Distributed Scheduling Algorithms to Improve the Performance of Parallel Data Transfers

Dannie Durand, Ravi Jain and David Tseytlin
Bellcore, 445 South Street, Morristown, NJ 07960
{durand,rjain}.bellcore.com

Abstract. The cost of data transfers, and in particular of I/O operations, is a growing problem in parallel computing. A promising approach to alleviating this bottleneck is to schedule parallel I/O operations explicitly. 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. Examples of such applications include 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 as bipartite graphs. These algorithms are based on the graph-theoretic notions of bipartite matching and edge-coloring. This paper is a condensed version of [4], where we discuss recent work on parallel I/O, in particular scheduling parallel I/O, and describe how our work fits into this context. We also describe our results in more detail in [4]. Here, we describe the problem we address (in Section 2), present a class of decentralized algorithms to solve it (Section 3) and present simulation results (Section 4)

2 Problem description

2.1 System model

Our system model is based on bipartite architectures such as the architecture shown in Figure 1. Here,

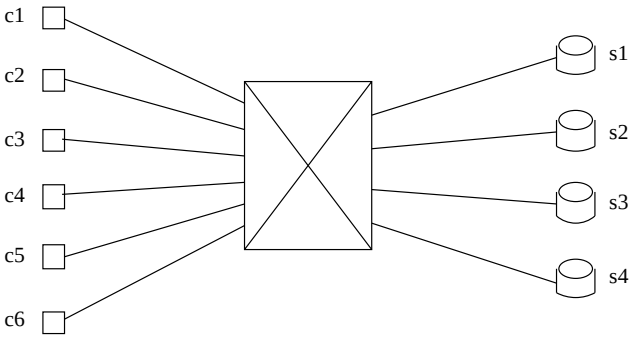

Figure 1: I/O System Architecture Abstraction

clients (e.g., processors, workstations or disk controllers) on the left initiate 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 realistic models that will allow us to study various architectural refinements are discussed in [4].

Our algorithms will consist of two stages: a scheduling stage, during which a schedule is generated, followed by a data transfer stage. The total cost is the sum of the cost of generating the schedule and the time to transfer the data according to that schedule. 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.

2.2 Scheduling as edge coloring

For an example of data transfer scheduling, see the bipartite transfer graph, G, of Figure 2. Here the ver-

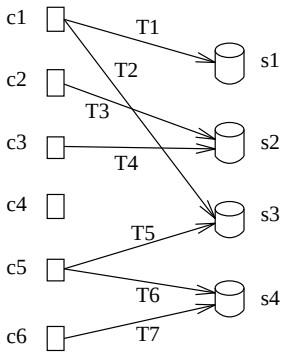

Figure 2: Data Transfer Graph

tices are the clients and servers shown in Figure 1. The edges $(T1, T2, \ldots, T7)$ are pending I/O transfers between the clients and the servers. (Note that our algorithms 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, for example, 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)).

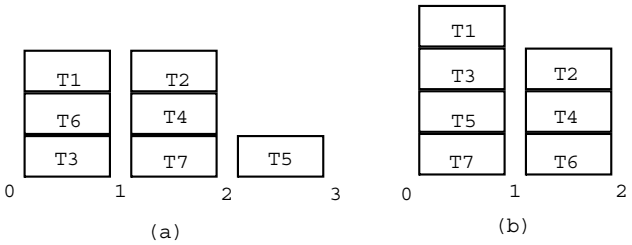

Figure 3: Schedules represented as Gantt charts.

Our algorithms are based upon two relevant problems from graph theory: 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. Thus 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. 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. Since each color in an edge-coloring is a matching on G, each color in the coloring represents a set of transfers that can occur simultaneously. 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 [3] that Δ colors are necessary and sufficient to edge-color a bipartite graph of degree Δ.

3 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.

All algorithms in our class are based on an outer loop which generates a set of matchings (i.e. a partial coloring) at each iteration until all edges are colored. The algorithms differ in the way the matchings are generated and in the number of matchings generated (or colors used) in each iteration of the outer loop, also called one *phase* of the algorithm. The pseudocode for a uniprocessor simulation of our algorithm is shown in Figure 4. We begin by describing the case

where only one matching is computed in each phase.

The inner loop consists of a simple bidding scheme, similar to those used in [8, 1], during which clients and servers exchange *messages* to generate a schedule. This schedule determines the order in which the actual data *transfers* take place during the transfer stage. In the first step of the single matching bidding scheme, 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*. This two step bidding strategy results in a matching, but not necessarily a maximal matching (i.e., a matching where no edge can be added without creating a conflict at one of the vertices). We discuss how to improve the matchings in the next section.

