DATA TRANSFER SCHEDULING*

Kiran Kumar Somalwar

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-88-31 September 1988

* This work was partially supported by the IBM Corporation under Contract Number 616513.

DATA TRANSFER SCHEDULING

by

KIRAN KUMAR SOMALWAR, B. Tech.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF ARTS

THE UNIVERSITY OF TEXAS AT AUSTIN

December, 1988

Abstract

The problem of data transfer scheduling is a problem of multiple resource allocation. It is cast as an edge coloring problem. Several exact algorithms for edge coloring, in the literature, are described. A new algorithm for edge coloring, when no color can be used more than k times, is designed and analyzed. Several inexact algorithms are described and evaluated for their cost effectiveness.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Data transfers are an intrinsic part of any computation. Data transfer operations occur in unit sizes ranging from individual bytes or words, typically on a bus within a processing element, to megawords on a channel between high-speed disks and large RAMs. It is frequently the case that many sources and sinks for data transfer operations use some common set of resources for the execution of their operations. These resources again span the scale from buses, coupling functional units in memory, and pools or channels coupling multiple disk controllers to multiple processors in a multi-processor architecture.

The use of shared resources introduces a contention, and thus the need, for scheduling of operations to effectively utilize the shared resources. It is becoming clear that in many circumstances scheduling may become an important factor in the performance of data transfer operations. Scheduling problems were studied in the 1960's when many processors typically contended for a few disks. There has been, however, less interest in recent years. This data transfer scheduling is now again becoming important as macro-level parallelism is becoming more and more prevalent in the architecture computer of systems. Increase in number of processors is requiring a much greater focus on management of resources for external data transfer operations.

Scheduling of data transfer operations is intrinsically different from

the classical scheduling problems of mapping requests for service to processors or allocating blocks of memory to processes. The state of the system is typically distributed and the information necessary to make a scheduling decision may not be immediately available at any one site. It is sometimes the case that multiple entities must elaborate to effect a data transfer. Scheduling must often coordinate the scheduling of the requests of several entities. Therefore the concepts typically used for scheduling of computer work to processors, etc. are not directly applicable to data transfer scheduling. This thesis defines scheduling problems in terms of familiar concepts from graph theory and defines algorithms for execution of these scheduling operations, investigates their properties and evaluates their effectiveness.

The next section expands upon the problem statement, the approach and the results.

1.1 Overview of the Thesis

In this section, we define the problem, briefly explain our approach, and summarize the results.

1.1.1 Problem Statement

Optimal scheduling of a set of data transfers in a parallel system requires the optimal assignment of resources required for the data transfers. An optimal schedule is one which meets the criteria of the minimum schedule length for the set of data transfers, or the maximum throughput of the channels. It must map the resources of the system, in the form of processing nodes and channels, to the data transfers. The minimum resources required to complete a data transfer are the processing nodes involved in the data

transfer, and the channel connecting them. More resources, in the form of channels and processing nodes may be required if there is no direct channel connecting the two nodes between which the data transfer is required.

A set of data transfers can be represented by a weighted undirected graph, G=(V,E) where V defines the set of processing nodes, and E specifies the data transfers that are required to be done. If an edge (x,y) belongs to E, then it means that a data transfer is required to be done between x and y. The weight associated with each edge gives the cost of the data transfer. In other words, the weight associated with an edge gives the time required to complete the data transfer, assuming a channel of unit bandwidth.

We also study the problem of dynamic scheduling of data transfers. This analysis applies to a more realistic situation in which new data transfers are added to the graph G, at the end of each time unit. That is, new data transfers are required to be done, even when the original set has not been completely scheduled.

The research described in this thesis is probably the first attempt at the problem of scheduling data transfers, in its general form. This problem has been ignored by researchers earlier mainly because parallel systems have become popular only recently, and due to the general feeling that scheduling is not really cost-effective, if the data transfers are simple and have low cost. Scheduling is definitely cost-effective if the data transfers have a high cost. Such data transfer operations typically arise in interactions with I/O systems, and I/O intensive applications.

1.1.2 Approach

We use a well studied problem in classical graph theory, namely the edge-coloring problem, as the basic paradigm for optimal scheduling of the data transfers. The edge-coloring problem[6] is to assign colors to the edges of a graph, so that no two adjacent edges[1] have a common color. The edge-coloring paradigm is directly applicable to the class of data transfer scheduling problems in which there is a direct path between every pair of nodes, and all transfers are of unit cost. The paths need not be unique or seperate. Other data transfer scheduling problems can be mapped to this simpler problem. Specifically, the problem of data transfer scheduling, when the number of channels in the system is limited, is cast as what we call the limited edge coloring problem. Consequently, the edge-coloring paradigm is applicable to more complex data transfer scheduling problems.

It should be noted, however, that not all data transfer scheduling problems can be effectively and optimally solved using this paradigm. For example, consider the situation wherein we have multiple paths connecting any pair of nodes. The edge-coloring paradigm, defined and characterised herein, will fail to give the optimal solution to this problem, because it does not have the 'power' to make the correct choice of the path, in addition to making the choice of the pair of nodes between which the data transfer is to be done. We hypothesize that the edge coloring paradigm, will however give a good solution.

[1]Two edges are adjacent if and only if they have a common vertex.

1.1.3 Results

Several exact algorithms are defined and evaluated for their cost-effectiveness. Heuristic algorithms, with lower cost, are proposed and evaluated. The applicability of the algorithms to the dynamic situation wherein new data transfers are added to the graph, at the end of each time unit, and the set of data transfers have not been completely scheduled, is also studied and evaluated.

New algorithms with improved run-time performance are developed for the edge-coloring problem, and applied to the data transfer scheduling problem

1.2 Organization of the Thesis

In chapter 2, the definitions and the notation used through the rest of the thesis are given. The problem is defined in complete detail in Chapter 2. Classes of data transfer scheduling problems are defined. We discuss the edge-coloring paradigm, and study the mappings required to be done to the various data transfer scheduling problems, so that the paradigm can apply. In Chapter 4, we discuss two representative algorithms in the literature. In chapter 5, we present a new algorithm with improved time performance for limited edge coloring, and heuristic algorithms for edge coloring. In Chapter 6, we present the analysis and cost-effectiveness of the algorithms. The concluding remarks are contained in Chapter 6. We present a summary of our research, and suggest future research directions.

Chapter 2

Basic Definitions and Notation

In this chapter, we present the definitions of terms and the notation we use in the rest of this thesis. Most of the definitions are for graph-theoretic terms[6].

A *graph* G is a pair (V, E) where V is a finite non-empty set of elements called *vertices* and E is a finite set of distinct unordered pairs of elements of V called *edges*. V is called the vertex set of G, and E is called the edge-set of G. Figure 2.1 gives an example of a graph. If $e = (x, y)$ is an edge in G, then e is said to *join* the vertices x and y. We also say that e is *incident* to x and y. We can also say that x is a *neighbor* of y and vice-versa. Two edges are *adjacent* to each other if they have a common vertex. Two edges are *independent* if they are not adjacent to , that is, they have no vertex in common. For example, in Figure 2.1, (a,f) and (c,f) are adjacent to each other, the common vertex being f.

A *multigraph* is a graph in which the edges may occur several times. Figure 2.2 gives an example of a multigraph. If G is a multigraph, then two or more edges of G joining the same pair of vertices x and y are *multiple edges*; the number of such edges is called the *multiplicity* of (x,y).

For each vertex x of a graph or multigraph G, the number of edges incident to x is called the *degree* of the vertex. The *maximum degree* of a

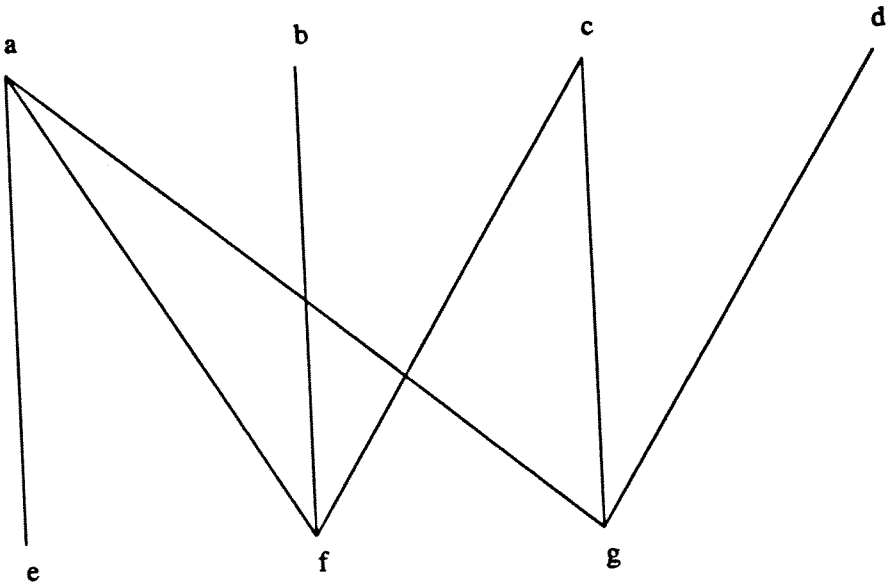

Figure 2.1: A graph.

