DAYANANDA SAGAR UNIVERSITY

SCHOOL OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

FOURTH SEMESTER

ALGORITHM DESIGN AND ANALYSIS

(16CS209)

TASK SCHEDULING USING GREEDY ALGORITHM

MINI PROJECT REPORT

submitted by

KAVUTURI DEVA BHARGAV- ENG17CS0107

KIRAN.M-ENG17CS0111

DAYANANDA SAGAR UNIVERSITY

CERTIFICATE

This is to certify that the Algorithm Design and Analysis Mini-Project report entitled **"Task Scheduling using Greedy Algorithm"** being submitted by

Mr.Kiran.M (ENG17CS0111)

to Department of Computer Science and Engineering, School of Engineering, Dayananda Sagar University, Bangalore, for the fourth semester Bachelor of Technology of this university during the academic year January – May 2019.

Date: _____

_____

Signature of the Faculty in Charge

_____

Signature of the Chairman

ACKNOWLEDGEMENT

I am pleased to acknowledge **Dr. Jasma Balasangameshwara**, Associate Professor, Department of Computer Science and Technology, School of Engineering, for her invaluable guidance, support, motivation and patience during the course of this mini project work.

I extend our sincere thanks to **Dr. M.K. Banga**, Professor and Chairman of the Department of Computer Science and Engineering, School of Engineering, who continuously helped me throughout the project and without his guidance, this project would have been an uphill task.

I have received a great deal of guidance and cooperation from my friends and family and I wish to thank one and all that have directly or indirectly helped me in the successful completion of this mini-project work.

Kiran.M (ENG17CS0111)

TABLE OF CONTENTS

PROBLEM STATEMENT

Given an 2D array of tasks where every task has a deadline and associated bonus if the task is finished before the deadline the task should be selected and performed. It is also given that every task takes single unit of time, so the minimum possible deadline for any task is one. It is all about how to maximize total bonus if only one task can be scheduled per unit of time.

The aim of this project is to develop an efficient algorithm to calculate the maximum possible number of bonus points by performing the tasks on or before the given deadline .

PRINCIPLE ASSUMPTIONS OF SCHEDULING

- One machine can process only one operation at a time.
- Each operation once started must be completed first.
- Preceding operation must be completed before the succeeding one can proceed
- Processing time is known and fixed and the time taken in transfer of tasks between machines is neglected.
- Machines to be used are of different types.
- Tasks to be processed are known and ready for processing before the period under consideration begins.[1]

RESTRICTIONS AND LIMITATIONS

- The inputs should be in integer form only.

INTRODUCTION

ABOUT THE DESIGN TECHNIQUE

Greedy Technique: A greedy strategy always makes the choice that seems to be the best at that moment. A greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized. The greedy algorithm has only one shot to compute the optimal solution so that it never goes back and reverse the decision. it is quite easy to come up with a greedy algorithm for a problem. Analysing the run time for greedy algorithm will generally be much easier than for other techniques.

Task scheduling : as the selection of an order for a series of tasks to be done on a number of service facilities (machine).The purpose of scheduling problems is to complete the task within the minimum possible time, keeping the minimum idle time of the machines (or services).

ABOUT THE PROBLEM

This problem consists of n tasks each associated with a deadline and bonus and our objective is to earn maximum bonus. We will earn bonus only when task is completed on or before deadline. We assume that each task will take unit time to complete.

- In this problem we have n tasks t1, t2, … tn each has an associated deadline d1, d2, … dn and bonus b1, b2, ... bn.
- Bonus will only be awarded or earned if the task is completed on or before the deadline.
- We assume that each task takes unit time to completed.
- If two or more tasks has the same deadline then the tasks with maximum bonus is selected to perform.
- The objective is to earn maximum bonus when only one task can be scheduled or processed at any given time. [2]

ALGORITHM

Step 1: Start

Step 2: Input the array of bonus and deadline of the task

Step 3: Sort the task in the decreasing order based on bonus

Step 4: Take the maximum deadline and create an array(slot)

Step 5: And add one task after the anther checking if the slot is empty

Step 6:

```
for(Cnt=1;Cnt<=TotalTasks;Cnt++)
{
   if(Tasks[Cnt].Deadline >= Days)
    {
      a[Cnt]=1;
      TotalBonus+=Tasks[Cnt].Bonus;
      Days++;
    }
  else

     TotalBonus=TotalBonus;
}
```

Step 7: The table with slot and deadline is displayed

Step 8: Print the tasks which are performed, tasks that are not performed and maximum bonus.

Step 9: Exit

PSEUDO CODE

```
struct Homework
{
  int Deadline;
  int Bonus;
};
struct Homework Tasks[Size];
int Compare(const void *_a,const void *_b)
{
  struct Homework *a,*b;
  a=(struct Homework *)_a;
  b=(struct Homework *)_b;
  if(a->Deadline>b->Deadline)return 1;
  if(a->Deadline<b->Deadline)return -1;
  if(a->Bonus<b->Bonus)return 1;
  if(a->Bonus>b->Bonus)return -1;
  return 1;
}

void printTasks(int N)
```

```
  {
  int Cnt;
  printf("\nTASK_ID\tDEADLINE\tBONUS");
  for(Cnt=1;Cnt<=N;Cnt++)
    printf("\nTask%d\t%d\t\t%d",Cnt,Tasks[Cnt].Deadline,Tasks[Cnt].Bonus);
  }

int main()
  {
   int i;
   int TotalTasks;
   int TotalBonus=0;
   int Cnt;int Days=1;
    printf("\n\t\t\t***Task Scheduling Using Greddy Strategy***\n");
    printf("Enter the total no. of tasks ");
    scanf("%d",&TotalTasks);
      int a[TotalTasks];
      for(i=1;i<=TotalTasks;i++)
      {
          a[i]=0;
      }
    for(Cnt=1;Cnt<=TotalTasks;Cnt++)
      scanf("%d %d",&Tasks[Cnt].Deadline,&Tasks[Cnt].Bonus);

    qsort(&Tasks[1],TotalTasks,sizeof(struct Homework),Compare);
    for(Cnt=1;Cnt<=TotalTasks;Cnt++)
```

```
    {
      if(Tasks[Cnt].Deadline >= Days)
        {
          a[Cnt]=1;
          TotalBonus+=Tasks[Cnt].Bonus;
          Days++;
         }
       else
         TotalBonus=TotalBonus;
    }
    int j=1;
    printTasks(TotalTasks);
      printf("\n\n1) TASKS THAT ARE PERFORMED");
    for(i=1;i<=TotalTasks;i++)
    {
        if(a[i]==1)
        {
            printf("\nTASK_ID %d sholud be performed on day %d to get %d
bonus",i,j,Tasks[i].Bonus);
            j++;
        }
    }
    printf("\n\n2) TASKS THAT ARE NOT PERFORMED");
    for(i=1;i<=TotalTasks;i++)
    {
        if(a[i]==0)
        {
```

```
            printf("\nTASK_ID %d sholud be performed on day %d to get %d
bonus",i,j,Tasks[i].Bonus);

            j++;

         }

      }

      printf("\n total bonus=%d",TotalBonus);

      return 0;

    }
```

