Heuristics for Scheduling I/O Operations

Ravi Jain, *Senior Member*, *IEEE*, Kiran Somalwar,
John Werth, *Member*, *IEEE*, and J.C. Browne

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.

Index Terms—Data transfer scheduling, graph edge coloring, I/O bottleneck, multiprocessor operating systems, parallel I/O, scheduling heuristics, simultaneous resource scheduling.

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 [15], [18].

There have been two major application domains where I/O in parallel computer systems has traditionally been found to be a bottleneck. One is scientific computing with massive datasets, such as those found in seismic processing, climate modeling, etc. [10]. The second is databases [6]. While the I/O bottleneck remains a central concern for these domains [7], in recent years the concern has spread to other applications (such as image visualization [2], multimedia applications [13], [14], and parallel text retrieval [24]), executing on a variety of parallel architectures [34], [15], [1], [26], [8].

Most solutions to the I/O bottleneck 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 [29], [15]. 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 I/O bottleneck. However, scheduling parallel I/O operations has received relatively little attention. 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. In this paper, we will develop scheduling algorithms and explore this tradeoff by considering two criteria for evaluating them: the time that an algorithm takes to produce a schedule and the length of the schedule produced.

Much of the previous work on scheduling deals with tasks which each require only a single resource at any given time [18] 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. In addition, the obvious extension of single resource scheduling, i.e., serial acquisition of multiple resources, does not in general lead to optimal schedules. Consequently, 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 has made assumptions which are not relevant for scheduling parallel I/O operations (see [18] 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 [21], [19], [18].

- *R. Jain is with Applied Research, Bellcore, 331 Newman Springs Rd., Red Bank, NJ 07701. E-mail: rjain@thumper.bellcore.com.*
- *K. Somalwar is with Digital Equipment Corp.,*
- *J. Werth and J.C. Browne are with the University of Texas at Austin, Austin, TX*

Manuscript received May 1993.
For information on obtaining reprints of this article, please send e-mail to: transpds@computer.org, and reference IEEECS Log Number D95259.

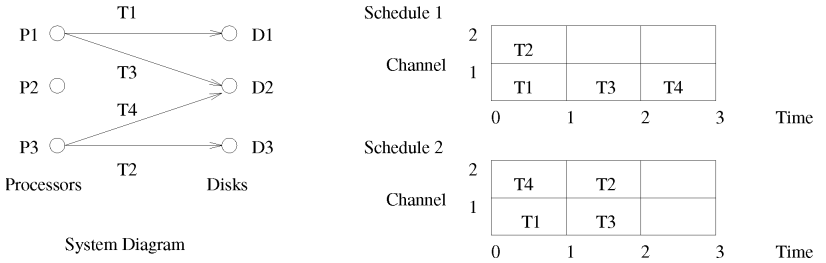

Fig. 1. Parallel I/O scheduling example.

In Section 2, we sketch the role of parallel I/O scheduling in the context of an integrated approach which addresses the I/O bottleneck at several levels of the system, and the requirements for scheduling to be feasible. In Section 3, we describe a simple parallel I/O scheduling problem and a scenario in which it may be likely to arise. We have previously designed optimal algorithms which provide optimal solutions to this problem [19]. In Section 4, we present fast heuristics which provide approximate solutions to the problem, and examine their performance theoretically. In Section 5, we present results of an experimental evaluation of the performance of these heuristics.

We evaluate performance using two criteria: the time required to produce schedules and how close the schedules produced are to the optimal schedule. We compare the heuristics both against the optimal algorithm and to the situation in which 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 heuristics 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 Section 6, we end with some conclusions.

2 Scheduling Parallel I/O Tasks

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 [33]. One important innovation was to schedule the I/O operations by reordering the requests in the queues at devices [11], [33]. We propose that a scheduling approach be applied to I/O operations in parallel computer systems also. 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 I/O devices, where a device may be a single independent disk or a striped disk array. For simplicity, consider each I/O device to be a single disk (see Fig. 1). The I/O transfer requests are numbered in the order they arrived at the system, and have been collected to form a batch of transfers which is to be scheduled. 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. Schedule 1 shows the transfers which take place at each time slot if the transfers are scheduled in the order they arrived at the system, i.e., First Come First Served (FCFS). Schedule 2 shows that transfers can be completed in a shorter time by rearranging the order in which they are performed. As mentioned earlier, we will be interested in two criteria for evaluating scheduling algorithms: the time that the algorithm takes to produce a schedule, and the length of the schedule produced. Thus, for example in Fig. 1, an algorithm like FCFS which takes very little time to produce a schedule (and hence has good performance with respect to the first criterion) may produce schedules longer than the minimum length (i.e., bad performance with respect to the second criterion). Conversely, the algorithm which produces Schedule 2 may take longer than that which produces Schedule 1, but also produces an optimal (i.e., minimum-length) schedule.

In general, scheduling parallel I/O tasks can be effective if the following conditions hold.

C1. It is possible to control the order and timing of the execution of tasks,
C2. There is sufficient information to make choices about task order and timing,
C3. There are resource bottlenecks which can be reduced by making such choices, and
C4. The overhead incurred in making these choices is sufficiently small.

Condition **C4** often holds, and is increasingly likely to hold in many systems, as the gap between I/O speeds and processor speeds continues to increase and applications generate larger I/O requests. Whether conditions **C1-C3** hold for any particular situation depends upon the nature of the I/O requests generated by applications as well as the capabilities provided by the system software and architecture. We briefly discuss these below.

In general, condition **C3**, namely the existence of resource conflicts whose effects can be mitigated by scheduling, often holds in the I/O subsystem of many large-scale parallel architectures, a typical resource bottleneck being the number or effective bandwidth of I/O channels [25], [28]. However, whether **C3** holds also depends upon the structure of the I/O requests generated by I/O-intensive applications. Parallel I/O scheduling to mitigate resource conflicts can be beneficial if the multiprocessor is executing multiple programs, if the data access patterns cannot be predicted, or if the assignment of data to devices is fixed or too expensive to modify [20].

The information required for scheduling, i.e., condition **C2** above, depends upon the type of scheduling to be performed. In this paper, we consider centralized batch-oriented scheduling. Centralized batch scheduling requires a relatively large amount of control information (or "metadata") about the data to be transferred, such as the source, destination, and length of each transfer, any ordering between transfers, and the size of each data block constituting a transfer. (We discuss the availability of control information further below.) In addition, this information must be available at a central control point, and the nature of the I/O demand must permit batch operation. Despite these stringent requirements, there are interesting situations where they hold, such as the delivery of x-ray image files from a medical patient database, or the delivery of movies from an electronic video store [31]. In general, I/O requests from these kinds of interactive visualization and multimedia applications occur naturally in batches in response to user inputs, the size of archival image files is typically known in advance, and files are typically stored in fixed-length blocks.

