DIMACS Technical Report 94-38
July 1994

Distributed Scheduling Algorithms to Improve the Performance of Parallel Data Transfers[1]

by

Dannie Durand[2] Ravi Jain[3] David Tseytlin[4]
Bellcore
445 South Street
Morristown, New Jersey 07960

[1] Presented at Workshop on I/O in Parallel Computer Systems at the International Parallel Processing Workshop, Cancun, Mexico, April 26, 1994
[2] Permanent Member
[3] Permanent Member
[4] Visitor

DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Bell Laboratories and Bellcore.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

The cost of data transfers, and in particular of I/O operations, is a growing problem in parallel computing. This performance bottleneck is especially severe for data-intensive applications such as multimedia information systems, databases, and Grand Challenge problems. A promising approach to alleviating this bottleneck is to schedule parallel I/O operations explicitly.

Although centralized algorithms for batch scheduling of parallel I/O operations have previously been developed, they are not be appropriate for all applications and architectures. We develop a class of decentralized algorithms for scheduling parallel I/O operations, where the objective is to reduce the time required to complete a given set of transfers. These algorithms, based on edge-coloring and matching of bipartite graphs, rely upon simple heuristics to obtain shorter schedules. We present simulation results indicating that the best of our algorithms can produce schedules whose length is within 2 - 20% of the optimal schedule, a substantial improvement on previous decentralized algorithms. We discuss theoretical and experimental work in progress and possible extensions.

1 Introduction

In the last 20 years, advances in both processor and architectural design have resulted in a huge growth in computational speed. The speed of I/O subsystems has not kept up. As a result, there are now several important classes of application problems for which I/O is a bottleneck. The rate at which data can be delivered from disk to compute engine is a limiting factor on how fast these problems can be solved. Three examples of such applications are multimedia information systems, scientific computations with massive datasets and databases. Hence, using parallelism to improve the performance of the I/O subsystem is an important emerging research area.

In this paper, we present work in progress on distributed scheduling algorithms to improve performance in a class of parallel I/O subsystems which can be modeled by bipartite graphs. These algorithms are based on the graph-theoretic notions of bipartite matching and edge-coloring. In Section 2, we survey recent work on parallel I/O and discuss how our work fits into this context. Relevant previous work in scheduling is also reviewed. A detailed description of the problem is given in Section 3 and relevant ideas from graph theory are discussed. A class of decentralized algorithms to solve this problem is introduced in Section 4. Simulation results are presented in Section 5 and continuing work is described. Future work is discussed in Section 6. Our results are summarized in the conclusion.

2 Background

A variety of approaches to the I/O bottleneck, from algorithmic to low level hardware solutions, have been proposed. These include both methods to improve the rate of I/O delivery to uniprocessor systems by introducing parallelism into the I/O subsystem, and methods of improving the I/O performance of multiprocessors. At the highest level, new theoretical models of parallel I/O systems are being developed [1, 33, 25, 32], allowing the study of many fundamental algorithms in terms of their I/O complexity. At the next level, new language and compiler features are being developed to support I/O parallelism and optimizations, using data layout conversion [12] and compiler hints [29]. Operating systems optimizations include layer integration and integrated buffer management to reduce copying costs, and research in file systems [9, 22]. At the lowest level, performance improvements are being achieved at the hardware and network level. Fine-grain parallelism at the disk level has been proposed through mechanisms such as disk striping, interleaving, RAID and RADD [28, 31]. Finally, to support solutions to the I/O problem, new disk architectures must be sufficiently flexible and programmable that new I/O paradigms can be implemented and tested. Kotz and Cormen [21, 11] have studied these requirements.

In this paper, we describe an approach to reducing the I/O bottleneck using scheduling techniques. In a single out-of-core application with regular data patterns, the programmer can design a specific schedule to optimize data movement as, for example, in [10]. However, in time-sharing systems or in dynamic applications with irregular data movement, more general scheduling techniques must be considered. One important innovation that addresses the

I/O bottleneck in sequential computer systems was to schedule I/O operations by reordering the requests in the queues at devices [13, 30, and references therein]. Explicit scheduling of I/O operations is also a potentially significant contributor to an integrated approach towards solving the I/O bottleneck in parallel computer systems. Given a limited bandwidth interconnect between main memory and secondary storage on a multiprocessor, judicious scheduling of data transfers can make the best use of the available bandwidth, yielding substantial performance improvement. Previous simulation studies have shown that centralized, static scheduling algorithms can reduce the time required to complete a set of data transfers by up to 40% [17, 18]. Furthermore, scheduling becomes increasingly attractive as the I/O bottleneck becomes more severe: as processor speeds increase, the overhead for computing good schedules decreases while the importance of rapidly delivering data to the processors increases.

Much of the previous work on scheduling deals with tasks which each require only a single resource at any given time [19, 14], and is not relevant for I/O operations which each require a *pre-assigned* set of multiple resources (e.g. a processor, channel, and disk) simultaneously in order to execute. For example, most previous work concerns job-shop and multiprocessor scheduling, in which tasks must acquire several resources but only one at a time (see [7, 16, 23, 26, 19] for surveys). Serial acquisition of multiple resource does not, in general, lead to optimal schedules; algorithms which simultaneously schedule multiple resources are required.

The relatively little previous work on simultaneous resource scheduling does not apply directly to our problem. It has either considered very general resource requirements, leading to problems known to be NP-complete or requiring linear programming solutions of high time complexity, or made assumptions which are not relevant for scheduling parallel I/O operations (see [19] for a survey). For example, the results of Blazewicz et al. assume that tasks do not require specific pre-assigned resource instances, which is not relevant for the I/O situation (see [5, 6, and references therein]). As another example, the ground-breaking work by Coffman et al. [8] on file transfer scheduling assumes transfers cannot be preempted once started, leading to NP-complete problems in general. For I/O operations this assumption is not necessary, since in practice most I/O transfers are performed in terms of fixed-size blocks, and preemption is permitted at block boundaries. Relaxing this assumption allows efficient algorithms to be developed for many cases. In contrast to the previous work on scheduling, 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.

3 Problem Description

In this section, we describe our parallel I/O model precisely. We then discuss how the graph theoretic concepts of edge-coloring and bipartite matching can be applied to scheduling in the context of this model.

3.1 System Model

Our system model is based on bipartite architectures such as the architecture shown in Figure 1. Here, clients (e.g., processors, workstations or disk controllers) on the left initiate

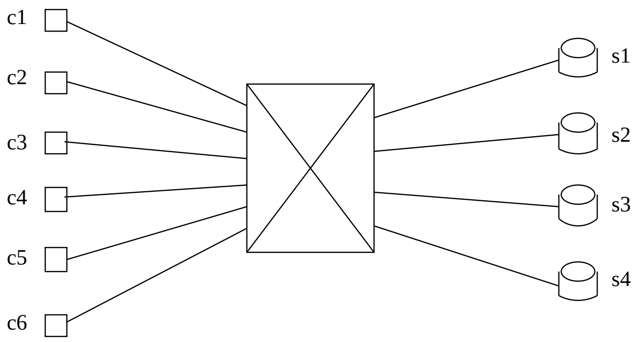

Figure 1: I/O System Architecture Abstraction

data transfers with servers (e.g., disks, disk arrays, disk controllers or file servers) on the right. Notice that the data may flow in either direction (i.e. reads or writes) but it is always the clients that initiate the transfer. Transfers take place in units of fixed-size blocks, and preemption is permitted at block boundaries. Every client can communicate with every server and, for this paper, we assume that the bandwidth of the interconnection network is not a limiting factor. Clients (and similarly servers) have no shared memory or private shared network allowing fast communication between them. Both clients and servers operate under the constraint that each unit can handle only one data transfer at a time. The architecture should be such that all clients can transmit data at the same time, and similarly for servers. This implies a common clock or regular synchronization of local clocks by a global clock signal.

