *completion time* is defined as

$$\omega_f = \max_{1 \le q \le n} \left(\sum_{f(t_i) = p_q} x_i + \sum_{\substack{f(t_i) = p_q \\ f(t_j) \neq p_q}} c_{ij} + \sum_{\substack{f(t_i) = p_q \\ f(t_j) = p_q}} I_{ij} \right).$$

i.e., the total sum of execution, communication, and interference costs incurred on the processor for which this total is maximal over all the processors. The concept of *completion time* is illustrated in Figure 7 using a Gantt diagram. In this figure, the communication costs are depicted as occuring in one lump, but it should be kept in mind that these costs are actually dispersed in time throughout the execution of the tasks.

The five values that we measured are listed below. The interpretation of the terms *total cost* and *completion time* depend on whether the configuration includes interference costs or not.

T_A, the total cost of an assignment by Algorithm A;

T_O, the optimal value of total cost;

ω_A, the completion time of an assignment by Algorithm A;

ω_O, the optimal value of completion time;

ω_T, the completion time of an assignment which optimizes total cost.

In each of the figures to be discussed below, we compare the ratio of suboptimal costs to optimal costs. For example, the ratio $\frac{\omega_A}{\omega_O}$ reflects the performance of Algorithm A in finding assignments with minimal completion time. If the ratio is 1.0 Algorithm A's assignment is optimal. If the ratio is 1.10 Algorithm A's assignment is 10% greater than optimal, and so on. In each figure, results are presented both in table form and graphically.

Figures 8 and 9 demonstrate empirically that addition of interference costs to the model yields assignments with a high degree of concurrency. Figure 8 shows this is true for assignments which have optimal values for total costs while Figure 9 shows this is also true for suboptimal assignments found by Algorithm A. Figures 8 and 9 also compare the degree of concurrency attained in assignments for the interference cost model with the degree of concurrency attained in assignments in the model without interference costs. While the improvement is as expected, the magnitude of the improvement is significant.

Figure 8 shows the distribution of the ratio $\frac{\omega_T}{\omega_O}$ for systems which include interference costs and for systems without interference costs. Recall that ω_T is the completion time of an assignment which is optimal with respect to total costs while ω_O is the optimal value of completion time. Thus, the ratio $\frac{\omega_T}{\omega_O}$ reflects the degree of concurrency attained by assignments with an optimal value for total costs. From the percentage figures in the first row

of the table in Figure 8, we see that assignments with an optimal value for total costs also have excellent values for completion time. For example, 22% of the assignments were also optimal with respect to completion time, 61.0% of the assignments are less than 1.1 times the optimal value, and 98.3% of the assignments were less than 1.5 times the optimal value. We also see that use of interference costs yields a marked improvement in the distribution of the ratio $\frac{\omega_T}{\omega_O}$. For example, when interference costs are included, 98.3% of the assignments that are optimal with respect to total costs also have completion times that are less than 1.5 times the optimal completion time. However, without interference costs, that figure is only 49.1%. While this difference is as expected, the magnitude of the difference is notable.

Figure 9 shows the distribution of the ratio $\frac{\omega_A}{\omega_O}$ for systems with interference costs and for systems without interference costs. Recall that ω_A is the completion time of an assignment found by Algorithm A while ω_O is the optimal value of completion time. This ratio demonstrates the degree of concurrency attained by suboptimal assignments found by Algorithm A. The results from this table show that Algorithm A finds assignments with a good degree of concurrency when interference costs are included in the model. For example, when interference costs are included, 93.2% of assignments found by Algorithm A have completion times that are less than 1.5 times the optimal completion time. However, in the model without interference costs, only 42.3% of the assignments found by Algorithm A have a completion time that is less than 1.5 times the optimal value.