Despite the potential benefits of scheduling for some important application classes, the architecture and system software of many parallel computers do not have the capabilities to permit effective I/O scheduling. In particular, they do not permit sufficiently fine-grained control over the timing and order of I/O operations, i.e., condition **C1** is not met. However, we expect this situation to change because these are precisely the capabilities which are required to address the I/O bottleneck at higher levels of the system. For instance, it has been argued that the low-level parallelism of disk striping and RAID [29] techniques which are becoming widely available may not be effectively utilized in the long run if such facilities are not provided at higher levels. In particular, it has been argued that such facilities, as well as control information, are needed for I/O-efficient application-specific scheduling algorithms [22], [36], [9], compilers [32], operating systems, and architecture [28]. These arguments are reviewed in [20]. In this paper, we assume that such facilities will be available.

We have discussed the role that scheduling can play in parallel computer systems, the applications for which it may be effective, and the capabilities required to effectively utilize it. We now describe a specific I/O scheduling problem and its solution.

3 A Parallel I/O Scheduling Problem

We first state the parallel I/O scheduling problem we will consider in this paper, and then describe an example application in which it may arise. The application we discuss is image visualization but numerous other applications can be considered. We show 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.

3.1 Problem Statement

The specific I/O problem we will consider can be stated, as a decision problem, as follows: In principle, the problem may arise between any two levels of the memory hierarchy, but is stated here in terms of processors and I/O devices. Given a batch of pending I/O transfers, where

1) all transfers are of the same length,
2) the transfers may occur in any order,
3) 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,
4) each processor and each each disk may perform at most one transfer at any given time, and
5) each processor can communicate via a direct link with each I/O device

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 [25] where the interconnection network connecting the I/O devices to processors is 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 [30]. 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 [27]. In general, *UnitIOS* may also arise in a shared-bus architecture; typically in such architectures the effective bandwidth of the bus limits the number of devices that can be attached to it, and scheduling data transfers will help alleviate this limitation.

3.2 Example 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 large amount of processing required to convert the raw data into images has been performed on a parallel computer, and the resulting image database is stored on disks at a central site. Users view the images by requesting image files to be displayed on their graphics workstations. The disks and workstations are connected, possibly via transparent high-speed interface cards, to a set of common buses (or a single high-speed system bus that is shared in a time-multiplexed fashion); the buses allow multiple I/O transfers to proceed in parallel. The disks have a central controller which receives requests for image files from the workstations, collects them into batches, and schedules the resulting I/O transfers, which take place across the set of common buses.

The central controller can be a separate intelligent disk controller, or one of the disks' controllers which has been designated as the master. We also consider the third possibility that the system is configured so that the parallel computer (where the raw data was converted into images) performs the role of the central controller and schedules the transfer of images between workstations and disks. In this third case, we consider the situation where the parallel computer itself is a shared-bus system, with both processors and disks on the common set of buses; then the algorithm for scheduling the data transfers is part of the parallel computer's operating system. Note that whether the scheduling algorithm is itself a parallel algorithm is an orthogonal issue; in general, it is desirable that the scheduling algorithm produces near-optimal schedules quickly enough that the algorithm need not be parallelized.

Regardless of the device used for the central controller, user requests for image files are sent from the workstations to the central controller. 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.

3.3 Definitions and Problem Formulation

In this section, we present some definitions and formulate the problem more formally. Some repetition of material in our previous work [19] is necessary in order to make this paper self-contained.

DEFINITION. *An* edge coloring *of a graph $G = (V, E)$ is a function $c : E \rightarrow N$ 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 would add the restriction that each color can be used to color at most k edges.

With this definition, Fig. 1 can be interpreted as an instance of the simple I/O scheduling problem. The graph is a bipartite graph, i.e., one in which the vertices can be partitioned into two sets such that each edge connects vertices from different partitions. In the graph, the vertices on the left represent processors and those on the right represent I/O devices, and the edges represent data transfers. The two schedules represent two ways of edge-coloring the graph, where the time slots correspond to different colors.

We state some definitions and known results which we will draw upon, sketching proofs where they provide intuition for results later in the paper. In the following, we will sometimes say that a vertex is "covered" by a given color if some edge incident on it has been assigned that color.

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 $|E| = m$.*

DEFINITION. *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.*

The degree of the bipartite graphs representing processors, disks, and data transfers (such as the one shown in Fig. 1) is an important parameter. It turns out, as explained below, that for any given bipartite graph, the minimum number of colors needed to edge-color the graph (or equivalently, to schedule the data transfers) equals the degree of the graph.

DEFINITION. *A* matching *$M \subseteq E$ is a set of edges such that no two edges have a common vertex. A* critical matching *is one which covers all critical vertices.*

LEMMA 3.1 [4]. *Every bipartite graph has a critical matching.*

LEMMA 3.2. *Exactly d colors are necessary and sufficient to color a bipartite graph of degree d.*

PROOF. [4] 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 3.1, 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. □

Since the minimum number of colors required to edge-color a bipartite graph equals its degree, we see that the key issue for any edge-coloring algorithm which is attempting a minimum coloring for bipartite graphs is to reduce the graph degree with every color used. Thus, with each color used, the algorithm should make sure that *every* critical vertex is covered with that color. As we will see in the next section, our algorithms are iterative, with one color being used to edge-color the graph during each iteration, and edges which are assigned a color being deleted from the graph at the end of each iteration. Thus, at every iteration, the algorithm should reduce the degree of *all* the highest-degree vertices by one. (Note that if some critical vertex does not have its degree reduced, the degree of the graph is not reduced and, hence, neither is the minimum number of colors required to color the remaining graph.) We will make use of this insight when we design efficient edge-coloring heuristics in the next section. We now consider the situation where each color may be used to color only a limited number of edges.

DEFINITION. *A* k-coloring *of a graph is an edge-coloring in which each color may be used to color at most k edges.*[1]

1. Our definition differs from [4], where k-coloring refers to the *vertex* coloring problem in which at most k colors may be used to color all the vertices in the graph.