TIME COMPLEXITY

Time Complexity of the above solution is **$O(n^2)$**.The program was successfully implemented by using greedy method. The maximum profit with the scheduling is obtained in the output screen. Task scheduling will minimize the work in progress and maximizes the utilization of production resource.

TESTING

Test Case no.	Title	Input	Expected output	Actual output	Result
1	with no repeated and no skipped deadlines	N=7 DEADLINE={1,2,3,4,5,6,7}BONUS={2,4,6,8,9,3,8}	Total bonus=40	Total bonus=40	pass
2	With repeated deadlines	N=5 DEADLINE={1,1,2,3,4,} BONUS={2,5,6,3,8}	Total bonus=22	Total bonus=22	pass
3	With skipped deadlines	.N=6 DEADLINE={1,2,4,5,6,7} BONUS={5,7,4,9,3,1}	Total bonus=29	Total bonus=29	pass
4	Tasks with same deadlines	.N=3 DEADLINE={2,2,2} BONUS={1,2,3}	Total bonus=5	Total bonus=5	pass

OUTPUT SNAPSHOTS

1.

```
                        ***Task Scheduling Using Greddy Strategy***
Enter the total no. of tasks 7
1       2
2       4
3       6
4       8
5       9
6       3
7       8

TASK_ID DEADLINE        BONUS
Task1   1               2
Task2   2               4
Task3   3               6
Task4   4               8
Task5   5               9
Task6   6               3
Task7   7               8

1) TASKS THAT ARE PERFORMED
TASK_ID 1 sholud be performed on day 1 to get 2 bonus
TASK_ID 2 sholud be performed on day 2 to get 4 bonus
TASK_ID 3 sholud be performed on day 3 to get 6 bonus
TASK_ID 4 sholud be performed on day 4 to get 8 bonus
TASK_ID 5 sholud be performed on day 5 to get 9 bonus
TASK_ID 6 sholud be performed on day 6 to get 3 bonus
TASK_ID 7 sholud be performed on day 7 to get 8 bonus

2) TASKS THAT ARE NOT PERFORMED
 total bonus=40
```

2.

```
                        ***Task Scheduling Using Greddy Strategy***
Enter the total no. of tasks 5
1       2
1       5
2       6
3       3
4       8

TASK_ID DEADLINE        BONUS
Task1   1               5
Task2   1               2
Task3   2               6
Task4   3               3
Task5   4               8

1) TASKS THAT ARE PERFORMED
TASK_ID 1 sholud be performed on day 1 to get 5 bonus
TASK_ID 3 sholud be performed on day 2 to get 6 bonus
TASK_ID 4 sholud be performed on day 3 to get 3 bonus
TASK_ID 5 sholud be performed on day 4 to get 8 bonus

2) TASKS THAT ARE NOT PERFORMED
TASK_ID 2 sholud be performed on day 5 to get 2 bonus
 total bonus=22
```

3.

```
                        ***Task Scheduling Using Greddy Strategy***
Enter the total no. of tasks 6
1       5
2       7
4       4
5       9
6       3
7       1

TASK_ID DEADLINE        BONUS
Task1   1               5
Task2   2               7
Task3   4               4
Task4   5               9
Task5   6               3
Task6   7               1

1) TASKS THAT ARE PERFORMED
TASK_ID 1 sholud be performed on day 1 to get 5 bonus
TASK_ID 2 sholud be performed on day 2 to get 7 bonus
TASK_ID 3 sholud be performed on day 3 to get 4 bonus
TASK_ID 4 sholud be performed on day 4 to get 9 bonus
TASK_ID 5 sholud be performed on day 5 to get 3 bonus
TASK_ID 6 sholud be performed on day 6 to get 1 bonus

2) TASKS THAT ARE NOT PERFORMED
 total bonus=29
```

4.

```
                        ***Task Scheduling Using Greddy Strategy***
Enter the total no. of tasks 3
2       1
2       2
2       3

TASK_ID DEADLINE        BONUS
Task1   2               3
Task2   2               2
Task3   2               1

1) TASKS THAT ARE PERFORMED
TASK_ID 1 sholud be performed on day 1 to get 3 bonus
TASK_ID 2 sholud be performed on day 2 to get 2 bonus

2) TASKS THAT ARE NOT PERFORMED
TASK_ID 3 sholud be performed on day 3 to get 1 bonus
 total bonus=5
```

CONCLUSION

We have successfully scheduled the tasks using this program . This program uses greedy

Strategy. We get the list of the tasks which should be performed and shoulnot be performed to get the maximum bonus points. The greedy algorithm has only one shot to compute the optimal solution so that it never goes back and reverse the decision. it is quite easy to come up with a greedy algorithm for a problem. Analysing the run time for greedy algorithm will generally be much easier than for other techniques.

REFERENCES

[1]: dyclassroom-greedy-algorithm-job-sequencing

[2]: dyclassroom-greedy-algorithm-job-sequencing(wikipedia)

Heuristic Algorithms for Task Assignment in Distributed Systems

VIRGINIA MARY LO, MEMBER, IEEE

Abstract—**In this paper, we investigate the problem of static 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 whose goal is the minimization of the total execution and communication costs incurred by an assignment. In addition, 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.**

Index Terms—**Distributed systems, load balancing, resource allocation, scheduling, task assignment, task scheduling.**

I. 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 systemwide operating system provides a message-passing mechanism among the processors, and it is assumed that the cost of transporting messages between processors is nonnegligible. 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

Manuscript received December 18, 1985; revised December 28, 1987. 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.