Figure 10 demonstrates the performance of Algorithm A in finding suboptimal assignments which minimize total execution, communication, and interference costs. (In other words, we now ignore the issue of concurrency and just look at the performance of Algorithm A in finding suboptimal assignments in the interference cost model.) The table shows the distribution of the

ratio $\frac{T_A}{T_O}$ for Algorithm A with and without interference costs. It is clear that Algorithm A does perform better for Stone's model than for the model with interference costs added. For example, without interference costs Algorithm A found an optimal assignment in 25% of the cases. However, with interference costs Algorithm A found an optimal assignment in only 3% of the cases. Similarly, the cost of 91.3% of the assignments were less than 1.5 times the cost of an optimal assignment without interference costs, but this figure fell to only 78.4% with interference costs. Thus, Algorithm A is not well-suited for minimizing total costs when interference costs are added to the model. This result is not surprising since Algorithm A was designed for Stone's model and thus considers interference costs only during the Grab part of the algorithm.

To summarize,

(1) In the model with interference costs, an assignment with optimal total costs also has excellent values of completion time. In the model without interference costs, an assignment with optimal total costs often has poor values of completion time.

(2) This same trend hold for suboptimal assignments found by Algorithm A.

(3) Algorithm A is not as well suited as a heuristic for the model with interference costs and we should investigate other heuristics for this model.

Thus, we have shown that it is desirable to augment Stone's model with interference costs and that heuristics designed to minimize total execution, communication, and interference costs will also yield assignments with a high degree of concurrency. However, Algorithm A is not a useful heuristic for this purpose.

4.3. Arbitrary Interference Costs

In this section we consider the general case when interference cost is dependent on both the processor and the tasks involved. Let $I_q(i,j)$ be the interference cost incurred when tasks t_i and t_j are both assigned to processor p_q. For task processor systems with two processors and under the assumption that

$$\frac{I_1(i,j) + I_2(i,j)}{2} \leq c_{ij} \tag{3}$$

an optimal assignment of tasks to processors can be found using network flow algorithms. This assumption states that the average interference cost between two tasks over the two processors be less than or equal to the communication costs between the two tasks. Again, if we consider interference costs as arising from the memory contention and synchronization overhead between communicating tasks, it is reasonable to make an even stronger assumption that

$$I_q(i,j) \leq c_{ij}; \quad q = 1,2$$

and thus (3) certainly holds true.

We represent the task processor system as a network as usual. The edge from task node t_i to processor node p_q is given the weight

$$x_{iq}' = x_{iq} + \sum_{1 \leq l \leq k} \frac{I_q(i,l)}{2}$$

and the edge between task nodes t_i and t_j is given the weight

$$c_{ij}' = c_{ij} - \frac{I_1(i,j) + I_2(i,j)}{2}$$

This construction is illustrated in Figure 11.

Theorem 4.2: A cut in such a network has cost equal to the total sum of execution, communication, and interference costs for the assignment corresponding to that cut. (Thus a minimum cut yields an assignment which minimizes the total sum of execution, communication,

and interference costs.)

The proof is analogous to the proof of Theorem 4.1 and can be found in [14]. Again, the Max Flow/Min Cut Algorithm can be applied to find optimal assignments for the 2-processor case.

For the n processor case, a suitable model has not yet been found.

5. Comparison of Algorithms

The complexity of each of the parts of Algorithm A is discussed below. Let e be the number of edges in the network representation of a task processor system with k tasks and n processors.

Grab: $O(nk^2e \log k)$

> There exist Max Flow/Min Cut Algorithms of complexity $O(ke \log k)$ [10] and there will be at most k total iterations wth n min cuts per iteration.

Lump: $O(k^2e \log k)$

> Computation of the lower bound L on the cost of an n-way cut when tasks are assigned to more than one processor involves finding $k-1$ mincuts in a network with task nodes only.

Simple Greedy: $O(en)$

> Simple Greedy examines each communication edge. For each edge Simple Greedy checks to see if there is a processor to which all the tasks can be assigned.

The Simple Greedy phase of Algorithm A was initially designed to "finish up" assignments that were not completed by Grab and Lump. However, because of the efficiency of Simple Greedy, we decided to investigate its performance alone. Figure 12 compares the performance of the phases of Algorithm A with that of Simple Greedy and two augmented versions of Simple Greedy which we shall call Complex Greedy and Sort Greedy.

