Parallel Processing Letters, Vol. 12, No. 1 (2002) 113–125
© World Scientific Publishing Company

A GRAPH THEORETICAL MODEL FOR SCHEDULING SIMULTANEOUS I/O OPERATIONS ON PARALLEL AND DISTRIBUTED ENVIRONMENTS

JOSE AGUILAR
CEMISID, Departamento de Computación
Universidad de los Andes, Facultad de Ingeniería Mérida, 5101, Venezuela
aguilar@ing.ula.ve

Received May 2001
Revised January 2002
Accepted by S. Akl

Abstract
The motivation for the research presented here is to develop an approach for scheduling I/O operations in distributed/parallel computer systems. First, a general model for specifying the parallel I/O scheduling problem is developed. The model defines the I/O bandwidth for different parallel/distributed architectures. Then, the model is used to establish an algorithm for scheduling I/O operations on these architectures.

1. Introduction

The motivation for the research presented herein is to develop effective and generally applicable methods scheduling I/O operations [1, 2, 3]. In this paper we present a graph-theoretic model for formally specifying scheduling problems. We present a model for scheduling of batched parallel I/O requests to eliminate contention for I/O ports while maintaining an efficient use of bandwidth. We apply the model to several parallel/distributed environments [4, 5, 6, 7, 8, 9]. We will explore this tradeoff by considering one criterion for evaluating it: the length of the schedule produced. The goal is to process all requests as fast as possible without violating the "one communication at a time" constraint. To reach this goal, our approach is based on the next idea: data transfers are prescheduled to obtain schedules that are conflict free and make good use of the available bandwidth.

Our approach consists of two phases: a scheduling phase where requests are assigned to time slots, and a data transfer phase where the data transfers are executed according to the schedule. We make the assumption that a request message is much shorter than the actual data transfer is, so that the cost of sending some pre-scheduling 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. The algorithm for the scheduling phase is essentially a K-coloring of a bipartite graph, where the vertices represent processors and disks, the edges represent I/O transfers, and K is the maximum bandwidth on the system. Our model allows scheduling data transfer under various architectural and logical constraints in the context of a general framework. The rest of the paper proceeds as follows. In section 2 we present our model and algorithm in detail. In section 3 we discuss our results. Finally, in section 4 we present the conclusions.

2. Our Model

In this section we describe the scheduling problem in which we are primarily interested: the scheduling of batched parallel I/O operations in a parallel/distributed computer system. We shall assume the existence of primitive objects called resources; intuitively these correspond to disks (but can be extended to machines, communication links, etc.). 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. We also assume the existence of primitive objects called units of computation; intuitively these correspond to tasks, programs (but can be extended to industrial processes, etc.). We assume a discrete time to be a primitive notion, represented in the model as a set of natural numbers (colors in the bipartite graph).

We assume that the task allocation problem for different parallel applications has been made before executing the scheduling algorithm. The objective function is to obtain a minimum-length schedule on a given parallel computer architecture. We consider a centralized batch-oriented scheduling. We make the following assumptions:

1. The transfers require units of fixed-size slots and preemption is permitted at slot boundaries.
2. Each transfer requires a specific pair of resources, one processor and one I/O device.
3. Each processor can communicate via a link with each I/O device.
4. There exists no partial order in which the transfers are to occur, but if there are a precedent relation between two tasks with data transfer, the precedent task must be executed first.
5. Only a given number of I/O operations may take place at any given time. This number is limited by K, the maximum quantity of data transfers between processors and disks that may take place at any given time (K is called the data transfer bandwidth).
6. Communication is synchronous, that is, all clients (servers) communicate at regular fixed intervals.
7. The overhead incurred in making these choices is sufficiently small.
8. Each processor and each disk may perform at most one transfer at any given time.