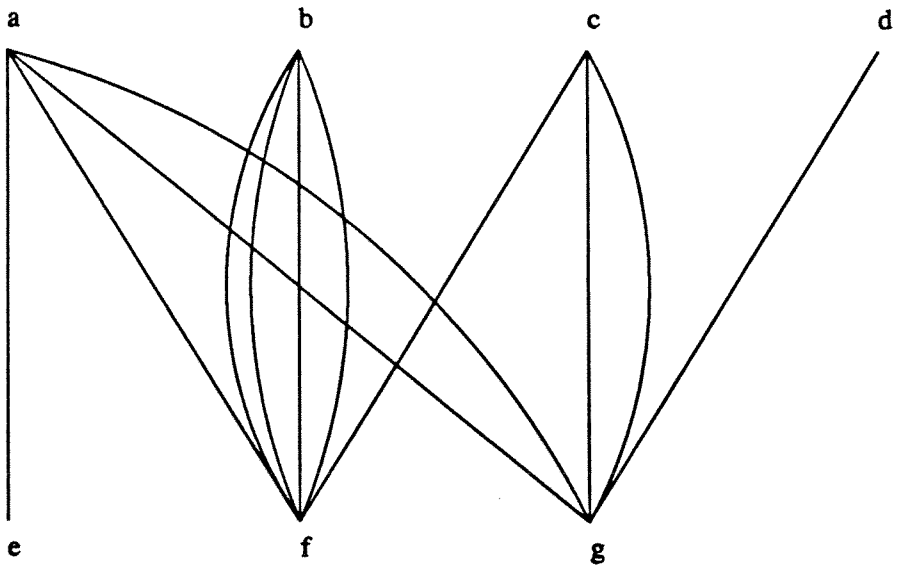

Figure 2.2: A multigraph.

graph is maximum of the degrees of all its vertices. The *combined degree* of an edge is the sum of the degrees of the two vertices it is incident on. We will usually use the term degree to refer to the maximum degree of the graph. If all the vertices of a graph G have the same degree, then G is called a *regular* graph. A graph, G = (V,E), is said to be *bipartite* if its vertex set can be partitioned into two sets A and B, such $V = A \bigcup B$ and $A \bigcap B = \phi$[1], and the only edges in the graph are those that join vertices in A to vertices in B. A bipartite graph will be represented by a triple of the form (A,B,E), through the rest of this thesis. A graph is said to be *k-complete* with respect to a constant p, if

1. All vertices have degree less than or equal to p.
2. The number of edges in the graph, m is equal to k × p.

Figure 2.3 gives an example of a k-complete graph.

A *matching*, on a graph, is a set of edges such that no two edges in the set are adjacent to each other. The set {(a,e), (b,f)} is a matching on the graph in Figure 2.1. A *maximum matching* is a matching that has the maximum number of edges in it. The *vertex cover*, or simply *cover* of a set of edges S is the set of vertices W such that S contains all and only vertices in W. A *critical matching* on a graph is the matching whose vertex cover includes all the maximum degree vertices of the graph. For the graph in Figure 2.1, the matching { (a,e), (c,f), (d,g) } is a critical matching. As an aside, a critical matching need exist for bipartite graphs only.

[1] ϕ represents the null set.

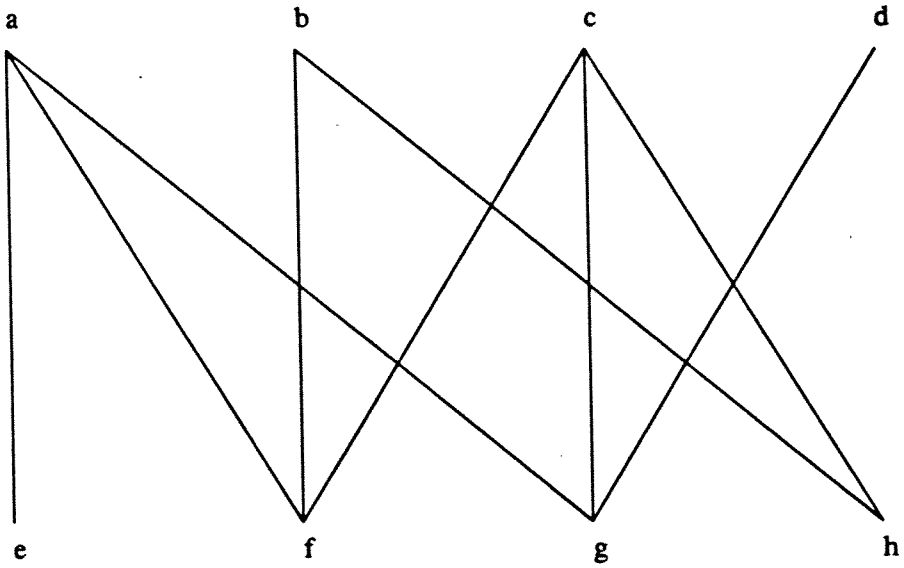

Figure 2.3: A 3-complete graph with respect to p = 3.

An *edge-coloring* of a graph (or multigraph) is an assignment of colors to edges so that no two adjacent edges have the same color. Figure 2.4 gives an edge-coloring for the graph in Figure 2.1. The numbers alongside the edges are the colors assigned to them. The minimum number of colors required for edge-coloring a graph is called the *chromatic index*, and is denoted by χ.

An *euler partition* of a graph is a partition of the edges of a graph into open and closed paths, so that each vertex of odd degree is at the end of exactly one path, and each vertex of even degree is at the end of no paths. An *euler split* of a bipartite graph G=(A,B,E) is a pair of bipartite graphs G_1=(A,B,E_1) and G_2=(A,B,E_2) where E_1 and E_2 are formed from an euler partition of E by placing alternate edges of the paths into E_1 and E_2. Any vertex of even degree in G will have half the degree of G, in both G_1 and G_2,

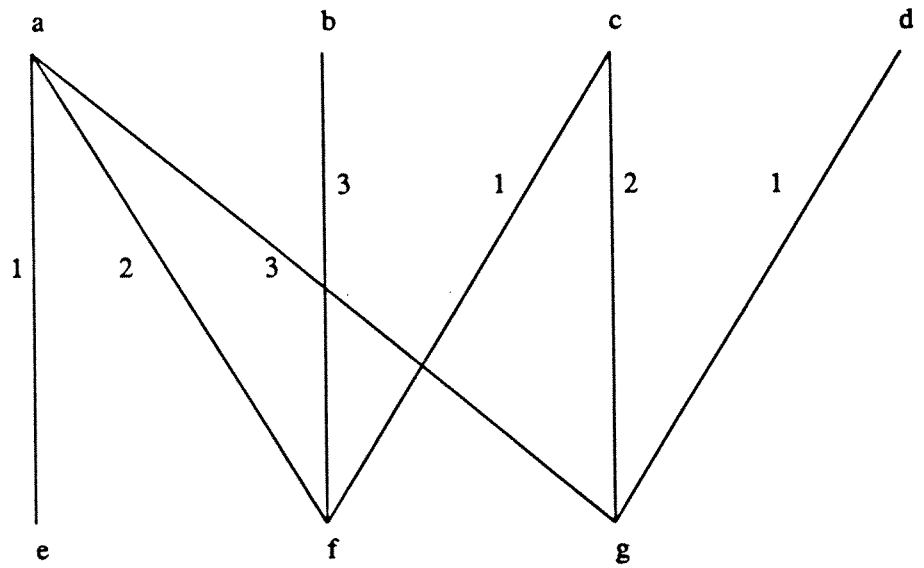

Figure 2.4: An edge coloring of the graph in Fig. 2.1.

while any vertex of odd degree in G will have degrees in G_1 and G_2 differing by 1. This implies that both G_1 and G_2 will have their maximum degree as $\lceil d/2 \rceil$, if d is the maximum degree of G. A *perfect euler split* is an euler-split (G_1, G_2) such that the number of edges in G_1 and G_2 differ by at most 1. That is, G_1 and G_2 have an equal number of edges, if the number of edges in G is even, or differ in the number of edges by 1, if the number of edges in G is odd. An euler split and a perfect euler split, as defined, need exist only for bipartite graphs. Figure 2.5 gives an euler split for the graph in Figure 2.1, and Figure 2.6 gives a perfect euler split for the same.

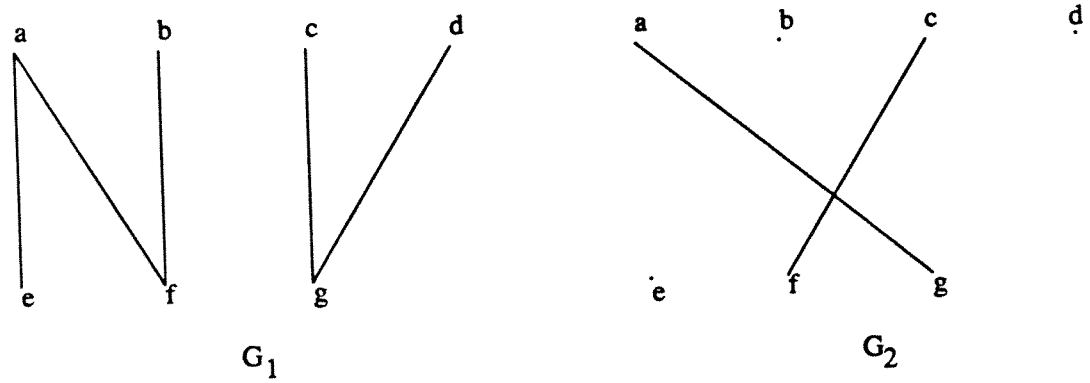

Figure 2.5: An euler split of the graph in Fig. 2.1.

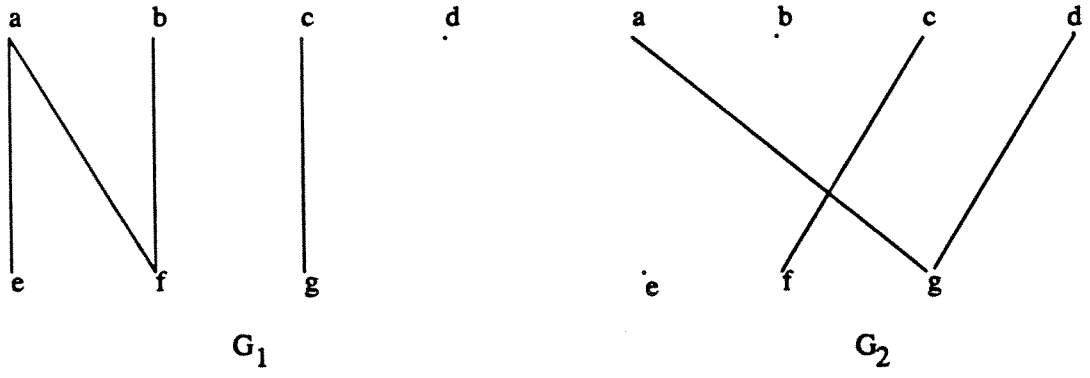

Figure 2.6: A perfect euler split of the graph in Fig. 2.1.

Chapter 3

Problem Statement

Typical scheduling algorithms for data transfer in a single channel system (eg. a system with all nodes connected on a bus) have been based on first come first served queues, priority queues, CSMA/CD[18]. These algorithms are obvious and easy to analyze because of the presence of only one channel in the system. Moreover, as the number of channels is just one, the choice of the algorithm is usually determined by other factors . For example, CSMA/CD protocols are used in systems where centralised control is not possible. First come first served queue algorithms are rarely used for bus allocation, or bus scheduling because of the need for an expensive centralised bus controller in hardware, to implement the algorithm. Typically, bus allocation algorithms have been based on static priority or dynamic priority.