LEMMA 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.*

PROOF. [5] If $d < \lceil m/k \rceil$, at least $\lceil m/k \rceil$ colors are required to color the graph. Otherwise, the argument of Lemma 3.2 applies. □

In the following section, we discuss algorithms for obtaining edge-colorings which meet, or come very close to, these lower bounds on edge-colorings.

4 HEURISTICS FOR SCHEDULING PARALLEL I/O

Algorithms for constructing minimum length schedules for *UnitIOS* have been designed. These are the algorithm **KT** [5] which takes time $O(mn(m + n))$, Somalwar's algorithm [35] which takes time $O(mn^{1.5}\log n)$, and algorithm **A2** [19] 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 [18].

While the optimal algorithm **A2** is faster than previous optimal algorithms, we would like to have even faster algorithms. This is because scheduling algorithms are typically 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. In this section, we describe several greedy edge-coloring algorithms, and present results on their theoretical worst-case behavior, both in terms of execution time and the lengths of the schedules produced.

4.1 Greedy Heuristics

An edge-coloring algorithm is called "greedy" if, for each color, it attempts to color as many edges as possible with that color; it is called a heuristic, or approximation algorithm, if it is not guaranteed to use the minimum number of colors. We use the pseudo-code template below for describing two of the greedy heuristics. Here the **break** statement exits the smallest enclosing loop (i.e., the **for** loop). The *Order()* function is described below.

Algorithm Greedy Vertex Heuristic

Input: Bipartite graph $G = (A, B, E)$, integer k, $0 < k \le \min\{|A|, |B|\}$;
(k is the maximum number of simultaneous transfers allowed).

Output: A k-coloring of G

```
1. F := Order(A, B, E);   /* F is some ordered sequence of the
   vertices in A ∪ B */
2. Initialize vcolor(i) = –1 for all i, 0 ≤ i ≤ |A| + |B|
3. i := 0;
4. while F ≠ ⟨ ⟩ {
5.    M := { };      /* M is the set of edges currently assigned
                        color i */
6.    for each vertex v read in sequence from F {
7.       if vcolor(v) ≠ i {
8.          Choose any edge e = (v, w) incident on v
9.          vcolor(v) := vcolor(w) := i;
10.         color(e) := i;
11.         E := E – {e};
12.         M := M ∪ {e};
13.      }   /* if */
14.      /* Start a new color if max. number of simultaneous transfers is reached */
15.      if |M| = k break;
16.   }   /* for */
17.   Delete vertices of degree 0
18.   i := i + 1;
19.   F := Order(A, B, E);   /* Reorder the remaining vertices in the graph */
20. }   /* while */
```

We assume that when the algorithm is called, the graph G contains only the vertices and edges corresponding to the transfers which have been requested (since not all workstations may make requests), and that the vertices are added to the graph in the order that the requests arrive.

Two heuristics we will investigate can be defined, assuming the template above, as follows.

1) First-Come First-Served, **FCFS**. Here *Order*() is the identity function. That is, **FCFS** examines the vertices of the bipartite graph in the order they are presented, and tries to cover as many vertices as possible.
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.

We present some intuition behind the **HDF** heuristic. For simplicity, consider the case where k is not the limiting factor, i.e., $k \ge \min\{|A|, |B|\}$. Recall that the minimum number of colors needed to edge-color the graph is the graph degree d. By the definition of edge-coloring, the degree of the graph cannot be reduced by more than one at any iteration of the **while** loop of the algorithm. Thus if, in each iteration, all the vertices of the highest degree are colored and the colored edges deleted, the degree of the graph is reduced by exactly one at each iteration, resulting in an edge-coloring using the minimum number of colors. Each edge which is chosen in line 8 precludes certain other edges (and a vertex) from being colored with the same color, thus motivating the idea of attempting to color the highest-degree vertices first.

Note that in considering the vertices of highest degree, it is necessary to consider vertices in both partitions of the graph; in terms of the data transfer application, this is because the "bottleneck" may arise at either a processor or an I/O device. Also note that after one coloring has been found, the list F may not necessarily correspond to an ordering of vertices by descending degree, so in line 19, it must be reordered.

We observe that one situation where the **HDF** heuristic may fail to cover all the critical vertices at an iteration occurs because it only considers the degree of the vertex at one endpoint of an edge. For instance, for the input graph of Fig. 1, consider the situation where the first vertex in the list F, and hence the first vertex examined by **HDF**, is the critical vertex P3. Now **HDF** may choose to color either T2 or T4, thus possibly making a poor choice, as in Schedule 1 of Fig. 1. Consequently, we propose the *Highest-Combined-Degree First* (**HCDF**) which we expect to perform better than **HDF** because it considers the degree of both endpoints

of an edge. For instance, for the input graph of Fig. 1, when no edges have been colored and **HCDF** has a choice between coloring T2 and T4, unlike **HDF** it will always choose the latter.

The **HCDF** heuristic is described using pseudo-code below. The pseudo-code is very similar to that given above, except it uses edge lists instead of vertex lists. We can use the template below to state **HCDF** as the third heuristic we will consider:

1) 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.

Algorithm Greedy Edge Heuristic

Input: Bipartite graph $G = (A, B, E)$, integer k, $0 < k \leq \min\{|A|, |B|\}$;
(k is the maximum number of simultaneous transfers allowed).

Output: A k-coloring of G

```
1. F := Order(A, B, E);      /* F is some ordered sequence of
                                the edges in E */
2. Initialize vcolor(i) = –1 for all i, 0 ≤ i ≤ |A| + |B|
3. i := 0;
4. while F ≠ ⟨ ⟩ {
5.    M := {};      /* M is the set of edges which are assigned
                       color i */
6.    for each edge e = (u, v) read in sequence from F {
7.       if vcolor(u) ≠ i and vcolor(v) ≠ i{
8.          vcolor(u) := vcolor(v) := i;
9.          color(e) := i;
10.         E := E – {e};
11.         M := M ∪ {e};
12.      }    /* if */
13.      /* Start a new color if max. number of simultaneous transfers is reached */
14.      if |M| = k break;
15.   }    /* for */
16.   Delete vertices of degree 0
17.   i := i + 1;
18.   F := Order(A, B, E);      /* Reorder the remaining edges
                                   in the graph */
19. }    /* while */
```

Intuitively, we expect **HCDF** to produce shorter schedules than **HDF** since it uses more information to choose the critical vertices it covers. For example, for the graph shown in Fig. 1, **HDF** may produce Schedule 1, while **HCDF** will always produce an an optimal schedule (e.g., Schedule 2). We now state some theoretical results which apply to all the greedy heuristics above; we omit most proofs for brevity.

