中央研究院
資訊科學研究所
Institute of Information Science, Academia Sinica • Taipei, Taiwan, ROC

TR-IIS-07-015

Optimizing Server Placement for Parallel I/O in Switch-Based Clusters

Jan-Jan Wu, Yih-Fang Lin, Da-Wei Wang, Chien-Min Wang

Oct. 30, 2007 || Technical Report No. TR-IIS-07-015

http://www.iis.sinica.edu.tw/page/library/LIB/TechReport/tr2007/tr07.html

Optimizing Server Placement for Parallel I/O in Switch-Based Clusters

Jan-Jan Wu[1] Yih-Fang Lin[1,2] Da-Wei Wang[1] Chien-Min Wang[1]

[1] Institute of Information Science
Academia Sinica
Taipei 115, Taiwan, R.O.C.
Phone: +886-2-2788 3799
Fax: +886-2-2782 4814
E-mails: {wuj,wdw, cmwang@iis.sinica.edu.tw}we

[2] Dept. Computer Science & Information Engineering
National Taiwan University
Taipei, Taiwan, R.O.C.
ice@iis.sinica.edu.tw

Abstract

In this paper, we consider how to optimize I/O server placement in order to improve parallel I/O performance in switch-based clusters. The significant advances in cluster networks in recent years have made it practical to connect tens of thousands of hosts via networks that have enormous and scalable total capacity, and in which communications between a host and any other host incur the same cost. The same cost property frees users from consideration of network contention and allows them to concentrate on load-balancing issues. We formulate the server placement problem on a cluster that has the same cost property as a weighted bipartite matching with the goal of balancing the workload on the I/O nodes. To find an optimal solution to this problem, we propose an $O(n^{\frac{3}{2}}m(\log n+\log m))$ algorithm, called *Load Balance Matching* (LBM), where n is the number of compute nodes and m is the number of I/O servers.

We also investigate server placement for general clusters in which multiple same-cost subclusters are interconnected to form a large cluster. This class of clusters typically adopt irregular topologies that allow the construction of scalable systems with an incremental expansion capability. Also, due to the limited bandwidth on network links between subclusters, network link contention is a major concern when distributing servers over the entire network. We show that finding an optimal placement strategy for general clusters with the goal of minimizing link contention is computationally intractable. To resolve this problem, we propose a hierarchical strategy that places servers in two steps. First, to minimize link contention, we decide which

subcluster each server should be assigned to. We propose a recursive tree-based heuristic algorithm, called *Load Balance Traversing* (LBT), which approximates an optimal solution to this problem. In the second step, the LBM algorithm decides the location of each server within a subcluster.

Our simulation results demonstrate that LBT achieves a significant improvement in parallel I/O performance over four other algorithms (randomized, even distribution, Shortest-Path, and Request-Volume).

Keywords: parallel I/O, I/O server placement, load balancing, switch-based cluster, irregular network, load-balancing matching algorithm, load-balancing tree-traversing algorithm.

1 Introduction

While the speed, memory size, and disk capacity of parallel computers continue to grow rapidly, the rate at which disk drives can read and write data is improving much more slowly. As a result, the performance of carefully tuned parallel programs can slow down dramatically when they read or write files. Parallel I/O technologies help increase parallelism in I/O by creating multiple data paths between the memory and the disks. In other words, the technologies increase parallelism in reading and writing a data file by striping it across multiple disks. PIOUS [21], VIP-FS [15], Galley [22], PPFS [16], and VIPIOS [5], to name but a few, are popular systems that provide parallel I/O for commercial parallel machines.

The significant advances in cluster networks in recent years have made it practical to connect tens of thousands of hosts via networks that have enormous and scalable total capacity, and in which communications between a host and any other host incur the same cost. This same-cost property is desirable because it allows computing processes to be assigned to hosts according to cluster-management or load-balancing considerations, without having to map the communication patterns of the computation to the network topology. For example, Clos networks provide multiple routes between hosts, and all the shortest routes are deadlock-free. Furthermore, since a host interface can send successive packets to another host over multiple routes, the traffic is dispersed in such a way that statistically avoids hot spots. In other words, communication in the network is contention-free. The concept of Clos networks has been used to build Myrinet-based clusters [1].

The same-cost concept has also been exploited in non-blocking networks [3, 27]. In fact, many clusters listed in the Top 500 Clusters (http://www.top500.org/) are based on the concept of non-blocking networks. For example, the processors of the Earth Simulator developed by NEC are connected by a 640 by 640 crossbar switch. The processors of the ASCI Q cluster located at Los Alamos National Laboratory, as well as those of Virginia Tech's X Telescale cluster, are

connected by high-speed switches interconnected in fat-tree configurations, while the Tungsten cluster developed by NCSA uses Myrinet switches that are interconnected in a Clos network configuration.