The author is with the Department of Computer and Information Science, University of Oregon, Eugene, OR 97403.

IEEE Log Number 8820887.

0018-9340/88/1100-1384$01.00

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 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 II gives background information on the task assignment problem. Sections III and V discuss the two models for this problem, heuristic algorithms for task assignment, and simulation results. Section IV compares the efficiency of these algorithms and Section VI discusses conclusions and directions for further research in this area.

II. 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 I. Stone and Bokhari [1]–[3], [22]–[24] conducted numerous studies of the task assignment problem for nonprecedence-constrained task systems with the objective of minimizing total execution and communication costs. Other researchers have looked at task assignment to minimize interprocess communications costs (IPC) with constraints on the degree to which the processors' loads are balanced [8]; 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], [24]. 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 decentralized 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].

III. 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 $T = \{t_1, t_2, \cdots, t_k\}$. In a distributed system containing n processors $P = \{p_1, p_2, \cdots, p_n\}$, 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 are somehow available and that these costs are expressible in some common unit of measurement. For tractability 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

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)

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

(b)

Fig. 1. Example of task assignment problem. (a) Cost of arbitrary assignment (total execution and communication costs) = $x_{12} + x_{22} + x_{32} + x_{43} + c_{14} + c_{24} + c_{34} = 4 + 5 + 4 + 10 + 8 + 4 + 23 = 58$. (b) Cost of optimal assignment (total execution and communication costs) = $x_{12} + x_{22} + x_{31} + x_{41} + c_{13} + c_{23} + c_{14} + c_{24} = 4 + 5 + 2 + 3 + 3 + 3 + 6 + 8 + 4 = 38$.

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 Fig. 1. This system is made up of four tasks and three processors. The execution costs x_{iq} and the communication costs c_{ij} are represented in tabular form. Fig. 1(a) shows the total execution and communication costs incurred by an arbitrary assignment and Fig. 1(b) 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 judicious choice of weights according to (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 Fig. 2. In a two-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.

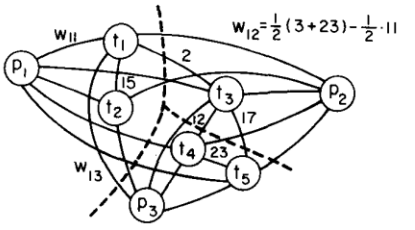


Fig. 2. n-processor network.

A. 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 follows.

1) For a given processor p_i we convert the n-processor network described above into a two-processor network consisting of p_i and a supernode $\bar{p}_i$ 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 two-processor network to find those tasks that would be assigned to p_i in the two-processor network.

3) Steps 1) and 2) are repeated for each processor, yielding a partial assignment of tasks to processors. This repeated application of the Max Flow/Min Cut Algorithm, once for each processor, constitutes *one pass* of Grab.

4) The n-processor network is then reconfigured by eliminating the tasks assigned in the previous pass and by recalculating the edge weights to reflect the partial assignment. Grab continues iteratively with the next pass using the reconfigured network.

5) 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

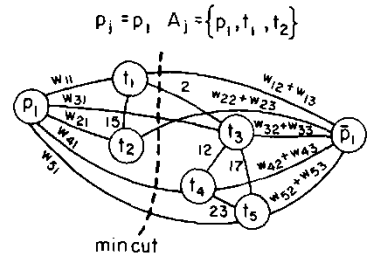


Fig. 3. Reduced network.

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.

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 $\Pi = \{S_1, S_2, \cdots, S_n, S^r\}$ 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, $i = 1, \cdots, n$ 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, $i = 1, \cdots, n$.

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 Fig. 2 by the following construction: the set of processor nodes $P - \{p_j\}$ are merged into a single supernode $\bar{p}_j$. For each task node t_i, $i = 1, \cdots, k$, 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 Fig. 3). The minimum cut in the network G_j with p_j and $\bar{p}_j$ as source and sink, respectively, induces a partition of nodes in G_j into two disjoint subsets, A_j containing p_j and $\bar{A}_j$ containing $\bar{p}_j$. 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 $A_{j_1} \cap A_{j_2} = \phi$. 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 $A_{j_1} \cap A_{j_2} \neq \phi$, 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:

- Initially, we begin with network $G^1 = G$ as defined above.
- $m = 0$ /* m is the pass number */
- Repeat until all tasks are assigned or no new tasks are assigned in a given pass.

/* **begin pass** */

•• $m = m + 1$

•• For $j = 1$ to n do /* for each processor j do */

Convert network G^m into a two-processor network with processor node p_j as source and the set $P - \{p_j\}$ as the sink (as described in Theorem 3.1) and apply the Max Flow/Min Cut Algorithm 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 tasks which remain unassigned. */

•• Let T^m denote the set of tasks which remain unassigned after m passes. 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

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

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. */

•• Recalculate the weight on each edge from t_i to p_q according to (1) with the new values of execution cost for all tasks in T^m.

/* **end of pass** */

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 the Appendix.

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.

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

[1] Theorem 3.1 as stated above was shown to be incorrect by Abraham and Davidson in [27]. 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." The corrected proof for Theorem 3.1 can be found in the Abraham and Davidson paper. 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.

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) 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 $T^m = \{t_1, t_2, \cdots, t_\gamma\}$ 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

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

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 $t_1, t_2, \cdots, t_\gamma$ for which $c_{ij} \leq C$.
- 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

$$\sum_{t_l \leftarrow G_i \cup G_j} x_{lq} < X \quad \text{then}$$

    - ••• Merge the two groups: $G = G_i \cup G_j$
    - ••• 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.

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 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 data sets can be found in [14].

