Scheduling Parallel I/O Operations[1]

Ravi Jain[2]
Bellcore

Kiran Somalwar
Digital Equipment Corp.

John Werth and J. C. Browne
Dept. of Computer Sciences, Univ. of Texas at Austin

Abstract

The I/O bottleneck in parallel computer systems has recently begun receiving increasing interest. Most attention has focused on improving the performance of I/O devices using fairly *low-level* parallelism in techniques such as disk striping and interleaving. Widely applicable solutions, however, will require an integrated approach which addresses the problem at multiple system levels, including applications, systems software, and architecture. We propose that within the context of such an integrated approach, scheduling parallel I/O operations will become increasingly attractive and can potentially provide substantial performance benefits.

We describe a simple I/O scheduling problem and present approximate algorithms for its solution. The costs of using these algorithms in terms of execution time, and the benefits in terms of reduced time to complete a batch of I/O operations, are compared with the situations in which no scheduling is used, and in which an optimal scheduling algorithm is used. The comparison is performed both theoretically and experimentally. We have found that, in exchange for a small execution time overhead, the approximate scheduling algorithms can provide substantial improvements in I/O completion times.

[1] This research was performed while the first two authors were at the University of Texas, and was partially supported by the IBM Corporation through grant 61653 and by the State of Texas through TATP Project 003658-237.

[2] Address correspondence to Ravi Jain, Applied Research, Bellcore, 445 South St, Morristown, NJ 07960. e-mail: rjain@thumper.bellcore.com

1 Introduction

The performance of parallel computers for many interesting classes of applications is often limited by the speed of I/O rather than the speed of computation. Technology and application trends indicate that this I/O bottleneck is likely to become increasingly important in the future [5, 7, and references therein]. While this problem has been receiving increasing attention in the past few years, most solutions have focused on improving the performance of a few components of the parallel computer system. Typically, attention has been paid to improving the performance of I/O devices using fairly *low-level* parallelism in techniques such as disk striping and interleaving [15, 5, and references therein]. It seems likely, however, that, just as is the case for sequential computers, widely applicable solutions to the I/O bottleneck in parallel systems will require an integrated approach which addresses the problem at all levels of the system, including applications, systems software, and architectures.

Explicit scheduling of parallel I/O operations is a potentially significant contributor to an integrated approach towards solving the the I/O bottleneck. Scheduling becomes increasingly attractive as the I/O bottleneck becomes more severe: the processing overhead for generating good schedules decreases while the importance of performing data transfers efficiently increases. However, scheduling parallel I/O operations has received relatively little attention.

Much of the previous work on scheduling deals with tasks which each require only a single resource at any given time [7], and is not relevant for I/O operations which each require multiple resources (e.g. a processor, channel, and disk) simultaneously in order to execute. Serial acquisition of multiple resource does not, in general, lead to optimal schedules; algorithms which simultaneously schedule multiple resources are required. Previous work on simultaneous resource scheduling has either considered very general resource requirements, leading to problems known to be NP-complete or requiring linear programming solutions of high time complexity, or made assumptions which are not relevant for scheduling parallel I/O operations (see [7] for a survey). In contrast, we seek to exploit the special structure and requirements of parallel I/O tasks to obtain polynomial-time algorithms

and simple heuristics which are effective for our application. In previous work we have developed a family of such algorithms for scheduling data transfers under various architectural and logical constraints in the context of a general model for specifying scheduling problems [12, 8, 7].

In section 2, we sketch the role of parallel I/O scheduling and describe a simple parallel I/O scheduling problem and a scenario in which it may be likely to arise. We have previously designed algorithms which provide optimal solutions to this problem [8]. Here we present fast algorithms which provide approximate solutions to the problem, and examine their performance both theoretically and experimentally. We compare the lengths of the schedules produced by these algorithms to those produced when scheduling is not employed, i.e., those in which I/O operations are performed on a first-come first-served basis. Our experimental results indicate that the approximate scheduling algorithms can potentially provide significant reductions in the maximum time to complete a batch of transfers, while paying only a small overhead to generate the schedule. In sec. 6 we end with some conclusions.