2.1 General Model

The formal specification of our I/O model consists of a bipartite graph where the edges represent data transfers from vertices of type processors to those of type disks or vice versa, representing the I/O operations to be scheduled. Each edge (e_{ij}) has a weight that specifies the quantity of data to transfer (W_{ij}). A fixed maximum quantity of data transfers between memory and disks may take place at any given time. This quantity of data transfers must be equal to or less than K. According to this approach, the I/O connections between the processors and the I/O devices are viewed as a single channel of higher bandwidth (K is the capacity of this bandwidth). According to the bipartite graph model, the scheduling data transfers can be viewed as an edge-coloring problem. Henceforth, we will use the term color and timeslot interchangeably. We first introduce some definition:

<u>Definition 1.</u> *An edge coloring of a graph $G=(V, E)$ is a function $c: E\to N$ which associates a color with each edge such that no two edges of the same color have a common vertex.*

Consider a collection of vertices representing processors and I/O devices, each of which can participate in at most one data transfer at any given time. Then an edge coloring for a graph G, where each edge of G represents a data transfer requiring one time unit, corresponds to a schedule for the data transfers. Note that all edges of G colored with the same color are independent in that they have no common vertex. Hence, the data transfer they represent can be performed simultaneously. An edge coloring of G represents a schedule where all edges e_{ij} with $c(e_{ij})=m$, for some *m*, represent data transfers that take place at time *m* ($e_{ij}\in Y_m$, where Y_m is the set of transfers that take place at the time *m*). The minimum number of colors (NC) required to edge-color G equals the length of the schedule. Consider an instance of the I/O system where K is the capacity of the data transfer bandwidth of the architecture. A schedule can be obtained as an edge-coloring of G, with the restriction that $\sum_{e_{ij}\in Y_m} W_{ij} <= K$, $\forall$ m=1, NC (a color *m* may be used a given number of times according to this restriction).

<u>Definition 2.</u> *A K-coloring of a graph is an edge-coloring in which each color m may be used to color a given number of edges according to the restriction $\sum_{e_{ij}\in Y_m} W_{ij} <= K$, $\forall$ m=1, NC*

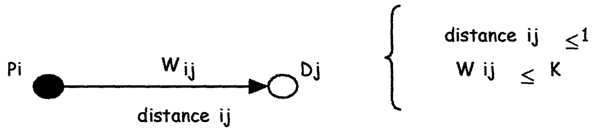
We present a parameterized algorithm to schedule data transfers based on edge coloring the transfer request graph. The parameterized algorithm can be tuned for a particular set of communication and computational cost, communication topology, etc. Our algorithm is based on an outer loop. We call one iteration of this outer loop a phase, and for each phase we use one new color *m* which generates *T* matchings at every iteration, such as T= number of elements of Ym, and $\sum_{e_{ij}\in Y_m} W_{ij} \leq K$, $\forall_{m=1,\ NC}$. We discuss the matching algorithm inside the loop for the case where m=1 (first color). In its simplest form, each client selects one of its incident edges uniformly. Then, the server resolves conflicts by selecting one of them. Clients assign the current color to the winning edges and remove those edges from the graph. If the communication required when m=1 uses more of the available bandwidth (K), deallocation of this color must be made. A fresh new color is obtained and the process is repeated in the next phase. The algorithm repeats until all edges are colored. Permutations can be made in the colors to define a new order of execution between the data transfer. This procedure will generate a matching, but not necessarily the best one. In the following, we present the details of our algorithm:

```
While $G(V,E)$ is not empty
          Get a new color ($m=m+1$)
          For all clients
                    Assign $m$ to untried edges ($c(e_{ij})=m$) according to the edge-coloring
                              problem restriction (no two edges with common vertex must
                              have same color)
          If $\sum_{e_{ij}\in Y_m} W_{ij} > K$ then
                    Discolor several edges with the $m$ color ($e_{ij}\in Ym$) until $\sum_{e_{ij}\in Y_m} W_{ij} \leq K$
          Delete colored edges and vertices of zero degree from G
```

On a given system, two types of I/O transfers may take place. Data transfers from one I/O device to a processor of the same place are called "local transfers", while those among a processor and an I/O device on separates places are called "remote transfers". A remote transfer requires simultaneous possession of an I/O device, a communication system, an I/O bus, and two processors. A local transfer requires a processor, an I/O device and an I/O bus. The bipartite graph that models the parallel I/O scheduling problem must be modify to take in account this situation (distance among the processors and the disks) and when one simple data transfer is bigger than K ($W_{ij} > K$). In general, we have four cases:

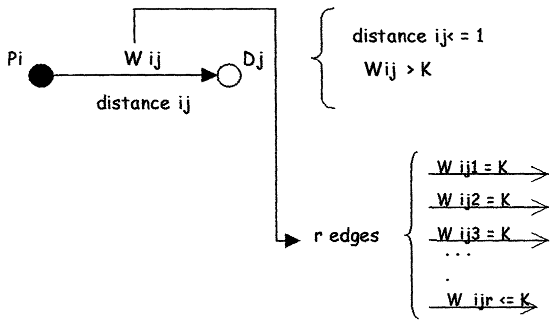
2.1.1 Case 1 ($distance_{ij} \leq 1$ and $W_{ij} \leq$, $\forall i=1$, proc and $j=1$, disks)

In this case, we do not need to modify our bipartite graph. A distance equal to 0 means that it is a local transfer. A weight equal to K means that this transfer must be executed only in a slot time (not to share the slot time with another transfer).