The results of these simulations show Algorithm A to be very successful in finding suboptimal assignments. Fig. 4 summarizes this information by showing the distribution of the ratio 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 33.4 percent of the cases, and for data sets 1 and 2 Algorithm A found an optimal assignment in 46.9 percent and 47.3 percent of the cases, respectively. In 92.7 percent of the cases, the cost of the

Table I: Distribution of T_A/T_O by Dataset

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%

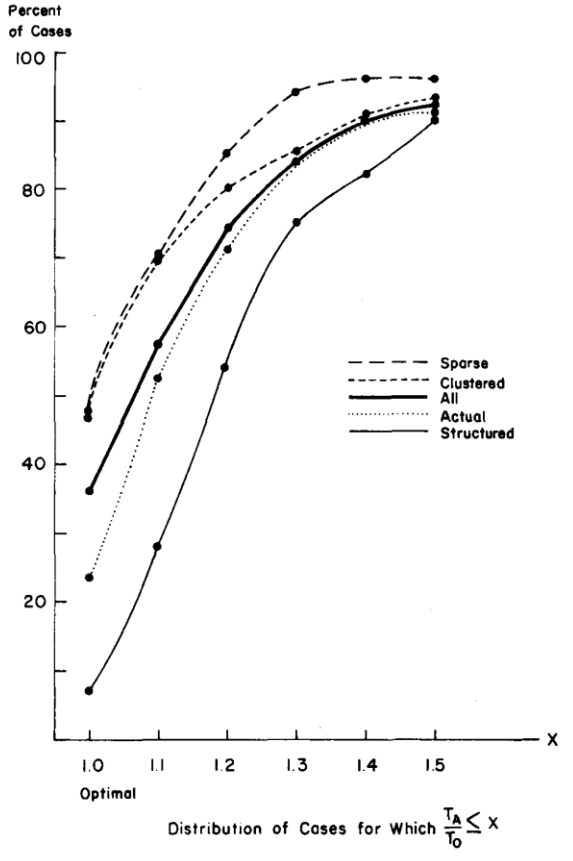


Fig. 4. Performance of Algorithm A for several data sets.

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 three cases of the 536 cases. The worst ratio T_A/T_O was 2.7. Algorithm A did not perform as well on the *actual* and *structured* data sets because Greedy presumes some clustering of tasks while the data sets did not exhibit this feature.

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. Fig. 5 compares the performance of the phases of Algorithm A to 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

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
Algorithm A	82	23.1%	62.2%	84.1%	90.2%	93.9%	96.3%
Simple Greedy	68	20.5%	52.8%	73.3%	77.7%	89.4%	93.8%
Complex Greedy	68	17.6%	55.8%	70.5%	82.2%	89.5%	93.9%
Sort Greedy	68	20.5%	54.3%	76.3%	83.6%	90.9%	92.3%

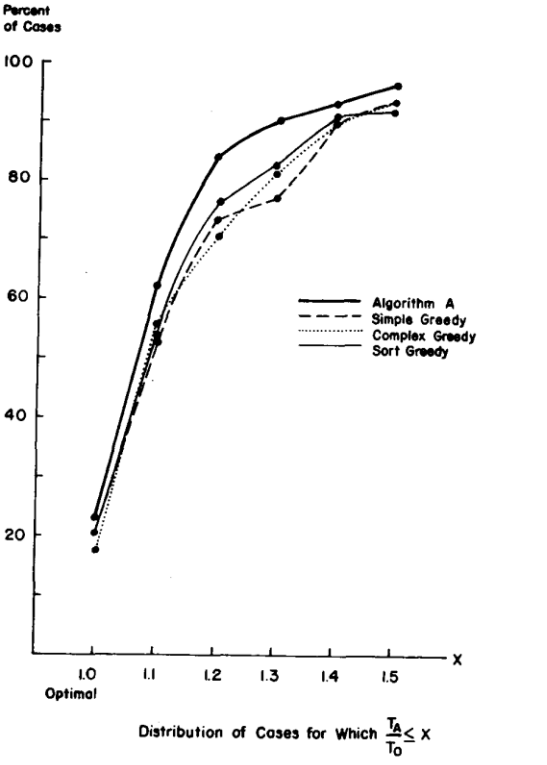


Fig. 5. Comparative performance of four algorithms.

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 nonincreasing 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 nonzero cost.

From Fig. 5, 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 percent of the cases while Simple Greedy found an optimal assignment in 20.5 percent 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 intercluster 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.

IV. Complexity 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 with 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 $k - 1$ min cuts in a network with task nodes only.

Simple Greedy: $O(e)$ or $O(nk^2)$
: If $X = \infty$, then Simple Greedy simply examines each communication edge. If $X < \infty$, Simple Greedy must check to see if there is a processor to which all the tasks in the groups to be merged can be assigned.

V. 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 byproduct 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 Fig. 6, 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_6 assigned to processor p_2). The assignment shown with dotted lines (t_1 through t_3 on p_1 and t_4 through t_6 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 a lot 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.

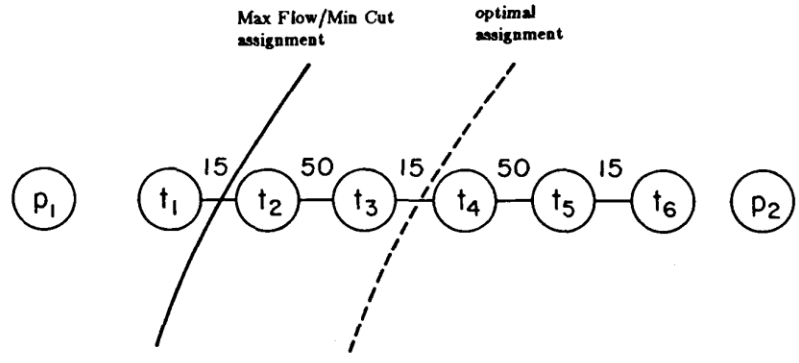
In particular, let $T = \{t_1, \cdots, t_k\}$ be the set of tasks, $P = \{p_1, \cdots, p_n\}$ be the set of processors, and let x_{ij}, $1 \le i \le k$, $1 \le j \le n$ and c_{ij}, $1 \le i, j \le k$ be the execution costs and communication costs, respectively, as defined before. Let $I_q(i, j)$, $1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