We also assume that the length of a request (a message describing the transfer), is much shorter than the transfer itself. This assumption is appropriate for data intensive, I/O bound applications. This simple model captures the key issues in data transfer scheduling on a range of multiprocessor I/O subsystem architectures. Extensions to more complex models that will allow us to study various architectural refinements are discussed in Section 6.

The constraint that servers cannot handle an arbitrary number of simultaneous data transfers must be addressed in order to obtain good performance from parallel I/O subsystems. Scheduling data transfers is a good solution when the cost of scheduling is smaller than the resulting performance improvement. The total communications cost is the sum of the cost of generating the schedule and the time to transfer the data according to that schedule. Our algorithms will consist of two stages: a scheduling stage, during which a schedule is generated, followed by a data transfer stage. While the scheduling stage adds computing and communications overhead, it can potentially significantly reduce the time required in the data transfer stage by avoiding the delays associated with the arrival of conflicting transfers at the servers.

3.2 Edge Coloring and Scheduling

For an illustration of data transfer scheduling, consider the example of the bipartite transfer graph, G, shown in Figure 2. Here the vertices are the clients and servers shown in Figure

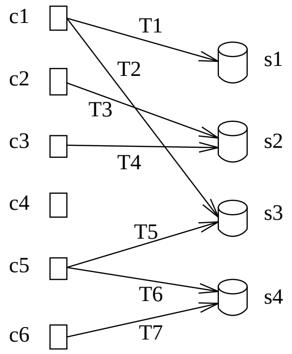

Figure 2: Data Transfer Graph

1. The edges $(T1, T2, \ldots, T7)$ are pending I/O transfers between the clients and the servers. (Note that while none are shown in Figure 2, multiple edges are allowed in G since more than one transfer may be pending between any given client, server pair. The algorithms discussed below are all designed to work for bipartite graphs with multiple edges.)

A schedule is a partition of the set of transfers into subsets such that all transfers in each subset can be executed simultaneously. The smaller the number of the subsets, the shorter the schedule length. Thus, in Figure 2, transfers $T2$ and $T5$ are competing for server $s3$ and so cannot take place at the same time. In addition, $T2$ cannot be scheduled at the same time as $T1$ since they both require $c1$. Finally, $T5$ and $T6$ cannot take place at the same time because they share $c5$. A legal schedule must take all of these constraints into account. If $T1$ and $T6$ are scheduled simultaneously in the first transfer, three steps will be needed to complete the schedule since $T2$ and $T5$ cannot occur at the same time as shown in Figure 3 (a). However, if $T1$ and $T5$ are scheduled together at the first step, the schedule can be completed in two steps (Figure 3 (b)).

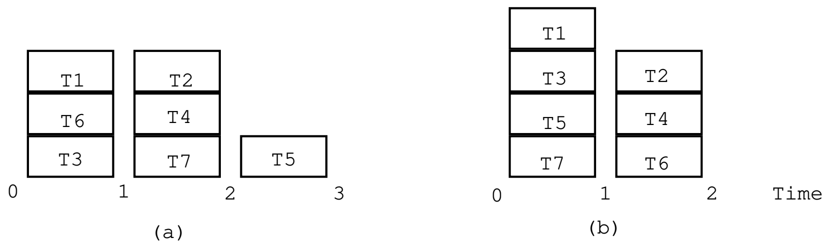

Figure 3: Schedules represented as Gantt charts.

We exploit two problems from graph theory that are relevant to the scheduling problem: bipartite matching and edge-coloring. Consider a bipartite graph $G = (A, B, E)$ where A

and B are the set of vertex partitions, and E is the set of edges such that every edge has one endpoint in A and one in B. A *matching* in G is a subset of E, with the property that no two edges in the matching share a vertex. Note that since a matching shares no vertices, a matching in the transfer graph constitutes a set of transfers that can be executed simultaneously. For example, the set of transfers $\{T1, T3, T6\}$ in Figure 2 is a matching since those three edges have no client and no server in common. A matching is *maximal* if no edge can be added to the matching without creating a conflict at one of the vertices. A matching is *maximum* if there is no other matching in the graph which is larger. The matching $\{T1, T3, T6\}$ is maximal since no edges can be added to it with out destroying the matching property. However, it is not a maximum matching because the set $\{T1, T3, T5, T7\}$ is a larger, legal matching.

An *edge-coloring* of a graph G is an assignment of colors to edges in such a way that no two edges of the same color share a vertex in G. Note that each color in an edge-coloring is a matching on G. Hence, each color in the coloring represents a set of transfers that can occur simultaneously and the coloring as a whole is a schedule for the transfer graph. As an example, the two schedules shown in Figure 3 constitute two different edge-colorings of the graph shown in Figure 2. Recall that the *degree* of a vertex is the number of edges incident upon it, and the degree of a graph, or *graph degree*, is the maximum degree of any vertex. It is well known [4] that Δ colors are necessary and sufficient to edge-color a bipartite graph of degree Δ.

4 A Distributed Scheduling Algorithm

In this section, we present a parameterized class of distributed bipartite edge coloring algorithms to solve the data transfer scheduling problem described above. The metrics used to distinguish the algorithms are the length of the schedule generated and the time required to generate that schedule. We expect that the choice of the "best" scheduling algorithm from the class will be strongly dependent on the characteristics of the particular architecture under consideration. In this article, we concentrate on schedule length. Work on the complexity of generating schedules will be reported in a future article.

4.1 Algorithm Design

The data transfers are effected by a scheduling stage, during which clients and servers exchange *messages* to generate a schedule, followed by a transfer stage, in which the actual data *transfers* take place. The algorithms are based on a simple bidding scheme similar to those used by [27, 2]:

```
For all Clients:  Color an incident edge chosen uniformly at random.
For all Servers:  Resolve conflicts uniformly at random.
```

In the first step, each client chooses an edge uniformly at random from those adjacent to it and sends a *proposal* message to the appropriate server, requesting a transfer. Since each client chooses only one of its adjacent edges, no conflict occurs at the clients. However, each server may receive more than one proposal. Hence in the second step, each server resolves conflicts by choosing one from its incoming requests uniformly at random as the winner. It sends a *response* message to the winning client confirming the request. We call one execution of this two-step bidding process a *pass*, i.e., an interval during which the clients assign colors to edges, followed by the proposal messages, and an interval during which the servers choose winners, followed by the response messages.