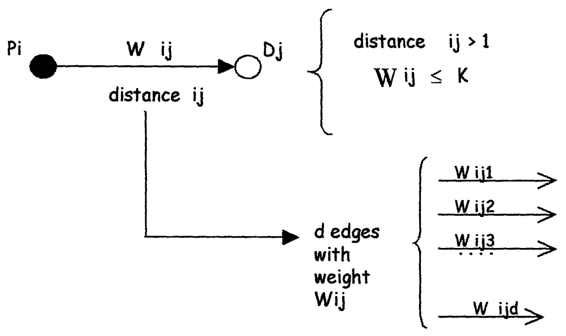
2.1.2 Case 2 ($distance_{ij} \leq 1$ and $W_{ij} > K$, $\forall$ i=1, proc and j=1, disks)

In this case, we need to decompose e_{ij} into r edges (where $r=\lceil W_{ij}/K \rceil$), r-1 edges with weight equal to K and the last one $\leq$ K. In this case, only the last one can possibly be executed in parallel with other transfers.



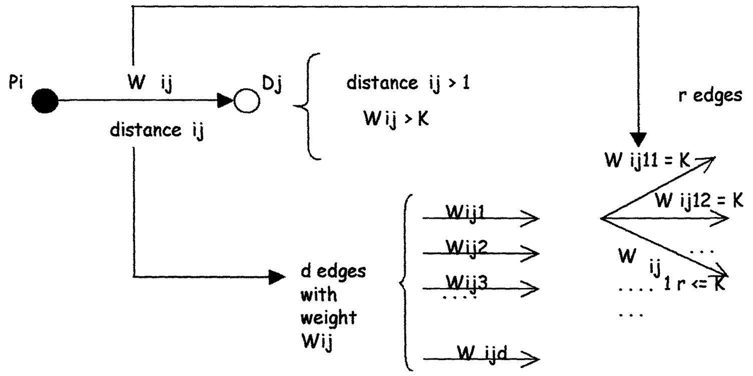
2.1.3 Case 3 ($distance_{ij} > 1$ and $W_{ij} \leq k$, $\forall$ i=1, proc and j=1,disks)

In this case, there are no direct communication links between disks and processors. That is, the data transfer between a disk and a processor must cross d nodes, $d \geq 1$. Therefore, the data transfer needs d time units for the data transfer.



2.1.4 Case 4 (distance$_{ij}$ >1 and W$_{ij}$ > K, ∀ i=1, proc and j=1, disks)

In this case, we need to decompose e$_{ij}$ into *d* edges (according to the case 3), and each one into *r* edges (according to the case 2). To execute this transfer we need d*r slot times.



2.2 Calculate the Data Transfer Bandwidth (K) for different parallel/distributed architectures

In this section, we present how to calculate *K* for different parallel/distributed platforms. In our model, we must consider system parameters to calculate *K*. These parameters are:

- *Proc* is the number of computational processors.
- n_d is the number of disks per processor.
- $n_{i/o}$ is the number of I/O nodes.
- n_{node} is the number of nodes.
- *n* is the total number of disks.
- B_d is the average bandwidth from the disks to the network interface.
- B_c is the average bandwidth from the processors to the network interface.
- B_d' is the average bandwidth from the disks to a local processor.
- $B_{i/o}$ is the average bandwidth from the I/O nodes to the network interface.
- B_{node} is the average bandwidth from nodes to the network interface between nodes.
- B_{bn} is the network interface bandwidth between nodes.
- B_n is the network interface bandwidth between processors.

