Applying Randomized Edge Coloring Algorithms to Distributed Communication: An Experimental Study.

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

Abstract

We propose a parameterized, randomized edge coloring algorithm for use in coordinating data transfers in fully connected distributed architectures such as parallel I/O subsystems and multimedia information systems. Our approach is to preschedule I/O requests to eliminate contention for I/O ports while maintaining an efficient use of bandwidth. Request scheduling is equivalent to edge coloring a bipartite graph representing pending I/O requests. Although efficient optimal algorithms exist for centralized edge coloring where the global request graph is known, in distributed architectures heuristics must be used. We propose such heuristics and use experimental analysis to determine their ability to approach the centralized optimal. The performance of our algorithms is also compared with the work of other researchers experimentally. Our results show that our algorithms produce schedules within 5% of the optimal schedule, a substantial improvement over existing algorithms. The use of experimental analysis allows us to evaluate the appropriateness of each heuristic for a variety of different architectural models and applications.

1 Introduction

The advent of huge data sets and a growing imbalance in CPU and I/O speeds has lead to a communications bottleneck in distributed I/O systems. New research to attack this problem is being undertaken at all levels from low level solutions, such as disk striping, through operating system and compiler support for I/O, to high level algorithmic solutions for "out-of-core" computations. In this paper, we propose and evaluate the use of scheduling techniques to eliminate contention and increase throughput.

We consider I/O intensive applications in an architecture based on clients and servers connected by a complete network where every client can communicate with every server. Clients and servers can handle at most one simultaneous communication. Each client has a queue of data transfer requests destined for various servers. (The requests may be reads or writes but they are always initiated by the clients.)

SPAA'95 Santa Barbara CA USA© 1995 ACM 0-89791-717-0/95/07.$3.50

The object is to process all requests as fast as possible without violating the "one communication at a time" constraint. One approach is to use direct algorithms in which conflicts at the servers are managed using retry algorithms, buffering or by discarding packets. However, for applications with very large packets, such as multimedia applications, these solutions can be very expensive. We consider a second approach: data transfers are prescheduled to obtain schedules that are conflict free and make good use of the available bandwidth. Our scenario involves a scheduling stage, during which requests are assigned to time slots, followed by a data transfer stage, where the data transfers are executed according to the schedule. The scheduling algorithm is based on a small number of rounds of bidding between clients and servers. We make the assumption that a request message is much shorter than the actual data transfer, so that the cost of sending some prescheduling messages is amortized by the reduction in the time required to complete the data transfers. This assumption is appropriate for data intensive, I/O bound applications. 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% [12, 13].

Figure 1 shows an example of a transfer request graph for a system with two clients and three servers. Outstanding data transfer requests are represented as edges in a bipartite graph with the clients on the left hand side and the servers on the right. Clients c_1 and c_2 each have two pending requests. A conflict exists at server s_2 which is the target of two requests. We can represent a schedule for this request graph as a Gantt chart with time slots on the x-axis and transfers scheduled for each slot on the y-axis. In Figure 2(a), transfers T_1 and T_4 are scheduled simultaneously in

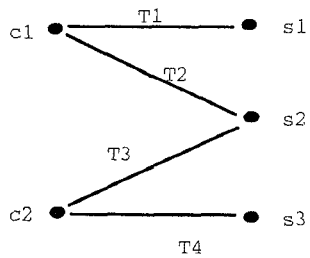

Figure 1: A transfer request graph

the first time slot, a legal schedule since T_1 and T_4 share neither a client nor a server. However, in order to avoid

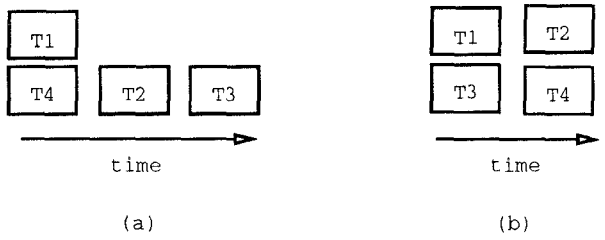

Figure 2: Two possible schedules for the example request graph

the conflict at s_2, T_2 and T_3 must be scheduled separately, resulting in a schedule with three slots. A shorter schedule can be obtained by scheduling T_3 with T_1 and T_2 with T_4, as shown in Figure 2(b). It can be seen that each time slot in a schedule is a matching in the bipartite transfer graph and that scheduling data transfers can be viewed as an edge coloring problem. Henceforth, we will use the terms color and timeslot interchangeably.

We present randomized, distributed algorithms to schedule data transfers based on edge coloring the transfer request graph. In the centralized case, it has been shown that at least Δ colors are required to edge color a bipartite graph with maximum degree Δ [3]. Efficient centralized algorithms to obtain optimal edge colorings exist (see [14] for a survey). Because in many distributed architectures global information about I/O requests is not centrally available, we use heuristics to generate near optimal solutions. We propose a parameterized algorithm which can be tuned for a particular set of communication and computational costs, since a wide variety of crossbar-based architectures with different performance characteristics exist (e.g. C.mmp [22], the Fujitsu VP500 [11], input queued packet switches [1], crossbar-based workstation farms).

We present an experimental analysis of our algorithms based on simulation and compare them with the work of other authors. Results show that our algorithm yields schedule lengths close to the optimum solution achieved in the centralized case; within 5% for some parameter choices. This is a substantial improvement over the algorithms of other authors which give schedules 50% - 60% greater than the optimum. Experimental studies allow us to observe the impact of parameter choices and variations in input data on the behavior of distributed, randomized edge coloring algorithms. From these observations, we obtain insights into how to best apply them in a variety of architectural contexts. We see this approach as an important adjunct to theoretical analysis.

The rest of the paper proceeds as follows. Related work is presented in Section 2. In Section 3, we present our algorithm in detail. After describing our experimental method, results on schedule lengths achieved as a function of parameter choice and input distribution are given. In Section 4, we discuss how to handle the arrival of new requests, fairness and starvation, and issues required for an analysis of running time. We summarize our results in Section 5.

2 Related Work

Much of the previous work on scheduling deals with tasks which each require only a single resource at any given time 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 (see [4, 6, 10, 15, 18, 14] for surveys). 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 [14] for a survey).

A distributed, randomized algorithm for routing in distributed systems has been published by Panconesi and Srinavasan [19, 20]. Their algorithm, which will be described in detail in Section 3, requires at most $1.6\Delta + O(\log^{1+\delta} n)$ colors to edge-color the entire graph in $O(\log \Delta)$ phases, where Δ is the maximum degree of the transfer request graph and n is the number of nodes.