Figure 1: Parallel I/O scheduling example

2 The role of scheduling

In this section we discuss the role of scheduling and describe an application in which a parallel I/O scheduling problem may arise. We show that the problem can be formulated naturally as a problem of coloring the edges of a graph. We then state some theoretical results which characterize the solutions to these problems.

The performance bottleneck created by the delays in the movements of mechanical devices is an old problem in sequential computer system design, and has motivated many significant innovations, including the introduction of memory hierarchies, multiprogramming, and pipelining via buffering [17]. One important innovation was to schedule the I/O operations by reordering the requests in the queues at devices [4, 17, and references therein]. We propose that a scheduling approach be applied to I/O operations in parallel computer systems also, and in this paper consider centralized batch scheduling. In [10, 9] we discuss the structure of applications and I/O request streams for which batch scheduling is applicable, and the capabilities required in the system software and architecture to make it feasible. We observe that these capabilities are also required in order to address the I/O bottleneck at higher levels of the system. The following simple example illustrates the potential benefits of parallel I/O scheduling.

Example. Consider the I/O operations T1 - T4 which are required between a set of processors and a set of disks (Fig. 1). The I/O transfer requests are numbered in the order they arrived at the system. Assume each processor and each disk can participate in at most one data transfer at any given time, and each transfer is of unit length. In addition, we are given that the transfers may occur in any order. Clearly the minimum number of time slots required to complete the transfers, corresponding to an optimal schedule, is less than that required if the transfer requests are simply serviced in the order they arrive.

2.1 Application: image visualization

Consider a scenario where users, who may be physicians, health care workers, scientists, etc., need to share and access a large image database. The images may consist of medical information (e.g. Computer-Aided Tomography scans), or oil prospecting information (e.g. seismic data from acoustical depth soundings) etc. The database is processed and stored at a parallel computer site, and users view the images by requesting image files to be displayed on their graphics workstations. The parallel computer is a shared-bus system, in which processors and disks are connected to a set of common buses (or a single high-speed system bus that is shared in a time-multiplexed fashion), which allow multiple I/O transfers to proceed in parallel. In order to provide a reasonable response time for the users, the workstations are also connected to the common buses, possibly via a transparent high-speed interface card. A user request for an image file is processed by the CPUs at the parallel computer, and results in image data being transferred from the system disks to the user's workstation across the common buses.

In this scenario, the parallel computer's operating system batches the image file requests and schedules the resulting I/O transfers. Files are read from disks in fixed-size blocks, and the blocks may be sent in any order to the workstations, which are capable of assembling them and generating the image. The workstations also off-load some low-level image-processing tasks from the parallel computer, such as shading, etc.

2.2 Problem statement

The specific I/O problem we will consider can now be stated, as a decision problem, as follows. Since, in principle, the problem may arise between any two levels of the memory hierarchy, it is stated in terms of processors and I/O devices. Given a set of I/O transfers, where

1. all transfers are of the same length,
2. each transfer requires a specified pair of resources, one a processor and the other an I/O device, from two given sets of processors and I/O devices,
3. each processor can communicate via a direct dedicated link with each I/O device, and
4. the transfers may occur in any order,

is there a preemptive schedule for performing the transfers whose total length is at most some given bound?

We call this problem the Unit-Length Simple I/O Scheduling (*UnitSimpleIOS*) problem. (When the problem is stated as an optimization problem the objective is to minimize the schedule length). It is applicable to systems such as the Sequent [13] where I/O devices are connected to processors via a single shared bus.

In this paper we also consider an extension to *UnitSimpleIOS* that is useful for modeling some practical parallel I/O architectures. We call this problem *UnitIOS*, and it is identical to *UnitSimpleIOS* except that the system architecture differs: only a fixed limited number k of I/O operations may take place at any given time. The *UnitIOS* problem arises in multiple-bus systems such as the IBM RP3, where k parallel buses connect processor and I/O devices [16]. In general, such multiple parallel bus architectures are attractive for future high-performance parallel computers as they not only allow more than one data transfer to be in progress at any given time, but also allow more processors and devices to be interconnected and improve the system's fault tolerance [14]. In general, *UnitIOS* may also arise in a shared-bus architecture where the effective bandwidth of the bus is a limiting factor.

2.3 Definitions and problem formulation

Def. An *edge coloring* of a graph $G = (V, E)$ is a function $c : E \rightarrow \{0, 1, 2, ...\}$ which associates a color with each edge such that no two edges of the same color have a common vertex.