In the first part of the paper, we consider server placement in cluster networks that have the same-cost property, which enables concurrent transfer of messages from the sending node to any of the receiving nodes. As a result, multiple I/O servers can transfer data to their clients concurrently, and vice versa. The overall time for a parallel I/O operation is determined by the server that finishes its remote data transfer last. Therefore, the load balance of the I/O servers is the key optimization criterion. Based on this idea, we formulate the server placement problem as a weighed bipartite matching, with the goal of minimizing the maximum weight of the edges in the matching set, which is equivalent to balancing the workload on the I/O servers. We propose a fast algorithm, called *Load Balance Matching* (**LBM**), which finds an optimal solution to this problem. Instead of employing standard weighted bipartite matching, **LBM** intelligently combines binary search and cardinality bipartite matching to substantially reduce the execution time required to find an optimal solution. **LBM** takes $O(n^{\frac{3}{2}}m(\log n + \log m))$ time, where n is the number of compute nodes and m is the number of I/O servers.

Despite the attractiveness of the same-cost property, same-cost clusters require an excessive number of switches to provide multiple paths; therefore, building large-scale same-cost clusters is an expensive operation. A more cost-effective alternative is to connect multiple smaller same-cost clusters with commodity networks. We call this class of clusters *general networked clusters*. Such clusters typically adopt irregular topologies to allow the construction of scalable systems with an incremental expansion capability. However, because of limited bandwidth on network links, link contention between subclusters is one of the dominant factors that affect the performance of parallel I/O in general networked clusters.

In the second part of the paper, we investigate server placement in general networked clusters, and show that finding an optimal placement strategy that minimizes link contention is computationally intractable. To resolve the problem, we adopt a load balancing strategy that places servers in two steps. First, to minimize link contention between subclusters, the algorithm decides which subcluster each server should be assigned to. We propose a tree-based heuristic algorithm, called *Load Balance Traversing* (**LBT**), which assigns I/O servers to the subclusters. `LBT` balances the workload on the links by recursively traversing the routing tree of the network. In the second step, the location of each server within its assigned subcluster is determined by the `LBM` (*Load Balance Matching*) algorithm.

The results of extensive simulations conducted to evaluate the above algorithms demonstrate that our load balancing strategy outperforms four other algorithms (Random selection, Even distribution, Shortest distance, and Request volume) in most cases, with an improvement ratio

of 10% to 60% in terms of parallel I/O throughput.

The remainder of the paper is organized as follows: In Section 2, we define the problem of server placement in same-cost clusters and present our *Load Balance Matching* algorithm. In Section 3, we describe our model of general networked clusters with irregular topologies, define the problem of server placement in such clusters, and present the *Load Balance Traversing* algorithm. Section 4 details our experiment results. Section 5 contains an overview of related works. Then, in Section 6 we present some concluding remarks.

2 Server Placement in Same-Cost Clusters

In this section, we consider server placement in clusters that have the same-cost property. The property enables concurrent transfer of messages from a sending node to any receiving nodes. In other words, multiple I/O servers can transfer data to their clients concurrently, and vice versa. The overall execution time of a parallel I/O operation is determined by the server that finishes its remote data transfer last. Therefore, the load balance of the I/O servers is the key optimization criterion.

2.1 System Model

A cluster typically consists of a number of general-purpose nodes, each of which has local disks and I/O support. The pool of local disks provides a massive space for storing and managing large data sets. Furthermore, to make full use of the computing power provided by the compute nodes, we adopt the concept of *part-time* I/O [8], which means there are no dedicated I/O nodes. Instead, a subset of compute nodes become I/O nodes during an input/output operation and return to computation afterwards. The notion of part-time I/O allows dynamic configuration of I/O systems according to the I/O traffic patterns of individual application programs.

We assume that an application program contains only one dominant phase (i.e., one phase that dominates the execution time of the entire program), and that the optimal data distribution strategy for the dominant phase is known (either by compiler or runtime analysis). For applications with several phases, each of which may prefer a different data distribution strategy, fast data shuffling between phases may need to be considered; however, this issue is beyond the scope of this paper.

Moreover, since the assignment strategy decides the optimal location for data storage, data must be migrated to that location if it is stored elsewhere in the system. Based on the application programs we have studied, we assume that the dominant phase must be executed iteratively many times; thus, the cost of migrating data to the optimal location can be amortized.