Some related algorithms that route h-relations [21] in an optical computing (OCP) model [2] have also been presented. In this model, when two or more messages are sent to the same recipient, none of the messages are successfully transmitted. If a single message is sent to a recipient, it is received and acknowledged. If a sender receives no acknowledgement, it knows that the transmission failed and must be repeated later. In an h-relation, every processor has at most h messages to send and is the destination of at most h messages. If we view clients as sending processors and servers as receiving processors and equate h with Δ, the h-relation problem is similar to our communication problem with the added constraint that OCP states explicitly how collisions will be handled (failure of all messages).

Gereb-Graus and Tsantilas [7] present a direct algorithm for completing an h-relation in the OCP model in $\Theta(h + \log\log n)$ steps. By direct we mean that the only messages sent are those in the h-relation (no request messages) and that they are only sent to their destination processor (no intermediate nodes.) Gereb-Graus and Tsantilas' algorithm, the best known direct algorithm for communicating an h-relation in OCP, is optimal to a constant factor for $h = \Omega(\log n \log\log n)$. We find their algorithm interesting because it is one example of an alternative approach when prescheduling is not desirable.

There is considerable evidence that it is hard to edge color a sparse graph. Gereb-Graus and Tsantilas posit that in order to achieve optimal coloring, h must be bounded below by $\log n \log\log n$. In [8], Goldberg et al. showed that for any random, direct algorithm, there exists a 2-relation that requires $\Omega(\log n)$ steps to route with success probability of 1/2. This result was extended by MacKenzie et al. [17] by showing that for any $h \geq 2$, there exists an h-relation such that any randomized, direct algorithm will require $\Omega(h + \log h \log n)$ expected steps to route. In [9], Goldberg et al. showed that for *any* randomized, distributed algorithm, there exists a 2-relation which requires at least $\Omega(\sqrt{\log\log n})$ steps to route.

To address this problem of routing sparse graphs, Goldberg et al. [8] present indirect algorithms for routing h relations, where messages are first sent to intermediate nodes to reduce contention by balancing the communication load. In

a complete optical network, it is possible to reduce the difficulty of coloring sparse graphs by redistributing messages because no distinction between clients and servers is made. Any processor can communicate with any other processor. Indirect routing is not possible in our model since there is no client-client or server-server communication.

In a packet switching context, Anderson et al. [1] present a randomized, distributed algorithm for computing matchings to route data cells from inputs to outputs in DEC's AN2 ATM communications switch. Because of their application, this work is geared towards meeting real time constraints and minimizing delay associated with individual requests rather than minimizing the time required to complete a set of outstanding requests. We discuss some aspects of this tradeoff between delay and throughput in Section 4.

3 Experimental Analysis of Distributed Edge Coloring Algorithms.

In this section we present the details of our algorithm and that of Panconesi and Srinavasan. Simulation is used to study the performance of these algorithms experimentally, revealing the impact of parameter choices on behavior.

We make the following assumptions about our system. 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. Transfers take place in units of fixed-size blocks and preemption is permitted at block boundaries. Communication in these algorithms is synchronous; that is, all clients (servers) communicate at regular, fixed intervals.

3.1 A Parameterized Edge Coloring Algorithm.

Our parameterized scheduling algorithm, shown in Figure 3, is based on an outer loop which generates *Ncolors* matchings at every iteration. We call one iteration of this outer loop a *phase*.

We first discuss the matching algorithm inside the loop for the case where $Ncolors = 1$. This algorithm is based on a distributed two-step bidding system. In its simplest form, each client selects one of its incident edges uniformly at random ($H1 = UAR$[1]) and sends a message to the associated server, stating its intention to color the edge. In the second step, each server resolves conflicts by selecting one of its proposals uniformly at random ($H2 = UAR$) and sending back an accept message. The other proposing clients are rejected. Clients assign the current color to the winning edges and remove those edges from the graph. A fresh new color is obtained and the process is repeated in the next phase. The algorithm repeats until all edges are colored.

This bidding process will generate matchings, but not necessarily very good ones. We use two approaches to obtain better matchings. The first is to use *multiple passes* per phase (*MPASSES*). (In our terminology, a *pass* is a single round trip in the bidding process.) With *MPASSES*, those clients whose proposals were rejected in the first pass have an opportunity to assign the current color to a different adjacent edge in subsequent passes by selecting proposals from previously untried edges.

[1] Uniformly at random

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

Figure 3: A parameterized scheduling algorithm

The second approach is to use heuristics in selecting proposals and winners. For example, clients might use information about traffic patterns to guide the choice of an edge to color. Similarly, servers may use a heuristic instead of a uniform distribution to select a winner. In this paper, we focus on one heuristic which has proven to be very effective in centralized algorithms [13], the *Highest Degree First* heuristic or *HDF*. When the clients send their proposals to the servers, they also send their current degree. Each server grants the request of the highest degree client (or selects a winner at random from among the highest degree clients if there are more than one.)

In order to obtain an optimal edge coloring, every matching must be a *critical* matching; i.e. a matching including an edge adjacent to each node of maximum degree. In contrast, the primary goal of a matching algorithm is to obtain *maximum* matchings, which is not necessary for edge coloring. By favoring high-degree nodes, *HDF* increases the probability of obtaining a critical matching. However, it also introduces some fairness issues as discussed in Section 4. In contrast to *HDF*, *MPASSES* explicitly increases the size of the matchings and only indirectly increases the probability of a critical matching.

To reduce the number of phases required to edge color the graph, we can compute several matchings simultaneously by using $Ncolors > 1$, which will result in a reduction of total running time if the complexity of each phase does not increase proportionally. Whether or not this is advantageous will depend on the specific architecture. For example, if the communication required when $Ncolors = 1$ does not use all of the available bandwidth, additional messages may be sent without increasing communication costs. When $Ncolors > 1$, each client selects *Ncolors* edges uniformly at random and then assigns a random permutation of the *Ncolors* fresh new colors to these edges. (If the degree of the client is less than *Ncolors*, some colors will not be used.) It then sends *Ncolors* messages to the servers. In this case, the servers must select up to *Ncolors* winners. Although increasing *Ncolors* can result in performance improvement for some architectures, our experimental results suggest it can have an adverse effect on schedule quality.