Simple Greedy is the Greedy algorithm of Algorithm A.

Complex Greedy is an augmented version of Simple Greedy in which an estimate is made of the cost of assigning two task groups to different processors. Complex Greedy merges the two groups if and only if there exists a processor for which the cost of assigning all tasks in the two groups is smaller than the estimate.

Sort Greedy is an augmented version of Simple Greedy in which communication edges are examined in order of non-increasing cost. (In Simple Greedy, these edges are examined in random order.) The sorting of communication edges adds a factor of $O(e \log e)$ to the complexity of the algorithm, where e is the number of communication edges with non-zero cost.

From Figure 12, we see that the performance of Simple Greedy was close to that of the more complex Algorithm A. For example, Algorithm A found an optimal assignment in 23.1% of the cases while Simple Greedy found an optimal assignment in 20.5% of the cases. We conclude that when an assignment is subject to further adjustment through dynamic task migration, efficient algorithms like Simple Greedy are more useful for a quick initial assignment of tasks to processors. If the assignment is permanent, the better performance of Algorithm A is worth the increased overhead.

From the table we also see that there was no significant difference in the performance of the three Greedy Algorithms. It is surprising that Sort Greedy did not yield better results. One would expect that elimination of communication edges in the order most expensive to least expensive would be more effective in the identification of communicating clusters and thus yield better assignments. One possible explanation for this phenomenon is the fact that in both Simple Greedy and Sort Greedy, communication edges whose costs are less than the average communication cost are eliminated from consideration. As a result, primarily inter-cluster edges remain, reducing the probability that tasks from two different clusters will be merged. The order in which the communication edges are examined would thus be less crucial.

6. Conclusions and Areas for Further Research

Our investigation of the static task assignment problem has resulted in the development of several heuristic algorithms for Stone's model which considers execution and communication costs only and for our model which introduces the concept of interference costs. Simulation results indicate that these algorithms perform well on a variety of task-processor systems. In addition, we have shown that highly efficient algorithms perform almost as well as more complex ones and thus are feasible for use in practical applications.

Our current research continues to look at the task assignment problem. An obvious extension to this research is to increase the complexity of the model to include such factors as memory requirements, deadlines, precedence constraints, and communication link loads. It will be interesting to see how much complexity we can include in the model before we are forced to move from elegant graph theoretic algorithms to increasingly empirical techniques. We are also interested in characteristics of the algorithms themselves. In particular, a task assignment algorithm should incorporate qualities such as

> monotonicity - as the resources of the distributed system increase (e.g. more processors available), the cost of the assignment produced by the algorithm should not increase. In other words, the algorithm should not display anomalous behavior such as the FIFO page replacement algorithm.
>
> sensitivity - since execution, communication, and interference costs will always be approximations, the algorithm should not be overly sensitive to small variations in these quanitites.
>
> robustness (fault tolerance) - the algorithm should adapt to failures in the system such as removal of nodes, failure of communication links, etc.

In addition, we are looking more closely at the relationship between the two goals of achieving load balancing among processors and the minimization of interprocess communication (IPC). There is general agreement that these two goals are often in conflict with one another, but there

is no data available about the degree and circumstances for this conflict. Experiments are underway to determine parameters of the task force that may affect the degree of conflict between these two goals. In addition, while there is a quantifiable measure for IPC, a precise definition of load balancing does not exist. Completion time is often used as a measure of the degree to which an algorithm achieves load balancing, but this metric can yield fairly unbalanced assignments. We are looking into new ways to measure load balancing within the context of the task assignment problem.

Task assignment thus continues to offer a wide variety of challenging problems. While much work has focused on this problem by itself, it is also time to integrate our view of task assignment into the overall picture of task managment (task definition, task assignment, task scheduling, and task migration) -- to see its place in the total life cycle of the task force in the distributed system.