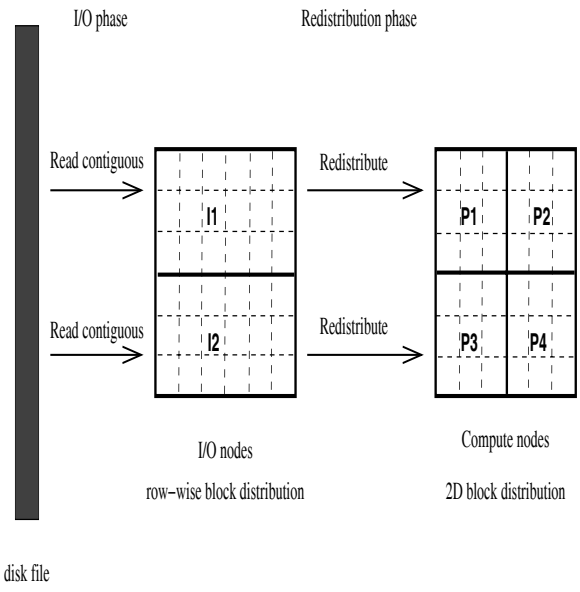


Figure 1: An example of data transfer between I/O nodes and compute nodes.

First, we use an example to illustrate why load balancing on I/O servers is the key optimization criterion for server placement within a same-cost cluster. For a read request, the I/O nodes read the data from disks and send it over the network to the compute nodes. Meanwhile, for a write request, the compute nodes send the data over the network to the I/O nodes, which then write it on the disks. We use the term "*remote data transfer*" to refer to moving data between compute nodes and I/O nodes.

Figure 1 shows a 2D array of 6×6 elements. The array is distributed in a 2D-block fashion across the memories of four compute nodes configured as a 2×2 grid. The two I/O nodes are configured as a linear array, and the data array is stored across the I/O nodes in a row-wise block distribution. I/O node I_1 reads 18 data elements from its disk, and then sends 9 of them to compute nodes P_1 and P_2 respectively. Similarly I_2 reads 18 data elements and sends 9 of them to P_3 and P_4 respectively. These remote data transfers are represented by an m by n matrix called an I/O matrix [25], in which the rows correspond to I/O nodes and the columns to compute nodes. Each entry $w_{i,j}$ represents the number of data elements to be transferred between I/O node I_i and compute node P_j. For example, the I/O matrix for remote data transfers depicted in Figure 1 is

$$W = \begin{pmatrix} 9 & 9 & 0 & 0 \\ 0 & 0 & 9 & 9 \end{pmatrix}.$$

The pattern of remote data transfers in Figure 1 is called "regular" because uniform data

distributions are used for both the disks and the compute nodes. However, many applications are characterized by non-uniform data references or an uneven computational load of data elements. To balance the load on compute nodes and reduce inter-processor communication, an "irregular" data distribution on the compute nodes may be more suitable for this type of application. Figure 2 shows the same array with a non-uniform distribution over the four compute nodes, which results in an irregular pattern of data transfers, as shown in the following I/O matrix:

$$\begin{pmatrix} 7 & 11 & 0 & 0 \\ 3 & 7 & 4 & 4 \end{pmatrix}.$$

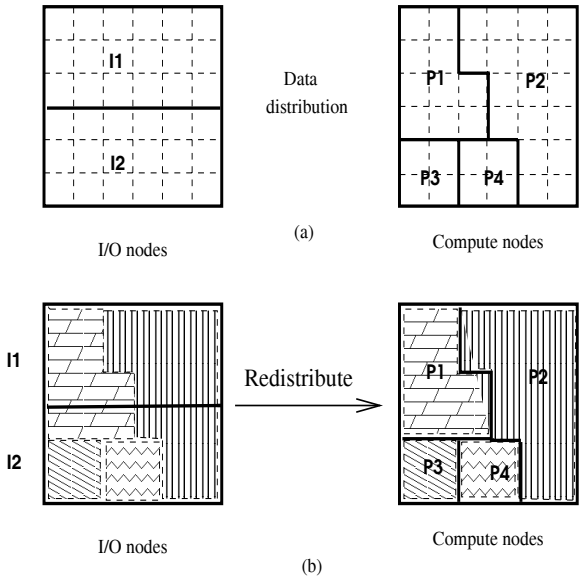


Figure 2: Irregular Data Transfer Between I/O Nodes and Compute Nodes. (a) Data distribution: regular row-wise blocks on disks, irregular blocks on compute nodes. (b) Data transfer patterns: the shaded areas with the same pattern represent the source, the destination, and the data elements transferred between them.

The problem involves choosing one entry in each row under the constraint that no two selected entries lie in the same column or the same row. If the second entry is chosen for the first row (that is, compute node P_2 is selected to be the part-time I/O node I_1, denoted by $I_1 \leftarrow P_2$), then the remote data transferred between P_2 and I_1 will be accessed via a local disk in P_2. Therefore, the amount of remote data transferred on I_1 is reduced to 7 elements.