In general, instead of computing one matching in each phase, we will compute *Ncolors* matchings in each phase, as shown in Figure 4. 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.

```
1. While (G = (A, B, E) is not empty)
2.     {
3.     Get Ncolors new colors.
4.     For i = 1 to Npasses
5.         {
6.         ∀ clients: Assign Ncolors to edges(s)
7.                    chosen by strategy H1.
8.         ∀ servers: Resolve conflicts
9.                    by strategy H2.
10.        }
11.    Delete colored edges and
               vertices of zero degree from G.
12.    }
```

Figure 4: A parameterized class of scheduling algorithms

3.1 Improving matching quality

In order to increase the number of edges in the matching achieved in each phase, we consider two approaches: using heuristics and multiple passes. The two methods can also be combined.

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 by the clients to guess which edges have a higher probability of success. In addition, instead of selecting the winner uniformly at random, the servers can also use heuristics.

For this paper, we focus on one heuristic which has proven to be very effective in centralized algorithms [6], 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. Note that *HDF* represents one type of heuristic, in which extra computation is performed to try to reduce communication time. We are also considering heuristics where additional communication penalty may be paid.

Multiple passes. Clients who lost their bid on the first pass can make bids using the same color on different edges in subsequent passes. Note that with multiple passes, a phase consists of more than one pass. In each pass in the phase, the clients choose from the same set of colors when 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.

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.

3.2 Resolving conflicts in the scheduling stage

Although scheduling eliminates conflicts during the data transfer stage, conflicts can still occur in the scheduling stage, 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, a hardware solution may be feasible, 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. For example, both the slotted and retry protocols for sending messages have been used in the BBN TC2000 parallel computer [2]. The protocols are provided at a hardware level transparently to the user.

In our 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.

3.3 Related work

Some related distributed communications algorithms have been presented by [8, 1, 5]; see [4] for a discussion.

Panconesi and Srinavasan [8] presented a distributed algorithm for edge-coloring on general graphs henceforth called *PS*, which differs from ours as follows. 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 client 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 [7]. 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.

3.4 An algorithm requiring global information

Note that, unlike the algorithm of Gereb-Graus and Tsantilas [5] and the *PS* algorithm of Panconesi and Srinavasan [8], 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*. In the first phase of *mPS*, $Ncolors = \Delta$, the true degree of the graph. Thereafter the algorithm estimates the current degree of the graph from the degree in the previous phase using a lower bound on Panconesi and Srinivasan's formula, yielding $Ncolors = \Delta(i) = \Delta(i-1)/e$ for phase $i > 1$. However, unlike *PS*, the modified algorithm *mPS* does not switch to Luby's algorithm when $\Delta(i)$ drops below the threshold Δ_t. Instead *mPS* continues to use the *PS* scheme to color edges using $Ncolors = \Delta(i-1)/e$ at phase i, until all edges have been colored.

4 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, listed below. (For comparison, we performed a series of experiments with the *mPS* algorithm also).

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.

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 and 6. The y-axes of these plots show the mean normalized schedule length. The mean normalized schedule length is the the ratio of the schedule length generated by our simulated algorithm to the optimal schedule length, averaged over the number of 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.

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

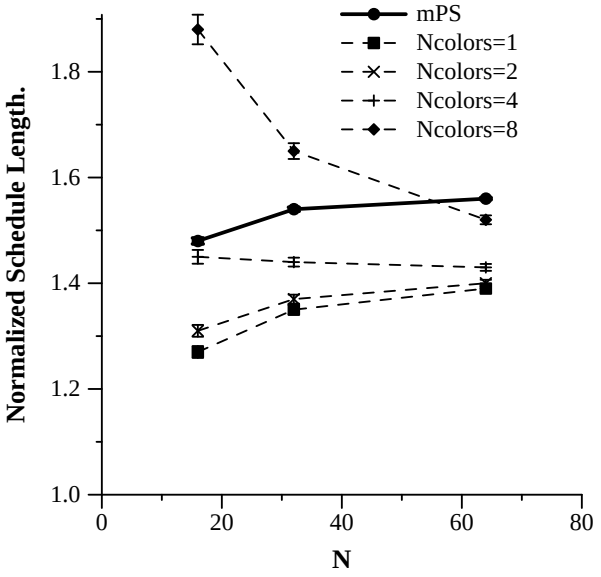

Figure 5: Impact of *Ncolors* on Schedule Length.

shows the schedule lengths produced by *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 number of edges which were assigned to each color for a single experiment. We have found (see [4] for an example) that sometimes there are colors which are not used at all, i.e., not assigned any edges. 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. 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.