This two step bidding strategy results in a matching (or a set of simultaneous transfers), but not necessarily a maximal matching. We wish to increase the number of edges in the matching achieved. We consider two approaches: using heuristics and multiple passes. In the case of heuristics, instead of selecting an edge uniformly at random, the clients can use heuristics to color an edge. For example, information about what happened in previous passes can be used to guess which edges have a higher probability of success. In addition, instead of selecting the winner uniformly at random, the servers can use heuristics. In the case of multiple passes, clients who lost their bid on the first pass can make bids using the same color on different edges in subsequent passes. The two methods can, of course, be combined.

In order to obtain a complete edge-coloring, the bidding process is repeated until the entire graph is colored. The pseudo-code for a uniprocessor simulation of our algorithm is shown in Figure 4. Note that in Figure 4, instead of using a single color during the bidding process, we use *Ncolors* colors. We call a *phase* the execution of a number of passes such that during each pass, the clients choose from the same set of colors while assigning colors to edges. Thus in Figure 4, each iteration of the **For** loop corresponds to a pass, and each iteration of the **While** loop corresponds to a phase. By using multiple colors per phase, instead of computing one matching in each phase, we will compute *Ncolors* matchings in each phase. Each client chooses *Ncolors* adjacent edges (or less if its degree is less than *Ncolors*) and assigns fresh new colors to these edges. Up to *Ncolors* conflicts can now occur at each server. Each server chooses a winner for each conflict in the phase. Using more than one color in each phase can reduce the number of phases required to color the graph, but, as we shall see in Section 5, potentially results in longer schedules.

The parameters of the algorithm are shown in bold face in Figure 4. Parameters *H1* and *H2* are the coloring heuristics discussed above, *Npasses* is the number of passes per phase, and *Ncolors* is the number of colors used in each phase. For *Ncolors* > 1, we implement the proposal of colors for edges during multiple passes as follows. At each pass, the client chooses a color uniformly at random from the set of colors which have not yet been successfully assigned to any edge, and then proposes that color for an edge chosen uniformly at random ¿from the set of edges which have never been proposed that color.

```
While (G = (A, B, E) is not empty)
   {
   Get Ncolors new colors.
   For i = 1 to Npasses
      {
      For all clients:  Assign Ncolors to edges(s)
                        chosen by strategy H1.
      For all servers:  Resolve conflicts
                        by strategy H2.
      }
   Delete colored edges and vertices of zero degree from G.
   }
```

Figure 4: A parameterized class of scheduling algorithms

4.2 Resolving Conflicts in the Scheduling Stage

In the algorithm design above conflicts can occur if more than one client sends a request to the same server simultaneously, or if more than one server responds to the same client.

There are several possibilities to resolve or avoid these conflicts. Since the messages are assumed to be much shorter than the data transfers, it is feasible that a hardware solution can be used, in which the clients and servers have a separate low bandwidth connection and a small amount of buffering for exchanging request and response messages. Another alternative is that the time interval of each phase is divided into slots, with clients sending messages during pre-assigned *slots*. Thus, for instance, if the number of servers and clients is equal, we might use a slotted communication protocol where at slot i each client j sends a message to server $(j + i) \ \ mod \ \ n$. A third alternative is to have a *retry* communication protocol, in which clients choose messages to send at random, and use an exponential backoff scheme similar to Ethernet if a conflict is detected.

As an example, slotted and retry protocols for sending messages have been used in the BBN TC2000 parallel computer [3]. The protocols are provided at a hardware level transparently to the user, and their relative performance for different workloads has been studied. In the current work, we assume that a suitable mechanism has been chosen and implemented in the underlying architecture, and focus on using these facilities to schedule the much longer data transfers.

4.3 Heuristic Design

Given the algorithm design above, it is possible to experiment with different heuristics. For this paper, we focus on one heuristic which has proven to be very effective in centralized algorithms [18]. The heuristic we use is called *Highest Degree First* (*HDF*). With *HDF*, clients continue to select an edge to color uniformly at random in line 7. However, when each client sends its proposed colors to the servers, it now includes its current degree as well. The servers, in line 9, do not choose a winner uniformly at random. Instead, the client with highest degree becomes the winner, with ties broken arbitrarily. The intuition behind this heuristic is that, since the optimal schedule length equals the graph degree, a heuristic that reduces the graph degree as fast as possible is likely to promote shorter schedules.

There are numerous heuristics which can be designed, with the choice of heuristic being guided by the communication and computation characteristics of the application and architecture under consideration. *HDF* represents one type of heuristic, in which extra computation is performed to try to reduce communication time. On the other hand, we are considering heuristics where some additional communication penalty may be paid, e.g., heuristics where servers provide clients with additional information in their responses, so as to help clients make better choices at subsequent passes. For this paper, we focus on the *HDF* heuristic.

4.4 Related Work

Some related distributed communications algorithms have been presented by [27, 2, 15]. Anderson et al. [2] describe a matching-based scheduling algorithm for routing data cells from inputs to outputs in an ATM communications switch. The scale of their problem is quite different since their algorithm must be implemented in hardware and complete in real time. Nevertheless, the approach is interesting and can offer some insight into the parallel data transfer problem.

Gereb-Graus and Tsantilas [15] have presented some work on distributed communication algorithms for optical computers. In the optical communication parallel (OCP) computer model, if two or more messages arrive at the same port simultaneously, neither message is accepted. If a single message arrives at a server, the server sends back an acknowledgement. If the sender receives no acknowledgement, it concludes that a collision occurred at the receiving port and tries again later. Gereb-Graus and Tsantilas' algorithm does not use scheduling techniques to avoid these conflicts. Instead, they use a probabilistic analysis to decide with what probability senders should refrain from sending a packet in order to reduce collisions and maximize throughput. This approach does not incur the cost of scheduling, but may result in substantially longer communication times for certain architectures. This tradeoff will be discussed in detail in Section 5.3.

Panconesi and Srinavasan [27] presented a distributed algorithm for edge-coloring on general graphs. They developed sophisticated mathematical tools which allow them to prove rigorous results concerning the complexity of their algorithm and the number of colors used. There are two major differences between our algorithm and the algorithm of Panconesi and Srinavasan [27], henceforth called *PS*. First, *PS* does not use heuristics or multiple passes to improve the matchings obtained in each phase. Second, as it stands, the algorithm is not suitable for implementation on a fully distributed system because it requires some global information. This is because *PS* uses a variable number of colors, $Ncolors = \Delta(i)$, in each phase, where $\Delta(i)$ is an estimate of the graph degree in phase i. Thus in the first phase, *PS* requires each sender to know the graph degree. In subsequent phases, *PS* uses a probabilistic analysis to estimate that, for any fixed $\epsilon > 0$, $\Delta(i) \leq ((1+\epsilon)\Delta(i-1))/e$ with high probability. This estimate is valid when $\Delta(i)$ exceeds a threshold $\Delta_t = \log^{2+\delta} n$, where n is the number of vertices in the graph, and δ is a parameter which affects the probability with which the estimate of $\Delta(i)$ is accurate. When $\Delta(i) < \Delta_t$, *PS* switches to a distributed algorithm proposed by Luby [24]. Luby's algorithm, which is a *vertex*-coloring algorithm, is used to color the line graph of G, thus requiring up to $2\Delta_t - 1$ colors. Panconesi and Srinavasan show analytically that *PS* requires at most $1.6\Delta + \log^{2+\delta} n$ colors to edge-color the entire graph.