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

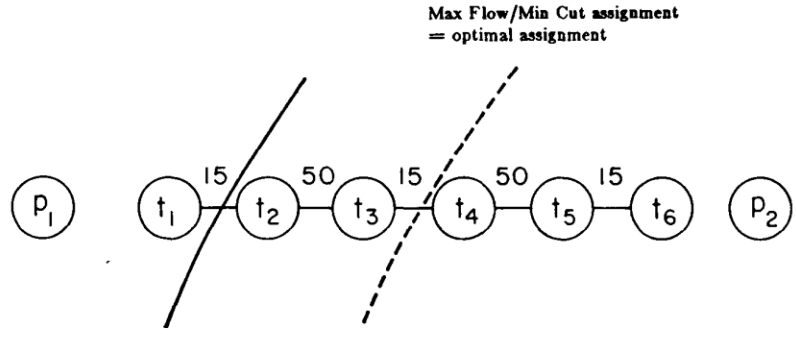


——— Cost $= x_{11} + x_{22} + x_{32} + x_{42} + x_{52} + x_{62} + c_{12} = 115$

– – – Cost $= x_{11} + x_{21} + x_{31} + x_{42} + x_{52} + x_{62} + x_{34} = 115$

(a)

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



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

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

(b)

Fig. 6. (a) Poor assignment without interference costs. (b) Better assignment with interference costs.

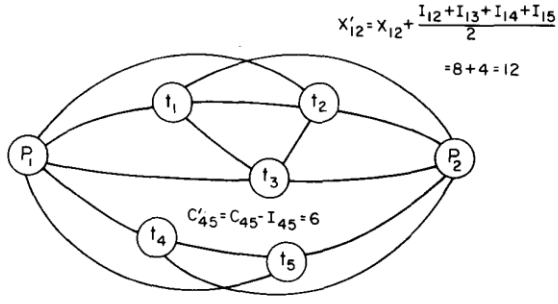


Fig. 7. Processor-independent interference costs.

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.

A. 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 Fig. 7.

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 the Appendix.

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 III 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 two-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.

B. 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 III. For each configuration, interference costs were generated from the uniform distributions over the intervals $[1, 2\bar{c}]$, $[1, (3/2)\bar{c}]$, $[1, (\bar{c}/2)]$, and $[1, (\bar{c}/10)]$, where $\bar{c}$ 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

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

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

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

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 Fig. 8 using a Gantt diagram. In this figure, the communication costs are depicted as occurring in one lump, but is should be kept in mind that these costs are actually dispersed in time throughout the execution of the tasks.

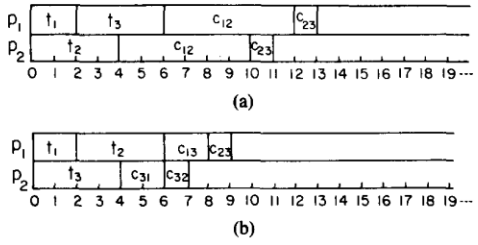


Fig. 8. Completion time (execution and communication costs). (a) One assignment $\omega_f = 13$. (b) Optimal assignment $\omega_f = 9$.

Distribution of ω_T/ω_O							
Model	Total No. of Simulations	Percent of Simulations					
		(Optimal) = 1.00	≤ 1.10	≤1.20	≤1.50	≤ 2.00	≤ 2.75
With I_{ij}	130	22.0%	61.0%	84.7%	98.3%	100.0%	100.0%
No I_{ij}	130	8.4%	10.1%	18.6%	49.1%	83.0%	100.0%

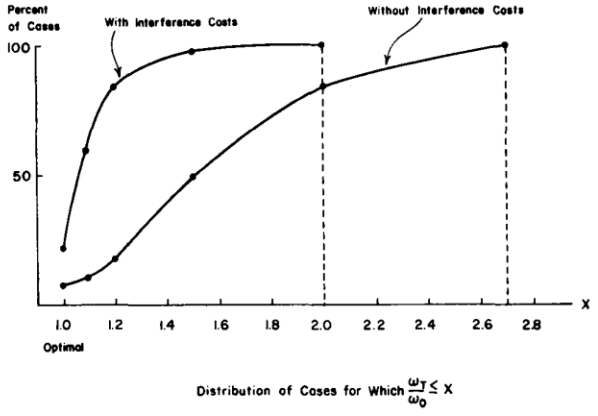


Fig. 9. Degree of concurrency attained by assignments that are optimal w.r.t. total costs.

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 ω_A/ω_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 percent greater than optimal, and so on. In each figure, results are presented both in table form and graphically.

Figs. 9 and 10 demonstrate empirically that addition of interference costs to the model yields assignments with a high

Distribution of ω_A/ω_O							
Model	Total No. of Simulations	Percent of Simulations					
		(Optimal) = 1.00	≤ 1.10	≤1.20	≤1.50	≤ 2.00	≤ 2.75
With I_{ij}	130	6.7%	18.6%	45.7%	93.2%	100.0%	100.0%
No I_{ij}	130	1.6%	5.0%	10.1%	42.3%	86.4%	100.0%

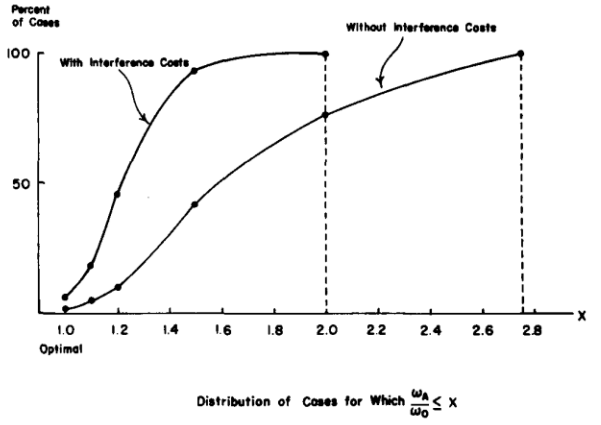


Fig. 10. Degree of concurrency attained by assignments that are suboptimal w.r.t. total costs.

degree of concurrency. Fig. 9 shows this is true for assignments which have optimal values for total costs while Fig. 10 shows this is also true for suboptimal assignments found by Algorithm A. Figs. 9 and 10 also compare the degree of concurrency attained in assignments for the interference cost model to 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.

Fig. 9 shows the distribution of the ratio ω_T/ω_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 ω_T/ω_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 Fig. 9, we see that assignments with an optimal value for total costs also have excellent values for completion time. For example, 22 percent of the assignments were also optimal with respect to completion time, 61.0 percent of the assignments are less than 1.1 times the optimal value, and 98.3 percent 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 ω_T/ω_O. For example, when interference costs are included, 98.3 percent 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 percent. While this difference is as expected, the magnitude of the difference is notable.