Minimizing the total number of data transfers results in the assignment $(I_1 \leftarrow P_2, I_2 \leftarrow P_3)$. Clearly, this assignment generates an imbalanced data transfer workload on the I/O nodes (7 elements on I_1 and 14 elements on I_2) and will therefore perform inefficiently in switch-based

systems. In contrast, if we choose the assignment ($I_1 \leftarrow P_1, I_2 \leftarrow P_2$), the workload will be balanced (11 elements on each I/O node).

2.2 Problem Definition

Let W be an I/O-matrix, as shown below. The m rows correspond to I/O nodes and the n columns correspond to compute nodes. A non-zero entry $w_{i,j}$ represents the data transfer time between I/O node I_i and compute node P_j. For simplicity, we use the number of data elements in a data transfer to represent the cost of the transfer. This assumption is reasonable for parallel I/O because (1) the same-cost property of cluster networks avoids the possibility of delays due to network contention; and (2) since messages in remote data transfers in parallel I/O are usually large, the start-up overhead is negligible; therefore, the number of data elements is the dominant factor in the data transfer time.

$$W = \begin{pmatrix} w_{1,1} & w_{1,2} & \dots & \dots & w_{1,n} \\ w_{2,1} & w_{2,2} & \dots & \dots & w_{2,n} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ w_{m,1} & w_{m,2} & \dots & \dots & w_{m,n} \end{pmatrix}$$

If the kth entry in the ith row is chosen (that is, compute node P_k is selected as the part-time I/O node I_i), the total time required for remote data transfers by I_i, denoted by T_i, is the sum of all the entries in the ith row, excluding the entry $w_{i,k}$:

$$T_i = (\sum_{j=1}^{n} w_{i,j}) - w_{i,k}.$$

Since data transfers on different I/O nodes can be performed in parallel in switch-based systems, the overall time for the whole system is determined by the time taken by the last I/O node to complete the transfer operation. Hence, the data transfer time for the whole system, denoted by T, is calculated as follows:

$$T = \max_{i=1,m} T_i$$

The goal is to choose m matrix entries $w_{1,j_1}, \dots, w_{m,j_m}$ such that the overall transfer time T is minimized, under the constraint that no two selected entries lie in the same column or the same row. For ease of presentation, we construct a *workload matrix* from the I/O matrix. Let Y be the workload matrix as shown below. An entry $y_{i,j} = (\sum_{k=1}^{n} w_{i,k}) - w_{i,j}$ in Y represents the

total data transfer time of I/O node I_i when compute node P_j is chosen for I_i.

$$Y = \begin{pmatrix} (\sum_{j=1}^{n} w_{1,j}) - w_{1,1} & \dots & (\sum_{j=1}^{n} w_{1,j}) - w_{1,n} \\ (\sum_{j=1}^{n} w_{2,j}) - w_{2,1} & \dots & (\sum_{j=1}^{n} w_{2,j}) - w_{2,n} \\ \dots & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & \dots \\ (\sum_{j=1}^{n} w_{m,j}) - w_{m,1} & \dots & (\sum_{j=1}^{n} w_{m,j}) - w_{m,n} \end{pmatrix}$$

If the entry $y_{i,k}$ were chosen as a part-time I/O, the total data transfer time of I/O node I_i and the overall transfer time of the whole system would be

$$T_i = y_{i,k} \; 1 \leq k \leq n, \qquad T = \max_{i=1,m} T_i.$$

In this case, the goal is to choose m matrix entries $y_{1,j_1}, \dots, y_{m,j_m}$ such that the overall transfer time T is minimized, under the constraint that no two selected entries lie in the same column or the same row. This problem is equivalent to matching in bipartite graphs, as shown in Figure 3. The two sets of vertices are equal to the set of I/O nodes and the set of compute nodes respectively. The graph is a bipartite graph with weighted edges, where the weight of an edge (i, j) is equal to $y_{i,j}$. The problem now becomes one of finding a maximum cardinality matching in which the largest edge weight in the matching set is minimized.

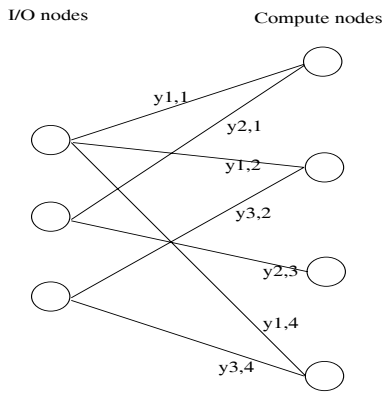


Figure 3: I/O node assignment formulated as a bipartite matching.

2.3 Load Balance Matching Algorithm

Let $G = (U, V, E)$ be a bipartite graph, where U and V are two sets of vertices and E is a set of weighted edges. $|U| = m$, $|V| = n$, and $m \leq n$. Given a set of edges, S, we define the cost of S, denoted by $C(S)$, as the weight of the largest edge weight in S. Since we are only looking for a

maximum cardinality matching[1], we use the term "matching" to denote "maximum cardinality matching" hereafter.

