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 the 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 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 also required in order to address the I/O bottleneck at higher levels of the system.

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 observe that, in exchange for a small execution time overhead, the approximate scheduling algorithms can provide substantial improvements in I/O completion times compared to the situation in which no scheduling is done.

[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 [17, 20, and references therin]. 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. 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.

Review of previous work reveals that most previous research on parallel scheduling deals with tasks which require only a single resource at any given time [20]. This is not relevant for scheduling of parallel I/O operations where each operation requires 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. In previous work 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 [20].

In section 2, we sketch examples of applications for which the I/O bottleneck is likely to be significant and 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. In sec. 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 algorithms which provide optimal solutions to this problem [21]. 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., 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

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 double-buffering [38]. One important innovation was to schedule the I/O operations by reordering the requests in the queues at devices [13, 38, and references therein]. We propose that the 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 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.

In general, scheduling parallel I/O operations can be effective if it is possible to make choices about the order and timing of the execution of tasks, if there are resource bottlenecks which can be utilized more efficiently by making such choices, and if the overhead incurred in making these choices is sufficiently small. Resource bottlenecks often exist in the I/O subsystem of many large-scale parallel

architectures, a typical example being the number or effective bandwidth of I/O channels [27, 31]. It also seems clear that as the gap between I/O speeds and processor speeds continues to increase, and applications generate larger I/O requests, the computational overhead of generating schedules for parallel I/O operations will become much less significant. Thus the effectiveness of scheduling hinges upon the ability to make choices about the order and timing of task execution. The nature of the I/O requests generated by applications and the capabilities provided by the system software and architecture determine whether these conditions hold. We discuss these in the following sections.

2.1 Applications

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 [7]. While the I/O bottleneck remains a central concern for these domains [8], in recent years the concern has spread to applications executing on general-purpose supercomputers as well as lower-end machines [39, 17, 1, 28, 11], a case in point being hypercube systems [9, 18, 34, 35, 36, 15, 16]. The appearance of the I/O bottleneck for wide classes of applications has motivated the argument that the performance of a supercomputer systems should be measured in terms of its data transfer rates, both within the system and across a network, instead of the peak floating-point computation rate [39, 23].

Meanwhile, a new class of applications that is rapidly becoming ubiquitous is image visualization [3] and multimedia information processing [14, 41]. It seems likely that multimedia information will be found in many, if not all, computing environments in the future [41, 31]. Multimedia information systems not only can impose much higher throughput demands than traditional applications (e.g. 5 MBps for HDTV after compression) but also introduce additional constraints, such as real-time and synchronized data transfers [42, 2], not found in the traditional applications.

2.2 The structure of I/O requests

We now consider the structure of the I/O requests generated by I/O-intensive applications of the type described above, and its effect on scheduling. In general, the stream of I/O requests generated by multiprogramming workloads is largely uncorrelated and has traditionally been scheduled dynamically at the level of device controllers. There is usually sufficient randomness among requests to avoid long queues at any given disk so that scheduling parallel I/O operations above the controller level is of little benefit. However for some applications and parallel architectures there are correlations between the I/O requests, leading to substantial resource conflicts.

For some scientific applications, such as 3D migration codes in seismic processing [10], the so-

lution progresses systematically across a coordinate space, yielding highly structured and totally predictable patterns of requests for data. For these applications scheduling of I/O requests is of little benefit since the order of the requests can be predicted in advance and thus the problem reduces to one of assigning data to storage devices so as to minimize conflicts [10, 24]. On the other hand, some families of applications, such as 3D visualization [44] and multimedia applications, pass through phases where they generate I/O requests in bursts as the locality of the data to be displayed or analyzed changes, and the entire set of requests must be satisfied before the computation can proceed. These families of applications may benefit substantially from batch-oriented scheduling of parallel I/O operations.

2.3 Capabilities required for scheduling

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. However, these are precisely the capabilities which are required to address the I/O bottleneck at higher levels of the system. It has been argued that the low-level parallelism of disk striping and RAID [32] techniques which are becoming widely available may not be effectively utilized in the long run if such facilities are not provided at higher levels. We briefly survey these arguments.

Algorithms. The fundamental algorithms used in a parallel system should ideally be asymptotically efficient in terms of the I/O activity they generate. In fact, several such algorithms have been designed, including algorithms for sorting, FFT and matrix operations. However, these algorithms do not remain optimal if disks cannot be accessed independently in parallel, e.g. if disk striping is used, where the read/write heads of all the disk drives move synchronously. Kotz [25] discusses several additional requirements that the system software and architecture must satisfy in order to utilize these optimal algorithms. (On the other hand, we observe that the system models underlying these algorithms [30] are unrealistic, and perhaps better formal models may yield more widely applicable optimal algorithms).