Fig. 10 shows the distribution of the ratio ω_A/ω_O for systems with interference costs and for systems without interference costs. Recall that ω_A is the completion time of an assignment

Distribution of T_A/T_O							
Model	Total No. of Simulations	Percent of Simulations					
		(Optimal) = 1.00	≤ 1.10	≤1.20	≤1.30	≤ 1.40	≤ 1.50
With I_{ij}	130	3.0%	12.3%	27.6%	46.1%	67.6%	78.4%
No I_{ij}	356	25.0%	50.8%	68.5%	80.3%	87.6%	91.3%

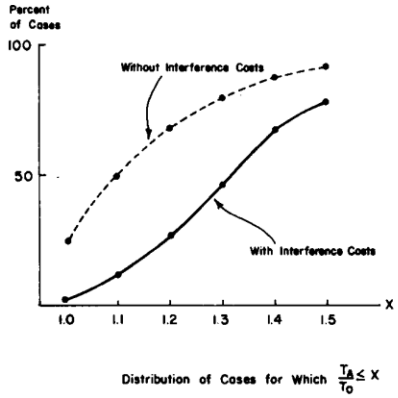


Fig. 11. Performance of Algorithm A on the interference cost model.

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 percent 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 percent of the assignments found by Algorithm A have a completion time that is less than 1.5 times the optimal value.

Fig. 11 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 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 percent of the cases. However, with interference costs, Algorithm A found an optimal assignment in only 3 percent of the cases. Similarly, the cost of 91.3 percent 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 percent 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 holds 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.

C. Arbitrary Interference Costs

In this section, we consider the general cost 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. We will show that 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

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

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 Fig. 12.

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.)

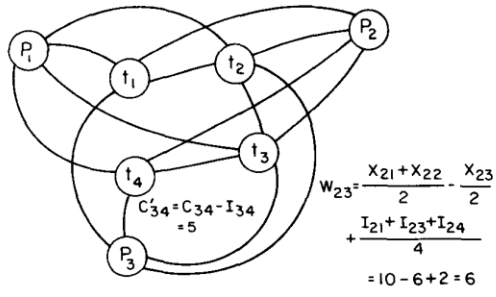


Fig. 12. Arbitrary interference costs.

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

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

VI. 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.

One 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 quantities.

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 are 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 management (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.

Appendix

Below are the proofs for Theorems 3.2 and 4.1.

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

Proof: We prove the theorem by induction on the number of passes m in Algorithm Grab.

a) Suppose Grab halts after one pass. In one pass of Algorithm Grab, the Max Flow/Min Cut Algorithm is applied for each processor node as described in Theorem 3.1 to assign tasks to each processor. Let S_i^1 be the set of tasks assigned to processor p_i in pass 1 of Grab, $i = 1, \cdots, n$. By Theorem 3.1, each of the n individual assignments which assigns S_i^1 to one processor p_i is a prefix of every optimal assignment. The assignment produced by pass 1 of Grab can be represented as the union of these n individual assignments and thus is also a prefix of every optimal assignment.

b) We now prove the theorem for $m + 1$ passes of Algorithm Grab. Let Grab be applied to network G^1. Suppose that after m passes it produces a partial assignment f^m represented by a partition Π^m of the tasks of G^1:

$$\Pi^m = \{S_1^m, \cdots, S_n^m, T^m\}$$

where S_i^m is the set of tasks assigned to processor p_i and T^m is the set of tasks which remain unassigned. By our induction hypothesis, we know that the (partial) assignment f^m forms a prefix of every optimal assignment for G^1. The cost of any optimal assignment f_{OPT} is

$$C(f_{\text{OPT}}) = \sum_{\substack{f_{\text{OPT}}(t_i)=p_q \\ t_i \in T}} x_{iq} + \sum_{\substack{f_{\text{OPT}}(t_i)\neq f_{\text{OPT}}(t_j) \\ t_i,t_j \in T}} c_{ij}$$

$$= \sum_{\substack{f_{\text{OPT}}(t_i)=p_q \\ t_i \in T-T^m}} x_{iq} + \sum_{\substack{f_{\text{OPT}}(t_i)=p_q \\ t_i \in T^m}} x_{iq} + \sum_{\substack{f_{\text{OPT}}(t_i)\neq f_{\text{OPT}}(t_j) \\ t_i,t_j \in T-T^m}} c_{ij}$$

$$+ \sum_{\substack{f_{\text{OPT}}(t_i)\neq f_{\text{OPT}}(t_j) \\ t_i,t_j \in T^m}} c_{ij} + \sum_{\substack{f_{\text{OPT}}(t_i)\neq f_{\text{OPT}}(t_j) \\ t_i \in T-T^m \\ t_j \in T^m}} c_{ij}.$$

By rearranging terms, we have

$$C(f_{\mathrm{OPT}}) = C(f^m) + C[T^m]$$

where

$$C(f^m) = \sum_{\substack{f_{\mathrm{OPT}}(t_i)=p_q \\ t_i \in T - T^m}} x_{iq} + \sum_{\substack{f_{\mathrm{OPT}}(t_i) \neq f_{\mathrm{OPT}}(t_j) \\ t_i, t_j \in T - T^m}} c_{ij}$$

$$C[T^m] = \sum_{\substack{f_{\mathrm{OPT}}(t_i)=p_q \\ t_i \in T^m}} x_{iq} + \sum_{\substack{f_{\mathrm{OPT}}(t_i) \neq f_{\mathrm{OPT}}(t_j) \\ t_i, t_j \in T^m}} c_{ij} + \sum_{\substack{f_{\mathrm{OPT}}(t_j) \neq f^m(t_i) \\ t_i \in T - T^m \\ t_j \in T^m}} c_{ij}.$$

In other words, the cost of any optimal assignment for G^1 can be broken down into two components: 1) the cost $C(f^m)$ of first assigning tasks in $T - T^m$ according to m passes of Grab and 2) the cost $C[T^m]$ of *subsequently* assigning tasks in T^m. Since tasks in $T - T^m$ must be assigned according to f^m in every optimal assignment for G^1, $C(f^m)$ is fixed and thus an optimal assignment for G^1 is one which minimizes $C[T^m]$.