4.5 A Decentralized Algorithm Requiring Global Information

Note that, unlike the algorithm of Gereb-Graus and Tsantilas [15] and the *PS* algorithm of Panconesi and Srinavasan [27] described in Section 4.4, our algorithms do not rely upon any global information about the graph degree in order to operate correctly. For the purposes of comparison, we implemented an algorithm which does use global information. This algorithm is a modified version of *PS*, called *mPS*, which sets $Ncolors = \Delta(i)$ at phase i, where we use a lower bound on Panconesi and Srinivasan's formula to estimate $\Delta(i) = \Delta(i-1)/e$ for phase $i > 0$. However, unlike *PS*, the modified algorithm *mPS* does not switch to Luby's algorithm when $\Delta(i)$ drops below the threshold Δ_t. The algorithm *mPS* instead continues to use the *PS* scheme to color edges using the formula $Ncolors = \Delta(i-1)/e$ at phase i, until all edges have been colored. We suspect that switching over to Luby's algorithm will be expensive (at least in terms of constant factors), since it requires first constructing a line graph. In addition, since using Luby's algorithm for vertex coloring can require upto $2\Delta_t - 1$ colors for coloring edges, it may not produce good edge colorings in practice.

5 Experimental Results

In this section, we present preliminary results on schedule lengths generated by some of the algorithms in our class. These results were produced with a uniprocessor program, sketched as pseudo-code in Figure 4, which simulates the functionality of the scheduling algorithms but not their computation and communication costs. We report on the results of three series of experiments:

1. The effect of varying *Ncolors*, the number of colors per phase, on the schedule length.
2. The effect of the servers using the *Highest-Degree-First* heuristic on the schedule length.
3. The effect of varying *Npasses*, the number of passes per phase, on the schedule length.

For comparison purposes, we performed a series of experiments with the *mPS* algorithm also.

These series of experiments were run on NxN bipartite graphs of size 16x16, 32x32 and 64x64 respectively. Graphs were generated such that each graph has $N^2/2$ edges selected at random. Multiple edges are permitted. The results of these series of experiments are shown in Figures 5, 7 and 9. The y-axes of these plots show the mean normalized schedule length, where the mean normalized schedule length is the ratio of the schedule length generated by our simulated algorithm to the optimal schedule length, averaged over the number graph colorings. Recall that the optimal schedule length for a particular graph is the degree of the graph. Each data point in the plots represents the average of 100 experiments: ten colorings of each of ten graphs. We computed 95% confidence intervals, shown as error bars, to verify that the trends seen are a function of the independent variables and not due to statistical variations. In the following sections we discuss our results in detail.

5.1 Varying the Number of Colors per Phase

Experimental results showing the impact of *Ncolors* on schedule length are given in Figure 5. The solid line shows the schedule lengths produced by the modified version of Panconesi

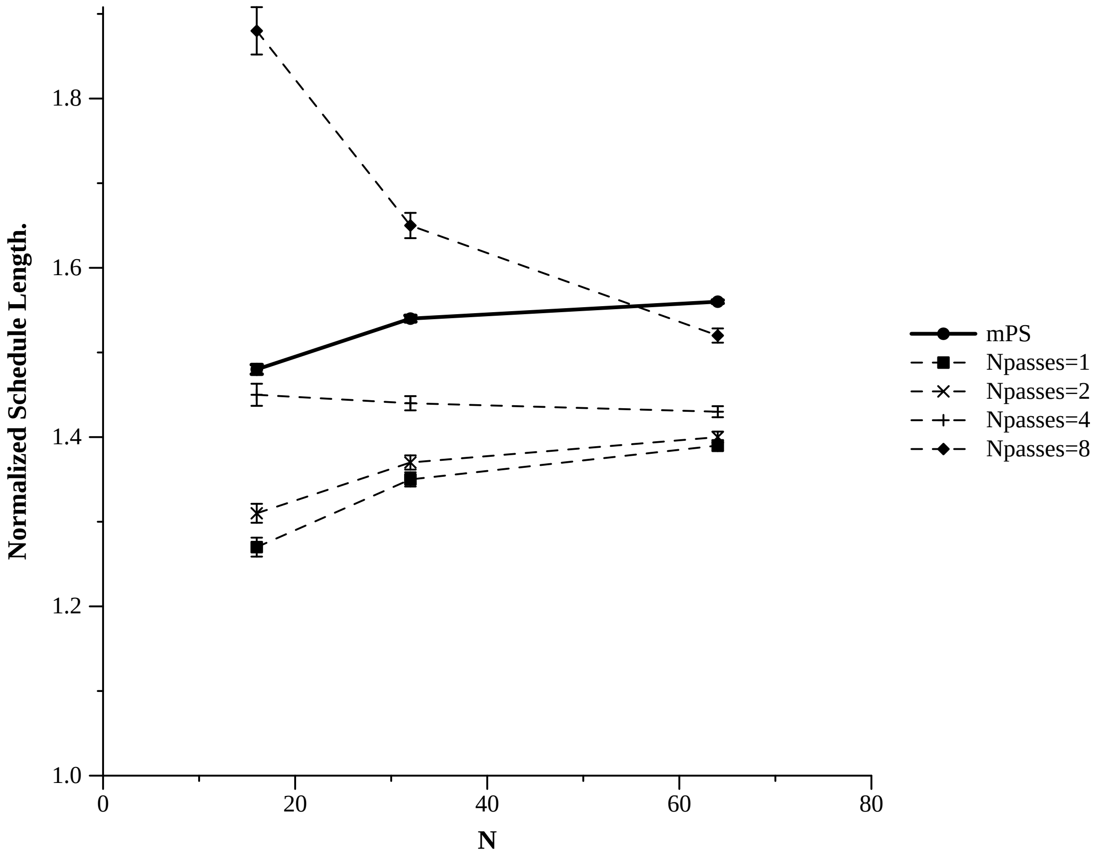

Figure 5: Impact of *Ncolors* on Schedule Length.

and Srinavasan's algorithm, *mPS*. The dotted lines were generated with our algorithm using

$Npasses = 1$ and $H1 = H2 =$ "uniformly at random". The parameter $Ncolors$ takes on the values 1, 2, 4 and 8. As can be seen in the figure, the schedules tend to get longer as $Ncolors$ increases. This effect is much more pronounced for small graphs than for large graphs.