The situation is drastically different when there are multiple channels in the system. The reason for having multiple channels (through which several nodes can complete their data transfers simultaneously), is to provide large bandwidth and thus satisfy system requirements. It is important that algorithms be designed so that the total bandwidth provided by the channels is utilized to the maximum possible extent. Very little work has been done on the problem of scheduling data transfers in a multiple channel system. Data transfers between memories and processors in many current parallel systems [19] have been supported by multistage interconnection networks (e.g.,

RP3[19]). The multistage interconnection networks provide multiple simultaneous paths for data transfers. Data transfer operations in such systems are low-cost memory operations, and it has not been cost-effective to schedule them. They have been scheduled on an ad-hoc basis. In the case of multistage interconnection networks, each node in the interconnection network schedules the data transfers that need to go through it, on a first come first served basis.

In this thesis, we are concerned with the usage of such multichannel systems to provide data transfer paths and bandwidth between the processors and external memory units. Since the data transfers that are involved are themselves quite expensive, it is a good idea to schedule them. That is, the overhead of scheduling the data transfers is cost-effective in terms of the lesser time it takes to complete the data transfers, if they are scheduled. For example, in many seismic applications, the basic unit of transfer from the external memory device is a 1000 by 1000 matrix of real numbers (4 bytes). Assuming a channel rate of 4 MB/sec, which is typical of many systems, time required to complete the transfer is 1 second. This is a large time, and hence scheduling such data transfers should be cost-effective.

3.1 Graph-theoretic Representation of the Problem

We present a graph-theoretic model for the problem of scheduling data transfers. The set of data transfers that need to be accomplished can be represented by a graph G = (A, B, E) where A is the set of processor nodes involved in the data transfers, B is the set of external memory units involved in the data transfers, and E is the set of data transfers themselves. An edge e = (x, y) in E represents the data transfer between x and y. There can be

multiple edges between a pair of nodes in the graph. Strictly speaking, such a graph is called a multigraph. The weight of the edge represents the cost of the data transfer. Our goal is to schedule the data transfers so that the total time taken is the minimum possible. In the rest of this thesis, will use the expression 'schedule of a graph G' to refer to the schedule of the data transfers represented by G.

3.2 Method of Solution

We shall initially consider the situation wherein all the edges of the graph have equal weights, and then extend the analysis to the general problem, wherein edges have different weights. We assume a full connectivity network currently, and extend the method to the multiple bus configuration in Section 3.2.3.

3.2.1 Uniform Cost Problem

We assume that all the edges in the graph have the same unit cost. The problem of scheduling a graph G, assuming edges in G have equal weights, is equivalent to the problem of edge coloring of G. An edge coloring of G is an assignment of colors to edges so that no two adjacent edges are assigned the same color. The minimum edge coloring is a coloring that uses the minimum number of colors.

Lemma 3.1 *An edge coloring for a graph G gives a schedule for the data transfers represented by G. The converse is also true.*

Proof: Assume that an edge coloring for G, using r colors $1 \ldots r$, was found. All edges colored with a single color, say i, are independent of each other ,

and represent concurrently schedulable data transfers. This means that they can be scheduled to run simultaneously. Since there are r colors, and edges (actually data transfers represented by the edges) in each color are scheduled to run concurrently, this method of scheduling gives a schedule that takes r time units. Conversely, given a schedule, all edges which are scheduled at one time can be given the same color. Consequently, if the schedule takes r time units, then the corresponding coloring has r colors.

Corollary 3.1 *A minimum edge coloring for a graph G gives a minimum schedule for the data transfers represented by G.*

Proof: In the lemma above, we proved that for any graph, if there is a coloring for the graph with r colors, then there is a schedule that takes r time units. It was also shown that for every schedule that takes r time units, there is an edge-coloring with r colors. As a result of this one to one mapping, a minimum edge-coloring for G gives the minimum schedule for the data transfers represented by G.

We assume here that there is no limit on the available communication resources, that is, the number of available communication channels is not less than the cardinality of the largest edge set with the same color. In Section 3.2.3, we consider a situation wherein the number of communication channels is limited.

3.2.2 Extension of the Method to Weighted Edges

We extend our method to the situation in which edges have weights corresponding to the cost of the data transfers. However, a few assumptions are made about the problem. It is assumed that the weights are integers, and

that each data transfer can be split up into parts, and each part transferred separately. Specifically, we assume that each data transfer can be split up into as many parts as its weight. Unless, we make this assumption, it seems difficult to use the algorithms and transformations we define in this thesis to solve the problem.

The problem of scheduling a weighted graph G is equivalent to the problem of coloring a multigraph G' obtained as follows. Each edge of weight x is replaced with w edges each of weight one, between the same pair of vertices. G' has the same edges as G, but the multiplicity of each edge is equal to the weight of the edge in G. More accurately, if G has an edge (v, w) of weight y, then G' has y edges between v and w. Notice that, for the schedules derived by this method, a required data transfer is partitioned into a set of smaller data transfers, and these smaller data transfers may be scheduled at different times.

3.2.3 Extension of the Method to the Multiple Bus Configuration

In a multiple bus configuration, with k buses, not more than k data transfers can be done concurrently. We assume full connectivity through the multiple bus system. Figure 3.1 gives a diagrammatic representation of a typical system under consideration. An example of such a system is the I/O organization of the parallel machine RP3 [19]. Figure 3.2 gives the I/O organization of the RP3. In addition, only independent data transfers are permitted to take place concurrently, as was the case with the previous problems. So, the problem of finding an optimal schedule is equivalent to the problem of finding the minimum coloring of a multigraph, under the constraint that any color cannot be used more than k times. In the rest of

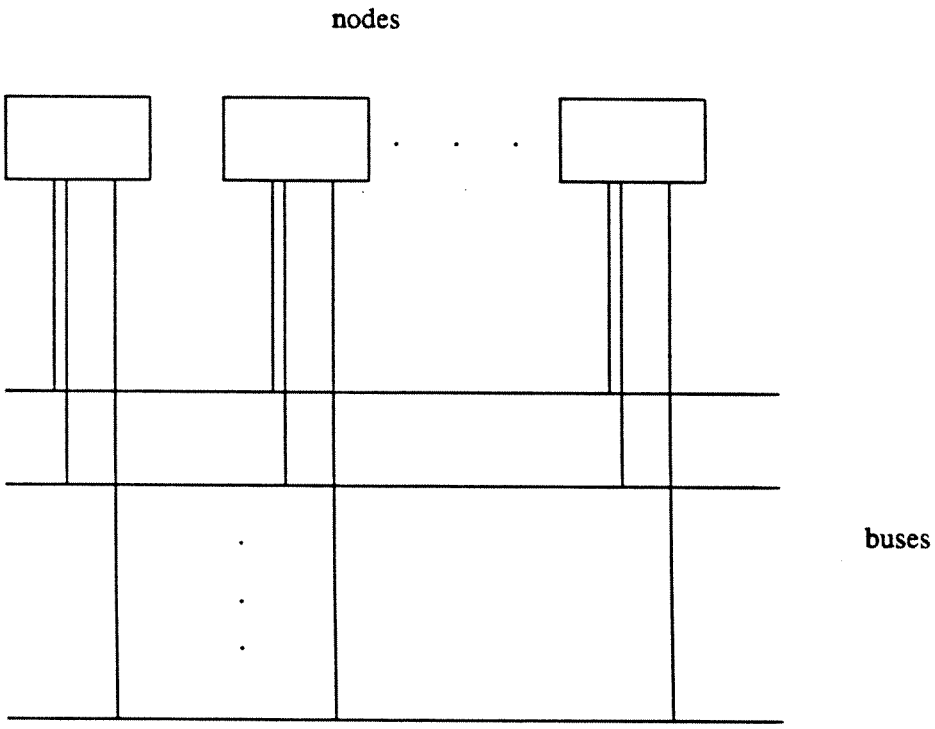

Figure 3.1: A typical multiple bus system organisation.

this thesis, we will refer to the problem of finding the minimum coloring of a multigraph as a *general coloring* problem, and the problem of finding the minimum coloring of multigraph, under the constraint that any color cannot be used more than k times as the *limited coloring* problem.