LEMMA 4.1 [18], [3]. *For a bipartite graph of degree d, and no restriction on the number of edges which can be colored with a single color, a greedy heuristic produces a coloring using at most* $2d - 1$ *colors.*

The intuitive explanation for Lemma 4.1 is roughly as follows. (For simplicity, assume $k \geq \min\{|A|, |B|\}$). A vertex v will not be colored with some color only if *all* its partners are colored with that color. Thus each color used either reduces the degree of v by one, or reduces the degree of all its partners by one. Thus at most $2d$ colors are needed to reduce the degree of v and all its partners to zero. We can lower this bound to $2d - 1$ by the observation that when the last edge incident on v is colored, it necessarily reduces the degree of both v and one of its partners.

LEMMA 4.2. *For a bipartite graph with m edges and degree d, and no restriction on the number of edges which can be colored with a single color, the greedy algorithms take time* $O(md)$ *to solve UnitSimpleIOS, and produce a schedule of length at most* $2d - 1$.

PROOF. The bound on schedule length follows from Lemma 4.1. For running time, observe that for **FCFS**, the **for** loop takes time $O(n)$ and *Order*() takes no time, so from Lemma 4.1, it follows that **FCFS** takes time $O(nd)$. For **HDF** the **for** loop also takes time $O(n)$, but *Order*() requires sorting. A bucket sort can be used for *Order*(), so that the first invocation of *Order*() takes $O(m)$ time. The invocations of *Order*() inside the **while** loop only need to reorder vertices due to the deletion of edges in M, and since $|M| \leq n$, take time $O(n)$. Thus, for **HDF**, each iteration of the **while** loop takes time $O(n)$. From Lemma 4.1, it follows that **HDF** takes time $O(m + nd)$. Finally, for **HCDF**, the **for** loop takes time $O(m)$, so from Lemma 4.1, it follows that **HCDF** takes time $O(md)$. □

It can be shown that the upper bound on the schedule length is "tight" for the heuristics above, when there is no restriction on the number of edges which can be colored by the same color. That is, 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 heuristic uses exactly $2d - 1$ colors. For **FCFS**, this construction [18], [3] is relatively straightforward. For **HDF** and **HCDF** it is quite involved.

LEMMA 4.3 [18]. *For both the* ***HDF*** *and* ***HCDF*** *algorithms, and no restriction on the number of edges which can be colored with a single color, 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.*

We can find some results similar to those above for the situation in which $k < \min\{|A|, |B|\}$, i.e., there exists a limitation on the number of simultaneous transfers possible.

LEMMA 4.4 [18]. *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 greedy algorithm produces a coloring using at most* $\lfloor m/k \rfloor + (2d - 1)$ *colors.*

Using a reasoning similar to Lemma 4.2, the greedy algorithms run in time $O(m^2/k + md)$ when the number of simultaneous transfers is limited to k.

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 for an optimal algorithm in Section 5.

5 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 **A2** [19] for unit-weight edges implemented by Somalwar [35]. 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 total number of data sources and sinks n, the number of transfers m, and the maximum number of simultaneous transfers k were varied. For each experiment, one hundred input instances (bipartite graphs) were generated using a pseudo-random number generator [23].

The range over which the parameters were varied was chosen keeping in mind the example application scenario of Section 3, i.e., image visualization using a multiple-disk shared-bus system. For example, suppose there are 16 disks and 16 workstations, and the images requested by workstation users are around 1 MB in size. To amortize the cost of disk accesses, images are typically stored in relatively large blocks (e.g., see [16]), say of size 64 KB. If all 16 users request images simultaneously, 250 transfers will be required. We performed experiments over a wide range of parameters; we present results from a representative set of experiments whose parameters are shown in Table 1. Here m is the number of transfers to be scheduled, n is the total number of disks and workstations, and k is the maximum number of simultaneous transfers. Note that we consider a wide range of edge densities, from cases where m is less than 10% of the number of edges in a complete graph to where it is more than three times that number (since multiple edges are allowed).

Our goal is to evaluate the feasibility and relative benefits of heuristic and optimal parallel I/O scheduling algorithms. For the design of any specific system it would be necessary to consider detailed application and system characteristics in order to select an appropriate algorithm. These characteristics would include specific parameters of the I/O interconnection network such as bus bandwidth and latency, the relative cost and number of processors and I/O devices, etc. As will be apparent from the results below, the experiments we have performed cover the interesting regimes of the algorithms' behaviors, and provide insight about how their behavior changes as relevant system parameters are varied. These results could then be used to guide the choice of algorithm for specific situations.

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.

TABLE 1
Parameters of the Experiments Reported Below

Experiment	m	n	k
Experiment 1	Varied from 100 to 800	32	8
Experiment 2	100	32	Varied from four to 16
	600	32	Varied from four to 16
Experiment 3	200	Varied from 24 to 64	12

Experiment 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 ($k = 8$). 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. (To obtain a reasonable granularity of measurement, the time for processing each graph 100 times was measured, and then divided by 100 to obtain the average time for processing that graph). 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.

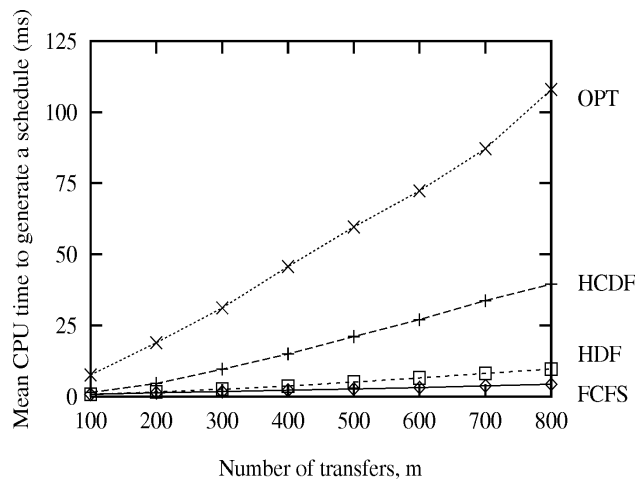

Fig. 2. CPU time (ms) versus number of transfers for $n = 32$, $k = 8$.