Execution Costs			
	p_1	p_2	p_3
t_1	31	4	14
t_2	1	5	6
t_3	2	4	24
t_4	3	28	10

Communication Costs				
	t_1	t_2	t_3	t_4
t_1	0	35	3	8
t_2	35	0	6	4
t_3	3	6	0	23
t_4	8	4	23	0

Arbitrary Assignment	
t	$f(t)$
t_1	p_2
t_2	p_2
t_3	p_2
t_4	p_3

(a) Cost of arbitrary assignment (total execution and communication costs)

$$= x_{12} + x_{22} + x_{32} + x_{43} + c_{14} + c_{24} + c_{34}$$

$$= 4 + 5 + 4 + 10 + 8 + 4 + 23$$

$$= 58$$

Optimal Assignment	
t	$f(t)$
t_1	p_2
t_2	p_2
t_3	p_1
t_4	p_1

(b) Cost of optimal assignment (total execution and communication costs)

$$= x_{12} + x_{22} + x_{31} + x_{41} + c_{13} + c_{23} + c_{14} + c_{24}$$

$$= 4 + 5 + 2 + 3 + 3 + 3 + 6 + 8 + 4$$

$$= 38$$

Fig. 1: Example of task assignment problem

Execution Costs			
	p_1	p_2	p_3
t_1	3	11	23
t_2	1	∞	9
t_3	14	10	21
t_4	15	6	8
t_5	∞	2	7

Communication Costs					
	t_1	t_2	t_3	t_4	t_5
t_1	0	15	2	0	0
t_2	15	0	0	0	0
t_3	2	0	0	12	17
t_4	0	0	12	0	23
t_5	0	0	17	23	0