We will use two node types: compute nodes (they are optimized to perform floating-point and numeric calculations, and normally have no local disk), and I/O nodes (they contain the system's secondary storage, and provide the parallel file system services). In some cases, an individual node can serve as more than one type. According to this, we can calculate *K* according to the next formulas:

2.2.1 Shared-bus system with computer nodes with local I/O devices (tightly coupled secondary storage)

The simplest architecture attaches disks to computer nodes, and the information exchange between disks of different processors must go through an interconnection network. In this architecture, *K* is calculated according to the following equation:

$$K = \min (B_n, proc{*}B_c, \sum_{i=1}^{proc} n_{di}{*}\ B_d')$$

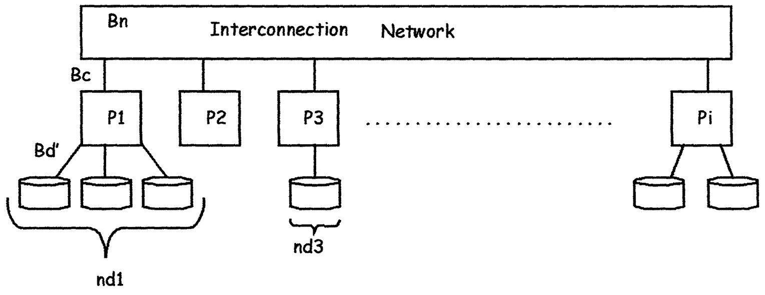


Fig. 1. Tightly coupled secondary storage architecture.

2.2.2 Shared-bus system with I/O devices connected directly to the bus

The parallel architecture is a shared-bus system, in which processors and disks are connected to a set of common busses (or a single high-speed system bus that is shared in a time-multiplexed fashion), which allow multiple I/O transfer to proceed in parallel. That is, the disks are attached to the network. All computer nodes can access each disk, so each node should be able to access any data with equal performance. In this architecture, *K* is calculated according to the next equation:

$$K = \min (proc{*}B_c\ , B_n, B_d{*}\ n_d)$$

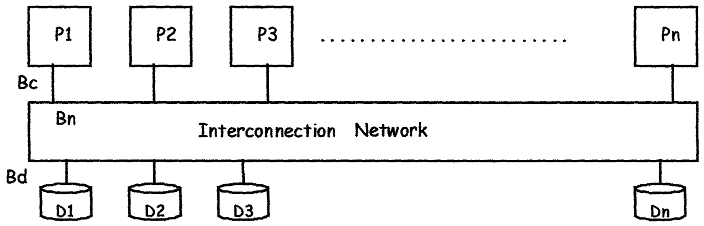


Fig. 2. Shared-bus system with I/O devices connected directly to the bus.

2.2.3 Shared-bus system with dedicated I/O nodes (loosely coupled secondary storage)

These platforms encompass a collection of I/O nodes, each one managing and providing I/O access to a set of disks. The I/O nodes connect to other nodes in the system by the same switching network that connects the compute nodes. Each I/O node acts as part of a distributed file server and operates as an intermediary between a set of disks and the computational array. The organization between the disks and its I/O node can take on any of the numerous existing host-to-disk architectures. These range from the more conventional host/disk head-of-string type of format to any RAID-level architecture. In this architecture, *K* is calculated according to the next equation:

$$K = \min\left(proc^* B_c,\ B_n,\ B_{i/o}^*\ n_{i/o},\ \sum_{i=1}^{ni/o} B_d'^* n_{di}\right)$$

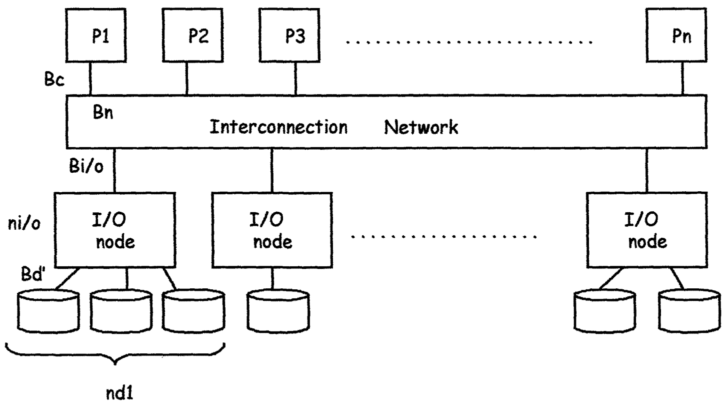


Fig. 3. Loosely coupled secondary storage.