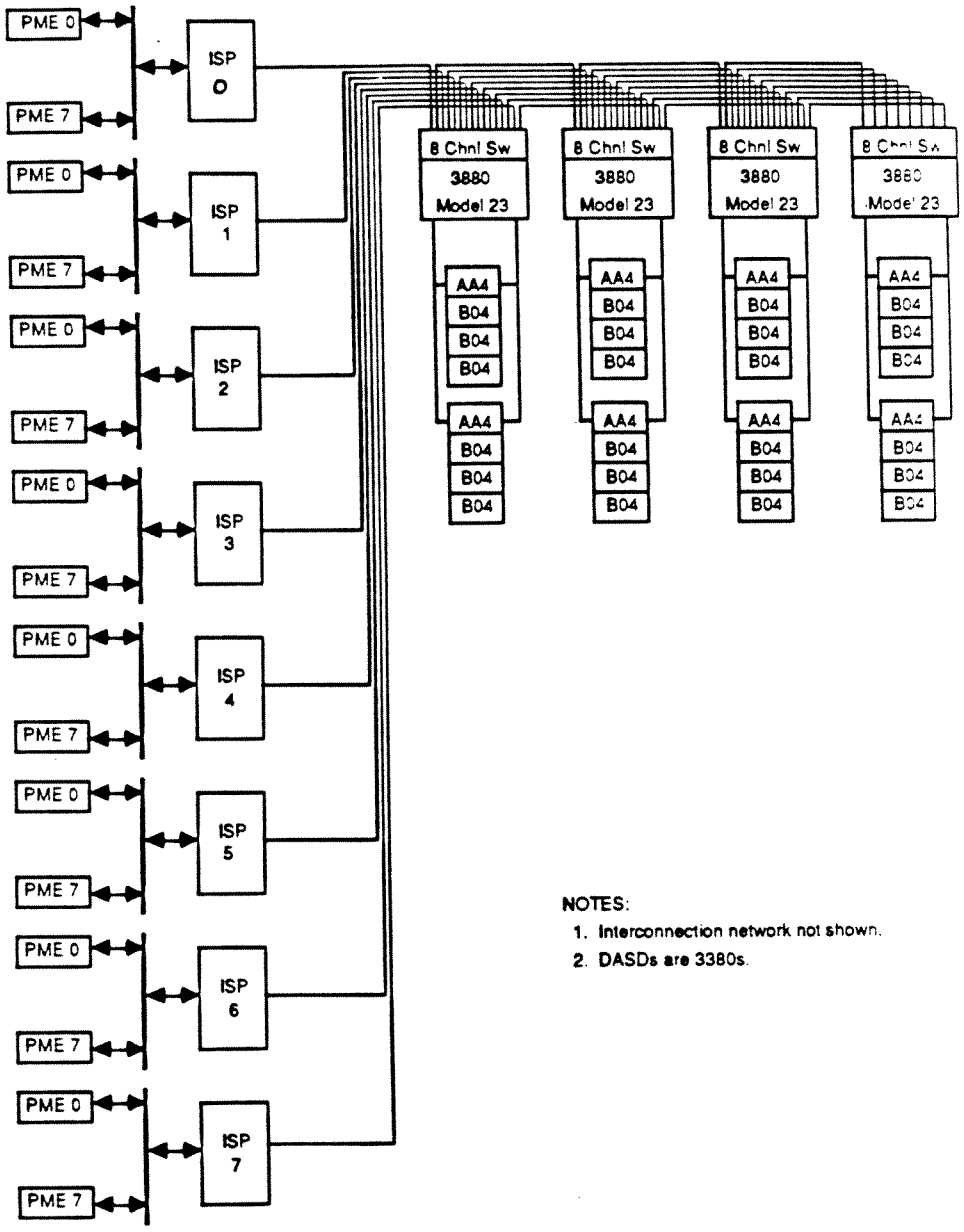

Figure 3.2: RP3 I/O organisation.

Chapter 4

Results and Previous Work

The edge-coloring problem has been studied by graph-theorists quite extensively. A substantial amount of work has been done on the problem, in terms of algorithms and theoretical results [4,5,6,7,8,10,11,13,16]. We are interested only in edge-coloring of bipartite graphs, hence we will concentrate on it. Edge-coloring of general graphs is a tougher problem, and the results are somewhat different. The interested reader is referred to [6,11,13].

The following theorem is useful in proving the theorem after that, which specifies the minimum number of colors required to color a bipartite graph. Theorem 4.3 addresses the problem of limited coloring, when no color can be used more than k times.

4.1 Basic Theorems on Edge Coloring

Theorem 4.1 *[1] Every bipartite graph has a matching which covers all vertices of the maximum degree.*

Proof: The proof is given in [1].

Theorem 4.2 *[6] The minimum number of colors needed to color a bipartite graph, or multigraph, of maximum degree d, is d.*

Proof: The proof is by induction on the degree of the graph. From Theorem 4.1 above, for every bipartite graph there exists a matching which covers all the vertices whose degree is equal to the maximum degree of the graph. If M is such a matching, the graph $G - M$ would be a bipartite graph of degree $d - 1$. By induction hypothesis, the graph $G - M$ can be colored with $d - 1$ colors. The edges in M are all independent of each other, and can be assigned a single color. Hence G can be colored with d colors.

As an aside, if G is a general graph, then either d or $d+1$ colors will be required to color G. It is an NP-complete problem to determine whether d or $d + 1$ colors are required to color the graph.

We now turn our attention to the limited coloring problem, namely the problem of edge-coloring a bipartite graph, assuming that no color can be used more than k times. The following theorem states the minimum number of colors required to color a bipartite graph of degree d, when no color can be used more than k times. In fact, in the next chapter, we show constructively the exact number of colors required to find a limited coloring of a graph.

Theorem 4.3 *The minimum number of colors required to color a bipartite graph, which has m edges, and whose degree is k, such that no color is used more than k times is* $\geq \max(d, \lceil m/k \rceil)$.

Proof: The proof is rather straightforward. Since there are m edges to be colored, and not more than k edges can be colored with a single color, we need at least $\lceil m/k \rceil$) colors to color the graph. Also, since the maximum degree of the graph is d, there is a vertex which has d edges incident on it. As each of the d incident edges on this vertex needs to be colored with a different color,

at least d colors are required to limited-color the graph. Hence, the minimum number of colors required to limited-color a bipartite graph is $\max(d, \lceil m/k \rceil)$.

In the following two sections, we present two representative algorithms, one for the general edge-coloring problem [7], and one for the limited edge-coloring problem [2]. Several other algorithms are available for both these problems [4,5,8]

4.2 Gabow's Algorithm[7]

This algorithm is based on the divide and conquer technique. If the graph is of even degree, then the graph is divided into two graphs, each of whose degree is half the degree of the original graph. Each of these two new graphs is colored seperately. If the graph is of odd degree, additional work is done before divide and conquer can be applied to obtain the minimum coloring. The procedural description of the algorithm follows.

procedure color1(G,d) **comment** Obtains the edge-coloring
of graph G, of degree d.
begin

1. **if** d is odd
   begin
2. Find a matching M that covers every vertex of degree d,
   and assign the edges in M a single color.
3. G = G - M.
   end
   comment G has even degree now.
4. **if** G is not an empty graph
   begin

5. Divide the graph into two graphs, G_1 and G_2, such that both G_1 and G_2 have degree $\lfloor d/2 \rfloor$.

6. color1($G_1, \lfloor d/2 \rfloor$).

7. color1($G_2, \lfloor d/2 \rfloor$).

end

end

Steps 2 and 5 are elaborated further. If the graph has even degree, then it is divided (actually, the edge set is divided) into two graphs G_1 and G_2 such that both have maximum degree $d/2$. In fact, G_1 and G_2 are the euler split of G. By Theorem 4.1, both G_1 and G_2 can be colored with $d/2$ colors each, thus giving a coloring for G using d colors. If the graph has odd degree, a matching covering all the maximum degree vertices is found. the edges in the matching are removed from the graph, to obtain a graph G', whose degree is $d-1$. The edges in M are all independent of each other, hence they can be assigned a single color. Since G' has its degree even now, divide and conquer technique can be applied on G' to obtain a coloring for G' with $d-1$ colors only. Thus G can be colored with d colors.

Step 2 is implemented, utilizing a technique due to Mendelsohn-Dulmage [14]. The procedural description for realizing Step 2 is given below, followed by an explanation.

procedure MD(G,d) **comment** Finds a matching covering all the maximum degree vertices in G. Let G=(S_1, S_2, E)

begin

1. **for** i=1,2 **do comment** find a matching M_i that covers every vertex of maximum degree d in S_i

begin

2. Let T be the set of vertices in S_i that do not have maximum degree
3. Let H be the multigraph G - T
4. Let M_i be a maximum matching on H

end

comment form the required matching M from M_1 and M_2

5. $M = M_1 \bigcap M_2$
6. $N = M_1 \bigoplus M_2$
7. **for** each connected component C of N **do**

begin

8. Let C be the sequence of edges e_1, ... e_r
9. **without loss of generality** assume that C starts with a vertex of degree d
10. **for** i=1 **stSep** 2 **to** r **do**
11. put e_i in M

end

end

From Theorem 4.2, it is obvious that there is a matching covering all the maximum degree vertices in S_1. The same is true for S_2. The respective matchings M_1 and M_2 are found in steps 1 to 4. To find M_i, all the vertices (and edges incident on them) in S_i that do not have the maximum degree are deleted from the graph, and a maximum matching found on the resulting graph. Such a maximum matching will cover all the maximum degree vertices in S_i.

Steps 5 to 11 find the required matching. The edges that are common to both M_1 and M_2 are included in M directly. Since a vertex is incident to at most two edges in $M_1 \oplus M_2$, one in M_1 and one in M_2; so a connected component of $M_1 \oplus M_2$ is a path with edges alternately in M_1 and M_2. The symmetric difference of M_1 and M_2 consists of 5 types of paths and cycles, as shown in Figure 4.1. In each case it is possible to select a matching M' $\subseteq M_1 \oplus M_2$ such that M' covers all the vertices of S_1 covered by $M_1 - M_2$, and all the vertices of S_2 covered by $M_2 - M_1$. Then $M = M' \bigcup (M_1 \bigcap M_2)$ is the required matching. Steps 7 to 11 find the matching M'. For each connected component C, put all the edges of $M_i \bigcap C$ in M, where i is chosen as follows. If C is an open path of odd length, choose i so that $\mid M_i \bigcap C \mid$ is maximum. If C is an open path of even length, exactly one end of C is a vertex of degree d; choose i so that M_i covers that vertex. If C is a closed path, choose i arbitrarily.

Step 5 in color1 is realized by the method of euler partition. $\{G_1, G_2\}$ is the euler split of G. The euler partition of G is obtained as follows[1]: Choose a vertex of odd degree; if none exists, choose a vertex of even, nonzero degree. Traverse an edge from that vertex to another and erase the edge. Continue traversing and erasing the edges until a vertex with zero degree is reached. This gives a path in the partition. Then choose a new start vertex and repeat the process. Do this until no possible start vertex remains. The euler split is formed from the euler partition by placing alternate edges of the paths into E_1 and E_2. The procedural description for the euler split is as follows.