Consider a collection of vertices representing processors and I/O devices, each of which can participate in at most one data transfer at any given time. Then an edge coloring for a graph G, where each edge of G represents a data transfer requiring one time unit, corresponds to a schedule for the data transfers, and vice versa. To see this, note that all

edges of G colored with the same color are independent in that they have no common vertex. Hence the data transfers they represent can be performed simultaneously. An edge coloring of G represents a schedule, where all edges e with $c(e) = i$, for some i, represent data transfers that take place at time i, and vice versa. The number of colors required to edge-color G equals the length of the schedule, and vice versa. To model the *UnitIOS* problem, we add the restriction that each color can be used to color at most k edges. An instance of the simple I/O scheduling problem is given in Fig. 1.

Notation. Let $G = (A, B, E)$ denote a bipartite graph where A and B are two disjoint sets of vertices and $E \subseteq A \times B$ is the set of edges. Let $|A| + |B| = n$ and $|E| = m$.

We have been able to use this more abstract formulation of the scheduling problem to show, using a general model for specifying scheduling problems [18, 12, 7], that *UnitIOS* is identical to the problem of obtaining optimal time-slot assignments in a satellite switching system [3].

We state some definitions and known results which we will draw upon, sketching proofs where they provide intuition for results later in the paper.

Def. The *degree* of a vertex is the number of edges incident upon it. The degree of a graph is the maximum of the degrees of its vertices. A *critical vertex* is one of maximum degree.

Def. A *matching* $M \subseteq E$ is a set of edges such that no two edges have a common vertex. A *maximum matching* is one such that no other matching has a larger cardinality. An edge in a matching is said to *cover* the vertices that are its endpoints.

Def. A *critical matching* is one which covers all critical vertices.

Lemma 2.1 *[2] Every bipartite graph has a critical matching.*

Lemma 2.2 *Exactly d colors are necessary and sufficient to color a bipartite graph of degree d.*

proof. [2] Clearly, at least d colors are necessary, since a critical vertex requires that each incident edge have a different color. The proof of sufficiency is by induction, sketched as follows. Find a critical matching M, which, from Lemma 2.1, must exist. Color the edges in M a single color and delete them from the graph. The remaining graph has degree $d - 1$, and by the induction hypothesis can be colored using $d - 1$ colors. Hence the graph can be colored with d colors. □

Def. A *k-coloring* of a graph is an edge-coloring in which each color may be used to color at most k edges.

Lemma 2.3 *[3] At least $p = \max(d, \lceil m/k \rceil)$ colors are necessary to k-color a bipartite graph with m edges and degree d.*

3 Heuristics for scheduling parallel I/O

Algorithms for constructing minimum length schedules for *UnitIOS* have been designed. These are the algorithm **KT** [3] which takes time $O(mn(m + n))$, Somalwar's algorithm [18] which takes time $O(mn^{1.5} \log n)$, and algorithm **A2** [8] which takes time $O(mn^{0.5} \log n)$. For a survey of optimal algorithms, and an experimental evaluation of the performance of these algorithms, see Jain [7].

While the optimal algorithm **A2** is faster than previous optimal algorithms we would like to have even faster algorithms, since in most applications scheduling algorithms are executed repeatedly, and any gain in speed helps overall system performance. Thus we consider heuristics for the *UnitIOS* problem, which are essentially approximation algorithms for k-coloring the edges of a bipartite graph, and present some theoretical results.

3.1 Greedy Heuristics for *UnitSimpleIOS*

An algorithm is called "greedy" if, for each color, it attempts to color as many edges as possible with that color; it is a heuristic, or approximation algorithm, if it is not guaranteed to use the minimum number of colors. Since we will be presenting several greedy heuristics, we establish a template for describing them using pseudo-code as follows.

Algorithm Greedy Heuristic
Input: Bipartite graph $G = (A, B, E)$
Output: An edge-coloring of G

```
1.  F := Order(A, B, E);
        /* F is some ordering of the edges in E */
2.  i := 0;
3.  while F ≠ ⟨ ⟩{
4.      M := { };
        /* Edges which are assigned color i */
5.      for each e read in sequence from F {
6.          if neither endpoint of e is colored i {
7.              color(e) := i;
8.              E := E − {e};
9.              M := M ∪ {e};
10.         }
11.     }
12.     i := i + 1;
13.     F := Order(A, B, E);
        /* Re-order remaining edges of the graph */
14. }
```

We assume that when the algorithm is called, the transfer requests have been added to the edge list of a bipartite graph in the order in which the requests arrived.