2.2.4 Distributed-Shared Memory

This model combines distributed and shared memory. That is, we distribute the disks among sets of processors, called nodes. Each node is a shared-bus system, in which processors and disks are connected to a set of common busses. In addition, another shared-bus system connects the nodes. When a node needs to communicate with disks in another node, it uses this second shared-bus system. In this architecture, *K* is calculated according to the following equation:

$$K = \min\left(\sum_{i=1}^{nnodes} proc_i^* B_c,\ B_{bn},\ \sum_{i=1}^{nnodes} B_d^* n_{di},\ B_{node}^* n_{node}\right)$$

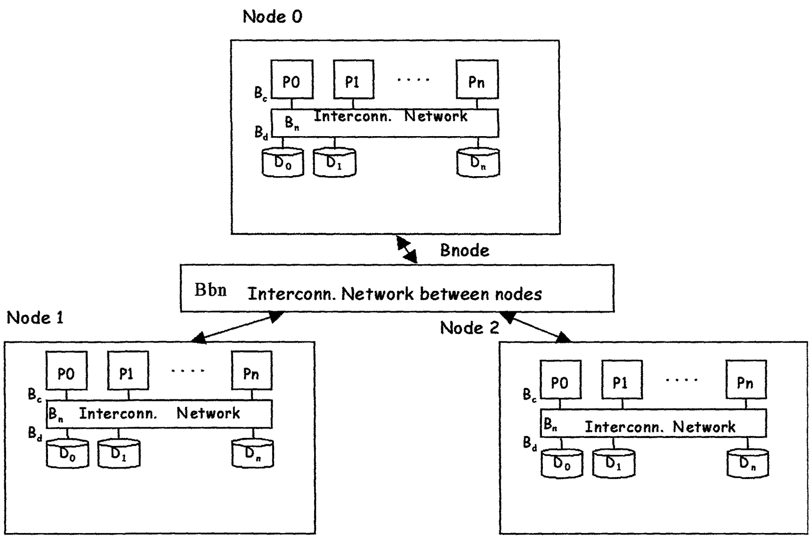


Fig. 4. Distributed-Shared Memory Architecture

For other architectures, see [4, 10] (Hypercube, etc.).

3. Experiments

We studied the impact of various parameters on the scheduling quality obtained by our approach. Scheduling quality is presented as the schedule length. We suppose an optimal data placement to guarantee the parallel I/O execution. Experiments were run on Proc*n_d random bipartite graphs, where the data transfers among the processors and disks are generated randomly. For each figure, every point is an average of 30 experiments. The experiments we have performed cover the interesting regimes of the algorithms' behaviors, and provide insight about how their behavior changes as relevant system parameters varied. The values of the parameters that we have used are:

Parameters	**Values**
Number of Transfers (tran)	20, 40, 60, 100, 150, 200
Proc	4, 8, 16, 32
B_n	5, 10, 20, 30, 50, 100
B_c	2, 5, 10,50, 100
B_d	2, 4, 20, 100
B_{bn}	20, 50, 100
B_d'	1, 2, 5
$B_{i/o}$	2, 5, 10, 50, 100
B_{nodes}	5, 10
n_d	[1, 5], [1, proc/2], [1, proc], [1, 2*proc]
$n_{i/o}$	[1, 5], [1, 10]
n_{nodes}	1, 2, 4, 8
W_{ij}	[1, K/2], [1, K], [1, 2*K], [1, 5*K]

The standard case is: tran=60, Proc=8, B_n=10, B_c=2, B_d=2, $B_{i/o}$=2, B_d'=1, B_{bn}=20, B_{nodes}=5, n_d = [1, proc/2], $n_{i/o}$ = [1, 5], n_{nodes} = 4, Wij= [1, K/2]. $n_{i/o}$ = [1, 5] means that the set of values for $n_{i/o}$ are chosen uniformly from this interval. We are going to show only a set of results, for the rest see [4].

3.1 Result analysis for different *tran* *values*

Table 1 shows the schedule length for different architectures and *tran*. The first architecture, the second one and the third one have similar behavior (*K* has a similar value). Although *K* has the biggest value for the fourth architecture, this architecture has a very long schedule length because the communication times are large.

tran	Archit. 1	Archit. 2	Archit. 3	Archit. 4
20	7	8	7	18
40	16	17	15	32
60	20	20	20	47
100	32	34	35	82
150	45	49	49	115
200	59	62	62	144

Table 1. Schedule length for different *tran* versus different architectures.