procedure ES(G) **comment** finds an euler split of G.
begin

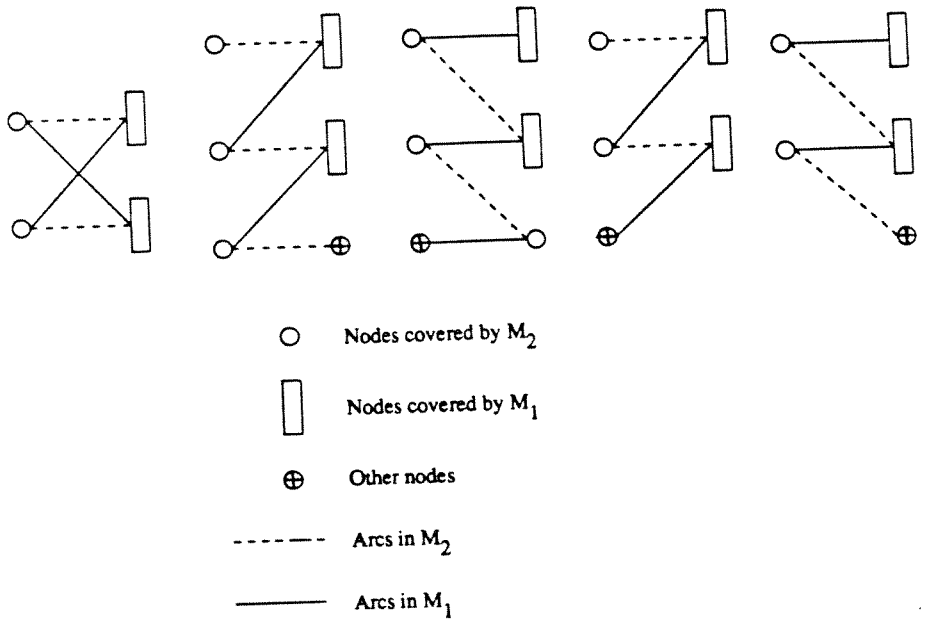

Figure 4.1: Symmetric difference $M_1 \oplus M_2$.

```
comment find the euler partition first.
1.    make P an empty list, and S an empty queue
2.    add all vertices of odd degree to the queue S
3.    add all vertices of nonzero even degree to the queue S
4.    while S is nonempty do
      begin
5.        let s be the vertex at the front of the queue in S
6.        delete s from S
          comment vertex s may have degree 0, since edges are deleted
          from G in line
7.        if vertex s has nonzero degree
          begin
8.            let p be an empty path
9.            v = s
10.           while vertex v has nonzero degree do
              begin
11.               let (v,w) be an edge in G
12.               put(v,w) in p, and delete it from G
13.               v=w
              end
14.           put the path p in P
15.           if vertex s has nonzero degree
16.               put s in S
          end
      end
      comment P is an euler-partition of the graph
17.   let E1 and E2 be empty edge sets
```

18. **for** each path p in P

19. traverse p putting alternate edges into E_1, and E_2

comment $G_1 = (S_1, S_2, E_1)$ and $G_2 = (S_1, S_2, E_2)$ form the euler split.

end

This coloring algorithm takes $O(n^{1/2}m \log n + n)$ time and $O(n+m)$ space, where n is the number of vertices, and m is the number of edges in the graph. The time and space analysis is given in the reference. An improved algorithm, in which the matching M in Step 2 of color1 is found more efficiently, is described by Cole et al [4]. The time required is $O(m \log n)$.

4.3 Bongiovanni's Algorithm[2]

This algorithm solves the problem of limited edge-coloring for a bipartite graph. Bongiovanni et al [2] describe the algorithm in a matrix notation, and do not use graph theory concepts and notation. We present the algorithm here in a graph-theoretic framework. From theorem 4.3, the minimum number of colors required to color a graph, such that no color is used more than k times, is $s = max(d, \lceil m/k \rceil)$, where d is the degree of the graph, m is the number of edges in the graph. Let n be the number of vertices in each of the partitions.

The following pre-processing work is done. The graph is converted into a k-complete graph with respect to s, by adding edges to the graph. We can easily see that the minimum number of colors to limited color the k-complete graph is s. The added edges are later 'dummied' out at the end of the computation. We describe the method to convert any graph into a k-complete graph with respect to the corresponding s, in the next chapter,

where we present our algorithm. The algorithm, assuming the graph has already been pre-processed, is as follows:

procedure color2(G,s) **comment** G is a k-complete graph with respect to s.
begin
1. **for** i=1 to s **do**
   begin
2. Find a critical matching, M, of size k
3. Assign all the edges in M a single color
4. $G = G - M$
   end
end

It is obvious that Step 2 is not easy. We present the details and analysis of this step in the next chapter. It seems like the time complexity of th algorithm is $O(n^2s^2)$ and the space complexity is $O(ns)$, where s is the number of colors required for the coloring.

Chapter 5

Algorithms for Edge-Coloring

In this chapter, we present our divide and conquer algorithm for solving the limited edge-coloring problem. The heuristic algorithms are also described.

5.1 Divide and Conquer Algorithm

We present some theorems in the following subsection to facilitate the presentation of the algorithm after that. Through the rest of this chapter, let us assume that the degree of the graph is denoted by d, and the number of edges in the graph is denoted by m. Let $s = \max(d, \lceil m/k \rceil)$, and let G = (A,B,E). Let $n_A = |A|$, and $n_B = |B|$. We also implicitly assume that $k \leq \min(n_A, n_B)$, since the number of times a color can be used is automatically restricted by the number of vertices in the partitions A and B, whichever is smaller.

5.1.1 Preliminary Theorems

Theorem 5.1 *Any bipartite graph can be converted into a k-complete graph with respect to s, by adding appropriate edges.*

Proof: The proof is constructive. If the number of edges, m, is less than $k \times s$, then there exists a vertex $v_a \in$ A such that v_a has degree less than s, since the

number of vertices in A is greater than or equal to k. Similarly, there exists a vertex $v_b \in$ B such that v_b has degree less than s. Add the edge (v_a,v_b) to the graph, and update the degrees of the vertices, and m. This procedure is repeated until $m = k \times s$, when the graph has become a k-complete graph with respect to s.

Throughout the construction, the degree of the graph is not increased. It is always equal to s. The number of edges in the graph, at the end of the above construction, is $m = k \times s$. Hence, the resulting graph is a k-complete graph with respect to s.

Now, we present a procedure to convert a graph into a k-complete graph.

procedure KC **comment** converts a graph G=(A,B,E) into a k-complete graph
begin
comment let A = { $a_0, a_1 \ldots a_{n_A-1}$ }, and B = { $b_0, b_1 \ldots b_{n_B-1}$ }
1. i=0; j = 0
2. **while** ($m < ks$) **do**
case:
3. (degree(a_i) $< s$) and (degree(b_j) $< s$) :
begin
4. Add an edge between a_i and b_j
5. Update the degrees of a_i and b_j
6. Update m
end
7. (degree(a_i) $= s$) and (degree(b_j) $< s$) :
8. i = i+1
9. (degree(a_i) $< s$) and (degree(b_j) $= s$) :

10. j = j+1
11. (degree(a_i) = s) and (degree(b_j) = s) :
12. i = i+1; j = j+1

end

Lemma 5.1 *The time and space required to convert a given bipartite graph into a k-complete graph with respect to s are $O(n)$ and $O(n)$ respectively.*

Proof: The number of edges added to the graph is $ks - m$. Hence, the time required is $O(ks - m + n_A + n_B) = O(n)$, where n is the number of vertices in the graph. The space required is equal to the number of edges added, which is $O(ks - m) = O(n)$.

The following theorem is easily derived from Theorem 4.1, and help in proving Theorem 5.3.

Theorem 5.2 *A regular bipartite graph has a matching of size $n_A (= n_B)$.*

Proof: It should be noted that, in a regular bipartite graph, $n_A = n_B$. From Theorem 4.1, there exists a matching that covers all the maximum degree vertices. Since there exist exactly $2 \times n_A$ maximum degree vertices, and each edge in the matching can cover only two vertices, there exists a matching of size n_A.

Theorem 5.3 *A k-complete graph with respect to s has a critical matching of size k.*

Proof: We recollect that a critical matching is a matching that covers all the maximum degree vertices. This theorem is different from Theorem 4.1,

because it is applicable to only k-complete graphs, and more importantly, it requires that the size of the matching be k. The proof is constructive. The following paragraph defines an algorithmic procedure to find a critical matching of size k. The construction is presented, for ease of description, when $n_A = n_B = n$; we later show how the construction can be modified for the situation when $n_A \neq n_B$. We use the notation $a_0, a_1 \ldots a_{n-1}$ for the n vertices in A and $b_0, b_1 \ldots b_{n-1}$ for the n vertices in B.

1. Add $n-k$ vertices to both A and B. Let the new vertices added to A and B be $A_1 = a_n, a_{n+1} \ldots a_{2n-k-1}$, and $B_1 = b_n, b_{n+1} \ldots b_{2n-k-1}$. Denote the original vertices in A and B by A_0 and B_0.
2. Add edges between the vertices in the set A_0 and vertices in the set B_1 such that the degree of each vertex in $A_0 \bigcup B_1$ is equal to s. While adding the edges, degree of no vertex should become greater than s. Call these edges F_{01}.
3. Add edges between the vertices in the set B_0 and vertices in the set A_1, such that the degree of each vertex in $B_0 \bigcup A_1$ is equal to s. While adding the edges, degree of no vertex should become greater than s. Call these edges F_{10}.
4. The resulting graph is a regular bipartite graph of degree s. Find a maximum matching M on this graph. $M \bigcap E$ gives the required critical matching of size k.

Steps 2 and 3 are possible. Assume that Step 2 is not possible. Then two situations are possible:

1. All vertices in A_0 have degree s, but there exists a vertex in B_1 which has degree less than s. The total number of edges incident on A_0 is

$n \times s$. Of these $n \times s$ edges, $k \times s$ belong to E. The number of edges that belong to F_{01} is $n \times s - k \times s = (n-k) \times s$. Hence, no vertex of B_1 can have degree less than s. This is a contradiction.

2. All vertices in B_1 have degree s, but there exists a vertex in A_0 which has degree less than s. The number of edges in F_{01} is $(n-k) \times s$, and since the number of edges in E is $k \times s$, the number of edges incident on A_0 is $n \times s$. Hence, no vertex in A_0 can have degree less than s. This is a contradiction.

Following the same method, we can prove that Step 3 is also possible.

All the edges incident on vertices in A_1 are in F_{10}. Hence, $n-k$ edges of any maximum matching, of the resulting graph, should belong to F_{10}. Similarly, $n-k$ edges of any maximum matching, of the resulting graph, should belong to F_{01}. From Theorem 5.2, the matching M in Step 4 contains $2 \times n-k$ edges. Of these, exactly $2 \times (n-k)$ edges do not belong to E. Hence, $M \bigcap E$ contains $(2 \times n-k)-2 \times (n-k) = k$ edges. If a vertex was a maximum degree vertex in the original graph, then all the edges, incident on the vertex, in the constructed graph are in E. Therefore, $M \bigcap E$ covers every maximum degree vertex in the original graph. Hence $M \bigcap E$ is a critical matching of size k.