The improvement in CPU time for the heuristic algorithms comes at the expense of worse schedules. To evaluate this, we define two quantities as follows. Let $opt(i)$ denote the optimal schedule length for graph i, $1 \le i \le 100$, and let $alg(i)$ denote the schedule length for a schedule generated by some heuristic algorithm *alg*. Then, for algorithm *alg*,

Mean percentage increase in schedule length =

$$\sum_{i=1}^{100} \frac{alg(i) - opt(i)}{opt(i)}$$

Maximum percentage increase in schedule length =

$$\max\left\{\frac{alg(i) - opt(i)}{opt(i)}, i = 1, \ldots 100\right\} * 100$$

We observed that the mean schedule length produced by the heuristic 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 **FCFS** can be over 45% longer than optimal. On the other hand, the schedules produced by **HDF** and **HCDF** were found to be always of optimal length, for this experiment. (Of course, there exist graphs, such as that shown in Fig. 1, for which the schedules produced by **HDF** and **HCDF** may have different lengths).

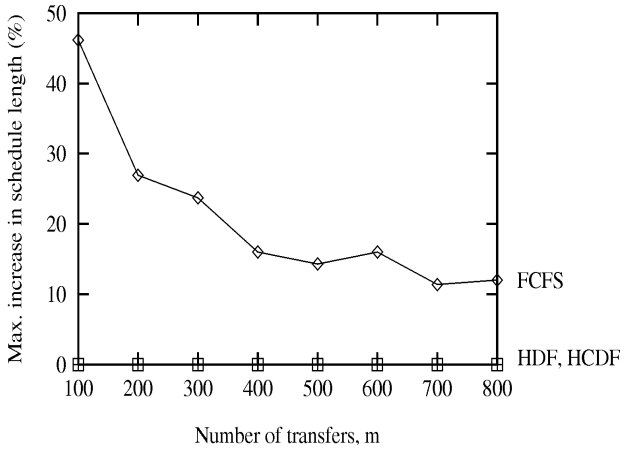

Fig. 3. Percentage increase in schedule length versus number of transfers for $n = 32$, $k = 8$.

We discuss a few features of the experimental results. In Experiment 1 the execution time of all the algorithms increases with the number of transfers to be scheduled (Fig. 2), as expected. It is interesting to see that the execution time for **HDF** is only slightly greater than **FCFS**. In Fig. 3, we see that the maximum percentage increase in schedule length produced by **FCFS** decreases as *m* increases. This is because as *m* increases, the optimum schedule length increases while the absolute increase in the schedule length produced by **FCFS** does not increase as fast, so that the percentage increase drops.

Experiment 2. Effect of varying I/O channel capacity, *k*. In this experiment, we fix $n = 32$ (i.e., 16 vertices in each partition, as before), $m = 100$, and vary *k* from four to 16; at $k = 16$ the *UnitIOS* problem reduces to the *UnitSimpleIOS* problem. In Fig. 4, we plot the mean CPU time for all the algorithms as *k* is varied. In Fig. 5, we plot again the maximum percentage increases in schedule length compared to the optimal schedule, over the input instances.

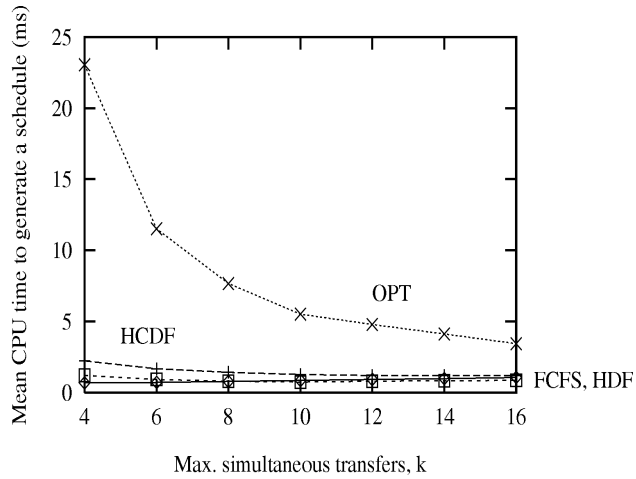

Fig. 4. CPU time (ms) versus maximum number of simultaneous transfers for $n = 32$, $m = 100$.

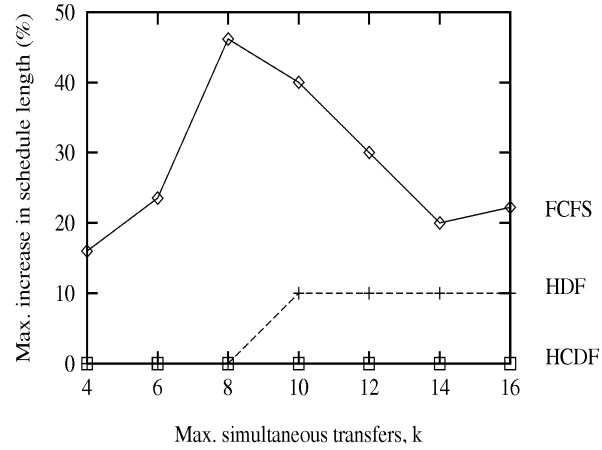

Fig. 5. Percentage increase in schedule length versus maximum number of simultaneous transfers for $n = 32$, $m = 100$.

From Fig. 4, we see that the heuristics are significantly faster than the optimal algorithm, and also show much less variability with *k*. We also see that the mean CPU time of **FCFS** barely differs from **HDF** and is only slightly less than **HCDF**. The CPU time decreases as *k* increases. This is because as *k* increases, more transfers are scheduled at each time slot, so the schedule length, and hence the number of iterations of the **while** loop in the greedy algorithm, decreases.

From Fig. 5, we see that while the maximum deviation of **HDF** from the optimum schedule is limited to 10%, the **FCFS** algorithm can produce schedules up to 46% longer. The **HCDF** heuristic produces optimal schedules as before.

The variation of the maximum percentage increase in schedule length of **FCFS** and **HDF** with *k* in Fig. 5 shows two phases. This is quite interesting and is explained as follows. First note that as *k* is varied from four to 16, the value $\lceil m/k \rceil$ varies from 25 to seven, while *d*, the degree of each input graph, remains the same. For the one hundred instances in our experiment, the majority of graphs have degree 11 or 12.[2] In other words, as *k* is increased from four

2. This clustering of the graph degrees is not surprising since the vertex degree has a binomial probability distribution, and the graph degree is bounded above by the bound on the tail of the binomial distribution [12].

to 16, we pass from a phase in which the optimal schedule length is determined by $\lceil m/k \rceil$ (for $k \le 8$) to one in which it is determined by d (for $k \ge 10$).