3.2 Result analysis for different W_{ij} *values*

Table 2 shows the schedule length per architecture. When the W_{ij} average is high, the schedule length is large. That is logical because for a large W_{ij} we can execute less I/O simultaneously.

W_{ij}	[1, K/2]	[1, K]	[1, 2K]	[1, 5K]
Archit. 1	20	45	62	149
Archit. 2	20	46	64	145
Archit. 3	20	47	61	150
Archit. 4	47	57	69	176

Table 2. Schedule length for different *Wij* intervals versus different architectures.

3.3 Result analysis for different B_n *values*

According to table 3, for B_n=50 we obtain the shortest schedule lengths, and for B_n=5 we obtained the largest schedule lengths. That is due to *K*, which depends of this parameter. A large B_n implies that more B_c and Bd' might also be required.

$B_c \backslash B_n$	5	10	20	50
5	25	20	17	13
10	25	21	16	10
50	24	21	17	10

Table 3. Schedule length for different B_n versus different B_c, for the first architecture.

3.4 Result analysis for different $\underline{B_{i/o}}$ values

For $B_{i/o}=50$, or $B_{i/o}=5$ and $n_{i/o}=[1, 10]$ we have the best results. For this example, we suppose $B_n=40$ and $B_c=5$. In general, for a big value of $n_{i/o}$ we obtain the best results because K is bigger due to the formula $B_{i/o}*n_{i/o}$. $B_{i/o}*n_{i/o}$ should therefore be such that their aggregate average bandwidth (rather than peak bandwidth) matches the B_c*proc and B_n values.

$B_{i/o}\backslash n_{i/o}$	[1, 5]	[1, 10]
2	22	19
5	19	12
50	12	12

Table 4. Schedule length for different *Bi/o* versus different $n_{i/o}$ for the third architecture

3.5 Result analysis for different $\underline{B_{node}}$ values

For this experiment, we suppose Bn=40. If we increase B_{node}, the schedule length is minimized because K increases. The minimal value is for $B_{nodes}>10$ and $n_{node}=2, 4$. For $B_{node}=2$ and $n_{node}=2$ or $n_{node}>4$ we obtain the largest schedule length. In general, for large n_{node} we cannot improve the results because the communication cost among the nodes is larger.

$n_{node}\backslash B_{node}$	2	5	10	20
2	45	42	35	26
4	42	29	25	25
8	43	46	47	49

Table 5. Schedule length for different *Bnode* versus different n_{node}, for the fourth architecture

3.6 Result analysis for different Proc values

According to table 6, the schedule length has a large dependence on Proc, but sometimes for different values of Proc we obtain the same schedule length because K is similar. In our experiments, only when Proc<8 K has a small value.

W_{ij} \Proc	4	8	16	32
[1, K/2]	37	20	20	20
[1, K]	63	45	43	44
[1, 2K]	81	62	63	60
[1, 5K]	171	149	149	149

Table 6. Schedule length for different *Proc* versus different Wij, for the first architecture

3.7 Comparison with other works

We have tested our approach on the same architecture where Jain et al. have tested their heuristics [2]. They tested their approach on a computer similar to the system defined on section 2.2.2 (Shared-bus system with I/O devices connected directly to the bus), and they suppose that $K = proc*B_c = B_n = B_d * n = 8$. We use the same input graphs for each value of *tran*. In their experiments they vary tran=[100, 800] and n=proc= [12, 32]. To evaluate our algorithm we use the same criterion as [2], the mean percentage increase, which is equal to $\sum_{I=1}^{100} (heu_i - opt_i)/opt_I$, where heu_i is the schedule length generated by our approach and opt_i denotes the optimal schedule length for graph *i*. In addition, the maximum increase in schedule length is equal to $max_i\{(heu_i - opt_i)/opt_i$, for $i=1, 100\}*100$. These results show that our model can be used in this architecture. These results do not improve the results obtain into [2] because the search of the optimum is not the goal of this work. In general, our algorithm has the same performance than the FCFS algorithm proposed into [2] (the schedules produced by them can be over 40% longer than optimal). On the other hand, the schedules produced by HDF and HCDF algorithms proposed into [2] were found to be always of optimal length, for this experiment. The reason of the low quality of our results is because the algorithm proposed into the section 2.1 is not an optimal heuristic to solve the edge coloring graph problem.