The procedure for realizing Step 2 is as follows. The procedure for realizing Step 3 will be very similar.

procedure F01 **comment** implements Step 2 in proof of Theorem 4.3.
begin

1. i=0; j = n
2. **while** $(i < n)$ and $(j < 2n-k)$ **do**

case:

3. (degree(a_i) < s) and (degree(b_j) < s) :

begin

4. Add an edge between a_i and b_j

5. Update the degrees of a_i and b_j

end

6. (degree(a_i) = s) and (degree(b_j) < s) :

7. i = i+1

8. (degree(a_i) < s) and (degree(b_j) = s) :

9. j = j+1

10. (degree(a_i) = s) and (degree(b_j) = s) :

11. i = i+1; j = j+1

end

Lemma 5.2 *The time and space required to find a critical matching of a k-complete graph with respect to s, are* $O(n^{1.5}s)$ *and* $O(ns)$ *respectively.*

Proof: Step 1 of the constructive proof for Theorem 5.3 takes $O(n-k) = O(n)$ time. In Step 2, (n-k)s edges are added to the graph, and in the procedural description of Step 2, the counters i and j are changed from 0 to n, and n to 2n-k respectively. Hence Step 2 takes $O((n-k)s + n + n - k) = O(ns)$ time. Step 3 takes $O(ns)$ time. The resulting graph, at the end of Step 3, is a regular bipartite graph with $2n - k$ vertices in each partition, and $(2n-k)s$ edges. Finding a maximum matching on this graph, using Hopcroft and Karp's algorithm [12] takes $O((2n-k)s \times (2n-k)^{.5}) = O(n^{1.5}s)$. Finding the edges common to M and E, in Step 4, will take $O(2n - k) = O(n)$ time, as it just involves checking the indices of the end vertices of M. Hence, the total time required for finding the critical matching is $O(n^{1.5}s)$.

In Step 1, $2(n-k)$ vertices are added to the graph, and in steps 2 and 3, $2(n-k)s$ edges are added to the graph. The space required for storing the matching M is $O(2n-k) = O(n)$. Hence, the total space required for finding the critical matching is $O(ns)$.

5.1.2 The Algorithm

We can now describe the algorithm. We assume that the given graph is preprocessed so that it is k-complete with respect to s.

procedure color3(G,s) **comment** Obtains the edge-coloring of
G, which is a k-complete graph with respect to s.
begin

1. **if** s is odd
   begin
2. Find a critical matching M of size k, and assign the edges in M a single color.
3. G = G - M.
   end
   comment G has even degree now.
4. **if** G is not an empty graph
   begin
5. Divide the graph into two graphs, G_1 and G_2, such that both G_1 and G_2 are k-complete with respect to $\lfloor s/2 \rfloor$.
6. color1(G_1, $\lfloor s/2 \rfloor$).
7. color1(G_2, $\lfloor s/2 \rfloor$).
   end end

The method for implementing Step 2 is given in the proof of Theorem 5.3. Step 5 is realized through a perfect euler split. The perfect euler split is easily achieved through a modification of the procedure described for achieving the euler split, in Chapter 4. The method for finding the euler partition is the same. Care is taken at the time of the traversal of the paths to form the euler split. It is always ensured that, at the end of traversal of a path, that E_1 has no lesser edges than E_2, and that E_1 has, at most, one more edge than E_2. A boolean variable 'balance' is used to indicate if both E_1 and E_2 have an equal number of edges. 'balance' is true when both E_1 and E_2 have an equal number of edges, and false when E_1 has one more edge than E_2. If 'balance' is true at the beginning of the traversal of a path, then E_1 gets the first edge of the path. If 'balance' is false at the beginning of the traversal of a path, then E_2 gets the first edge of the path. The algorithm for the perfect euler split is given below.

procedure PES(G) **comment** finds an euler split of G.
begin
comment find the euler partition first.

1. make P an empty list, and S an empty queue
2. add all vertices of odd degree to the queue S
3. add all vertices of nonzero even degree to the queue S
4. **while** S is nonempty **do**
   begin
5. let s be the vertex at the front of the queue in S
6. delete s from S
   comment vertex s may have degree 0, since edges are deleted from G in line

```
7.      if vertex s has nonzero degree
        begin
8.          let p be an empty path
9.          v = s
10.         while vertex v has nonzero degree do
            begin
11.             let (v,w) be an edge in G
12.             put(v,w) in p, and delete it from G
13.             v=w
            end
14.         put the path p in P
15.         if vertex s has nonzero degree
16.             put s in S
        end
    end
    comment P is an euler-partition of the graph
17. let E1 and E2 be empty edge sets
18. balance = true
19. for each path p in P
20.     if balance
        begin
20.         traverse p putting alternate edges into E1, and E2,
            putting the first edge into E1.
21.         if p is of odd length then balance = false
        end
        else begin
22.         traverse p putting alternate edges into E1, and E2,
```

putting the first edge into E_2.

23. if p is of odd length **then** balance = true

end

comment $G_1 = (S_1, S_2, E_1)$ and $G_2 = (S_1, S_2, E_2)$ form the perfect euler split.

end

The procedure finds the perfect euler split. It is a modification of the procedure for an euler split given in [7]. The proof of correctness is given in the reference. Hence, the procedure here finds an euler split. That the euler split found by this procedure is indeed perfect is true because, at the end of the traversal of a path, the number of edges in E_1 is at most one more, and never less than the number of edges in E_2. The time and space required for finding the perfect euler split are $O(|E| + |V|)$ **and** $O(|E| + |V|)$ respectively.

Theorem 5.4 *The algorithm color3 finds a limited coloring for a given graph, which is k-complete with respect to s, in* $O(n^{1.5} s \log s)$ *time, and* $O(ns)$ *space.*

Proof: The proof is by induction. Consider the situation when s is even. In this case, a perfect euler split of the graph is obtained. Both the graphs G_1 and G_2 of the euler split, have maximum degree $s/2$and have exactly $ks/2$ edges. Hence, they are k-complete with respect to $s/2$. By induction, the two graphs G_1 and G_2 can be colored with s/2 colors each. Hence, G can be colored with s colors. If s is odd, a critical matching, M, covering all the maximum degree vertices is found. The edges in the critical matching are removed from the graph, to obtain a graph G'. Since G' has $ks - k = k(s-1)$ edges, and has a maximum degree of $s - 1$, it is k-complete with respect to

$s-1$. The edges in M are all independent of each other, hence they can be assigned a single color. Since G' is k-complete with respect to s-1, which is even, divide and conquer can be applied to G' to obtain a coloring of s-1 colors. Thus, G can be colored with s colors, when s is odd. The base case situation, when s = 1, is trivial to prove.

Let T(n,s) denote the time required to find a limited coloring of a k-complete graph with respect to s. The time for finding a critical matching is $O(n^{1.5}s)$. Hence, we get the following obvious recurrence relation:

$$T(n,s) = O(n^{1.5}s) + 2T(n, \lfloor s/2 \rfloor)$$

Solving the recursive equation, we get $T(n,s) = O(n^{1.5}s \log s)$.

Space required for calculating the critical matching is $O(ns)$. The depth of recursion is $\log s$. Notice that, to form the perfect euler split, no extra edges are added. Additional storage is needed to store the copies of the vertices for the two graphs in the euler split. This storage is $O(n)$ for each level of recursion, thus requiring a total of $O(n \log s)$. Hence, the total storage required by the algorithm color3 is $O(ns)$.

The time and space required for limited coloring an arbitrary bipartite graph must include the time and space required for converting it to a k-complete graph with respect to s, and the time and space required by color3. Hence, the time required for limited coloring, from Lemma 5.1, and Theorem 5.4, is $O(n^{1.5}s) + O(n) = O(n^{1.5}s)$. The space required for limited coloring, from Lemma 5.1, and Theorem 5.4, is $O(ns) + O(n) = O(ns)$.

5.1.3 The Asymmetric Situation

We presented our algorithm for limited coloring assuming that the number of vertices in A is equal to the number of vertices in B. The only place where we used this assumption was in the proof of Theorem 5.3. The theorem also holds for the situation when $n_A \neq n_B$. In this case, in Step 1 of the construction, $n_B - k$ vertices are added to A, and $n_A - k$ vertices are added to B. At the end of Step 3 of the construction, a regular graph with $n_A + n_B - k$ vertices in each partition is obtained. The matching, M, in Step 4, contains $n_A + n_B - k$ edges. It can be easily that $M \bigcap E$ contains exactly k edges. The other theorems and steps of the algorithm hold even if the partitions A and B of the bipartite graph are not of the same size.

5.2 Heuristic Algorithms

In this section, we present inexact algorithms for edge coloring. The motivation for the heuristic algorithms is as follows:

- The exact algorithms are expensive and require a substantial amount of memory. The heuristic algorithms, hopefully, are not as expensive.
- The heuristic algorithms are, typically, much more simpler than the exact algorithms. Hence, they are easier to implement. This is true of the heuristics we present.
- THe heuristic algorithms typically take less time than the exact algorithm.

However, there is a price to be paid. The schedules obtained from the heuristic algorithms are not guaranteed to be optimal. In the next two subsections, we present two heuristic algorithms.

5.2.1 Highest Degree Algorithm

The idea of this heuristic is as follows: Choose edges, whose end vertices have high degrees, to be assigned a single color. This heuristic attempts to reduce the maximum degree of the graph, as well as the degree of the high degree vertices, as the maximum degree of the graph is a lower bound on the number of colors needed.

procedure HD(G)
begin
1. Arrange the vertices in descending order of their degrees
2. **while** G is not an empty graph
   begin
3. currentset = ϕ
4. Choose an edge e, such that e is not adjacent to any edge in currentset and one of its vertices has as large a degree as possible. If no such edge exists, then go step 6.
5. Add the edge e to currentset, and remove it from the graph G. Go to step 4.
6. Rearrange the vertices in order of their degrees.
7. Assign all edges in currentset a single new color. Goto step 4.

   end
end

The procedure above can be easily modified to do limited coloring. No more edges are chosen in Step 4, if the number of edges in currentset becomes k. Step 1 does not require a costly sort algorithm. As the degree of the graph can vary only from 0 to d, we can implement a bucket sort algorithm, with

a 'bucket' for each degree from 0 to d. The same reasoning applies for the Step 6. Step 4 is realized by scanning the list of vertices in descending order of their degrees, and checking the corresponding edges from the vertex, until an edge which satisfies the condition, of not being adjacent to any edge in currentset, is found.

The time required for the bucket sort in Step 1 is $O(d+n)$. To form currentset for a single color, all the edges may be scanned, in the worst case. Hence, to form currentset for a single color, time required is $O(d+n+m)$. The time required for Step 7, is $O(m)$. Hence, the total time required by the heuristic algorithm is $((m+n+d) \times numberofcolors)$.