For $k \le 8$, the optimum schedule length decreases as k increases. Meanwhile, the difference between the schedule length generated by **FCFS** and the optimal does not vary much, so that its ratio to the optimum schedule length increases. **HDF** and **HCDF** perform well when k is small, since they succeed in covering the same relatively small number of critical vertices which the optimal algorithm does.

For $k \ge 10$, the optimal schedule length is independent of k; essentially, the optimal algorithm cannot take further advantage of the increased parallelism provided by the larger value of k. On the other hand, **FCFS** now has more opportunity to cover the same number of critical vertices as the optimal algorithm. Thus the difference between the schedule length generated by **FCFS** and the optimal algorithm decreases slowly, so that its ratio to the fixed optimal schedule length also decreases. For **HDF**, we see that the maximum increase in schedule length becomes 10%; for these parameters, it corresponds to a schedule length of one slot more than the optimal. This occurs because, unlike the phase where $k \le 8$, in this phase **HDF** must cover all the critical vertices in order to do as well as the optimal algorithm.

In Figs. 6 and 7, we see the same kind of behavior for $m = 600$ as we have discussed for $m = 100$, although not as pronounced.

Experiment 3. Effect of varying number of disks and workstations, *n*. In this experiment, we fix $k = 12$, $m = 200$, and vary n from 32 to 64, with equal numbers of vertices in each partition. Figs. 8 and 9 show the mean CPU time, and the maximum percentage increases in schedule length as n is varied. Once again, while the maximum schedule length produced by **HDF** or **HCDF** is optimal or very close to optimal, the schedule length produced by **FCFS** can be significantly larger.

In Experiment 3, we see that keeping the same number of transfers while changing the architecture (the number of sources and sinks) does not affect the mean CPU time and maximum schedule length increase of the heuristic algorithms significantly; this is as expected. The execution time of the optimal algorithm (see Fig. 8), on the other hand, is quite sensitive to n.

5.1 Discussion and 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 [17]. 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" [17]. Note that their algorithm does not consider the practical constraint of a limited number of simultaneous data transfers, i.e., $k \le n$. For our applications, this constraint corresponds to the realistic situation of limited bus bandwidth being available in the system. Also, since the **HDF** and **HCDF** 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.

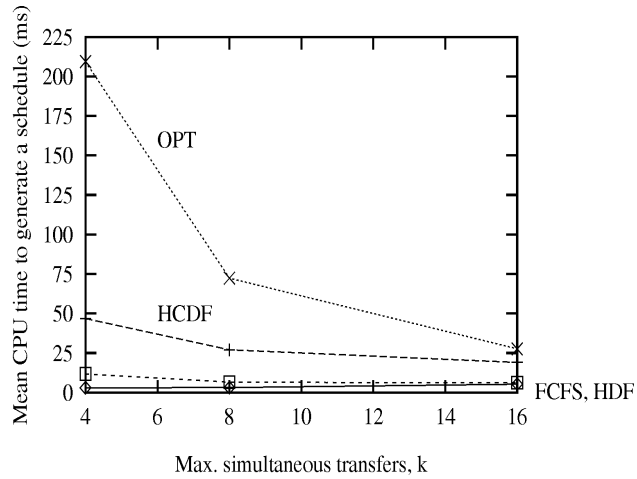

Fig. 6. CPU time (ms) versus maximum number of simultaneous transfers for $n = 32$, $m = 600$.

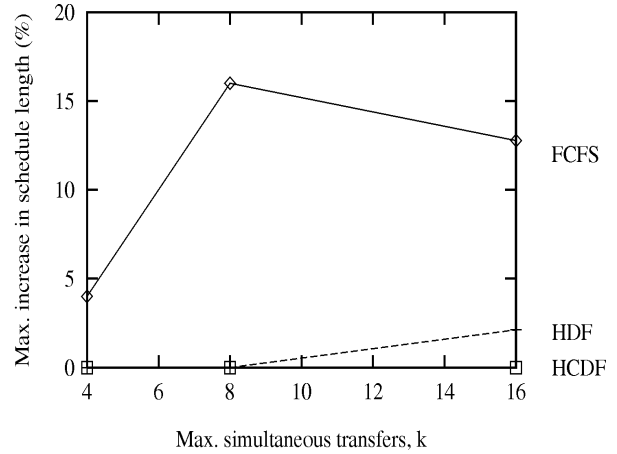

Fig. 7. Percentage increase in schedule length versus maximum number of simultaneous transfers for $n = 32$, $m = 600$.

Note that **FCFS**, **HDF**, **HCDF,** and the interchange heuristic form a sequence of heuristics which tend to produce shorter schedules at the expense of increasing processing time. Thus **FCFS**, **HDF**, **HCDF,** and the interchange heuristic can be viewed as a range of heuristics which allow the system designer to trade off communication cost versus processing time. Here we have quantified the relative performance of **FCFS**, **HDF**, **HCDF,** and the optimal algorithm (**A2** from [19]), for the situations we have studied, both in terms of processing time and schedule length. For architectures and applications where communication cost is high but processing cost is not (e.g., where data transfers are relatively long or processing is fast), the **HCDF** heuristic, or possibly even the optimal algorithm **A2**, should be used. On the other hand, where the amount of time available for scheduling is relatively limited (e.g., if many requests arrive frequently in a system where communication bandwidth is high), the **HDF** heuristic should be used. The parameters of each application and architecture can be estimated and the estimates used to guide the particular choice of heuristic for

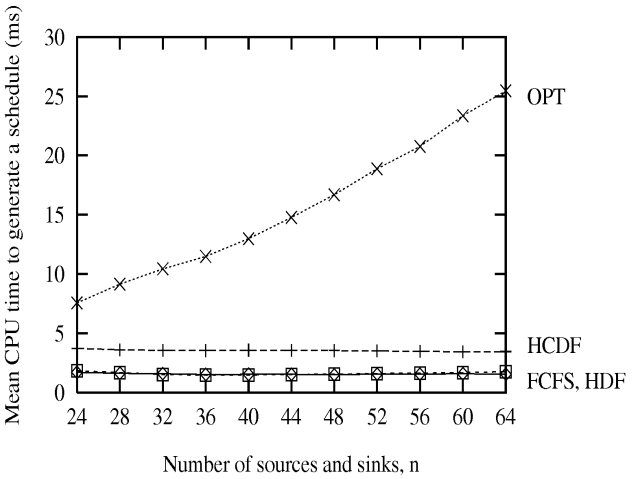

Fig. 8. CPU time (ms) versus number of sources and sinks for m = 200, k = 12.