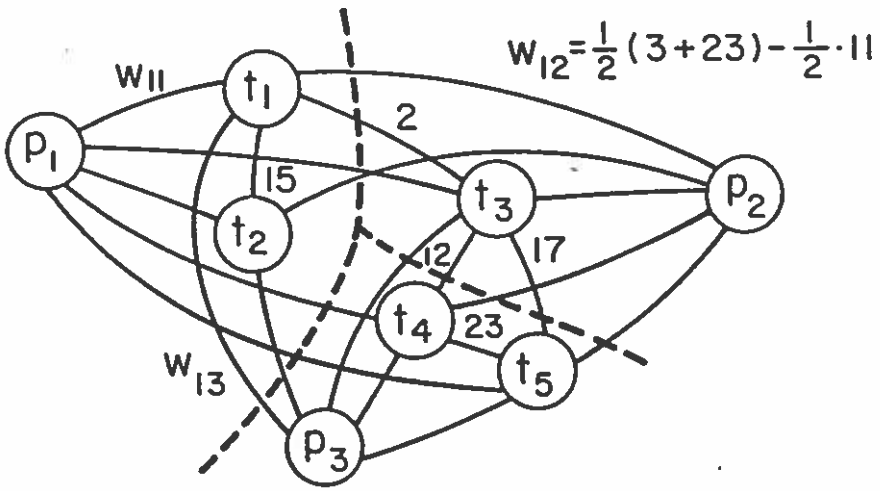


Fig. 2: n-processor network

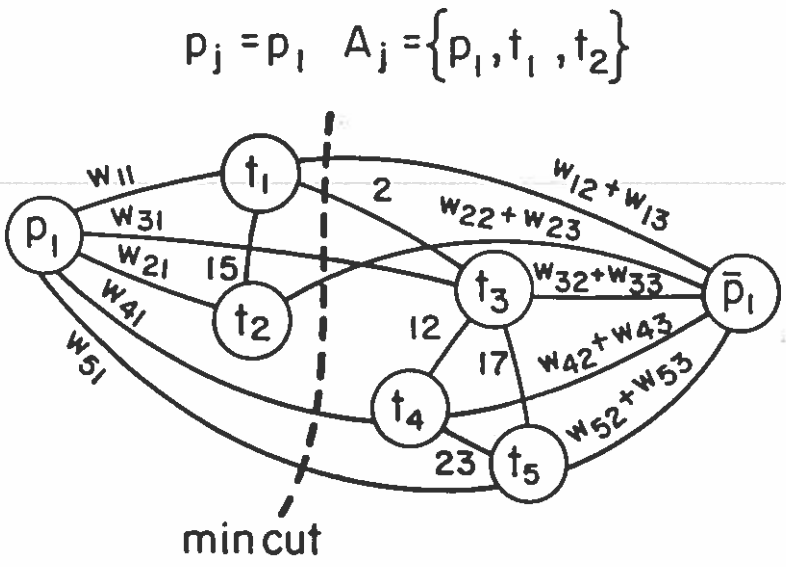


Fig. 3: Reduced network

Distribution of T_A/T_O							
Dataset	Total No. of Simulations	Percent of Simulations					
		(Optimal) = 1.00	≤ 1.10	≤1.20	≤1.30	≤ 1.40	≤ 1.50
All Data	536	33.4%	57.8%	74.1%	84.3%	89.9%	92.7%
(1) Clustered	228	46.9%	69.3%	80.3%	85.6%	90.9%	93.5%
(2) Sparse	55	47.3%	70.9%	85.4%	94.5%	96.3%	96.3%
(3) Actual	168	23.8%	53.0%	71.5%	84.0%	90.5%	91.7%
(4) Structured	85	7.1%	28.2%	55.3%	75.3%	82.4%	90.6%