We will include an obvious enhancement to the approximate algorithm by improving the Step 4: If there are several edges that can be selected from the highest degree vertex, chose the one whose other end vertex has the highest degree.

5.2.2 Highest Combined Degree Algorithm

The idea of the heuristic is to select edges that have a high combined degree, to be assigned a single color. This heuristic attempts to reduce the degrees of the high degree vertices.

procedure HCD(G)
begin
1. Arrange the vertices in descending order of their degrees
2. **while** G is not an empty graph
   begin
3. currentset = ϕ

4. Choose an edge e, such that e is not adjacent to any edge in currentset and has as large a combined degree as possible. If no such edge exists, then go step 6.
5. Add the edge e to currentset, and remove it from the graph G. Go to step 4.
6. Rearrange the edges in order of their combined degrees.
7. Assign all edges in currentset a single new color. Goto step 4.

end

end

The procedure above can be easily modified to do limited coloring. No more edges are chosen in Step 4, if the number of edges in currentset becomes k. Step 1 does not require a costly sort algorithm. As the combined degree of the edges in the graph can vary from 0 to $2d$, we can implement a bucket sort algorithm, with a 'bucket' for each number from 0 to $2d$. The same reasoning applies for the Step 6. Step 4 is realized by scanning the list of edges in descending order of their combined degrees until an edge which satisfies the condition, of not being adjacent to any edge in currentset, is found.

The time required for the bucket sort in Step 1 is $O(m)$. To form currentset for a single color, all the edges may be scanned, in the worst case. Hence, to form currentset for a single color, time required is $O(m)$. The time required for Step 7, is $O(m)$. Hence, the total time required by the heuristic algorithm is $(m \times numberofcolors)$.

Chapter 6

Evaluation of the Algorithms

The evaluation of the algorithms, in terms of how cost effective they are, is presented in this chapter. We evaluate the exact algorithm and the heuristic algorithms on several sample input data. The sample data that we consider in our evaluation are from three sources: random graphs, data transfer requests generated in a 3-D visualization application, and data transfer requests generated for a split-step 3-D migration program.

6.1 Sample Data

The sample data for evaluating our algorithms is from three different sources.

6.1.1 Random Graphs

The graphs are generated using a random number generator to generate the edges. The number of vertices in each partition, of the graph, is 64 and 16. The edges in the graph are generated randomly. The number of edges in the graph determines the density of the graph. We present our results for different densities in the graph, in the next section.

6.1.2 3-D Visualization

This is a highly I/O intensive application. The aim is to visualize a large volume of three dimensional data. It is assumed that the volume of data does not fit into the available memory . All the points of the volume, which is a read only data, on a cutting plane, are displayed. We restrict our attention to the cutting planes, which are horizontal, or vertical. The cutting plane can move, and the display of data should be done in real time. The 3-D visualization problem is I/O intensive only when all of the volume does not fit into memory.

The large volume of data is divided into cuboids, each side of which is 16. Thus, there will be $64^3 = 2^{18}$ such blocks in the whole volume. To improve performance, and to achieve load balancing, we stagger the blocks, so that each successive cuboid in the x dimension (and y dimension) is staggered, in the vertical direction, from the previous block by 1. When the horizontal cutting plane moves vertically, the staggering scheme described above achieves the effect of load balancing. To achieve load balancing for the moving vertical cutting planes, cross-sections, other than the rectangular one, that we consider here need to be considered. We do not consider other cross-sections here.

We map the partitions onto storage units. We use a (d_1, d_2) skewing scheme to map the partitions onto the storage units. All blocks sitting on a single cross-section, are stored on a single mass storage unit. The (d_1, d_2) scheme is described as follows: Each successive cross-section in the x dimension is stored at a storage unit $d_1 mod M$ away from the previous one, where M is the number of mass storage units. Each successive cross-section, in the y dimension, is stored at a storage unit $d_2 mod M$ away from the previous one.

By a simple analysis, we can easily show that M should be relatively prime to d_1 and d_2. The 3-D visualization application is defined in greater detail in [3]. We chose M to be a prime close to 32, which is 31.

The number of mass storage units being considered is 31. We assume that the processing work to determine the points in the cutting plane, and to initiate data transfer requests with the mass storage units is divided among 16 processors. Thus, the graphs generated by this application are bipartite graphs with 32 and 16 vertices in each partition. We generate requests for two situations:

1. A horizontal cutting plane which moves.
2. A vertical cutting plane which is parallel to the x axis and moves.

All other vertical cutting planes generate requests similar to the requests generated above.

6.1.3 Split-Step Migration

Split-Step Fourier Migration is one of the more accurate migration algorithms. The 'Split-Step' refers to the phase-shifts being performed separately - once in the frequency domain before the inverse FFT and then in the time domain.

There are 2 input data volumes(3D arrays) which represent the seismic data and the velocity volume as sets of frequency(ω) planes and velocity(v) planes respectively. ω planes are complex and so each of them occupies twice the memory of a v plane which is real. Typical volume sizes are 1024 by 1024 by 1024.

Each ω plane goes through several stages in the migration processing for every depth step. Each depth step corresponds to a v plane. First, a 2D FFT is performed, followed by a phase-shift($\phi 1$) and then by the inverse FFT(FFT^{-1}).Subsequently, there is a second phase-shift($\phi 2$) for which a velocity plane is required. This yields a partial result plane(p) which needs to be accumulated to the partial results from other ω planes corresponding to that depth step. This accumulation is the evaluation of a single point of a discrete Fourier transform in the vertical direction. The accumulation of all the p planes, which are real, is also the last stage of data processing at that depth step for the ω plane. Mutual exclusive access to the partial result buffer is necessary as the partial result - a shared variable - is updated. Finally, the ω plane repeats the above for each depth step. This process is carried out for all the ω planes to yield the data volume comprising of the p planes.

We do our evaluation assuming that that there are 64 processors in the system. Each plane in each of the 3-D volume is split or 'striped' into 8 segments and stored on seperate sets of storage units. The three volumes are stored on seperate storage units. This implies that the data is stored on a total of 24 storage control units. We assume that only the first 64 ω planes are going through their depth steps, and that the the 64 ω planes are saved onto the mass storage, after going through 64 depth steps.

6.2 Results

Tables 6.1 to 6.4 present the analysis for random graphs, when the number of buses, varies from k=4 to k=16. In each table, the number of edges, m, for the graph is varied from 100 to 1000, representing different densities of the graphs. The numbers on the left hand side of the tables correspond to

Table 6.1: Comparison of performance and execution time of the algorithms in a random graph on a 4 bus system.

	exact		*hd*		*hcd*		*hd(unsorted)*		*hcd(unsorted)*	
	colors	time	colors	time	colors	time	colors	time	colors	time
100	25	62.4	25	1.2	25	2.6	26	0.79	26	1.6
200	50	135.4	50	2.79	50	7.6	51	0.8	52	5.8
400	100	257.6	100	5.4	100	26	102	1.8	101	19.4
600	150	406.2	150	8.6	150	53.8	152	5.8	157	45.2
800	200	579.2	200	12	200	83	204	3.2	213	66
1000	250	684	250	16	250	129.8	254	3.8	263	100.4

m. The algorithms in the table are as follows:

exact The exact algorithm that we described in Chapter 5.

hd The highest degree heuristic.

hcd The highest combined degree heuristic.

hd(unsorted) The highest degree heuristic, with the difference that the vertices are not rearranged at the end of each iteration. The vertices are sorted, according to their degree, only initially.

hcd(unsorted) The highest combined degree heuristic, with the difference that the edges are not rearranged at the end of each iteration. The edges are sorted, according to their combined degree, only initially.

We can easily see that the heuristics perform extremely well. The highest degree heuristic and the highest combined degree heuristic give optimal or near optimal schedules all the time. The unsorted versions of these algorithms come up with near optimal schedules when the density of the

Table 6.2: Comparison of performance and execution time of the algorithms in a random graph on a 8 bus system.

	exact		*hd*		*hcd*		*hd(unsorted)*		*hcd(unsorted)*	
	colors	time	colors	time	colors	time	colors	time	colors	time
100	13	31.4	13	1.2	13	3	14	0.6	13	1.6
200	25	59.2	25	1.4	25	6.4	27	0.8	28	5.2
400	50	119.8	50	3.2	50	21.8	54	1.4	55	19.2
600	75	206.4	75	4.8	75	46.8	78	1.8	87	40
800	100	306.39	100	7.2	100	88.2	103	2.8	114	67.2
1000	125	307.8	125	8.8	125	106.6	133	3.6	145	109.99

Table 6.3: Comparison of performance and execution time of the algorithms in a random graph on a 12 bus system.

	exact		*hd*		*hcd*		*hd(unsorted)*		*hcd(unsorted)*	
	colors	time	colors	time	colors	time	colors	time	colors	time
100	11	21	11	0.8	11	1.6	11	0.4	11	1.4
200	18	36.8	18	1.2	18	5.8	21	0.6	19	5.2
400	34	96.6	34	2.8	34	24.2	44	1.8	40	20
600	50	116.4	50	3.6	50	47.4	70	2.4	67	39.8
800	67	161	67	5.2	67	129.8	93	6.59	88	108.4
1000	84	194	84	7.2	84	143.59	114	4.8	108	148.8

Table 6.4: Comparison of performance and execution time of the algorithms in a random graph on a 16 bus system.

	exact		*hd*		*hcd*		*hd(unsorted)*		*hcd(unsorted)*	
	colors	time	colors	time	colors	time	colors	time	colors	time
100	11	32.99	11	1	11	2.6	11	0.6	11	2.39
200	18	36.2	18	1.6	18	5.6	18	1.6	18	5.6
400	33	97.2	33	7.6	33	24.59	33	1.2	33	22.2
600	47	139.19	47	12	47	51.79	47	2.6	47	43.99
800	63	131	63	6.8	63	72.19	63	2.4	63	69.2
1000	81	196.8	81	18	81	164.8	82	3.4	81	109.99

graph is low, or when the number of buses, in the system, k , is low. When k is small, colorings generated by even a naive heuristic are good because, at most k independent edges need to be selected at any time, so that they can be assigned a single color.