Compilers. Reddy et al [37] argue that unless parallel compiler technology is extended to automatically generate I/O-efficient code for general applications, individual programs may not be able to effectively utilize low-level parallelism in the I/O subsystem. Parallel compilers can exploit the structure of individual programs to employ techniques such as overlapping I/O with computation, which can reduce program execution times significantly. We observe that improved compiler techniques could potentially allow the compiler to provide hints to enable scheduling of I/O operations.

Operating systems. Some features of parallel computer operating systems may not be well-suited to support high data rate applications such as multimedia information systems. One such feature of operating system kernels is the large amount of data copying which takes place in order to perform

data transfers [31]. Separating the data from the control information about the data (or 'meta-data'), as in the IEEE Mass Storage Standard, can help bypass some copying and reduce CPU involvement by allowing some transfers to be done by DMA [31, 12]. We observe that separating data from control information is also highly desirable in order to perform effective data transfer scheduling.

Architecture. Possibly the area that has received the most attention in terms of the I/O bottleneck has been the system architecture. The use of low-level parallelism in schemes such as disk interleaving, striping, RAID, RADD, etc. is well-known [17, 43, and references therein] and will not be reviewed here. Some of these low-level techniques, however, may be in conflict with the requirements at higher levels, as discussed above. The emergence of multimedia applications such as DVA also motivates further architectural requirements. Pasquale [31] argues that the high data rates and timing constraints of DVA necessitate better control over low-level timing of I/O transfers. In particular, system I/O channels should be interruptable. We observe that these are precisely the capabilities required to schedule data transfers across the channels.

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 describe an application in which a parallel I/O scheduling problem may arise. The application we consider is image visualization but numerous other applications can be envisaged. We then state the problem, and show that it 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 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 (CAT) 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. 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.

3.2 Problem definition

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 an 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 [27] where I/O devices are connected to processors via a single shared bus.

In this paper we also consider an extension to *SimpleIOS* that is useful for modeling some practical parallel I/O architectures. We call this problem *UnitIOS*, and it is identical to *SimpleIOS* 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 [33]. 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 [29].

3.3 Definitions and problem formulation

Def. An *edge coloring* of a graph $G = (V, E)$ is a function $c : E \to N$ 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 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 [40, 22, 20], that *UnitIOS* is identical to the problem of obtaining optimal time-slot assignments in a satellite switching system [6].

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 3.1 *[5] 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. [5] 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. □

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 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. [6] 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** [6] which takes time $O(mn(m+n))$, Somalwar's algorithm [40] which takes time $O(mn^{1.5}\log n)$, and algorithm **A2** [21] 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 [20].

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. 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 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(G);      /* F is some ordered sequence of the edges in E */
2. i := 0;
3. while F ≠ ⟨ ⟩{
4.     M := { };      /* Edges 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(G);
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; for any given vertex, the order of edges incident upon it is arbitrary. Ties are broken arbitrarily.
3. Highes Combined Degree First, **HCDF**. $Order()$ sorts the edges in descending order of the sum of the degrees of their endpoints. Ties are broken arbitrarily.

We can now state some theoretical results which apply to all the greedy heuristics above omitting most proofs for brevity.

Lemma 4.1 *[20, 4] For a bipartite graph of degree d, a greedy heuristic produces a coloring using at most $2d - 1$ colors.*

Lemma 4.2 *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.*

proof. The **for** loop takes time $O(m)$. A bucket sort can be used to implement $Order()$ for **HDF** and **HCDF** [40], so sorting takes time $O(m)$. Using Lemmas 4.1 and 3.2, the result follows. □

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 [20, 4] is relatively straightforward. For **HDF** and **HCDF** it is quite involved.

Lemma 4.3 *[20] 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**, is no worse than using the heuristics **HDF** and **HCDF** which actually perform some intelligent scheduling. In section 5 we present the results of an experimental evaluation of these algorithms, as well as a divide-and-conquer optimal algorithm.

4.2 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 4.4 *[20] 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.*

The analysis of the modified greedy algorithms is slightly more complicated than for the greedy algorithm. In particular, we are not able to prove a bound that is "tight" for all inputs. Using a reasoning similar to lemma 4.2, the modified greedy algorithms run in time $O(m^2/k + md)$.