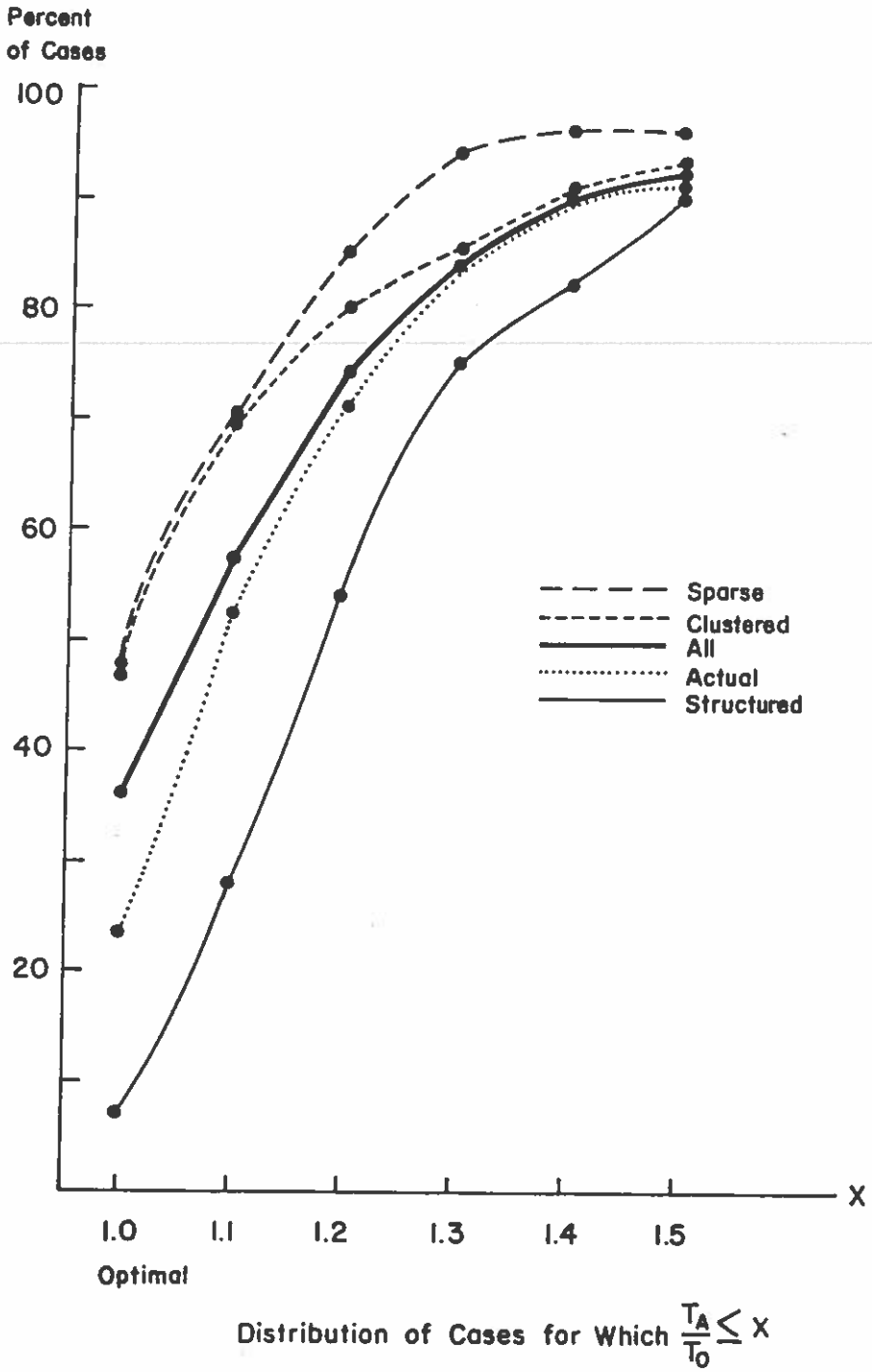
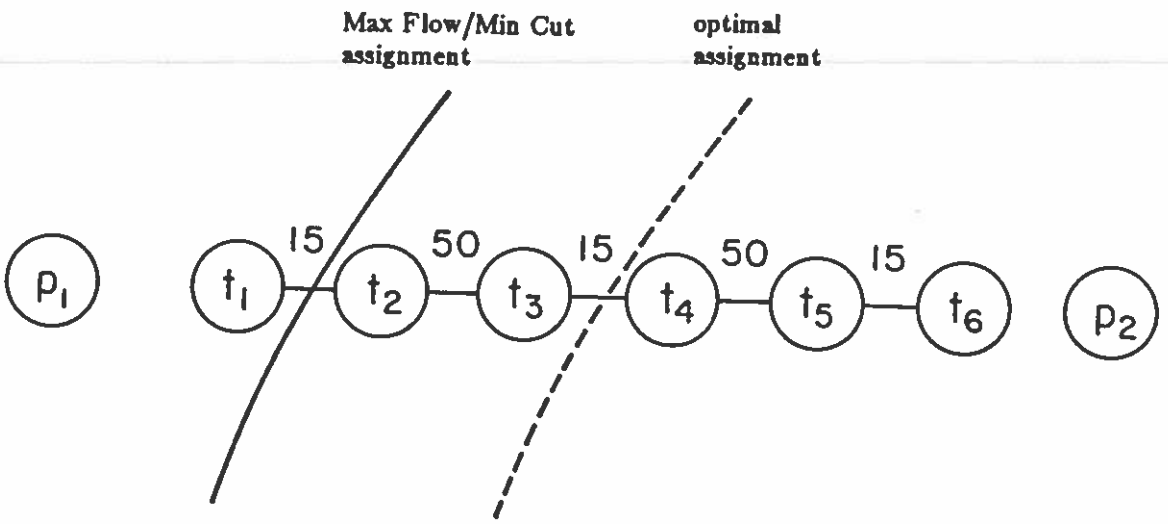


Fig. 4: Performance of Algorithm A for several datasets

Execution Costs		
	p_1	p_2
t_1	20	50
t_2	25	10
t_3	5	20
t_4	10	20
t_5	10	20
t_6	50	10

Communication Costs						
	t_1	t_2	t_3	t_4	t_5	t_6
t_1	0	15	0	0	0	0
t_2	0	0	50	0	0	0
t_3	0	0	0	15	0	0
t_4	0	0	0	0	50	0
t_5	0	0	0	0	0	15
t_6	0	0	0	0	0	0