The simplest heuristic is the greedy *First-Come First-Served* heuristic **FCFS**, which, for each color, examines the edges in the order they are presented and tries to color as many edges as possible. For **FCFS**, the *Order*() function is thus simply the identity function, i.e., edges are processed in the order that they arrived at the operating system.

The heuristics we will investigate can be defined, assuming the template above, as

1. First-Come First-Served, **FCFS**. *Order*() is the identity function.
2. Highest Degree First, **HDF**. *Order*() sorts the vertices by descending degree, and for each vertex in turn, chooses edges incident upon it arbitrarily. Ties between vertices are broken arbitrarily.
3. Highest Combined Degree First, **HCDF**. *Order*() sorts the edges in descending order of the sum of the degrees of their endpoints. Ties between edges are broken arbitrarily.

We can now state some theoretical results which apply to all the greedy heuristics above, omitting proofs for brevity (see [11]).

Lemma 3.1 *[7, 1] For a bipartite graph of degree d, a greedy heuristic produces a coloring using at most $2d - 1$ colors.*

Lemma 3.2 *For a bipartite graph with m edges and degree d, the greedy algorithms take time $O(md)$ to solve UnitSimpleIOS, and produce a schedule of length at most $2 - \frac{1}{d}$ times the optimal length.*

It can be shown that the upper bound on the schedule length is "tight" for the heuristics above i.e., for any given d, it is always possible to construct degree d graphs for which, if the heuristic makes the "worst" choices, (or *Order*(G) produces a worst-possible ordering), the heurisitic uses exactly $2d - 1$ colors. For **FCFS**, this construction [7, 1] is relatively straightforward. For **HDF** and **HCDF** it is quite involved.

Lemma 3.3 *[11] For both the* **HDF** *and* **HCDF** *algorithms, for any positive integer d there exists a bipartite graph G of degree d and a sequence of choices made by the algorithm, such that exactly $2d-1$ colors are used to edge-color G.*

From the theoretical results above we see that in the worst case, processing I/O requests in the order in which they appear (i.e., **FCFS**), produces schedule lengths which are no longer than those produced by the heuristics (**HDF** and **HCDF**) which actually perform some intelligent scheduling. However, an experimental evaluation of these algorithms shows that the heuristics are preferable to **FCFS** as they exhibit far less variability in the schedule lengths produced. We will present experimental results for the heuristics as well as a divide-and-conquer optimal algorithm in section 4.

3.2 Greedy Heuristics for *UnitIOS*

The program for implementing the greedy heuristics for when a color may be used to color at most $k \leq n$ edges is a slight modification of the program given above. We add the following line (the **break** statement exits the smallest enclosing loop, i.e., the **for** loop):

```
10.5    if |M| = k, break;
```

That is, after every edge is added to M, the program checks if $|M| = k$, and if so, does not add any more edges to M for the current color. We call the resulting algorithms *modified* greedy algorithms, **MFCFS**, **MHDF**, **MHCDF**.

Lemma 3.4 *[11] For a graph of n vertices, m edges and degree d, if at most $k \leq n$ edges may be colored with a single color, a modified greedy algorithm produces a coloring using at most $\lfloor m/k \rfloor + (2d-1)$ colors, and run in time $O(m^2/k + md)$.*

4 Experimental evaluation

We have experimentally evaluated the performance of the greedy algorithms on instances of the *UnitSimpleIOS* and *UnitIOS* problems. We compare their behavior to that of the exact algorithm **A**, which is a special case of **A2** [8] for unit-weight edges implemented by Somalwar [18]. We also compare their behavior to the theoretical bounds discussed in the previous section.

The algorithms were implemented as C programs and evaluated for two criteria: the CPU time taken to produce a schedule, and the length of the schedule produced. Several experiments were carried out in which the the total number of data sources and sinks n, the number of transfers m, and the maximum number of simultaneous transfers k were varied. For brevity we describe here the results for only one experiment; results for the other experiments are qualitatively similar and can be found in [10].