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 **A**, which is a special case of **A2** [21] for unit-weight edges implemented by Somalwar [40]. 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 each experiment, one hundred input instances (bipartite graphs) were generated using a pseudo-random number generator [26]. The range over which the parameters were varied was chosen keeping the example application scenario of sec. 3, i.e., image visualization using a shared-memory shared-bus parallel computer system, in mind.

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. (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

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

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 obviously 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 was calculated. The mean and maximum values of this percentage increase were then calculated over all the one hundred input instances. We observe that while the mean schedule length produced by the approximation algorithms does not differ significantly from the optimal, 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 upto 20% 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.

Expt. 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 4 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.

From Fig. 5 we see that while the maximum deviation of **MHDF** from the optimum schedule is limited to 10%, the **MFCFS** algorithm can produce schedules upto 50% longer. The **MHCDF** heuristic produces optimal schedules as before.

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

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

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

Expt. 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. 6 and 7 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 **MHDF** or **MHCDF** is optimal or exceedingly close to optimal, the schedule length produced by **MFCFS** can be significantly larger.

5.1 Discussion

We discuss a few features of the experimental results. In Fig. 3 we see that the maximum percentage increase in schedule length produced by **MFCFS** decreases as m increases. For $m = 100$ and $k = 4$, the optimum schedule length is determined by $\lceil m/k \rceil$ rather than d. As m increases, the optimum schedule length increases while the absolute increase in the length of the schedule produced by **MFCFS** does not vary significantly, so that the percentage increase drops.

From Fig. 2, the execution time of all the algorithms increases with the number of transfers to be scheduled. As expected, this increase is close to linear with m. It is interesting to see that **MHDF**'s execution time is very close to **MFCFS**, although the schedules it produced in this experiment were always optimal.

Figure 6: CPU time (ms) versus number of sources and sinks for $m = 200$, $k = 12$

Figure 7: Percentage increase in schedule length versus number of sources and sinks for $m = 200$, $k = 12$

In Fig. 5 we see that the maximum percentage overhead of **MFCFS** can be as high as 50%, while its mean CPU time barely differs from **MHDF** and is only slightly less than **MHCDF**. The CPU time decreases as k increases. Informally, 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.

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 [19]. 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" [19]. Note that this 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 correspon ds 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.

This example shows a general characteristic of the process of designing heuristics to solve optimization problems. There are two classes of design parameters: the amount of the (input) state space that is examined, and the complexity of the function applied to it. In the case of **MHDF** and **MHCDF**, both these parameters are changed: **MHCDF** examines more of the state space, as well as applies a slightly more complex function to it. The interchange heuristic [19] is, in turn, less myopic and more complex than **MHCDF**. One can envision a range of heuristics that could be systematically designed to cover the gap between **MFCFS** and the optimal algorithm, each heuristic being appropriate for different input parameters and applications.

It seems that the **MHDF** and **MHCDF** heuristics are extremely well-behaved, and produce schedules at most 10% longer than optimal for these experiments, 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. Since **MFCFS** essentially models the situation in which no scheduling is performed, we see that for these experiments, using a simple greedy heuristic like **MHDF** or **MHCDF** 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.

We have defined a simple parallel I/O scheduling problem which may arise in practice. For this problem, we have 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. One can envision a range of approximate algorithms that could be systematically designed to cover the gap between employing no scheduling and employing an optimal scheduling algorithm, each heuristic being appropriate for different input parameters and applications.

Our work on scheduling I/O has been carried out in the context of a general scheduling model [20] 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] J. Akella and D. P. Siewiorek. Modeling and measurement of the impact of Input/Output on system performance. In *Proc. 18th Intl. Symp. Comp. Arch.*, pages 390–399, 1991.

[2] D. P. Anderson, Y. Osawa, and R. Govindan. Real-time disk storage and retrieval of digital audio and video. *ACM Trans. Comp. Sys.*, 1993. To appear.

[3] M. Arrott and S. Latta. Perspectives on visualization. *IEEE Spectrum*, pages 61–65, Sep. 1992.

[4] 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.

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

[6] 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.

[7] H. Boral and D. J. DeWitt. Database machines: An idea whose time has passed? A critique of the future of database machines. In *Third Intl. Workshop on Database Machines*, pages 166–187, 1983.

[8] H. Boral and P. Faudemay, editors. *Database machines*. Springer-Verlag, 1989.