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, it is too many colors to generate good matchings.

Using heuristics and multiple passes. 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 6. As before, the thick

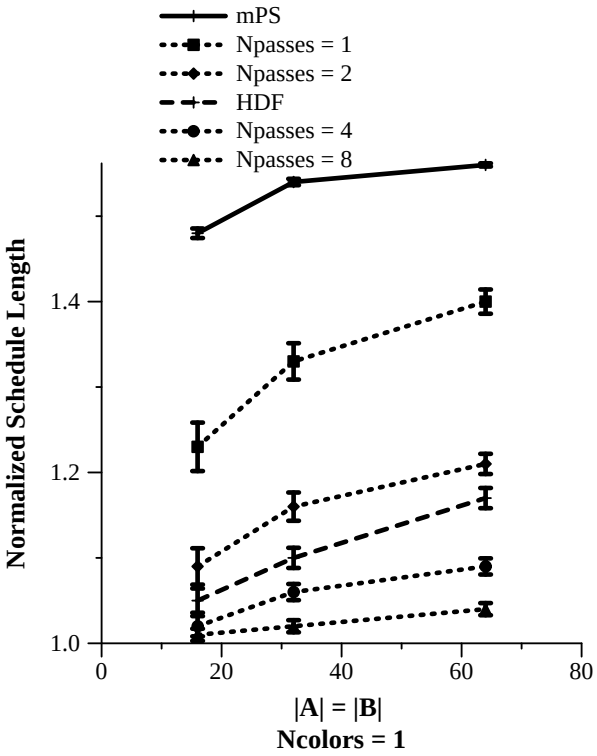

Figure 6: Impact of *HDF* and *MPASSES* on Schedule Length.

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 for all graph sizes. 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.

We have also investigated how the number of colors per phase affects the performance of *MPASSES* and *HDF*. Due to lack of space, we refer the reader to [4] for a discussion.

5 Discussion and conclusions

We discuss our results in the context of other work published in this area (see also [4]). For $Ncolors = 1$, our algorithms yield schedule lengths of $1.05\Delta - 1.2\Delta$ for HDF and $1.02\Delta - 1.09\Delta$ for four passes, a substantial improvement over the algorithms Panconesi and Srinavasan [8], 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 generate schedules faster, and reducing *Ncolors* adaptively as the algorithm progresses will offset the penalty in schedule length. It would be interesting to determine the appropriate rate at which to reduce *Ncolors* for different types of graphs.

The work of both Panconesi and Srinavasan and Gereb-Graus and Tsantilas [5] is based on probabilistic analyses that predict how the degree of the graph will decrease with time. 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.

In conclusion, we have introduced a class of distributed scheduling algorithms for data transfers in parallel I/O subsystems 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 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 $Npasses = 3$. 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 35% over previous decentralized algorithms. In [4] we have discussed extensions and our further work on this problem.

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.

References

[1] T. E. Anderson, S.S. Owicki, J. B. Saxe, and C. P. Thacker. High-Speed Switch Scheduling for Local-Area Networks. *ACM Trans. Comp. Sys.*, 11(4):319–352, Nov. 1993.

[2] Inside the TC2000, BBN, Cambridge, MA, 1990.

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

[4] D. Durand, R. Jain, and D. Tseytlin. Distributed scheduling algorithms to improve the performance of parallel data transfers. Tech. Rep. 94-38, DIMACS, June 1994.

[5] M. Gereb-Graus and T. Tsantilas. Efficient Optical Communication in Parallel Computers. In *1992 Symp. Par. Alg. Arch.*, pages 41–48, 1992.

[6] R. Jain, K. Somalwar, J. Werth, and J.C. Browne. Requirements and Heuristics for Scheduling Parallel I/O Operations. Submitted for publication, 1993.

[7] M. Luby. Removing Randomness in Parallel Computation without a Processor Penalty. In *IEEE Symp. on Found. of Comp. Sci.*, pages 162–173, 1988.

[8] A. Panconesi and A Srinavasan. Fast Randomized Algorithms for Distributed Edge Coloring. In *ACM Symp. Par. Dist. Comp.*, pages 251–262, Aug. 1992.