The algorithms were compiled using the DEC C compiler for Ultrix on RISC, Release V1.0 with all optimizations enabled. The programs were executed on a DECstation 5000/200 workstation in single-user mode running the Ultrix Release 4.2 (Rev. 96) operating system. Each program and its data structures occupied less than 3 MB of the 32 MB main memory of the system, and so the programs did not perform any I/O during execution except to read input graph files and write result files. The CPU time was measured using the *gettimeofday()* Ultrix system call.

In the following we report the results for the modified approximation algorithms in terms of the *UnitIOS* problem only; the results for *UnitSimpleIOS* are similar or are subsumed by these results.

Expt. 1. Effect of varying number of transfers, m. In this experiment, the number of disks and workstations was fixed at $n = 32$ (i.e., 16 vertices in each partition), and it was assumed that the I/O channel transfer capacity was relatively low ($k = 4$). One hundred input graphs were generated for each value of m. For each set of input graphs corresponding to a given value of m, the CPU time required to calculate a schedule for each graph was recorded, and the mean over all graphs in a set was calculated. In Fig. 2 the mean CPU time (in milliseconds) taken by each of the algorithms is plotted as a function of m. The standard deviations of the CPU time are typically around 2-10% of the mean, and are not shown on the plot for clarity. It is clear that the heuristics run much faster than the optimal algorithm **A**.

The improvement in CPU time for the approximation algorithms comes at the expense of worse schedules. To evaluate this, for each input graph, the percentage increase in the schedule length over the optimum schedule length for that graph was calculated. The mean and maximum values of this percentage increase, over all one hundred input instances, were then calculated. We observed that the mean schedule length produced by the approximation algorithms does not differ significantly from the optimal or from each other. However, the *maximum* schedule length may. In Fig. 3 the maximum percentage increase is plotted as a function of m. The schedule produced by **MFCFS** can be almost 40% longer than optimal. On the other hand, the schedules produced by **MHDF** and **MHCDF** were found to be always of minimum length, for this experiment.

In Expt. 1 the execution time of all the algorithms increases with the number of transfers to be scheduled (Fig. 2). As expected, this increase is close to linear with m. It is interesting to see that **MHDF**'s execution time is slightly less than **MFCFS**. This is because of two effects. Firstly, the schedule lengths **MHDF** produces are shorter than those produced by **FCFS**, so that the number of iterations of the **while** loop is smaller. Secondly, the implementation of **MHDF** takes advantage of the fact that only the vertices are sorted, and edges in-

cident upon a selected vertex are chosen arbitrarily, so that it suffices to implement F as a list of vertices rather than edges.

5 Previous related work

The only relevant previous work, to our knowledge, deals with approximate edge-coloring of general graphs and multigraphs. An algorithm using no more than $(4/3)d$ colors, and running in time $O(m(n+d))$ was developed [6]. This algorithm uses an "interchange approach" as its basis: for each edge, check if some "simple" recoloring of the colored edges would eliminate the need for an additional color. As the authors say, "in order to prove better bounds, the 'simple' recolorings become more complicated" [6]. Note that their algorithm does not consider the practical constraint of a limited number of simultaneous data transfers, i.e., $k \leq n$. For our applications, this constraint corresponds to the realistic situation of limited bus bandwidth being available in the system. Also, since the **MHDF** and **MHCDF** heuristics do not perform any backtracking or recoloring, they are likely to have smaller constants for their time complexity, and are likely to be simpler to implement, than the interchange heuristic. Their performance for our set of experiments also seems satisfactory considering their simplicity.

Figure 2: CPU time (ms) versus number of transfers for $n = 32$, $k = 4$

Note that **MFCFS**, **MHDF**, **MHCDF** and the interchange heuristic form a sequence of heuristics which tend to produce shorter schedules at the expense of increasing time complexity. One can envision a range of heuristics that could be systematically designed to cover the gap between **MFCFS** and the optimal algorithm.

Figure 3: Percentage increase in schedule length versus number of transfers for $n = 32$, $k = 4$

6 Conclusions

In this paper we have defined a simple parallel I/O scheduling problem which may arise in practice. For this problem, we compared the costs and benefits of using simple approximate batch scheduling algorithms with the situation in which no scheduling is employed, and found that, in our investigation, scheduling can provide significant improvements in the

times to complete I/O transfers for a small execution overhead.