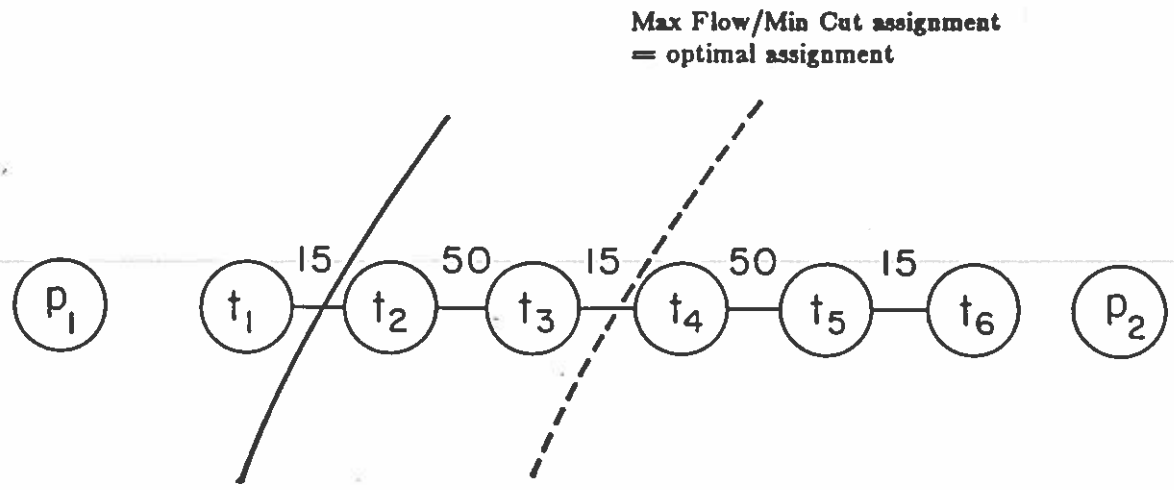


——— Cost $= x_{11} + x_{22} + x_{32} + x_{42} + x_{52} + x_{62} + c_{12} = 115$

– – – Cost $= x_{11} + x_{21} + x_{31} + x_{42} + x_{62} + x_{62} + x_{34} = 115$

Fig. 5a: Poor assignment without interference costs

Interference Costs						
	t_1	t_2	t_3	t_4	t_5	t_6
t_1	0	10	10	10	10	10
t_2	10	0	10	10	10	10
t_3	10	10	0	10	10	10
t_4	10	10	10	0	10	10
t_5	10	10	10	10	0	10
t_6	10	10	10	10	10	0



$$\begin{aligned}\text{Cost} &= x_{11} + x_{22} + x_{32} + x_{42} + x_{52} + x_{62} + c_{12} \\ &\quad + I_{23} + I_{24} + I_{25} + I_{26} + I_{34} + I_{35} + I_{36} \\ &\quad + I_{45} + I_{46} + I_{56} \\ &= 115 + \binom{5}{2} \cdot 10 \\ &= 215\end{aligned}$$

$$\begin{aligned}\text{Cost} &= x_{11} + x_{21} + x_{31} + x_{42} + x_{52} + x_{62} + c_{34} \\ &\quad + I_{12} + I_{13} + I_{23} + I_{45} + I_{46} + I_{56} \\ &= 115 + \binom{3}{2} \cdot 10 + \binom{3}{2} \cdot 10 \\ &= 175\end{aligned}$$

Fig. 5b: Better assignment with interference costs

Execution Costs		
	p_1	p_2
t_1	5	8
t_2	3	16
t_3	2	2
t_4	12	3
t_5	7	10

Communication Costs					
	t_1	t_2	t_3	t_4	t_5
t_1	0	12	8	0	0
t_2	12	0	5	0	0
t_3	8	5	0	0	0
t_4	0	0	0	0	9
t_5	0	0	0	9	0