We can understand this effect by looking at the histogram in Figure 6, which shows the number of edges which were assigned to each color for a single experiment. The figure shows

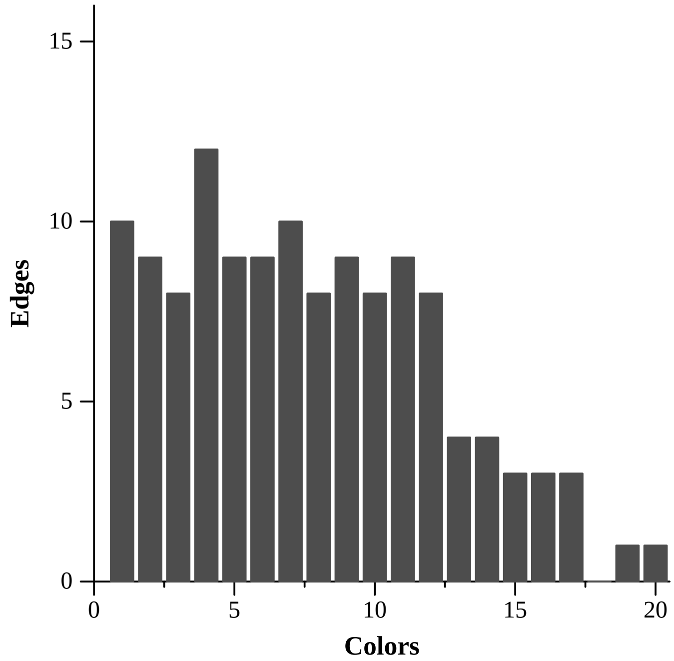

Figure 6: Histogram of Matching Density

that the later matchings (those associated with colors greater than 12) are much sparser than the earlier ones and the color 17 is not used at all. This occurs when $\Delta(i)$, the graph degree in phase i, drops below $Ncolors$. In this case, the algorithm has more colors than it needs to color the remaining edges and hence, on average, the algorithm does not assign as many edges to each color. Because the algorithm is non-deterministic, a given color may receive no edges at all, as occurred with color 17 in Figure 6. In a distributed situation, there is no central agent that can recognize the presence of an unused color, or "hole", in the schedule, resulting in an unnecessary but unavoidable delay.

When $Ncolors = 1$, the degree of the graph is never less than $Ncolors$ and holes never occur. As $Ncolors$ increases, the number of phases for which $\Delta(i) < Ncolors$ increases and hence the number of holes and underutilized colors increases, resulting in longer schedule lengths. For larger graphs, the percentage of phases for which this occurs is much smaller than for the smaller graphs so that the negative impact of a large value of $Ncolors$ is reduced.

We have shown in Lemma 7.1 (see the Appendix for a proof) that the maximum number of holes which can occur is $\Delta * (Ncolors - 1)$, when $Ncolors$ colors are used at each phase and the graph degree is Δ. Further, given any Δ and $Ncolors$, it is possible to generate a graph for which, in the worst case, this bound is met.

Obviously, the likelihood of $\Delta * (Ncolors - 1)$ holes being created is very low, as is confirmed by our experimental results. It is also possible to derive smaller upper bounds for restricted families of graphs which may be likely to occur in practice. However, since holes in the data transfer stage are very wasteful in terms of communication bandwidth, Lemma 7.1 helps motivates the development of techniques for improving the matchings generated by the algorithm, such as the use of multiple passes per phase, as discussed in the following section.

It is surprising that *mPS* yields relatively poor schedules, since *mPS* reduces the number of colors used in each phase to $Ncolors = \Delta(i)$, which should yield schedules with no holes. We posit that although $Ncolors = \Delta(i)$ is the minimum number of colors needed to color the remaining graph at each phase, and we used a lower bound on Panconesi and Srinivasan's formula for the estimate of $\Delta(i)$, it is too many colors to generate good matchings.

5.2 Using Heuristics and Multiple Passes to Improve Matchings

We ran one series of experiments using more than one pass to obtain a matching (*MPASSES*). We ran another series of experiments to obtain better matchings using the *Highest-Degree-First* heuristic (*HDF*). The impact of *MPASSES* and *HDF* on schedule length when $Ncolors = 1$ is shown in Figure 7. As before, the thick line represents the *mPS* algorithm. The dashed line shows the improvement in schedule length gained by using *HDF* and the four dotted lines show the effect of using 1, 2, 4 and 8 passes, respectively. Not surprisingly, as we increase the number of passes the schedule length decreases. *HDF* produces schedule lengths that are shorter than those obtained with two passes but not as good as those obtained with four passes. However, the communication cost associated with *HDF* is much smaller, so that for many architectures *HDF* may represent a good compromise between the cost of scheduling and the length of the schedule obtained. The 8-pass algorithm gives the best schedule lengths, within 5% of optimal. This represents a 20% improvement over the basic algorithm ($Ncolors = 1$, $Npasses = 1$ and no heuristics) and a 30% improvement over *mPS*. In contrast, *HDF* yields schedules between 5% - 17% of optimal depending on graph size.

The histograms in Figure 8 show how *HDF* and *MPASSES* behave differently even when they achieve the same result. This figure shows the results of two experiments in which the same graph was colored twice, once by *HDF* and once by *MPASSES* with $Npasses = 3$. The black bars show the size of the matchings obtained with the *HDF* algorithm whereas the grey bars show the matchings obtained using *MPASSES*. In each case, the same schedule length was obtained. However, the histograms have very different shapes. The *HDF* histogram indicates good load balancing: all the matchings are approximately the same size except at the very end of the algorithm. On the other hand, using *MPASSES*, better matchings were obtained at the beginning of the execution. As the algorithm progresses, the size of the matchings drops off sharply. In other words, this algorithm performs best when the graph is dense. If we apply this algorithm to a dynamic situation where new transfer requests arrive continually, *MPASSES* may perform even better.

Figure 9 shows how the number of colors per phase affects the performance of *MPASSES*