To find a minimum cost matching, we first define a simple decision version of the problem. That is, given a bipartite graph and a bound B, is there a matching whose cost is less than B? The problem can be solved by deleting all edges with weights greater than B, finding the matching of the resulting graph, and checking if the cardinality of the matching is equal to the cardinality of the matching in the original graph. The problem can be solved in $O(n^{\frac{3}{2}}m)$ time [23].

To find the minimum cost matching, we perform a binary search on all the possible bounds. There are at most $m \times n$ edges, so there are at most $m \times n$ different values to be searched. At most $O(\log m + \log n)$ searches are needed to find the minimum cost matching; therefore, the algorithm takes $O((n^{\frac{3}{2}}m)(\log m + \log n))$ time to find the optimal I/O assignment strategy.

3 Server Placement in General Networked Clusters

Next, we consider server placement in general networked clusters, which usually adopt irregular network topologies. First, we give a brief overview of the "up-down" routing strategy commonly used in irregular networks. We then define the server placement problem in general networked clusters, and prove that it is NP-hard. Finally, we present our *Load Balance Traversing* algorithm, which approximates an optimal solution to the problem.

3.1 Up-Down Routing

The up-down routing mechanism [12] first performs a breadth-first search to build a spanning tree T for a graph $G = (V, E)$, which represents a general networked cluster. Since T is a spanning tree of G, E is partitioned into two subsets, – T and $E - T$. The edges in T are referred to as *tree edges* and those in $E - T$ as *cross edges* (which provide adaptivity during routing). Since the tree is built by a BFS, the cross edges can only connect subclusters whose levels in T differ by at most 1. A tree edge going up the tree, or a cross edge going from a switch with a higher ID to a switch with a lower one, are referred to as *up links*. The communication channels going in the opposite direction are called *down links*. In up-down routing, a message must traverse all the up links before traversing any down links.

Figure 5(a) illustrates a general networked cluster comprised of seven subclusters. The connectivity of the subclusters in the network can be represented by a graph $G = (V, E)$,

[1]A maximum matching on a bipartite graph $G = (U, V, E)$, as defined above, is a matching whose matched set contains all the nodes in U. A maximum cardinality matching on a weighted bipartite graph G is a maximum matching that only considers the cardinality of the matched set. Readers should refer to [23] for further details.

Algorithm **Load_Balance_Match**

Input : $G = (U, V, E, Y), \quad |U| = m, |V| = n$

Output : a list of m edges

{

Step1 : L = sort Y in non-decreasing order

Step2 : $Bsearch\_Match(G, L, 1, |L|)$

}

Algorithm Bsearch_Match(G, L, l, u)

Input : G, a bipartite graph

L, sorted list

l, u lower bound and upper bound indices of L

Output : a list of m edges

{

if $l == u$, then return(max_cardinality_bipartite_match(G))

else {

$k = (l + u)/2$

$B = L(k)$ /* find the middle element of L */

/* Is there a matching with cost less than B? */

$G' = G - \{e_{i,j} | w_{i,j} > B\}$

if max_cardinality_bipartite_match(G') $\neq$ NULL, then

/* positive answer, proceed to lower half */

Bsearch_Match($G', \{L(l), \ldots, L(k-1)\}, l, k-1$)

else

/* negative answer, proceed to upper half */

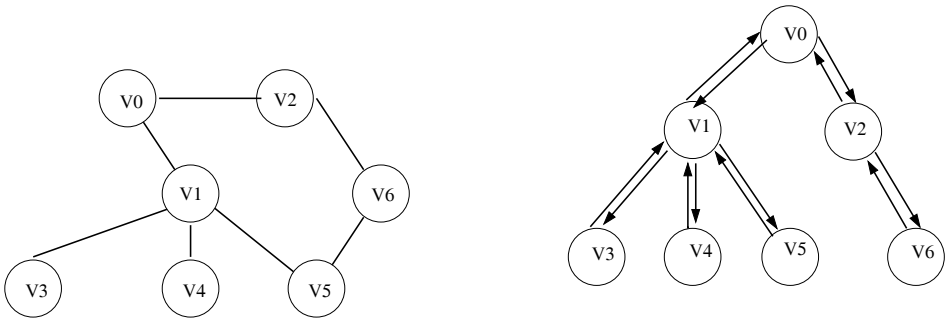
Bsearch_Match($G, \{L(k), \ldots, L(u)\}, k, u$)

}

}

Figure 4: The Load Balance Matching Algorithm for Assigning I/O Nodes Based on a Binary Search and Maximum Cardinality Bipartite Matching

where the set of nodes V represents same-cost clusters, and the set of edges E represents the bidirectional connection channels between the subclusters. We assume that all the connection channels have equal bandwidth. Figure 5(b) shows an up-down routing tree generated from a graph G.