The weaker heuristics, namely hd(unsorted) and hcd(unsorted), do poorly when the density of the graph is high. The schedules(or coloring) generated differ from the optimal coloring by as much as 20%. On the other hand, the better heuristics, namely the hd and hcd algorithms give near optimal solutions all the time. The reason why hd(unsorted) does not do as well as the hd, is that the vertices are not rearranged at the end of each iteration. Rearranging of vertices, according to their degree, at the end of each iteration, seems to be a very crucial step. A similar analysis applies for the comparison of hcd to hcd(unsorted).

On the other hand, hd(unsorted) and hcd(unsorted) take a lot less time than hd and hcd respectively. All the four heuristics have better time performance than the exact algorithm. The exact algorithm takes a lot more

time to complete the coloring. The heuristic hd(unsorted) takes the least amount of time. hd comes next best, in terms of time performance. The heuristic hcd takes more time than the heuristic hd, but is still better than the exact algorithm, in terms of time performance.

The time performance difference between the hd and hd(unsorted) algorithms and the difference in the quality of coloring by the two algorithms, suggests to us the following : we could have spectrum of algorithms depending on how frequently the rearrangement of vertices is done. In case of hd, it is done at the end of each iteration. In case of hd(unsorted), it is done only at the beginning. The quality of schedules generated by such an algorithm would not be as good as that of hd, and would be better than the quality of schedules generated by hd(unsorted). Similar remarks apply for hcd and hcd(unsorted). Tables 6.5 and 6.6 show the quality of schedules generated for the data transfers required for the 3-D visualization application. The numbers in the left hand side of the table are the number of buses in the system. Again, hd and hcd do very well, giving near optimal solutions. On the other hand, hd(unsorted) and hcd(unsorted) perform very badly, generating colorings that take as much as 30% more colors than the optimal coloring.

Table 6.7 shows the quality of schedules generated for the split step migration problem.

We also evaluate the 'goodness' of the algorithms for the situation when edges are added dynamically to the graph. The evaluation is done in the following manner. Exactly $\lfloor k/2 \rfloor$ Edges are added to the graph at the end of each iteration. After more than the number of edges in the original graph have been colored, the computation is stopped, and the number of colors used is determined. Table 6.8 gives the number of edges colored and

Table 6.5: Comparison of performance and execution time of the algorithms for the 3-D visualization problem, for a vertical cutting plane.

	exact		*hd*		*hcd*		*hd(unsorted)*		*hcd(unsorted)*	
	colors	time	colors	time	colors	time	colors	time	colors	time
4	45	48.4	45	1.6	45	6.8	46	0.6	46	4.8
8	23	20	23	3.2	23	11	24	0.4	27	6.6
12	15	10.6	16	0.8	15	4.6	20	0.4	20	6.79
16	12	7.39	13	1	12	7.19	14	0.4	14	5.4

Table 6.6: Comparison of performance and execution time of the algorithms for the 3-D visualization problem, for a horizontal cutting plane.

	exact		*hd*		*hcd*		*hd(unsorted)*		*hcd(unsorted)*	
	colors	time	colors	time	colors	time	colors	time	colors	time
4	256	361.19	256	20	256	185.4	260	15.2	276	189.2
8	128	238.6	128	16.2	128	199.6	136	4.19	148	122
12	86	148.6	86	12.2	86	136.2	123	6.8	120	77.8
16	64	52.6	67	5.4	64	98	76	3.4	80	75.4

Table 6.7: Comparison of performance and execution time of the algorithms for the split-step migration problem.

	exact		*hd*		*hcd*		*hd(unsorted)*		*hcd(unsorted)*	
	colors	time	colors	time	colors	time	colors	time	colors	time
2	168	725.4	168	13.8	168	88	171	3.8	171	72.4
4	84	249.6	84	6.39	84	71.6	91	2.6	91	64.6
6	64	170.6	64	5.8	64	69.4	64	3	64	64
8	64	157	64	6.8	64	69.4	64	3.8	64	64.4

the colors used to color them. The numbers on the left of the table correspond to the initial number of edges in the graph. The lesser the number of colors used the better the heuristic is for the dynamic situation. The data from the experiment seems to indicate that all the four heuristics use the same number of minimum possible colors to color the edges. Hence, all the four algorithms are good for the dynamic situation.

Table 6.8: Comparison of performance and execution time of the algorithms for dynamic random graphs, with k = 8

	hd		*hcd*		*hd(unsorted)*		*hcd(unsorted)*	
	colors	edges	colors	edges	colors	edges	colors	edges
100	13	104	13	102	13	104	13	104
200	25	200	25	200	25	200	25	200
300	38	304	38	304	38	304	38	304
400	50	400	50	400	50	400	50	400
500	63	504	63	503	63	504	63	504

Table 6.9: Comparison of performance and execution time of the algorithms for dynamic random graphs, with k = 12

	hd		*hcd*		*hd(unsorted)*		*hcd(unsorted)*	
	colors	edges	colors	edges	colors	edges	colors	edges
100	9	108	9	106	9	108	9	108
200	17	204	17	204	17	204	17	204
300	26	312	26	308	26	312	26	312
400	34	408	34	408	34	408	34	408
500	42	504	42	504	42	504	42	504

Chapter 7

Conclusions

7.1 Summary

We summarize the results and the contribution of this thesis. The problem of data transfer scheduling is cast as an edge coloring problem. Data transfer scheduling, when the number of channels, k , is limited, is cast as an edge coloring problem, wherein no color can be used more than k times. We call this problem the limited edge coloring problem. We present a new divide and conquer algorithm for limited edge coloring. The algorithm takes $O(n^{1.5}s\log s)$ and $O(ns)$ space, where n is the number of vertices in the bipartite graph, and s is the number of colors required to color the graph. The value of s is $\lceil m/k \rceil$, where m is the number of edges in the graph. We define and evaluate four new heuristic algorithms for edge coloring. All the four heuristic algorithms are based on the idea that the edges attached to the maximum degree vertices need to be chosen first when edges are to be assigned a color. The complexity of the heuristics is $O(m \times number of colors)$. Of these four inexact algorithms, two algorithms, namely hd and hcd, give very good and near optimal solutions for the sample data. For the other two heuristic algorithms, hd(unsorted) and hcd(unsorted), the schedules generated by them differ from the optimal by a substantial amount, when the graph is dense. On the average, the time taken by the algorithms are in the following order:

exact > hcd > hcd(unsorted)> hd > hd(unsorted)

If we consider the length of the schedules generated by the algorithms, the ordering would be as follows:

hd(unsorted), hcd(unsorted) > hd, hcd > exact

It seems that hd is the best algorithm to use, if we consider both time performance and the quality of the schedules. Of course, the exact algorithm should be used if it is necessary to obtain the optimal solution.

The algorithms are also analyzed for for their performance when edges are added to the graph dynamically. All the four heuristic algorithms perform very well by being able to schedule the maximum number of edges possible with the given number of colors.

7.2 Future Work

In the thesis, we assumed that there was a direct path between every pair of nodes in the system. The extension of the data transfer scheduling problem, when not all pairs of nodes are directly connected to each other, is an important problem. Further work needs to be done to extend the solution to be applicable to this situation. Extension of the solution to the situation of hierarchical buses and multistage interconnection networks would have interesting applications. New exact algorithms for edge coloring with improved time complexity need to be designed. More analysis of the dynamic situation when new data transfer requests are added to the graph, needs to be done. Adequate measures of 'goodness' for the dynamic situation need to be defined.

BIBLIOGRAPHY

[1] C. Berge. Graphs and Hypergraphs. North-Holland, Amsterdam, 1973.

[2] G. Bongiovanni et al. An optimum time slot assignment algorithms for an SS/TDMA system with variable number of transponders, *IEEE Trans. Comm.*, **COM-29** (May 1981), 721-726.

[3] Browne, J. C. et al. Parallel structuring of the 3-D visualization problem, report available with James Browne, Dept. of Comp. Sci., Univ. of Texas at Austin, Austin, Texas 78712.

[4] R. Cole and J. Hopcroft. On edge coloring bipartite graphs, *SIAM Jl. Computing*, **11** (1982), 540-546.

[5] A. L. Dulmage and N. S. Mendelsohn Some graphical properties of matrices with non-negative entries, *Aequationes Mathematicae*, **2** (1969), 150-162.

[6] S. Fiorini and R. J. Wilson. Edge-colourings of Graphs. Pitman, London, 1977.

[7] Using euler partitions to edge color bipartite graphs, *Intl. Jl. Comp. & Info. Sci.*, **5** (1976), No. 4, 345-355.

[8] H. N. Gabow and O. Kariv. Algorithms for edge coloring bipartite graphs and multigraphs, *SIAM Jl. Computing*, **11** (1982), 117-129.

[9] Z. Galil. Efficient algorithms for finding maximum matching in graphs, *Comp. Surveys*, **18** (1986), No. 1, 23-38.

[10] M. K. Goldberg. Edge coloring of multigraphs: recoloring technique, *Jl. Graph Th.*, **8**(1984), 123-127.

[11] D. S. Hochbaum, T. Nishizeki and D. B. Shmoys. A better than 'best possible' algorithm to edge color multigraphs, *Jl. Algorithms*, **7** (1986), 79-104.

[12] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximal matchings in bipartite graphs, *SIAM Jl. Computing*, **2** (1975), 225-231.

[13] H. J. Karloff and D. B. Shmoys. Efficient parallel algorithms for edge coloring problems, *Jl. Algorithms*, **8** (1987), 39-52.

[14] E. L. Lawler. Combinatorial Optimization: Networks and Matroids. Holt, Reinhart and Winston, New York, 1976.

[15] G. Lev, N. Pippenger and L. G. Valiant. A fast parallel algorithm for routing in permutation networks, *IEEE Trans. Comp.*, **C-30** (1981), 93-110.

[16] D. Leven and Z. Galil. NP-completeness of finding the chromatic index of regular graphs, *Jl. Algorithms*, **4**(1983), 35-44.

[17] C. J. H. McDiarmid. The solution of a timetabling problem, *Jl. of the Institute of Mathematics and their applications*, **9** (1972), 23-34.

[18] R. M. Metcalf and D. R. Boggs. Ethernet: Distributed packet switching for local computer networks, *Comm. ACM*, **19** (July 1976), 395-404.

[19] G. F. Pfister et al. The IBM research parallel processor prototype(RP3): introduction and architecture, *Proc. Intl. Conf. Parallel Processing*, (Aug. 1985), 764-771.