3.2 An Adaptive Edge Coloring Algorithm

We compare our algorithm with a distributed edge coloring algorithm (*PS*) for bipartite graphs presented by Panconesi and Srinavasan [19, 20] as part of a divide-and-conquer approach to edge-coloring general graphs. *PS* also generates

```
Ncolors = Δ
Threshold = log^{1+δ} N
Repeat
   {
   Get Ncolors new colors.
   For all clients:
       Assign Ncolors to edges(s) chosen U.A.R.
   For all servers:
       Resolve conflicts U.A.R. for each color.
   Ncolors = Δ(i)
   } until  Ncolors < Threshold
Use Luby to color remaining graph.
```

Figure 4: Panconesi and Srinavasan's algorithm

Ncolors matchings at every phase as seen in Figure 4. Their algorithm is based on the same two-step bidding routine that we use but does not use any additional machinery to improve the size of the matchings. Unlike our algorithm, *PS* uses an adaptive approach to selecting the number of fresh new colors used in each phase. Initially $Ncolors = \Delta$, where Δ is the maximum degree of the initial graph (phase 0). In subsequent phases, $Ncolors = \Delta(i)$, where $\Delta(i)$ is a probabilistic estimate of the degree of the graph in phase i. Panconesi and Srinavasan's contributions include an extension to Chernoff-Hoeffding bounds which allows them to predict $\Delta(i)$ with high probability, as well as the number of phases required to color the graph. These bounds are not valid for sparse graphs, so the algorithm switches to a random deterministic vertex-coloring algorithm proposed by Luby [16] once $\Delta(i)$ drops below the threshold of $log^{1+\delta} n$, where n is the number of nodes and δ is a parameter related to the success probability. Luby's algorithm is used to vertex color the line graph of the remaining graph, requiring at least $2\Delta(t) - 1$ colors, where $\Delta(t) \leq log^{1+\delta} n$ is the degree of the graph at threshold. Panconesi and Srinavasan show analytically that their algorithm requires at most $1.6\Delta + O(\log^{1+\delta} n)$ colors to edge-color the entire graph in not more than $O(\log \Delta)$ phases.

We also compared our results experimentally with a modified version of Panconesi and Srinavasan's algorithm. Below we present our experimental approach and discuss our results.

3.3 Experimental Approach

We studied the impact of various parameters on the schedule lengths obtained by our algorithm experimentally using a parameterized simulator, written in C that runs on uniprocessor UNIX workstations. It takes the number of colors and the number of passes as input and selects winners either uniformly at random or using the *highest degree first* (*HDF*) heuristic. We also implemented a modified version of PS called *mPS*. *mPS* differs from PS in that it does not switch to Luby's algorithm when the threshold of $log^{1+\delta} n$ is reached. Instead, the algorithm continues until all edges are colored.

Experiments were run on NxN random bipartite graphs, where $N \in \{16,32,64\}$, with an edge density of one half (i.e., $G = (A, B, E)$, where $|A| = |B| = N$ and $E = N^2/2$). Multiple edges were allowed, since multiple requests between the same client and server are possible in a distributed system. We chose these sizes because in current networks of workstations and parallel I/O subsystems one rarely sees more than 100 components. As architectural sizes increase in the future, the impact of these algorithms on larger graphs should also be studied.

We ran experiments to study the impact of *Ncolors*, *Npasses*, the *HDF* heuristic and *mPS* on schedule length as shown in Table 1. Experiments varying *Ncolors* allow us to better understand the role of multiple colors per phase in the behavior of *mPS* and to explore the related tradeoff between running time and schedule length penalty. Experiments also allow us to compare the schedule lengths obtained with *HDF* and *MPASSES*, showing how *HDF* and *MPASSES* differ in their behavior even when the same schedule length is obtained. In addition, experimentation reveals the interaction of *Ncolors* with other parameters.

We ran these experiments on two types of graphs, *uniform* graphs and *hotspots*. In uniform graphs, the expected degree of all vertices is the same. In other words, the probability that vertex, v, is an endpoint for some edge, e, is equal to $1/N$ for all vertices, v. In hotspot graphs, a distinguished vertex v has a greater probability of having adjacent edges. This "hot" vertex has an expected degree of $\hbar\delta$, where δ is the expected degree of the other vertices. We refer to $\hbar$ as the *heat* of the hotspot. We considered graphs with either a single client hotspot or a single server hotspot and with $\hbar$ values of 1 (i.e. uniform graphs), 2 and 4.

The results are presented graphically in the next section. Schedule quality is presented as normalized schedule length, the schedule length divided by the maximum degree of the graph, Δ. This allows us to compare graphs of different sizes and densities. The data presented are averages over many experiments. For uniform graphs, every point is an average of 100 experiments; that is, ten colorings each of ten graphs. Because hot graphs have greater variance, we ran 225 executions of each hotspot experiment, or 15 colorings each of 15 graphs. We computed 99% confidence intervals and present them graphically as error bars.

3.4 Experimental Results

Uniform Graphs We first present the impact of the number of colors on scheduling length as shown in Figure 5. The black line shows the performance of *mPS* as a function of graph size. The dotted lines represent our algo-

	Npasses				*HDF*
Ncolors	1	2	3	4	
1	x	x	x	x	x
2	x	x	x	x	x
4	x	x	x	x	x
8	x	x	x	x	x
mPS	x				

Table 1: Experiments Performed

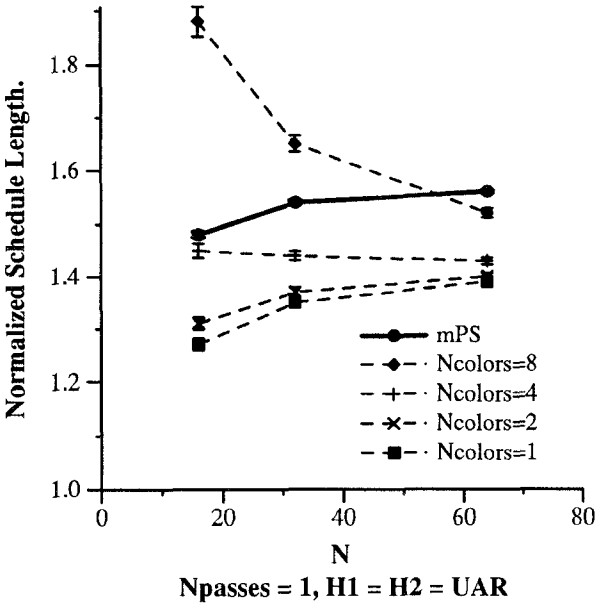

Figure 5: Impact of *Ncolors* on schedule length.