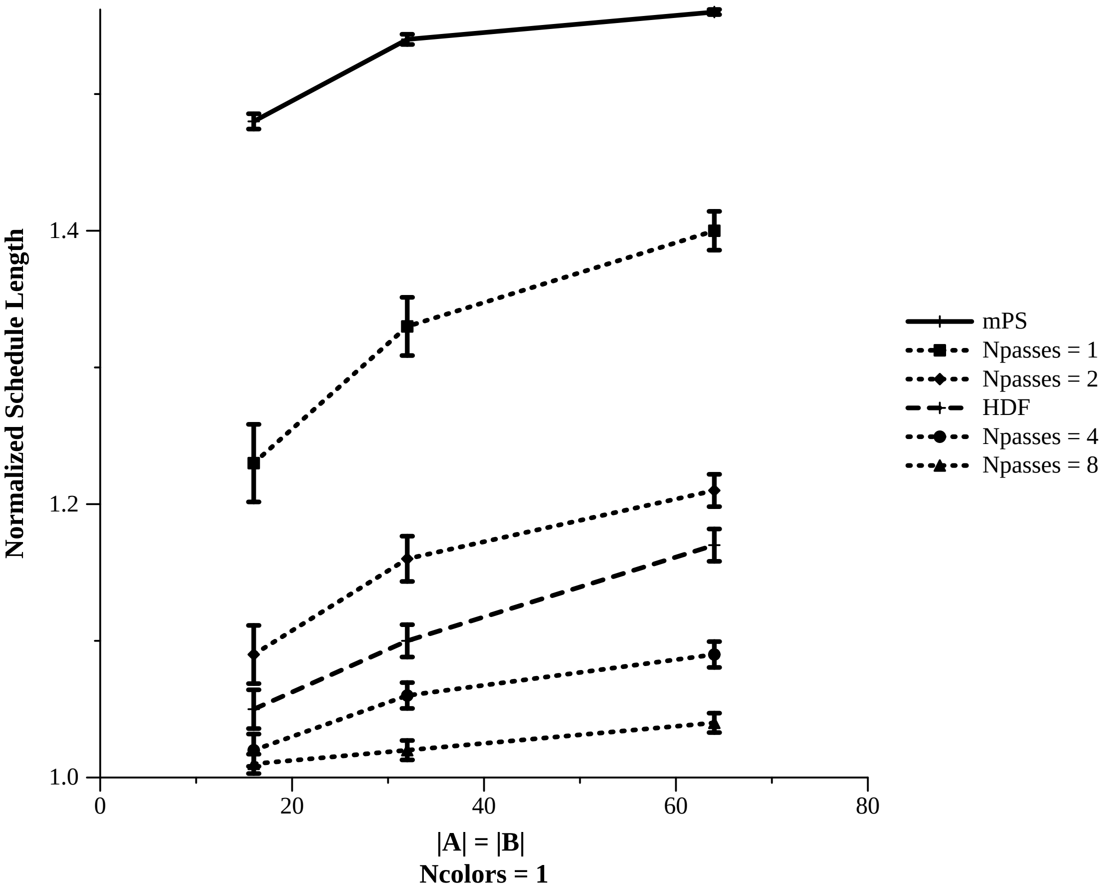

Figure 7: Impact of *HDF* and *MPASSES* on Schedule Length.

and *HDF*. In this figure, *Ncolors* increases along the x-axis. The solid line shows how schedule lengths obtained with *HDF* vary as *Ncolors* increases. The dashed and dotted lines show the effect of *Ncolors* on schedule lengths obtained by 1, 2, 4 and 8 passes. We see that *HDF* is much more sensitive to large values of *Ncolors* than *MPASSES*. This is because *HDF* cannot compensate for the poor utilization of colors that occurs once $\Delta(i)$ drops below *Ncolors*. This suggests that *MPASSES* would be better for situations in which *Ncolors* is large.

5.3 Comparison with Related Work

Here we discuss these results in the context of other work published in this area. For $Ncolors = 1$, our algorithms yield schedule lengths of 1.05Δ –1.2Δ for HDF and 1.02Δ –1.09Δ for four passes, a substantial improvement over the algorithms Panconesi and Srinavasan [27], whose schedule lengths have been shown to be $\sim 1.6\Delta$ theoretically. In our experiments, for the situations we studied, we saw that the modified algorithm *mPS* also produced schedule lengths of $\sim 1.6\Delta$. However, because *PS* uses $Ncolors > 1$, it may generate schedules faster than our algorithms. The *PS* approach is interesting because using $Ncolors > 1$ will

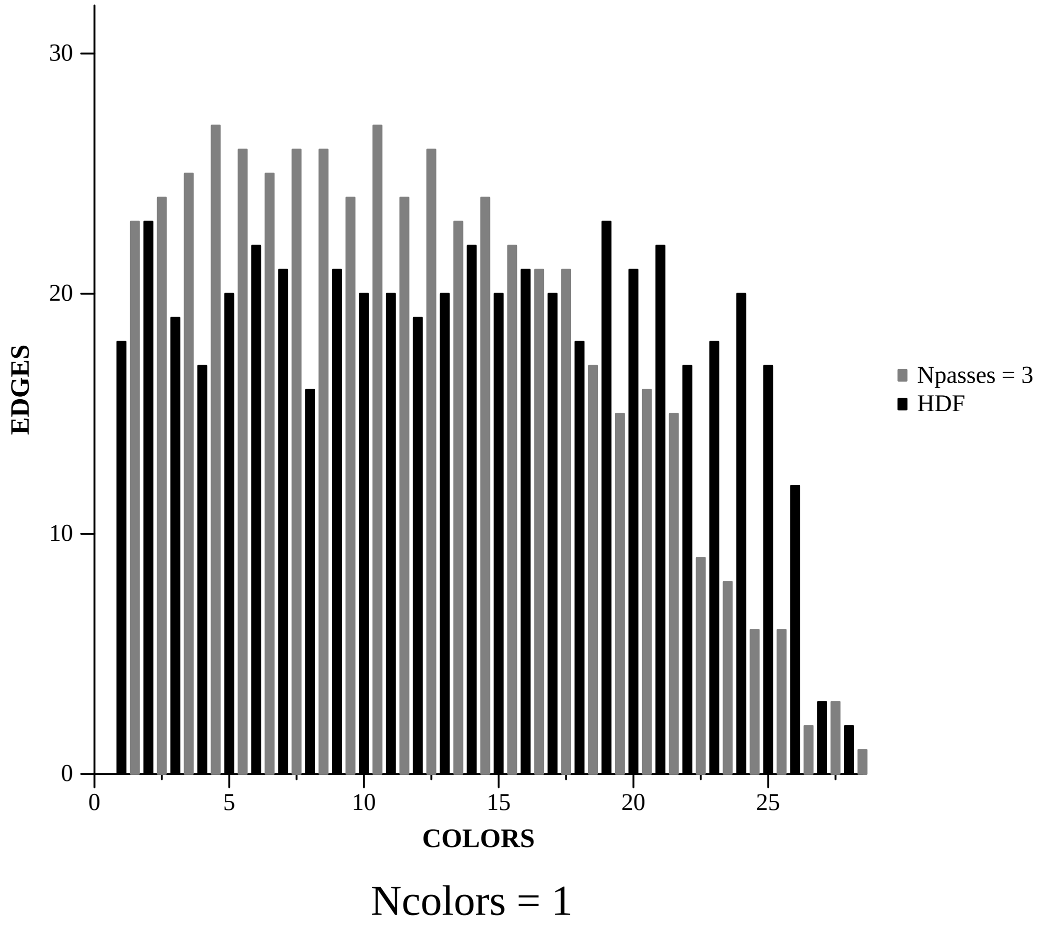

Figure 8: Comparison of *HDF* with *MPASSES*

.

generate schedules faster, and reducing *Ncolors* adaptively as the algorithm progresses will offset the penalty in schedule length. (Note that in a situation where I/O requests arrive continuously and the graph to be colored is created dynamically, it would be difficult to apply this method). It would be interesting to determine experimentally the appropriate rate at which to reduce *Ncolors* for different types of graphs. For the graph sizes we studied, our results suggest that the choice of $Ncolors = \Delta(i)$ is too large.