(a) A general networked cluster

(b) An up–down routing tree for the cluster in (a)

Figure 5: A routing tree for a general networked cluster comprised of seven same-cost subclusters.

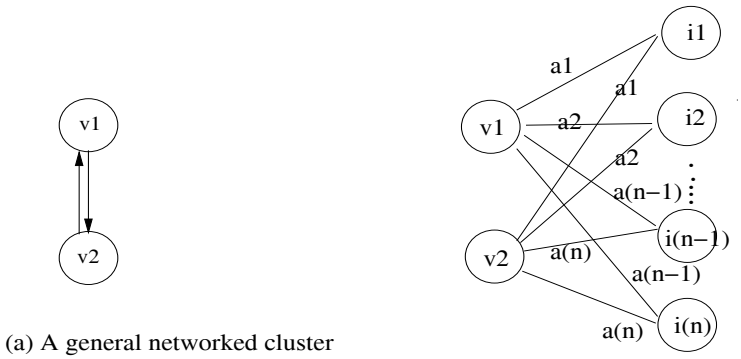
3.2 Problem Definition

In network-based systems, the link that has the maximum number of messages travelling through it simultaneously becomes the *dominating link* in the network. The lower the workload on the dominating link, the better the parallel I/O performance will be. In this paper, we assume that both the read and write operations co-exist in the application program and the ratios of the operations are known (e.g., by profiling). Under this model, the I/O server placement problem can be stated as follows. Given a graph $G = (V, E)$ that represents an irregular network with $|V|$ subclusters and $|E|$ links, a network routing function (up-down routing in this paper), the number of processing nodes in each subcluster, and the number of I/O nodes to be assigned, we need to determine the optimal locations for the I/O nodes such that the workload on the dominating link will be minimized. We show that the problem is NP-hard when the number of subclusters in the network is not less than two by reducing the partition problem to a decision version of the same problem.

Partition Problem. The partition problem, which is known to be NP-complete, is defined as follows. Given a set A of n numbers $\{a_1, a_2, \ldots, a_n\}$, partition A into two disjoint sets, A_1 and A_2, such that the sum of the values in set A_1 is equal to the sum of the values in set A_2.

Theorem 1 *The I/O server placement for general networked clusters is NP-hard when* $n >= 2$,

where n is the number of subclusters in the network.



(a) A general networked cluster comprised of 2 subclusters

(b) Data requests between I/O nodes and subclusters

Figure 6: A 2-subcluster general networked cluster and its data request pattern

Proof. Given a partition problem with n numbers $A = \{a_1, a_2, \ldots, a_n\}$, we construct a general networked cluster problem as follows. There are two subclusters $\{v_1, v_2\}$ and n I/O nodes, $i_1, i_2, \ldots, i_n$, and each subcluster requests a data packet of size a_j from node i_j. The routing tree for the cluster has two edges, an *upward* edge and an *downward* edge, as shown in Figure 6(a). Each I/O node can be assigned to either v_1 or v_2. For example, assigning i_1 to v_1 places the workload of a_1 on the downward edge from v_1 to v_2, because cluster v_2 now has to request data a_1 from v_1. Similarly, if i_3 is assigned to v_2, then the workload on the upward edge is increased by a_3, since v_1 will have to request data a_3 from v_2. Clearly, the assignment of the I/O nodes partitions the data traffic into two disjoint sets: A_1, to v_1 and A_2, to v_2. The sum of the data packet sizes in A_1 corresponds to the total downward edge load, while the sum of those in A_2 corresponds to the total load on the upward edge. The contention on the links between v_1 and v_2 is less than $\frac{M}{2}$ if and only if A can be partitioned evenly, where M is the sum of the set A. Since this decision problem is NP-complete, we can conclude that the optimization problem of I/O server placement is NP-hard. ■

3.3 Load Balance Traversing

We propose a hierarchical load balancing strategy that places servers in two steps. In the first step, the algorithm decides which subcluster each server should be assigned to, with the goal of minimizing network contention on the links between subclusters. For this purpose, we propose a tree-based heuristic algorithm, called *Load Balance Traversing* (**LBT**), which assigns I/O servers to the subclusters. `LBT` balances the workload on the links based on a recursive traversal

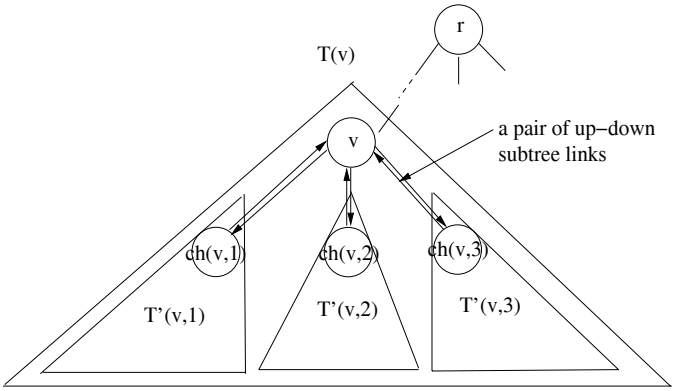


Figure 7: Graphical illustration of the tree-related variables

of the network's routing tree. In the second step, the location of each server within its assigned subcluster is determined by the `LBM` (*Load Balance Matching*) algorithm.