[9] D. Bradley and D. A. Reed. Performance of the Intel iPSC/2 input/output system. In *Proc. Conf. on Hypercubes, Concurrent Comp. and Appl.*, pages 141–144, 1990.

[10] J. C. Browne, G. E. Onstott, P. L. Soffa, Ron Goering, S. Sivaramakrishnan, Harish Balan, and Kiran Somalwar. Design and evaluation of external memory architectures for multiprocessor computer systems: Second quarter report to IBM Yorktown Heights Research Lab. Technical report, Univ. Texas at Austin, Dept. of Comp. Sci., 1987. Available from J. C. Browne.

[11] C. E. Catlett. Balancing resources. *IEEE Spectrum*, pages 48–55, Sep. 1992.

[12] S. A. Coleman and R. W. Watson. New architectures to reduce I/O bottlenecks in high-performance systems. In *Proc. 26th Hawaii Intl. Conf. Sys. Sci.*, page 5, 1993.

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

[14] E. A. Fox, editor. *CACM Special Issue on digital multimedia systems*. ACM, Apr. 1991.

[15] J. C. French, T. W. Pratt, and M. Das. Performance meaurement of a parallel Input/Output system for the Intel iPSC/2 hypercube. In *Proc. SIGMETRICS*, pages 178–187, 1991.

[16] J. Ghosh and B. Agarwal. Parallel I/O subsystems for hypercube multicomputers. In *Proc. Intl. Par. Proc. Symp.*, pages 381–384, 1991.

[17] 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.

[18] H. Hadimioglu and R. J. Flynn. The architectural design of a tightly-coupled distributed hypercube file system. In *Proc. Conf. on Hypercubes, Concurrent Comp. and Appl.*, pages 147–150, 1989.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] H. Jordan. Scalability of data transport. In *Proc. Scalable High Perf. Computing Conf.*, pages 1–8, 1992.

[24] A. Kandappan. Data allocation and scheduling for parallel I/O systems. Master's thesis, Dept. of Elect. and Comp. Eng., Univ. of Texas at Austin, 1990.

[25] D. Kotz. Multiprocessor file system interfaces. In *Proc. 2nd Intl. Conf. Par. Distrib. Info. Sys.*, pages 194–201, 1993.

[26] P. L'Eculyer. Efficient and portable combined random number generators. *Comm. ACM*, 31:742–774, June 1988.

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

[28] E. Miller. Input/Output behavior of supercomputing applications. Technical Report UCB/CSD 91/616, Univ. California, Berekeley, 1991.

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

[30] M. Nodine and J. S. Vitter. Paradigms for optimal sorting with multiple disks. In *Proc. 26th Hawaii Intl. Conf. Sys. Sci.*, page 50, 1993.

[31] J. Pasquale. Systems software and hardware support considerations for digital video and audio computing. In *Proc. 26th Hawaii Intl. Conf. Sys. Sci.*, page 15, 1993.

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

[33] 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.

[34] P. Pierce. A concurrent file system for a highly parallel mass storage system. In *Proc. Conf. on Hypercubes, Concurrent Comp. and Appl.*, pages 155–160, 1989.

[35] T. Pratt, J. French, P. Dickens, and Jr. S. Janet. A comparison of the architecture and performance of two parallel file systems. In *Proc. Conf. on Hypercubes, Concurrent Comp. and Appl.*, pages 161–166, 1989.

[36] A. L. N. Reddy and P. Banerjee. Design, analysis and simulation of I/O architrctures for hypercube multiprocessors. *IEEE Trans. Par. and Distrib. Sys.*, pages 140–151, Apr. 1990.

[37] 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.

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

[39] J. E. Smith, W. C. Hsu, and C.Hsuing. Future general purpose supercomputer architectures. In *Proc. Supercomp. '90*, pages 796–804, 1990.

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

[41] IEEE Spectrum. *Special Issue on interactive multimedia.* IEEE, Feb. 1993.

[42] R. Steinmetz. Synchronization properties in multimedia systems. *IEEE J. Sel. Areas Comm.*, page 401, Apr. 1990.

[43] M. Stonebraker and G. A. Schloss. Distributed RAID - a new multiple copy algorithm. In *Proc. 6th Intl. Conf. Data Eng.*, pages 430–437, 1990.

[44] R. H. Wolfe, Jr. and C. N. Liu. Interactive visualization of 3D seismic data: A volumetric method. *IEEE Comp. Graphics Appl.*, pages 24–30, July 1988.