The work of Gereb-Graus and Tsantilas [15] gives some intuition into the time required for unscheduled transfers. Recall that in their algorithm, described in Section 4.4, there is no scheduling stage. Gereb-Graus and Tsantilas estimate the communication time to be, with high probability:

$$T_{GGT} = \frac{e}{1-\epsilon}\Delta + c_1\sqrt{\Delta \ln N} + c_2 \ln N \ln\ln N + c_3 \ln N + c_4 \geq 2.7\Delta,$$

where the choice of $\epsilon < 1$ determines the probability with which the analysis holds and c_1, c_2, c_3 and c_4 are constants which depend on ϵ, N and Δ. Unfortunately, we have no

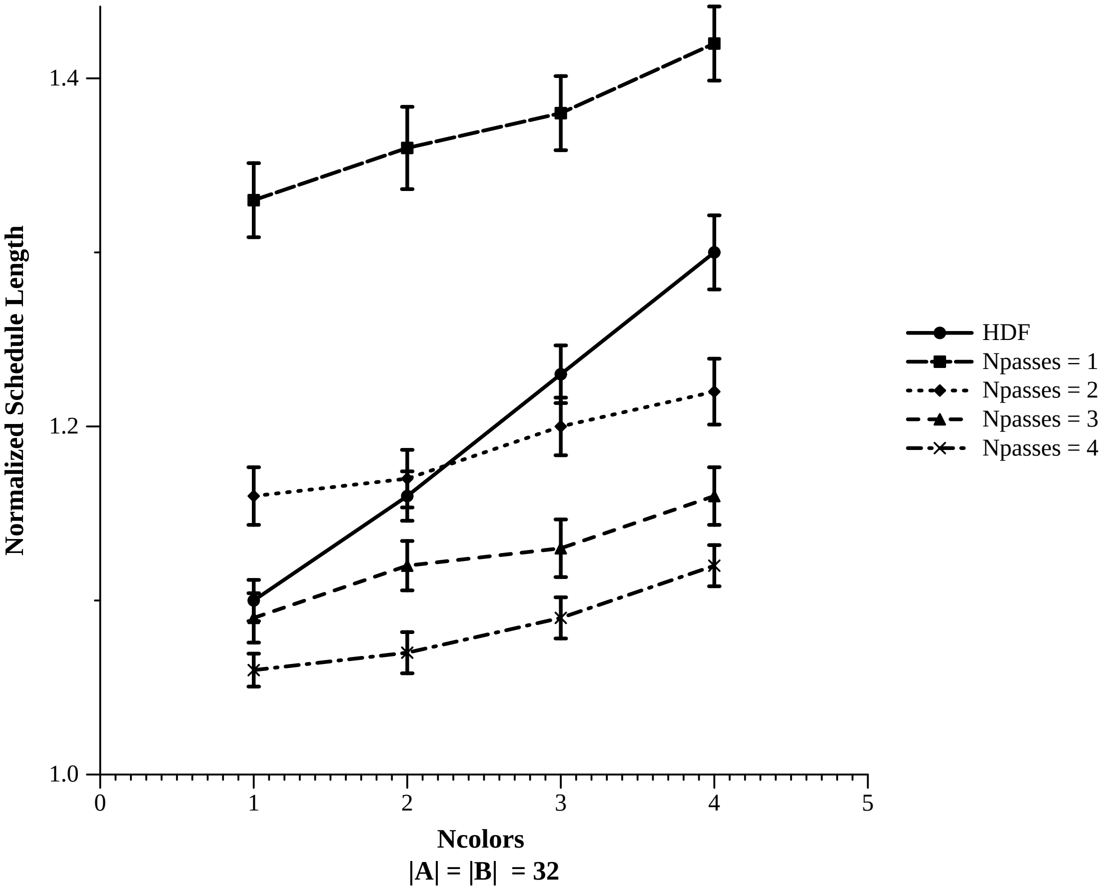

Figure 9: Impact of Ncolors on *MPASSES* and *HDF*.

experimental measurements of the speed of this algorithm. In general, which algorithm is best will depend on the ratio of the request size to the transfer size, which in turn will depend on the architecture and application at hand.

The work of both Panconesi and Srinavasan and Gereb-Graus and Tsantilas is based on probabilistic analyses that predict how the degree of the graph will decrease with time. Panconesi and Srinavasan use this approach to estimate the number of colors needed to color the remaining graph. Gereb-Graus and Tsantilas need this information to estimate the probability of collisions occurring at the servers. In a distributed context it may be difficult to obtain the graph degree, particularly if the I/O operations belong to different programs executing concurrently on a multiprocessor, e.g. in time-shared multimedia information systems. This is especially so if one considers an on-line scheduling situation in which the graph is created dynamically as new requests arrive. Our algorithms, based on a fixed number of colors per phase, do not have this problem.

6 Extensions

Currently, we are studying the communications and computational complexity of these algorithms analytically in order to obtain a better understanding of the tradeoff between the cost of generating a schedule and the length of the schedule generated. In addition, we plan to extend our algorithms to the dynamic case where new transfer requests arrive during the scheduling process. This requires that we address fairness issues. With the introduction of heuristics in lines 7 and 9 of Figure 4, there is a danger of starvation or reduced quality of service, especially in the presence of uneven work loads.

The simple model presented in Section 3.1, captures the essentials of an important class of parallel I/O subsystems. It can be easily generalized to express a wider class of more realistic architectures. For example, it is straightforward to extend our model to include multiple ports or I/O buffers. If a disk, d, can support M simultaneous transfers, the algorithm described in Section 4 can be modified so that then d will not reject proposed transfers unless it receives more then M proposals.

Important extensions to our work include the modeling of more complex I/O interconnection architectures, e.g. to model multimedia mail in a workstation cluster. It may be possible to extend our results for bipartite topologies to general graphs, as in the approach taken by [20]. Similarly, limitations in the bandwidth of the network can be considered, as well as variable-length transfers in which preemption is not permitted, as in [8].

Finally, we will consider this work in the context of disk striping. Some research has indicated that striping gives excellent speedups for some applications and is difficult to exploit for others. For example, del Rosario et al. [12] have shown that I/O performance is highly sensitive to the interaction of stripe size and in-core data layout. Hence, for those cases where disk striping is not effective, I/O scheduling may be promising. Thus, for I/O subsystems based on several independent RAID systems, I/O scheduling would exploit high level parallelism between RAIDs, while striping would exploit low level parallelism within RAIDs.

7 Conclusions

We have introduced a class of distributed scheduling algorithms for data transfers that can be parameterized to suit architectures and applications with different communication and computation trade-offs.

We have considered the effect of a potential improvement in the time required to generate a schedule by computing several bipartite matchings simultaneously (multiple colors per phase). We have shown that this approach results in longer schedules, in general, partly due to the formation of holes, i.e., time intervals during which no transfers take place. We have obtained a bound on the number of holes which can be created. Next, we have presented experimental results on the efficacy of two approaches to reducing the schedule length: *Multiple Passes* (*MPASSES*) and the *Highest-Degree-First* (*HDF*) heuristic. *HDF* and *MPASSES* gave about the same results when $Npasses = 3$. Each approach has its advantages. *HDF* has lower communication costs and provides good load balancing. *MPASSES* gives better results if the number of passes is high enough, is less sensitive to using multiple colors per phase and is more appropriate for dynamic situations where new transfer requests arrive continually. Our experimental results for the situations we studied showed that *MPASSES* and *HDF* produce schedules which are within 2 - 20% of optimal, constituting an improvement of up to 30% over previous decentralized algorithms. An analytical study to determine the best compromise between scheduling time and schedule length is currently in progress.

Acknowledgements

We thank Alessandro Panconesi for useful discussions in the initial stages of this work. Bill Aiello, Sandeep Bhatt and Mark Sullivan of Bellcore also provided helpful feedback.

Appendix

A color is called a *hole* if, for some phase, it is available for proposals by the clients but no edge is assigned that color. Here we derive an upper bound on the number of holes which can occur in a schedule generated by by our algorithm for the case where both the clients and the servers choose edges uniformly at random and the number of passes per phase is one.