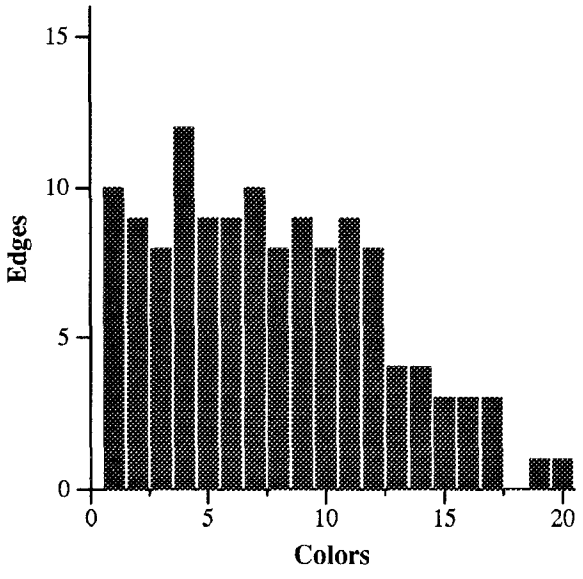

Figure 6: Histogram of matching density

rithm where $Npasses = 1$, $H1 = H2 = UAR$ and $Ncolors \in \{1,2,4,8\}$. We see that *mPS* requires between 1.5Δ and 1.6Δ colors to edge color the graphs we studied, about the number of colors Panconesi and Srinavasan predicted analytically. Our best performance is achieved when $Ncolors = 1$, ranging from 1.27Δ to 1.39Δ. As *Ncolors* increases, schedule length increases as well, particularly for the smaller graphs. To see why this happens consider the histogram shown in Figure 6. This figure shows a coloring generated in a single experiment, where each vertical bar corresponds to a single matching. The matchings are quite full for the first half of the run but then begin to decrease in size. In particular, color 18 is not used at all. Unused colors occur when *Ncolors* is larger than Δ because none of the clients can use all the available colors. As a result, there is a nonzero probability that some color will not be proposed by any client, resulting in a "hole" in the schedule. Since the system is distributed, it is impossible to recognize unused colors and the time slot will be wasted. Even when holes like this do not occur, unless *Ncolors* is small compared with Δ, colors will be used inefficiently, leading to smaller matchings. This explains why smaller graphs (with lower values of Δ) are more sensitive to increasing *Ncolors* than larger graphs, for a fixed edge density.

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. Our experience with varying *Ncolors* suggests that the performance of *mPS* can be improved by reducing the number of colors used in each phase to, for example, $\lambda\Delta(i)$, where $\lambda < 1$, with a concomitant increase of $1/\lambda$ in the number of phases required. The optimal value of λ would have to determined experimentally.

The impact of *HDF* and *MPASSES* on schedule length is shown in Figure 7. This figure shows the improvement that can be obtained from using *HDF* and 2, 4 and 8 passes per phase. From the data we see that these techniques substantially reduce schedule length, with eight passes bringing us within 5% of optimal for all graphs studied. Simply increasing the number of passes from one to two divides the percentage above optimal in half, reducing the schedule length from 40% to 20% above optimal for 64x64 graphs.

We see that for these inputs and parameters (uniform graphs, $Ncolors = 1$), *HDF* gives about the same schedule length as three passes. *HDF* requires much less communication, making it attractive for many architectures. The histogram in Figure 8 shows how these two approaches exhibit different internal behavior even when they achieve the same schedule length. The black bars correspond to *HDF*, the gray bars to $Npasses = 3$. We see that *HDF* gives better load balancing: except for the last two matchings, all matchings are quite large and roughly the same size. The execution based on three passes shows larger matchings at the beginning of the execution but the size of the matchings begins to decrease about half way through the run. Which approach is best, depends on the relative communication and computational costs of the architecture at hand.

Figure 9 shows how *MPASSES* and *HDF* behave as *Ncolors* increases. The figure shows schedule lengths obtained for a 32x32 graph when $Ncolors \in \{1, 2, 3, 4\}$. Clearly, *MPASSES* is less sensitive than *HDF* to increased values of *Ncolors*. This is not surprising, since *MPASSES* compensates for the sparse matchings that result when $Ncolors \gtrsim \Delta$. Hence, *MPASSES* is attractive for large values of *Ncolors*. Alternatively, if communication costs are so high that using *MPASSES* is prohibitive, a low value of *Ncolors* should be used in order to gain the maximum benefit from *HDF*.

Hotspots Experimental studies of schedule lengths as a function of heat obtained when coloring bipartite graphs with hotspots are shown in Figures 11 - 14. The experiments taken as a whole show that all parameters considered tend