Proc\Tran	100	300	500	800
12	42	27	19	14
32	50	35	23	20

Table 7. Comparison of our approach with [2] (max. increase in schedule length ((%))

4. Conclusions

The problem of scheduling I/O in parallel/distributed computer systems is of considerable practical interest but is only just beginning to receive attention. As a first step in this research, we have defined a general model for the scheduling of parallel computations that serves as a consistent framework for specifying scheduling. In general, our approach is a general, versatile and robust model to solve the I/O scheduling problem, which can be used in different parallel/distributed platforms. We have presented an algorithm for coordinating data transfers associated with a set of pending I/O requests over different parallel/distributed environments. In our approach all outstanding requests are scheduled before considering new arrivals. 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. An appropriate metric for our algorithm is the number of colors required to edge color the graph, or equivalently, the length of the schedule required to complete all requests. By pre-scheduling data transfers, we eliminate contention for I/O ports while maintaining efficient use of interconnection bandwidth.

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. When these applications are run alone on a multiprocessor, no new requests should be scheduled until the current batch of request is completed. One advantage of our algorithm is that starvation is not a problem since every request will be served within the current batch. We studied the performance and the behavior of our algorithm for different architectures experimentally and compared it to the work of others. According to our results, K has a large influence in the I/O operation performance. In addition, our approach gives scheduling solutions that are not the best one, but they are sufficient to allow I/O operations in a given system. Our approach produces longer schedules than previous works because our algorithm to solve the edge coloring graph problem is not optimal. Some open problems for future work are:

1. We did not address the arrival of new I/O requests. New dynamic approaches must be developed, for example, to consider a new request as soon as the current time slot has been scheduled. One of the ways is 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.

2. Edge coloring graph is a NP-Complete Problem. New heuristic approaches must be developed to reduce the schedule length.

3. We must continue to investigate the parallel I/O scheduling problem both theoretically and experimentally, particularly the use of precedence graphs of the applications corresponding to situations of practical interest (multimedia requirements that place different demands on the I/O system, database systems distributed file systems, etc.).

Acknowledgment

This work was partially supported by CONICIT grant AP-97003817, CDCHT-ULA grant I-620-98-02-AA and CeCalCULA (High Performance Computing Center of Venezuela).

References

[1] R. Jain, K. Somalwar, J. Werth and J. Browne, Scheduling Parallel I/O operations in Multiple Bus Systems, *Journal of Parallel and Distributed Computing*, **16** (1992) 352-362.
[2] R. Jain, K. Somalwar, J. Werth and J. Browne, Heuristics for Scheduling I/O operations, *IEEE Transaction on Parallel and Distributed Systems*, **8** (1997) 310-321.
[3] B. Narahari, S. Sherde, R. Simhu and S. Subramanya, Routing and Scheduling I/O Transfer on Wormhole-routed Mesh Network, *Journal of Parallel and Distributed Computing*, **57** (1999) 1-13.
[4] J. Aguilar, Ejecución Paralela de Operaciones de Entrada/Salida. *Proc. Simposio Español de Informática Distribuida*, (2000) 409-417.
[5] R. Barke, E. Shriver, P. Gibbons, B. Hillyer and Y. Matices, Modeling and Optimizing I/O throughput of Multiple disks on a bus, *Performance Evaluation Review*, **27** (1999) 83-92.
[6] A. Choudhary, Parallel I/O Systems, Journal of Parallel and Distributed Computing, **17** (1993) 1-3.
[7] D. Feitelson, P. Corbett, S. Johnson and Y. Hsu, Parallel I/O Subsystems in Massively Parallel Supercomputers, *IEEE Parallel and Distributed Technology*, **Fall** (1995) 33-47.
[8] J. Lee, I. Tsaur and S. Hwang, Parallel Array Object I/O Support on Distributed Environments, *Journal of Parallel and Distributed Computing*, **40** (1997) 227-241,

[9] Y. Patt, The I/O Subsystem: A candidate for improvement, *IEEE Computer*, **March** (1994) 15-16.

[10] J. Aguilar, File Decomposition and Assignment Problems on Distributed Systems, *Proc. Int. Conf. on Information Systems, Analysis and Synthesis*, eds. M. Torres, B. Sanchez, J. Aguilar (Vol. 5, 1999) 587-593.