Now consider the network G^{m+1} constructed for the $(m+1)$st pass of Grab. By definition of Grab, $m+1$ passes on G^1 can be viewed as m passes on G^1 followed by one pass on G^{m+1}. Thus, in order to prove that the (partial) assignment resulting from $m+1$ passes on G^1 is a prefix of all optimal assignments for G^1, we need to show that the assignment produced by one pass on G^{m+1} is a prefix of all assignments of tasks in T^m which minimize $C[T^m]$. By construction of G^{m+1}, we know that an optimal assignment $f_{\mathrm{OPT}'}$ for G^{m+1} minimizes $C[T^m]$ as shown below:

$$C(f_{\mathrm{OPT}'}) = \sum_{\substack{f_{\mathrm{OPT}'}(t_i)=p_q \\ t_i \in T^m}} x_{iq}^{m+1} + \sum_{\substack{f_{\mathrm{OPT}'}(t_i) \neq f_{\mathrm{OPT}'}(t_j) \\ t_i \in T^m}} c_{ij}$$

which by (2)

$$= \sum_{\substack{f_{\mathrm{OPT}'}(t_i)=p_q \\ t_i \in T^m}} x_{iq} + \sum_{\substack{f_{\mathrm{OPT}'}(t_j) \neq f^m(t_i) \\ t_i \in T - T^m \\ t_j \in T^m}} c_{ij} + \sum_{\substack{f_{\mathrm{OPT}'}(t_i) \neq f_{\mathrm{OPT}'}(t_j) \\ t_i \in T^m}} c_{ij}$$

$$= c[T^m].$$

Also, by our base induction hypothesis a), the (partial) assignment produced by one pass of Grab on G^{m+1} is a prefix of all optimal assignments $f_{\mathrm{OPT}'}$ for G^{m+1}. Thus, the (partial) assignment produced by m passes on G^1 followed by one pass on G^{m+1} is a prefix of all optimal assignments for G^1. Or equivalently, the (partial) assignment produced by $m+1$ passes on G^1 is a prefix of all optimal assignments for G^1. Q.E.D.

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.)

Proof: For a given assignment, if two tasks, t_i and t_j, are assigned to the same processor, say p_1, then tasks nodes t_i and t_j are cut off from all processor nodes other than p_1. Thus, the $n-1$ edges from t_i to the processor nodes other than p_1 are cut, incurring a cost of

$$\sum_{r \neq 1} w_{ir} = x_{i1} + (n-1) \times \frac{1}{2(n-1)} \sum_{1 \leq l \leq k} I_{il}.$$

Similarly, the $n-1$ edges from t_j to the processor nodes other than p_1 are cut, incurring a cost of

$$\sum_{r \neq 1} w_{jr} = x_{j1} + (n-1) \times \frac{1}{2(n-1)} \sum_{1 \leq l \leq k} I_{jl}.$$

Considering only the interference costs between t_i and t_j, we see that the combined cost is equal to

$$x_{i1} + x_{j1} + I_{ij}.$$

Hence, when two tasks are assigned to the same processor, the incurred cost is equal to the sum of execution and interference costs.

If t_i and t_j are assigned to different processors, e.g., t_i to p_1 and t_j to p_2, the edges from t_i to all processor nodes other than p_1 are cut, and the edges from t_j to all processor nodes other than p_2 are cut. Moreover, the edges between t_i and t_j are cut, incurring the costs

$$\sum_{r \neq 1} w_{ir} = x_{i1} + \frac{1}{2} \sum_{1 \leq l \leq k} I_{il}$$

$$\sum_{r \neq 2} w_{jr} = x_{j2} + \frac{1}{2} \sum_{1 \leq l \leq k} I_{jl}$$

and

$$c_{ij} - I_{ij},$$

respectively. Again, considering only the interference costs between t_i and t_j, we see that the incurred cost is equal to

$$x_{i1} + \frac{I_{ij}}{2} + x_{j2} + \frac{I_{ij}}{2} + c_{ij} - I_{ij} = x_{i1} + x_{j2} + c_{ij}.$$

Thus, when two tasks are assigned to different processors, the total cost incurred is equal to the sum of execution and communication costs but no interference costs. This reasoning may be extended to all possible pairs of tasks. Hence, the cost of an n-way cut is equal to the total sum of execution, communication, and interference costs incurred by the corresponding assignment. The cost of an assignment f is given by

$$
\begin{aligned}
c_f &= \sum_{q=1}^{n} \left(\sum_{f(t_i)=p_q} \sum_{r \neq q} w_{ir} \right) + \sum_{f(t_i) \neq f(t_j)} c'_{ij} \\
&= \sum_{q=1}^{n} \sum_{f(t_i)=p_q} \left(\sum_{r \neq q} \left(\frac{1}{n-1} \sum_{s \neq r} x_{is} - \frac{n-2}{n-1} \; x_{ir} \right. \right. \\
&\quad \left. \left. + \frac{1}{2(n-1)} \sum_{1 \le j \le k} I_{ij} \right) \right) + \sum_{f(t_i) \neq f(t_j)} c'_{ij} \\
&= \sum_{q=1}^{n} \sum_{f(t_i)=p_q} \left(x_{iq} + \sum_{r \neq q} \frac{1}{2(n-1)} \sum_{1 \le j \le k} I_{ij} \right) + \sum_{f(t_i) \neq f(t_j)} c'_{ij} \\
&= \sum_{q=1}^{n} \sum_{f(t_i)=p_q} \left(x_{iq} + (n-1) \times \frac{1}{2(n-1)} \sum_{1 \le j \le k} I_{ij} \right) \\
&\quad + \sum_{f(t_i) \neq f(t_j)} c'_{ij} \\
&= \sum_{q=1}^{n} \sum_{f(t_i)=p_q} \left(x_{iq} + \sum_{1 \le j \le k} \frac{I_{ij}}{2} \right) + \sum_{f(t_i) \neq f(t_j)} c'_{ij} \\
&= \sum_{q=1}^{n} \sum_{f(t_i)=p_q} x_{iq} + \sum_{q=1}^{n} \sum_{f(t_i)=p_q} \sum_{1 \le j \le k} \frac{I_{ij}}{2} \\
&\quad + \left(\sum_{f(t_i) \neq f(t_j)} c_{ij} - \sum_{f(t_i) \neq f(t_j)} I_{ij} \right) \\
&= \sum_{q=1}^{n} \sum_{f(t_i)=p_q} x_{iq} + \sum_{f(t_i) \neq f(t_j)} c_{ij} + 2 \times \left(\sum_{f(t_i)=f(t_j)} \frac{I_{ij}}{2} \right) \\
&\quad + 2 \times \left(\sum_{f(t_i) \neq f(t_j)} \frac{I_{ij}}{2} \right) - \sum_{f(t_i) \neq f(t_j)} I_{ij} \\
&= \sum_{q=1}^{n} \sum_{f(t_i)=p_q} x_{iq} + \sum_{f(t_i) \neq f(t_j)} c_{ij} + \sum_{q=1}^{n} \sum_{f(t_i)=f(t_j)=p_q} I_{ij} \\
&= \text{total execution, communication, and interference costs incurred by } f. \qquad \text{Q.E.D.}
\end{aligned}
$$