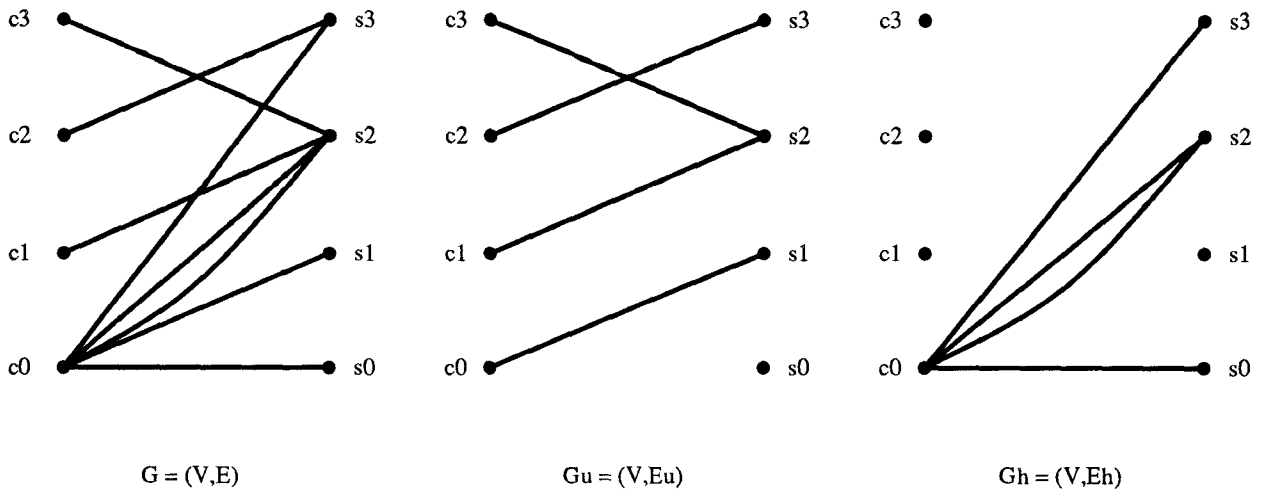

Figure 10: Decomposition of a hotspot graph

to get shorter schedules (i.e. closer to the centralized optimum) with graphs with both client and server hotspots than with uniform graphs. We can understand this intuitively, by observing that a graph with a hotspot, $G = (V, E)$, can be decomposed into two graphs: a uniform graph, $G_u = (V, E_u)$ and a graph, $G_h = (V, E_h)$, containing the extra edges making up the hotspot. $E = E_u \cup E_h$ as shown in Figure 10. Coloring G_h is equally easy (or hard) for both the distributed and centralized cases. When $Ncolors = 1$, our algorithm will require exactly Δ_h colors to color G_h, where Δ_h is the degree of G_h. As the heat $\hbar$ increases, the fraction of edges in G which come from G_h rather than G_u will increase and the schedule length will approach Δ.

Let us also consider how the behavior of the algorithms on server and client hotspots differs. Comparing Figures 11 and 12 shows that the vanilla algorithm ($Ncolors = 1$, $Npasses = 1$, $H1 = H2 = UAR$) does better on server hotspots than client hotspots. To see why, remember that optimal schedules are obtained when the degree of the graph is decreased at every time slot. With server hotspots the bottleneck is on the server side. Unlike clients, where sending a proposal does not guarantee a winner, as long as a server gets a proposal, it is guaranteed to reduce its degree. As heat increases, the probability of a hot server getting a proposal increases as well.

Figure 11 shows that *HDF* performs very well on client hotspots, decreasing much faster than the multiple pass schedules as heat increases. With the *HDF* heuristic, the degree of the graph is decreased at every time slot unless two highest degree nodes compete for the same server at some point during the algorithm. In this case, the server can only accept one of the two highest degree proposals and the degree of the graph will not be reduced. As heat increases, the advent of two highest degree clients is increasingly unlikely, since the degree of the hotspot is much greater than the degree of other clients.

In contrast, *MPASSES* does very well with server hotspots, with schedule lengths dropping dramatically as $\hbar$ increases. The curve for $Npasses = 4$ approaches optimal when $\hbar = 2$ and when $\hbar = 4$, all curves approach optimal. We hypothesize that this occurs because the server hotspot is colored on the first pass with high probability, guaranteeing a critical matching. With the hot server out of the running, clients have a chance to color edges connected to other servers on subsequent passes. This reduces the degree of other nodes and preserves the hotspot. In contrast, multiple pass curves flatten out with increasing heat in the case of client hotspots as shown in Figure 11. As observed above, the probability that a client hotspot will be colored is not as high as in the server case, so the probability of obtaining a critical matching is also lower . With server hotspots, the heat of the hotspot is preserved whereas with client hotspots the heat degrades with time, reducing the performance advantage associated with hotspots.

Figures 13 and 14 show that as heat increases, the tendency of large values of *Ncolors* to inflate schedule length decreases. This is not surprising since schedule length increases when $Ncolors \gtrsim \Delta$, which occurs less frequently in hotspot graphs because the initial degree is high. *mPS* seems only mildly sensitive to heat in the client case and not sensitive at all in the server case. This is because in *mPS* the value of *Ncolors* scales with the degree of the graph.

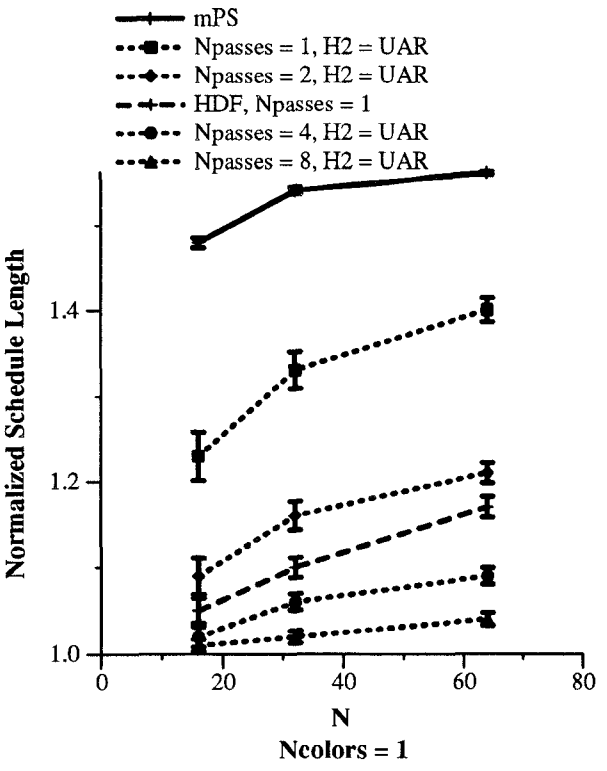

Figure 7: Impact of *HDF* and *Mpasses* on schedule length.

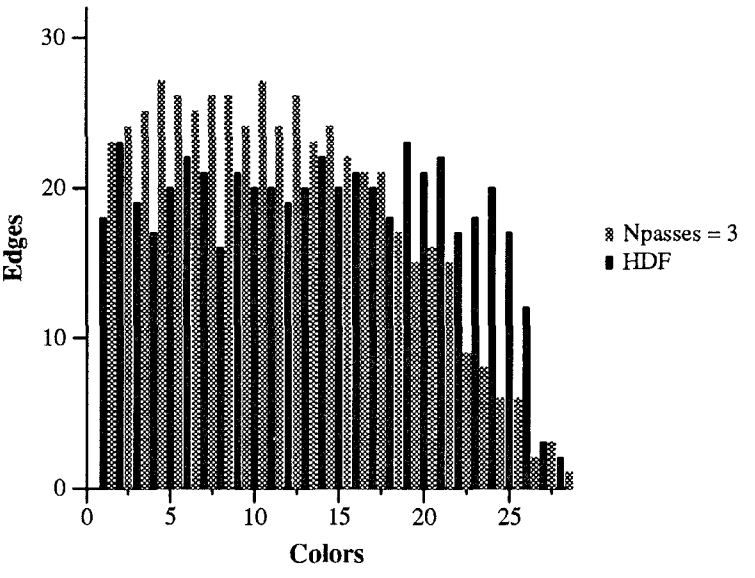

Figure 8: Comparison of *HDF* with *MPASSES*

4 Discussion

Below we discuss strategies for handling new I/O requests, fairness and starvation issues and problems involved in analyzing the running time. Some open problems are proposed.