The *Load Balance Traversing* algorithm is motivated by our observation that, on a complete d-ary routing tree T, parallel I/O performance is maximized when all the subtrees of T have approximately the same number of I/O nodes. This is reasonable because such a distribution balances the workload on the links that connect the root and the subtrees. These links are called the *subtree links* of tree T. The balanced workload on the subtree links prevents hot spots developing in remote data transfers. This concept can be extended to more general up-down routing trees.

3.3.1 Definitions

Before presenting the *Load Balance Traversing* algorithm, we define some relevant variables.

Tree related variables

- r: the root of the whole tree.
- $\mathbf{T}(v)$: the tree whose root is cluster v.
- $\mathbf{T}'(v, i)$: the ith sub-tree of $\mathbf{T}(v)$.
- $\mathbf{Ch}(v)$: the set of children of $\mathbf{T}(v)$.
- $\mathbf{N}(v)$: the set of clusters in $\mathbf{T}(v)$.
- $ch(v, i)$: the ith child of $\mathbf{T}(v)$.
- $nch(v)$: the number of children of $\mathbf{T}(v)$.

Figure 7 presents a graphical illustration of the above variables. The partial tree T has three subtrees and three subtree links. For instance, in the routing tree shown in Figure 5(b), the variables are computed as follows.

$r = v0$.
$ch(v0, 1) = v1, ch(v0, 2) = v2, ch(v1, 1) = v3,$
$ch(v1, 2) = v4, ch(v1, 3) = v5, ch(v2, 1) = v6.$
$nch(v0) = 2, nch(v1) = 3, nch(v2) = 1.$
$\mathbf{Ch}(v0) = \{v1, v2\}, \mathbf{Ch}(v1) = \{v3, v4, v5\}, \mathbf{Ch}(v2) = \{v6\}.$
$\mathbf{N}(v0) = \{v0, v1, v2, v3, v4, v5, v6\}, \mathbf{N}(v1) = \{v1, v3, v4, v5\},$
$\mathbf{N}(v2) = \{v2, v6\}.$

Workload related variables

- $\mathbf{I}$: the set of all I/O nodes in the network.
- $\mathbf{P}_v$: the set of compute nodes in cluster v.
- $\mathbf{I}_{v,i}$: the set of I/O nodes in tree $\mathbf{T}'(v, i)$.
- $req_r(v, p, i)$: the size of read requests to I/O node i issued by compute node p in cluster v.
- $req_w(v, p, i)$: the size of write requests to I/O node i issued by compute node p in cluster v.
- $r_r(v, i)$: the total size of read requests to I/O node i issued by the compute nodes in cluster v:

$$r_r(v, i) = \sum_{\forall p \in \mathbf{P}_v} req_r(v, p, i).$$

- $r_w(v, i)$: the total size of write requests to I/O node i issued by the compute nodes in cluster v:

$$r_w(v, i) = \sum_{\forall p \in \mathbf{P}_v} req_w(v, p, i).$$

- $r(v, i)$: the total size of read and write requests to I/O node i issued by the compute nodes in cluster v, i.e., $r(v, i) = r_r(v, i) + r_w(v, i)$.
- $R_r(v, i)$: the total size of read requests to I/O node i issued by the clusters in $\mathbf{T}(v)$, i.e., the subtree rooted at cluster v:

$$R_r(v, i) = \sum_{\forall k \in \mathbf{N}(v)} r_r(k, i).$$

- $R_w(v,i)$: the total size of write requests to I/O node i issued by the clusters in $\mathbf{T}(v)$:

$$R_w(v,i) = \sum_{\forall k \in \mathbf{N}(v)} r_r(k,i).$$

- $R(v,i)$: the total size of read and write requests to I/O node i issued by the clusters in $\mathbf{T}(v)$, i.e., $R(v,i) = R_r(v,i) + R_w(v,i)$.