In our experiments we found that the simple greedy **MHDF** and **MHCDF** approximation algorithms produced schedules at most 10% longer than optimal, for execution times which are close to the execution times of the **MFCFS** algorithm. On the other hand, while the mean length increase in schedules produced by **MFCFS** may not be very large, the maximum may be 50% over the optimal schedule. More importantly, the dependence of **MFCFS** on various input parameters, especially k, is quite large and hard to predict.

Our work on scheduling I/O has been carried out in the context of a general scheduling model [7] which we have used to address more complex scheduling problems, including I/O scheduling in the presence of precedence constraints, mutual exclusion constraints, and hierarchical system architectures. We are currently investigating algorithms for scheduling synchronized data transfers which arise in applications such as multimedia systems, and on-line I/O scheduling.

Acknowledgements

The first author thanks Ramesh Govindan, Peter Newton and Mark Sullivan for many helpful comments and discussions.

References

[1] A. Bar-Noy, R. Motwani, and J. Naor. The greedy algorithm is optimal for on-line edge coloring. *Inf. Proc. Lett.*, pages 251–253, Dec. 1992.

[2] Claude Berge. *Graphs.* North-Holland, 1985.

[3] G. Bongiovanni, D. Coppersmith, and C. K. Wong. An optimum time slot assignment algorithm for an SS/TDMA system with variable number of transponders. *IEEE Trans. Comm.*, 29(5):721–726, May 1981.

[4] P. J. Denning. Effects of scheduling on file memory operations. In *Proc. AFIPS Spring Joint Comp. Conf.*, pages 9–21, 1967.

[5] G. A. Gibson. *Redundant disk arrays: Reliable, parallel secondary storage.* PhD thesis, Univ. of Calif., Berkeley, Comp. Sci. Div, 1990. Also available as Tech. Rep. UCB/CSD 91/613.

[6] D. S. Hochbaum, T. Nishizeki, and D. B. Shmoys. A better than "best possible" algorithm to edge color multigraphs. *SIAM J. Comput.*, 7:79–104, 1986.

[7] Ravi Jain. Scheduling data transfers in parallel computers and communications systems. Technical Report TR-93-03, Univ. Texas at Austin, Dept. of Comp. Sci., Feb. 1993.

[8] Ravi Jain, Kiran Somalwar, John Werth, and J. C. Browne. Scheduling parallel I/O operations in multiple-bus systems. *J. Par. and Distrib. Comp.*, Dec. 1992. Special Issue on Scheduling and Load Balancing.

[9] Ravi Jain, Kiran Somalwar, John Werth, and J. C. Browne. Requirements and heuristics for scheduling parallel I/O operations. May 1993. Submitted for publication.

[10] Ravi Jain, Kiran Somalwar, John Werth, and J. C. Browne. Scheduling parallel I/O operations. In *Proc. Workshop on I/O in Par. Comp. Sys.*, Apr. 1993.

[11] Ravi Jain and John Werth. Analysis of approximate algorithms for edeg-coloring bipartite graphs. May 1993. Submitted for publication.

[12] Ravi Jain, John Werth, and J. C. Browne. A general model for scheduling of parallel computations and its application to parallel I/O operations. In *Proc. Intl. Conf. Par. Proc.*, 1991.

[13] T. Lovett and S. Thakkar. The Symmetry multiprocessor system. In *Proc. Intl. Conf. Par. Proc.*, pages 303–310, 1988.

[14] T. N. Mudge, J. P. Hayes, and D. C. Winsor. Multiple bus architectures. *Computer*, 20(6):42–48, June 1987.

[15] D. A. Pattreson, G. A. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. SIGMOD*, 1988.

[16] G. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM research parallel processor (RP3): Introduction and architecture. In *Proc. Intl. Conf. Par. Proc.*, pages 764–771, 1985.

[17] A. Silberschatz and J. Peterson. *Operating systems concepts.* Addison-Wesley, 1988.

[18] Kiran Somalwar. Data transfer scheduling. Technical Report TR-88-31, Univ. Texas at Austin, Dept. of Comp. Sci., 1988.