4.1 Batch versus Dynamic Schedules

In the previous section we presented algorithms for coordinating data transfers associated with a set of pending I/O requests. We did not address the arrival of new I/O requests. Here we discuss the relative advantages of two possible approaches: batch and dynamic algorithms. In *batch* algorithms, all outstanding requests are scheduled before considering new arrivals. In *dynamic* algorithms, new requests are considered as soon as the current time slot has been scheduled. Hybrid algorithms, with a small number of matchings per batch, are also possible.

Batch scheduling is equivalent to edge coloring the bipartite graph corresponding to the predetermined set of I/O requests that arrived since the last scheduling stage began. An appropriate metric for batch algorithms is the number of colors required to edge color the graph, or equivalently, the length of the schedule required to complete all requests. This approach is natural for applications where requests arrive in spurts such as "out of core" algorithms, where a program is manipulating a data set too large to fit in memory and must periodically perform a set of I/O operations to obtain the next chunk of data to work on (see, for example [5]). When these applications are run alone on a multiprocessor, no new requests should be scheduled until the current batch of requests is completed.

There are some problems associated with batch algorithms. Because of the distributed nature of the algorithm, there is no way to detect when all edges have been colored. Gereb-Graus and Tsantilas and Panconesi and Srinavasan use probabilistic analyses to predict the number of rounds required to complete all edges. This requires a knowledge of Δ, the maximum degree of the graph, at the beginning of execution. This is global quantity cannot be obtained without global communication. Furthermore, if the probabilistic analysis uses conservative upper bounds, the predicted number of rounds may be substantially greater than the actual number of rounds required, increasing the running time of the scheduling stage. Finally, towards the end of every scheduling stage, the remaining graph will be sparse and, hence, harder to color. On the other hand, an advantage of batch algorithms is that starvation is not a problem since every request will be served within the current batch.

Dynamic algorithms are equivalent to finding a series of matchings. In this case, the goal is to obtain the largest possible matching at each step. This differs from the batch case, where it is more important to reduce the degree of the graph at each step than to obtain large matchings. Some metrics used to evaluate dynamic communication algorithms are throughput and delay. Dynamic algorithms map naturally to applications where requests arrive randomly such as packet switches and multiprocessor I/O systems supporting multitasking. Dynamic algorithms do not have the problem of predicting when to stop that characterizes the batch algorithms. Furthermore, as long as there is incoming traffic, the graph is always dense, making it easier to maintain high throughput. However, in dynamic algorithms, the issues of fairness and starvation must be addressed. Some solutions include the use of time stamps and dividing traffic into priority classes (see, for example, [1]).

The algorithms discussed in this paper are batch algorithms. It would be straightforward to make our algorithm

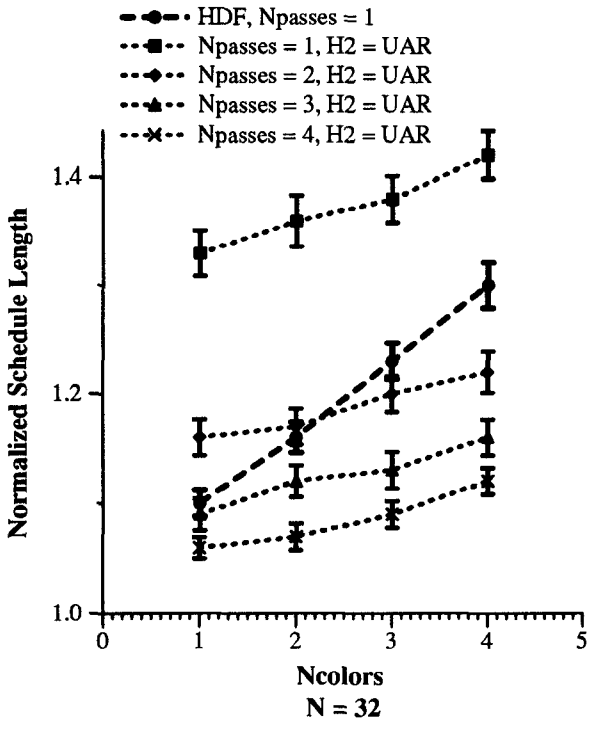

Figure 9: Impact of *Ncolors* on *MPASSES* and *HDF*.

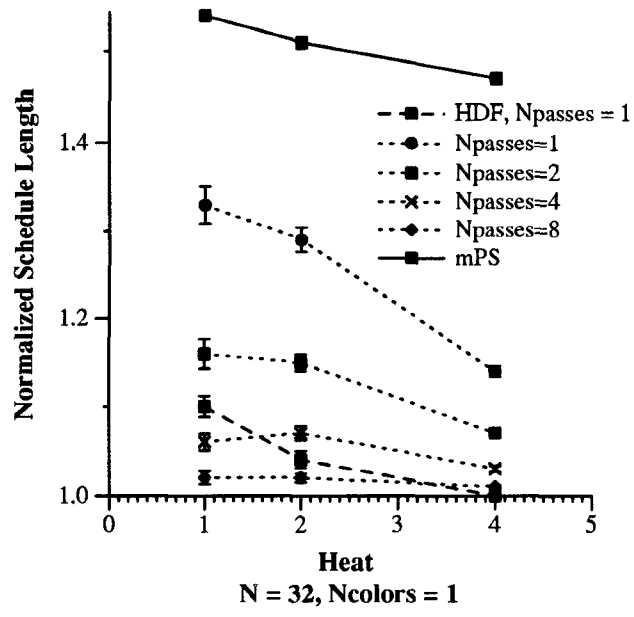

Figure 11: Impact of *HDF* and *MPASSES* on client hotspots

dynamic by dispatching each matching as soon as it is colored and then adding new arrivals as edges to the current graph before computing the next matching. However, for heuristics such as *HDF*, unreasonable delays may occur, suggesting that *MPASSES* may be more appropriate than *HDF* for the dynamic case. A heuristic intended to reduce the degree quickly in an edge coloring algorithm will not necessarily deliver unbiased service in a matching algorithm. A second reason for preferring *MPASSES* over *HDF* in dynamic algorithms can be seen in Figure 8. In the dynamic case, we expect the graphs to be denser because of new arrivals. This situation corresponds to the matchings seen on the left hand side of the figure, where *MPASSES* obtains larger matches than *HDF*.