References

[1] S. H. Bokhari, "Dual processor scheduling with dynamic reassignment," *IEEE Trans. Software Eng.*, vol. SE-5, pp. 341–349, July 1979.

[2] ——, "Optimal assignments in dual-processor distributed systems under varying load conditions," ICASE Rep. 79–14, July 1979.

[3] ——, "A shortest tree algorithm for optimal assignments across space and time in a distributed processor system," *IEEE Trans. Software Eng.*, vol. SE-7, pp. 583–589, Nov. 1981.

[4] Y. C. Chow and W. H. Kohler, "Models of dynamic load balancing in a heterogeneous multiple processor system," *IEEE Trans. Comput.*, pp. 354–361, May 1979.

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

[6] E. G. Coffman and P. J. Denning, *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[7] D. H. Cornett and M. A. Franklin, "Scheduling independent tasks with communications," Washington Univ., Dept. Elec. Eng., Tech. Rep., 1979.

[8] K. Efe, "Heuristic models of task assignment scheduling in distributed systems," *IEEE Computer*, pp. 50–56, June 1982.

[9] O. J. El-Dessouki, "Program partitioning and load balancing in network computers," Ph.D. dissertation, Illinois Instit. Technol., Dec. 1978.

[10] Z. Galil, S. Micali, and H. Gabow, "Priority queues with variable priority and an $O(EV \log V)$ algorithm for finding a maximal weighted matching in general graphs," in *Proc. 23rd Annu. Symp. Foundations Comput. Sci.*, Nov. 3–5, 1982, pp. 255–261.

[11] M. Gursky, "Some complexity results for a multi-processor scheduling problem," private communication from H. S. Stone, 1981.

[12] V. B. Gylys and J. A. Edwards, "Optimal partitioning of workload for distributed systems," in *Dig. Papers COMPCON*, Fall 1976, pp. 353–357.

[13] V. M. Lo, "Task assignment in distributed multiprocessor systems," in *Proc. IEEE Conf. Parallel Processing*, 1981, pp. 358–360.

[14] ——, "Task assignment in distributed systems," Ph.D. dissertation, Dep. Comput. Sci., Univ. Illinois, Oct. 1983.

[15] ——, "Heuristic algorithms for task assignment in distributed systems," in *Proc. IEEE 4th Int. Conf. Distributed Comput. Syst.*

[16] L. M. Ni and K. Hwang, "Optimal load balancing strategies for a multiple processor operating system," in *Proc. IEEE Conf. Parallel Processing*, 1981, pp. 352–357.

[17] C. C. Price, "Search techniques for a nonlinear multiprocessor scheduling problem," *Naval Res. Logist. Quarterly*, June 1982, pp. 213–233.

[18] C. C. Price and S. Krishnaprasad, "Software allocation models for distributed computing systems," in *Proc. 4th Int. Conf. Distributed Comput. Syst.*, May 1984, pp. 40–48.

[19] C. C. Shen and W. H. Tsai, "A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion," *IEEE Trans. Comput.*, vol. C-34, pp. 197–203, Mar. 1985.

[20] R. G. Smith, "The contract bid protocol: High-level communication and control in a distributed problem-solver," *IEEE Trans. Comput.*, vol. C-29, pp. 1104–1113, Dec. 1980.

[21] H. S. Stone and S. H. Bokhari, "Control of distributed processes," *IEEE Computer.*, pp. 97–106, July 1978.

[22] H. S. Stone, "Multiprocessor scheduling with the aid of network flow algorithms," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 85–93, Jan. 1977.

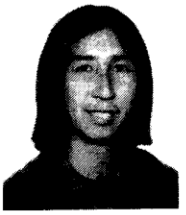
[23] ——, "Program assignment in three-processor systems and tricutset partitioning of graphs," Tech. Rep. ECE-CS-77-7, Dep. Elec. Eng., Univ. Massachusetts, Amherst, 1977.

[24] ——, "Critical load factors in two-processors distributed systems," *IEEE Trans. Software Eng.*, vol. SE-4, pp. 254–258, May 1978.

[25] A. M. Van Tilborg and L. D. Wittie, "Wave scheduling—Decentralized scheduling of task forces in multicomputers," *IEEE Trans. Comput.*, vol. C-33, pp. 835–844, Sept. 1984.

[26] C. S. Wu and M. T. Liu, "Assignment of tasks and resources for distributed processing," in *IEEE COMPCON Fall 1980, Proc. Distributed Processing*, pp. 655–662.

[27] S. G. Abraham and E. S. Davidson, "Task assignment using network flow methods for minimizing communication in n-processor systems," Center for Supercomput. Res. Develop., Tech. Rep. 598, Sept. 1986.



Virginia Mary Lo (A'85) received the A.B. degree from the University of Michigan, Ann Arbor, in 1969, the M.S. degree in computer science from the Pennsylvania State University, University Park, in 1978, and the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 1983.

She has been on the faculty of the Department of Computer and Information Science at the University of Oregon since January 1985. Her research interests include scheduling and load balancing in distributed and real-time distributed systems.

Dr. Lo is a member of the Association for Computing Machinery and the IEEE Computer Society.