Edges colored during a pass are deleted at the end of the pass, as are vertices of zero degree. For any phase, define the degree of a (client or server) vertex to be its degree at the start of the phase, i.e., before the clients make proposals. We number consecutive phases starting with zero. For all phases $i \geq 0$, let the *client degree* at phase i, denoted $\delta_c(i)$, be the maximum degree of any client at that phase, and the *server degree*, denoted $\delta_s(i)$, be the maximum degree of any server. Clearly the graph degree at phase i, $\delta(i) = \max\{\delta_c(i), \delta_s(i)\}$. (By definition, $\delta(0) = \delta$).

Lemma 7.1 *The maximum number of holes which can occur in a schedule generated by the algorithm when both clients and servers chooses edges uniformly at random, the number of passes per phase is one,* Ncolors *colors are used at each phase and the graph degree is δ, is* $\delta * (\text{Ncolors} - 1)$. *Given any δ and* Ncolors*, it is possible to generate a graph for which, in the worst case, this bound is met.*

proof. A hole occurs at any phase i only if $Ncolors\ > \delta_c(i)$. (Note that the converse does not hold, since different clients may collectively propose all available colors, and all such proposals may be accepted). Let $h(i) \geq 0$ denote the number of holes created at phase i. Clearly, for all i, $h(i) \leq \max\{0, Ncolors - \delta_c(i)\}$.

Now consider the first phase $j \geq 0$ such that $Ncolors > \delta_c(j)$. For all phases $i \geq j$, $h(i) \leq Ncolors - \delta_c(i)\ \leq Ncolors - 1$. In addition, for all phases $i \geq j$, each server will receive at least one proposal, so that each server's degree will decrease by at least one. Thus at most $\delta_s(j)$ phases are required to color the graph remaining at phase j. It follows that the total number of holes which can be generated, $H(G) = \sum_{i\ \geq\ 0} h(i)\ \leq \delta_s(j) * (Ncolors - 1)$ $\leq \delta * (Ncolors - 1)$.

One simple graph for which this bound is exact is the tree where the client degree is 1 and the server degree is δ. In the worst case, for all phases, all clients propose the same color. Then δ phases are required to color the graph, and at each phase $Ncolors - 1$ holes are created. □

References

[1] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, pages 1116–1127, Sep. 1988.

[2] T. E. Anderson, S.S. Owicki, J. B. Saxe, and C. P. Thacker. High-Speed Switch Scheduling for Local-Area Networks. *ACM Transactions on Computer Systems*, 11(4):319–352, November 1993.

[3] Inside the TC2000, 1990.

[4] Claude Berge. *Graphs*. North Holland, 1985.

[5] L. Bianco, J. Blazewicz, P. Dell'Olmo P, and M. Drozdowski. Scheduling multiprocessor tasks on a dynamic configuration of dedicated processors. Technical Report R-92/045, Institute of Computing Science, TU Poznan, 1992.

[6] L. Bianco, J. Blazewicz, P. Dell'Olmo P, and M. Drozdowski. Scheduling preemptive multiprocessor tasks on dedicated processors. *Perf. Eval.*, 1994. To appear.

[7] E. G. Coffman, Jr., editor. *Computer and job-shop scheduling theory*. John Wiley, 1976.

[8] E. G. Coffman, Jr., M. R. Garey, D. S. Johnson, and A. S. LaPaugh. Scheduling file transfers. *SIAM Journal of Computing*, 3:744–780, 1985.

[9] P. F. Corbett, S. J. Baylor, and D. G. Feitelson. Overview of the Vesta Parallel File System. In *The 1993 Workshop on Input/Output in Parallel Computer Systems*, pages 1–17, 1993.

[10] Thomas H. Corman. Fast permuting on disk arrays. *Journal of Parallel and Distributed Computing*, 17:41 –57, January 1993.

[11] T. H. Cormen and D. Kotz. Integrating Theory and Practise in Parallel File Systems. In *Proceedings of the DAGS 93 Symposium on Parallel I/O and Databases*, pages 64 – 74, 1993.

[12] Juan Miguel del Rosario, Rajesh Bordawekar, and Alok Choudhary. Improved Parallel I/O via a Two-phase Run-time Access Strategy. In *The 1993 Workshop on Input/Output in Parallel Computer Systems*, pages 56–70, 1993.

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

[14] M.D. Durand, T. Montaut, L. Kervella, and W. Jalby. Impact of Memory Contention on Dynamic Scheduling on NUMA Multiprocessors. In *Proceedings of the 1993 International Conference on Parallel Processing*, August 1993.

[15] M. Gereb-Graus and T. Tsantilas. Efficient Optical Communication in Parallel Computers. In *1992 Symposium on Parallel Algorithms and Architectures*, pages 41–48, 1992.

[16] Mario Gonzalez, Jr. Deterministic processor scheduling. *Computing Surveys*, 9:173, Sept. 1977.

[17] R. Jain, K. Somalwar, J. Werth, and J.C. Browne. Scheduling Parallel I/O Operations in Multiple Bus Systems. *Journal of Parallel and Distributed Computing*, 16:352 –362, December 1992.

[18] R. Jain, K. Somalwar, J. Werth, and J.C. Browne. Requirements and Heuristics for Scheduling Parallel I/O Operations. DRAFT; submitted to journal, 1993.

[19] 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.

[20] H. J. Karloff and D. B. Schmoys. Efficient Parallel Algorithms for Edge Coloring Problems. *Journal of Algorithms*, pages 39 –52, August 1987.

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

[22] O. Kreiger and M. Stumm. HFS: A Flexible File System for large-scale Multiprocessors. In *Proceedings of the DAGS 93 Symposium on Parallel I/O and Databases*, pages 6 – 14, 1993.

[23] E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Recent developments in deterministic sequencing and scheduling: A survey. In *Deterministic and Stochastic Scheduling*, pages 35–73. D. Reidel Publishing, 1982.

[24] M. Luby. Removing Randomness in Parallel Computation without a Processor Penalty. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 162–173, 1988.

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

[26] Krishna Palem. *On the complexity of precedence constrained scheduling.* PhD thesis, Univ. Texas at Austin, Dept. of Comp. Sci., 1986. Available as Tech. Rept. TR-86-11.

[27] A. Panconesi and A Srinavasan. Fast Randomized Algorithms for Distributed Edge Coloring. In *Proceedings of the 1992 ACM Symposium on Parallel and Distributed Computing*, pages 251–262, August 1992.

[28] David Patterson, Garth Gibson, and Randy Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD Conference*, pages 109–116, June 1988.

[29] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan. Informed Prefetching: Converting High Throughput to Low Latency. In *Proceedings of the DAGS 93 Symposium on Parallel I/O and Databases*, pages 41 – 55, 1993.

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

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

[32] J. S. Vitter and M. H. Nodine. Large-scale sorting in uniform memory hierarchies. *Journal of Parallel and Distributed Computing*, pages 107–114, Jan./Feb. 1993.

[33] J. S. Vitter and E. A. M. Shriver. Optimal disk I/O with parallel block transfer. In *Proc. ACM Symp. Theory of Comp.*, 1990.