Unlike our algorithm, PS cannot be easily converted into a dynamic algorithm because of the need to know the current degree of the graph. When new arrivals are added to the graph, it is no longer possible to use Panconesi and Srinavasan's analysis to predict Δ analytically and it is difficult to obtain the true graph degree in a distributed context.

4.2 Towards an Analysis of Running Time

The results in the previous section reveal much about the effects of heuristics and input characteristics on schedule length. However, we have not addressed the running time of the scheduling stage. The time required to generate a schedule is proportional to

$$\lceil \frac{L}{Ncolors} \rceil \times Npasses \times cost/pass, \qquad (1)$$

where L is the schedule length. Estimating the cost per pass depends on the details of the architectural model. Here we consider two extremes: a random crossbar network and communication via preassigned timeslots. Other protocols such as retry with exponential backoff and OCP are also possible.

In the first case, we consider a nonblocking crossbar switch in which each row and each column can only support one connection at a time. When two simultaneous rows or columns are requested, the collision is arbitrated in hardware. By a random crossbar switch, we mean a crossbar switch in which the hardware chooses from among all simultaneous requesters uniformly at random. When *H2* = *UAR*, we can allow the arbitration hardware in the crossbar to select winners instead of the servers. Of the clients sending a proposal to the server, only one will get through. The server acknowledges that client, thereby designating it as the winner. Clients that receive no acknowledgement are losers and must attempt another proposal. This approach has the advantage that the cost per pass is $O(1)$. However, it does not allow servers any strategy other than *UAR* for selecting winners. In particular, it would not support *HDF*. In a random crossbar network, there is no benefit to using *Ncolors* > 1 since the communication required for one phase with *Ncolors* proposals is identical to that required for *Ncolors* phases with one proposal.

At the other end of the spectrum we have preassigned timeslots. 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, at slot i each client j sends a message to server $(j+i)$ *mod* n. The communications cost of this protocol is $O(N)$ even if clients have fewer than N messages to send. If request lengths are short the impact of this expensive protocol is not so bad. In addition, the cost can be amortized over many communications by increasing *Ncolors*, up to N communications per phase.

Panconesi and Srinavasan permit each processor to receive and send one message to each neighbor during each phase, as well as unlimited computation. They associate

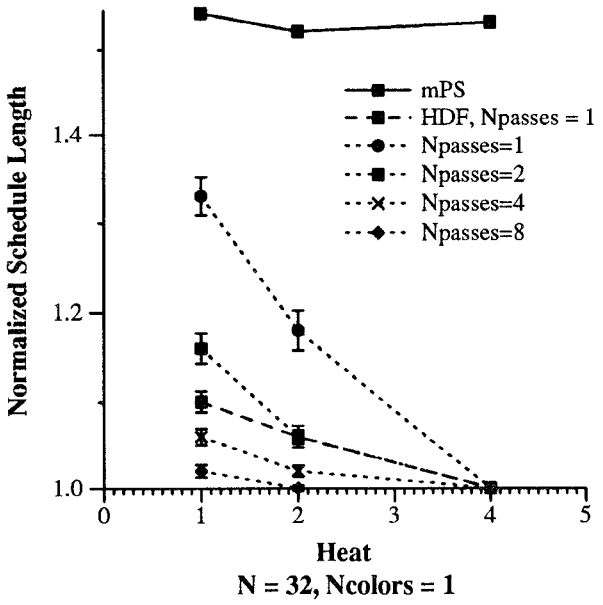

Figure 12: Impact of *HDF* and *MPASSES* on server hotspots

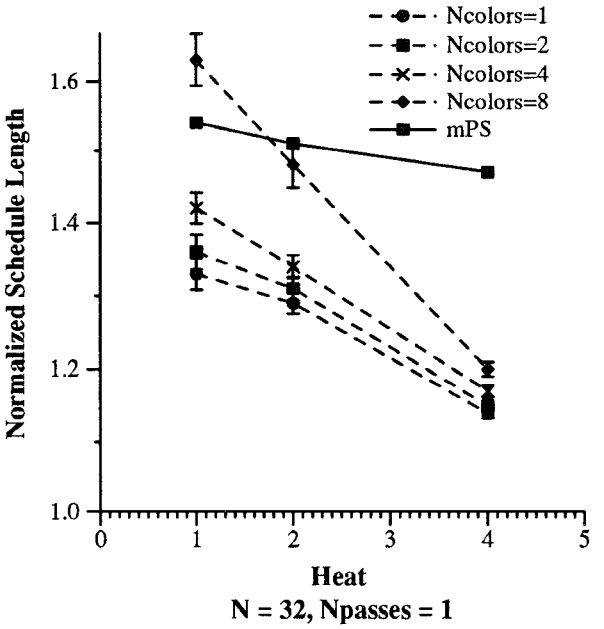

Figure 13: Impact of *Ncolors* on client hotspots

unit cost with each phase. In fact, this protocol permits up to $O(N\Delta)$ communications in each phase at unit cost. It is not clear how this is possible without Δ input ports on each processor, which would make prescheduling unnecessary. Using an adaptive number of colors per phase, *Ncolors* $= \Delta(i)$, allows PS to complete a schedule in $O(log\Delta)$ time. In comparison, we require L phases where experimental results suggest that $L \approx (1+\epsilon)\Delta$, where ϵ is generally less than .5. However, PS sends $O(\Delta(i))$ messages per phase whereas our algorithm requires only $O(1)$ messages per phase. A more detailed model of interprocessor communication is necessary in order to fully understand the complexity of our algorithm and that of PS.

Panconesi and Srinavasan use the initial degree of the graph as inputs. Their algorithm is based on probabilistic analysis to estimate the current degree of the graph at each stage and to estimate the number of steps required to complete a coloring. This is necessary for a batch algorithm since it is expensive to detect completion online. However, this approach can give a false measure of algorithm quality. If the probabilistic analysis to determine the number of phases required to route all messages is based on bounds which are not tight, an algorithm may run much longer than actually necessary to complete the communication. Comparing measured running times may give very different results than comparing predicted running times. In addition, obtaining the true degree of the graph may be difficult in a distributed system. Obtaining Δ also requires $O(\log \Delta)$ operations, potentially doubling the running time.