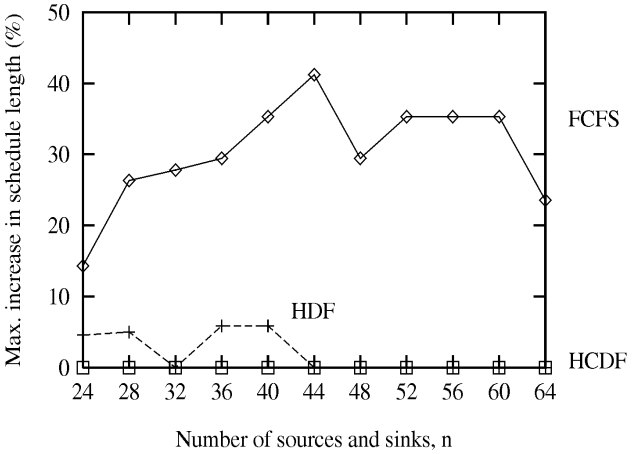

Fig. 9. Percentage increase in schedule length versus number of sources and sinks for m = 200, k = 12.

that situation.

It seems that the **HDF** and **HCDF** heuristics are extremely well-behaved. Despite the existence of graphs for which it can be proved that they produce poor schedules, in our experiments we found that they produced schedules at most 10% longer than optimal, for execution times which are close to the execution times of the **FCFS** algorithm. On the other hand, the maximum schedule length produced by **FCFS** in our experiments was almost 50% over the optimal schedule. More importantly, the dependence of **FCFS** on various input parameters, especially k, is quite large and hard to predict. Since **FCFS** essentially models the situation in which no scheduling is performed, we see that for these experiments, using a simple greedy heuristic like **HDF** or **HCDF** for batch scheduling of parallel I/O operations can be advantageous.

6 Conclusions

Most attention to the I/O bottleneck has focused on improving the performance of system components using low-level parallelism. An integrated approach to the I/O bottleneck which addresses the problem at multiple levels of the system is needed. Within the context of such an approach, scheduling parallel I/O operations will become increasingly attractive and can potentially provide substantial performance benefits.

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.

Our work on scheduling I/O has been carried out in the context of a general scheduling model [18] 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.

Acknowledgments

Ravi Jain thanks Ramesh Govindan, Peter Newton, and Mark Sullivan for many helpful comments and discussions. We also thank the referees for their helpful comments on the previous draft of this paper. This resarch was performed while Ravi Jain and Kiran Somalwar were at the University of Texas and was partially supported by IBM Corp. through grant 61653 and by the State of Texas through TATP Project 003658-237. A preliminary version of part of this work appeared in the informal proceedings distributed to participants at the Workshop on I/O in Parallel Computer Systems, Newport Beach, California, April 13, 1993. A much shorter version of the workshop paper appeared in ACM Sigarch's quarterly bulletin, *Computer Architecture News*, in December 1993.

References

[1] J. Akella and D.P. Siewiorek, "Modeling and Measurement of the Impact of Input/Output on System Performance," *Proc. 18th Int'l Symp. Computer Architecture*, pp. 390-399, 1991.
[2] M. Arrott and S. Latta, "Perspectives on Visualization," *IEEE Spectrum*, pp. 61-65, Sept. 1992.
[3] A. Bar-Noy, R. Motwani, and J. Naor, "The Greedy Algorithm Is Optimal for On-Line Edge Coloring," *Information Processing Letters*, pp. 251-253, Dec. 1992.
[4] C. Berge, *Graphs*. North-Holland, 1985.
[5] 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.*, vol. 29, no. 5, pp. 721-726, May 1981.
[6] H. Boral and D.J. DeWitt, "Database Machines: An Idea Whose Time Has Passed? A Critique of the Future of Database Machines," *Proc. Third Int'l Workshop Database Machines*, pp. 166-187, 1983.
[7] *Database Machines*, H. Boral and P. Faudemay, eds. Springer-Verlag, 1989.
[8] C.E. Catlett, "Balancing Resources," *IEEE Spectrum*, pp. 48-55, Sept. 1992.
[9] T. Cormen, "Fast Permuting in Disk Arrays," *J. Parallel and Distributed Computing*, pp. 41-57, Jan./Feb. 1993.
[10] J.M. del Rosario and A. Choudhary, "High-Performance I/O for Massively Parallel Computers: Problems and Prospects," *Computer*, pp. 59-68, Mar. 1994.
[11] P.J. Denning, "Effects of Scheduling on File Memory Operations," *Proc. AFIPS Spring Joint Computer Conf.*, pp. 9-21, 1967.
[12] W. Feller, *An Introduction to Probability Theory and Its Applications*. John Wiley, 1966.