Interference Costs					
	t_1	t_2	t_3	t_4	t_5
t_1	0	6	2	0	0
t_2	6	0	2	0	0
t_3	2	2	0	0	0
t_4	0	0	0	0	3
t_5	0	0	0	3	0

$$X'_{12} = X_{12} + \frac{I_{12}+I_{13}+I_{14}+I_{15}}{2}$$

$$= 8 + 4 = 12$$

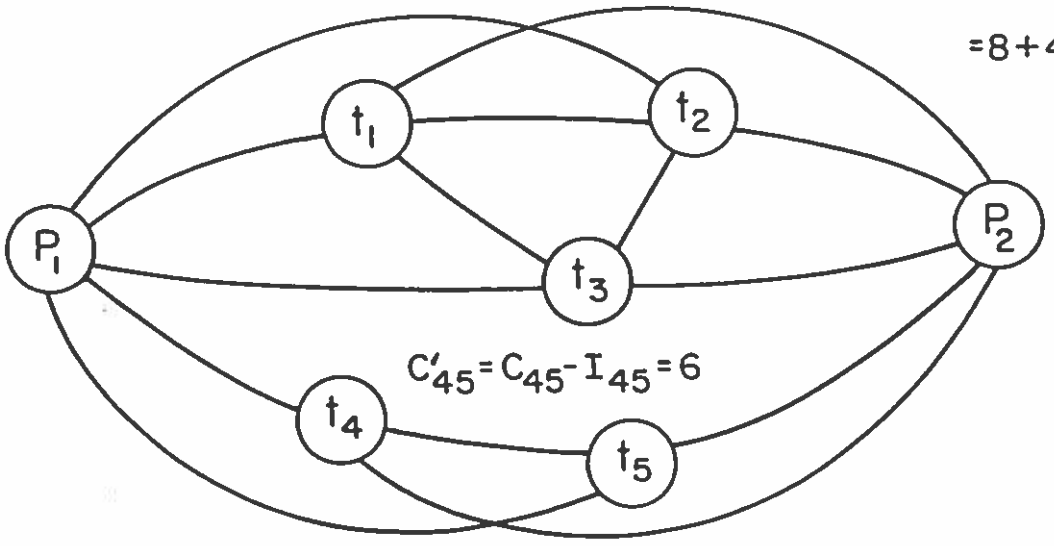
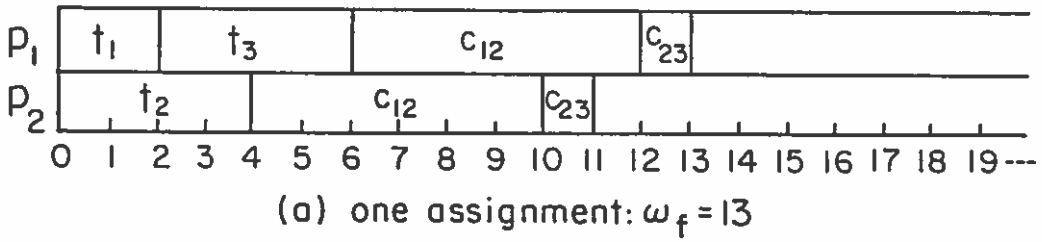


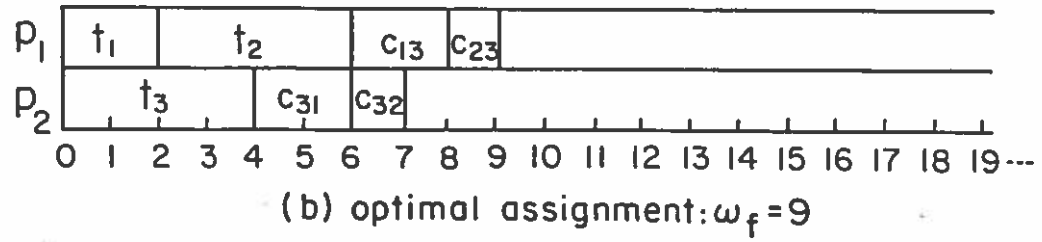
Fig. 6: Processor-independent interference costs

Execution Costs	
	p_1, p_2
t_1	2
t_2	4
t_3	4

Communication Costs			
	t_1	t_2	t_3
t_1	0	6	2
t_2	6	0	1
t_3	2	1	0



(a) one assignment: $\omega_f = 13$



(b) optimal assignment: $\omega_f = 9$

Fig. 7: Completion time (execution and communication costs)