4.3 Open Problems

The above reflections suggest some open problems for future work.

1. The test data used in this article corresponds to relatively small machines with heavy traffic loads. The behavior of these algorithms for large N and sparse graphs should be studied. Uneven graphs ($|A| \neq |B|$) and other input configurations (e.g. multiple hotspots) are also important.

2. An analysis of running time, including a formal model of client/server communication that captures the details of the underlying architecture, is required.

3. Dynamic matching-based algorithms are needed in the context of telecommunications and multi-tasking applications. Issues of fairness and starvation in such algorithms need to be addressed.

5 Conclusion

In this article we have presented a parameterized, randomized edge coloring algorithm to preschedule data transfers in a fully connected distributed architecture. Some examples of such architectures include parallel I/O subsystems, multimedia information systems and input queued packet switches. By prescheduling data transfers, we eliminate contention for I/O ports while maintaining efficient use of interconnection bandwidth. Modeling the outstanding I/O requests as a bipartite graph allows us to develop algorithms based on the graph theoretic notions of edge coloring and bipartite matching.

We used simulation to measure the length of the schedule required to complete a set of data transfers with these algorithms. This is equivalent to the number of colors required to color the edges of the associated bipartite transfer graph. We studied the performance and behavior of our algorithm experimentally and compared it to the work of others. We evaluated two approaches to reducing the schedule length: the *highest degree first* or *HDF* heuristic, which favors nodes with a heavy I/O load, and *multiple passes* (*MPASSES*)

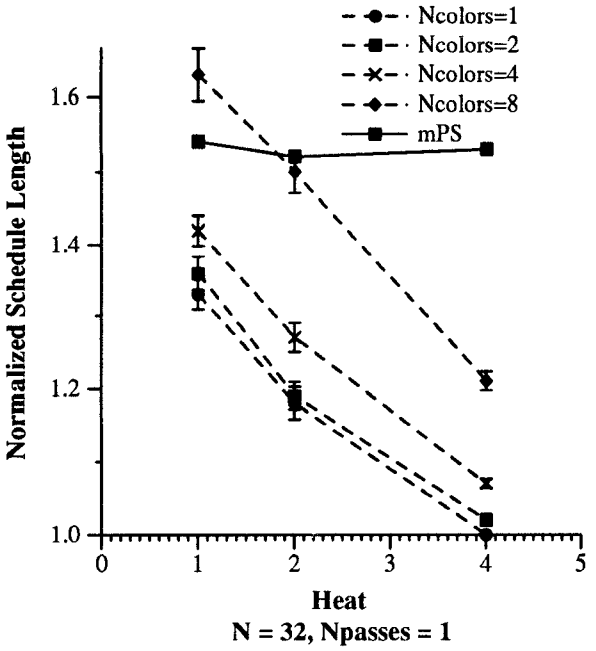

Figure 14: Impact of *Ncolors* on server hotspots

which uses additional communication to increase bandwidth utilization.

Both approaches yielded good performance. When compared with the optimal schedule achieved in the centralized case, our best algorithm yielded schedules within 5% of the optimal solution. Previous work by other authors required roughly 50% to 60% above the optimal schedule.

Our simulations also gave insight into how the two methods differed in achieving these results. From looking at traces of individual runs, we concluded that *MPASSES* is more suitable for dynamic applications such as communications switches and I/O in multitasked systems. *HDF* is more appropriate for batch oriented problems like "out-of-core" algorithms. *MPASSES* incurs a higher communication cost whereas *HDF* requires greater computational power on the disks. Our parameterized approach offers an algorithm that can be tailored to a wide variety of architectural settings.

We also considered computing several matchings simultaneously to reduce the time required to generate schedules. While this can reduce the number of phases required to obtain a solution, this approach was shown to incur a penalty in increased schedule length. This previously unidentified tradeoff partially explains the improvement in performance obtained by our algorithm when compared with those of other workers.

The use of experimental analysis allowed us to study the potential of heuristics in improving schedule lengths. Results demonstrated the ability of heuristics with different communication/computation tradeoffs to deliver comparable performance. Because experimental results reveal the internal behavior of these heuristics, our studies yield insights into how best to apply the algorithm in a variety of architectural contexts. They also allowed us to study how algorithms vary with input distribution.

Acknowledgements

We thank Alessandro Panconesi and Aravind Srinavasan 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 Transactions on Computer Systems*, 11(4):319–352, November 1993.

[2] R. J. Anderson andG. L. Miller. Optical Communication for Pointer-based Algorithms. Technical Report CRI-88-14, Computer Science Department, University of Southern California, 1988.

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

[4] E. G. Coffman, Jr., editor. *Computer and Job-Shop Scheduling Theory*. John Wiley, 1976.

[5] Thomas H. Corman. Fast Permuting on Disk Arrays. *Journal of Parallel and Distributed Computing*, 17:41 –57, January 1993.

[6] 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.

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

[8] L. A. Goldberg, M. Jerrum, T. Leighton, and S. Rao. A Doubly Logarithmic Communications Algorithm for the Completely Connected Optical Communication Parallel Computer. In *1993 Symposium on Parallel Algorithms and Architectures*, pages 300–310, 1993.

[9] L. A. Goldberg, M. Jerrum, and P. D. MacKenzie. An $\omega(\sqrt{\log \log n})$ Bound for Routing in Optical Networks. In *1994 Symposium on Parallel Algorithms and Architectures*, pages 147–156, 1994.

[10] Mario Gonzalez, Jr. Deterministic Processor Scheduling. *Computing Surveys*, 9:173, Sept. 1977.

[11] Kai Hwang, editor. *Advanced Computer Architecture*. McGraw Hill, 1993.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] P. D. MacKenzie, C. G. Plaxton, and R. Rajaraman. On Contention Resolution Protocols and associated Probabilisitic Phenomena. In *Proceedings of the ACM Symposium on the Theory of Computation*, volume 26, 1994.

[18] 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.

[19] 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.

[20] A. Panconesi and A Srinavasan. An Extension of the Chernoff-Hoeffding Bounds with an Application to Distributed Edge Coloring. To appear in SIAM Journal of Computing, 1995.

[21] L. G. Valliant. General Purpose Parallel Architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, page 967. Elsevier, 1990. Chapter 18.

[22] W. A. Wulf and C.G. Bell. C.mmp – A Multi-Miniprocessor. In *Proceedings of the Fall Joint Computer Conference*, pages 765–777, 1972.