[13] *CACM Special Issue on Digital Multimedia Systems*, E.A. Fox, ed. ACM, Apr. 1991.
[14] S. Ghandeharizadeh and L. Ramos, "Real-Time Display of Multimedia Data Using a Shared-Nothing Architecture," *Computer Architecture News*, Dec. 1993.
[15] G.A. Gibson, *Redundant Disk Arrays: Reliable, Parallel Secondary Storage*. MIT Press, 1992.
[16] R. Haskin, "The Shark Continuous-Media File Server," *Proc. Compcon Spring '93*, p. 12, 1993.
[17] D.S. Hochbaum, T. Nishizeki, and D.B. Shmoys, "A Better Than 'Best Possible' Algorithm to Edge Color Multigraphs," *SIAM J. Computing*, vol. 7, pp. 79-104, 1986.
[18] R. Jain, "Scheduling Data Transfers in Parallel Computers and Communications Systems," Technical Report TR-93-03, Univ. of Texas at Austin, Dept. of Computer Science, Feb. 1993.
[19] R. Jain, K. Somalwar, J. Werth, and J.C. Browne, "Scheduling Parallel I/O Operations in Multiple-Bus Systems," *J. Parallel and Distributed Computing*, special issue on Scheduling and Load Balancing, Dec. 1992.
[20] R. Jain, K. Somalwar, J. Werth, and J.C. Browne, "Scheduling Parallel I/O Operations," *Proc. Workshop I/O in Parallel Computing Systems*, Apr. 1993.
[21] R. Jain, J. Werth, and J.C. Browne, "A General Model for Scheduling of Parallel Computations and Its Application to parallel I/O Operations," *Proc. Int'l Conf. Parallel Processing*, 1991.
[22] D. Kotz, "Multiprocessor File System Interfaces," *Proc. Second Int'l Conf. Parallel and Distributed Information Systems*, pp. 194-201, 1993.
[23] P. L'Eculyer, "Efficient and Portable Combined Random Number Generators," *Comm. ACM*, vol. 31, pp. 742-774, June 1988.
[24] Z. Lin and S. Zhou, "Parallelizing I/O Intensive Applications for a Workstation Cluster: A Case Study," *Computer Architecture News*, Dec. 1993.
[25] T. Lovett and S. Thakkar, "The Symmetry Multiprocessor System," *Proc. Int'l Conf. Parallel Processing*, pp. 303-310, 1988.
[26] E. Miller, "Input/Output Behavior of Supercomputing Applications," Technical Report UCB/CSD 91/616, Univ. of California, Berkeley, 1991.
[27] T.N. Mudge, J.P. Hayes, and D.C. Winsor, "Multiple Bus Architectures," *Computer*, vol. 20, no. 6, pp. 42-48, June 1987.
[28] J. Pasquale, "Systems Software and Hardware Support Considerations for Digital Video and Audio Computing," *Proc. 26th Hawaii Int'l Conf. Systems Science*, p. 15, 1993.
[29] D.A. Patterson, G.A. Gibson, and R.H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proc. SIGMOD*, 1988.
[30] 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," *Proc. Int'l Conf. Parallel Processing*, pp. 764-771, 1985.
[31] P.V. Rangan, H.M. Vin, and S. Ramanathan, "Designing a Multi-User Multimedia On-Demand Service," Technical Report CS92-231, Univ. of California, San Diego, 1992.
[32] A.L.N. Reddy, P. Banerjee, and D.K. Chen, "Compiler Support for Parallel I/O Operations," Technical Report RJ 7918 (#72901), IBM Almaden Research Center, 1991.
[33] A. Silberschatz and J. Peterson, *Operating Systems Concepts*. Addison-Wesley, 1988.
[34] J.E. Smith, W.C. Hsu, and C. Hsuing, "Future General Purpose Supercomputer Architectures," *Proc. Supercomputing '90*, pp. 796-804, 1990.
[35] K. Somalwar, "Data Transfer Scheduling," Technical Report TR-88-31, Univ. of Texas at Austin, Dept. of Computer Science, 1988.
[36] J.S. Vitter and M.H. Nodine, "Large-Scale Sorting in Uniform Memory Hierarchies," *J. Parallel and Distributed Computing*, pp. 107-114, Jan./Feb. 1993.

Ravi Jain received the PhD in computer science from the University of Texas at Austin in 1992. Prior to his PhD degree, he worked at Syntrex Inc., SES Inc., and the Schlumberger Laboratory for Computer Science on developing communications and systems software, performance modeling, and parallel programming. He initiated and co-chaired the first workshop on Input/Output in Parallel and Distributed Systems (IOPADS) in 1992, and he is coeditor (with John Werth and J.C. Browne) of the book *Input/Output in Parallel and Distributed Computer Systems*, published by Kluwer in 1996.

Dr. Jain joined Bellcore as a research scientist in 1992. His current research interests include design and analysis of algorithms, architectures and protocols for mobile computing and communications, including protocols for locating mobile users and techniques for mobile database access. He was technical team leader for the SCOUT system, an application which delivers personalized road traffic information to mobile users; this project is being converted into part of Bellcore's AirBoss™ product line. Recently, Dr. Jain has been engaged in research on issues arising from supporting mobile users with fixed ATM backbone networks, and phone number portability.

Dr. Jain is a member of the Upsilon Pi Epsilon and Phi Kappa Phi honorary societies, a senior member of the IEEE, and a member of the ACM.

Kiran Somalwar received the BTech degree in computer science and engineering from the Indian Institute of Technology, Madras, in 1986, and an MSEE from the University of Texas at Austin, in 1988. After his MSEE, he joined Digital Equipment Corp. in Massachusetts, working on systems software and parallel computing.

John Werth received the BS and MS in mathematics from Emory University in 1962 and 1963, respectively, and the PhD in mathematics from the University of Washington in 1968. He is assistant dean for Information Technology in the College of Natural Sciences at the University of Texas at Austin, and has been a senior lecturer and research scientist in the Department of Computer Science since September 1985. His current interests are research and development in software engineering environments for parallel programming.

Dr. Werth was an assistant professor and later assistant department head of mathematics at New Mexico State University from 1968-1975, then associate professor and later chairman of Computer Science and Electrical Engineering at the University of Nevada, Las Vegas, from 1975-1985.

Since 1992, Dr. Werth has been chair of the ACM Education Board, and a board member of the Computer Science Accreditation Board (CSAB) as well as the Computing Research Association (CRA). He was program cochair for the 1991 Annual SIGCSE Technical Symposium and the 22nd annual Computer Science Conference (CSC '94), and is program cochair for the CRA 1996 Snowbird Conference. He was guest coeditor of special issues of *Computer Architecture News* on I/O in parallel computer systems, December 1993 and December 1994. Dr. Werth is a member of the IEEE and ACM.

J.C. Browne earned his PhD in chemical physics at tThe University of Texas in 1960. Dr. Browne is a professor of computer science and physics and holds the Regents Chair #2 in Computer Sciences at the University of Texas at Austin. He taught in the Physics Department at the University of Texas from 1960 through 1964. He was, from 1965 through 1968, a professor of computer science at Queens University in Belfast and directed the Computer Laboratory. Dr. Browne rejoined the University of Texas in 1968 as a professor of physics and computer science.

Dr. Browne's research interests span parallel architectures and parallel programming, performance measurement and analysis, operating systems, and VLSI synthesis. His early research career in computational physics adds familiarity with large scale computational problems. He has been a member of the Technical Advisory Committee for Lawrence Livermore Laboratories, Los Alamos National Laboratories, the National Bureau of Standards, the National Science Foundation–Computer Research Section, and the DARPA Information Science and Technology Office. He is a fellow of the British Computer Society, the American Physical Society, and the American Association for the Advancement of Science. He has been chairman of the ACM Special Interest Group on Operating Systems and has been an associate editor of several journals. Dr. Browne has published approximately 100 papers in computational physics and 150 papers in computer science. He has given approximately 200 invited lectures to most major computer science and physics conferences.