- $R'_r(v,i)$: the total size of read requests to I/O node i issued by the clusters, excluding those in $\mathbf{T}(v)$:

$$R'_r(v,i) = R_r(r,i) - R_r(v,i).$$

- $R'_w(v,i)$: the total size of write requests to I/O node i issued by the clusters, excluding those in $\mathbf{T}(v)$:

$$R'_w(v,i) = R_r(r,i) - R_r(v,i).$$

3.3.2 Load Balance Traversing Algorithm

Recall that, in a routing tree, the subtree link with the maximum number of messages travelling through it simultaneously becomes the *dominating link*. Clearly, the lower the workload on the dominating link, the better the parallel I/O performance will be. Our goal is to find the optimal locations for the I/O nodes such that the workload on the dominating link is minimized. Given a routing tree T, and the request functions $r_r(v,i)$ and $r_w(v,i)$ of each cluster v and I/O node i, the workload of each subtree link and the maximum workload of a set of subtree links are computed as follows.

Note that, since a network link is bidirectional, it can be represented by two directional edges: an upward subtree edge and a downward subtree edge. Let $\mathbf{I}$ be the set of all I/O nodes, and $\mathbf{I}_i$ be the set of I/O nodes assigned to subtree $\mathbf{T}'(v,i)$ so far. First, we define the *downward subtree-edge load*, denoted by $L_d(v,i,\mathbf{I_i},\mathbf{I})$, to represent the workload of the downward subtree edge connecting cluster v to its ith child.

Two kinds of downward message travel through this edge: one for writing data to the I/O nodes in set $\mathbf{I_i}$ in subtree $\mathbf{T}'(v,i)$ by the clusters outside the subtree, and the other for reading data from the I/O nodes in set $\mathbf{I} - \mathbf{I_i}$ outside subtree $\mathbf{T}'(v,i)$ by the clusters in the subtree. For write requests, the workload is $\sum_{k \in \mathbf{I}_i} R'_w(ch(v,i),k)$, and for read requests, the workload is $\sum_{k \in \mathbf{I}-\mathbf{I_i}} R_r(ch(v,i),k)$. The downward subtree-edge load is the sum of the read load and the write load, as shown in Equation 1. The *upward subtree-edge load* on the link, denoted by $L_u(v,i,\mathbf{I_i},\mathbf{I})$, is computed similarly by Equation 2. Furthermore, since two messages travelling

through the same bidirectional link in opposite directions do not interfere with each other, the subtree-link load, denoted by $L_L(v, i, \mathbf{I_i}, \mathbf{I})$, is the maximum of the link's downward and upward workloads, as shown in Equation 3.

$$L_d(v, i, \mathbf{I_i}, \mathbf{I}) = \{\sum_{k \in \mathbf{I}_i} R'_w(ch(v, i), k)\} + \{\sum_{k \in \mathbf{I} - \mathbf{I}_i} R_r(ch(v, i), k)\} \quad (1)$$

$$L_u(v, i, \mathbf{I_i}, \mathbf{I}) = \{\sum_{k \in \mathbf{I}_i} R'_r(ch(v, i), k)\} + \{\sum_{k \in \mathbf{I} - \mathbf{I}_i} R_w(ch(v, i), k)\} \quad (2)$$

$$L_L(v, i, \mathbf{I_i}, \mathbf{I}) = \max\{L_d(v, i, \mathbf{I_i}, \mathbf{I}), L_u(v, i, \mathbf{I_i}, \mathbf{I})\} \quad (3)$$

Table 1: Read requests from clusters to I/O nodes. The figures represent the number of units. The unit size is 100MB

I/C	v_0	v_1	v_2	v_3	v_4	v_5	v_6
i_0	1	3	1	2	1	2	2
i_1	1	2	2	1	2	1	1
i_2	3	1	1	2	4	3	1
i_3	1	1	3	1	1	2	1

Table 2: Write requests from clusters to I/O nodes. The figures represent the number of units. The unit size is 100MB.

I/C	v_0	v_1	v_2	v_3	v_4	v_5	v_6
i_0	1	1	1	1	0	2	2
i_1	2	2	1	0	2	1	1
i_2	0	1	0	1	0	0	3
i_3	1	0	1	1	1	1	2

Using Figure 5 as an example, the subtree-link loads on the four I/O nodes, $\mathbf{I} = \{i0, i1, i2, i3\}$, and the read/write requests listed in Table 1 and Table 2 respectively, can be computed as follows (assuming that $\mathbf{I_1} = \mathbf{I_2} = \{\}$ for $\mathbf{T}(v0)$ initially).

$L_d(v0, 1, \{\}, \mathbf{I}) = R_r(v1, i_0) + R_r(v1, i_1) + R_r(v1, i_2) + R_r(v1, i_3) =$
$8 + 6 + 10 + 5 = 29.$
$L_u(v0, 1, \{\}, \mathbf{I}) = R_w(v1, i_0) + R_w(v1, i_1) + R_w(v1, i_2) + R_w(v1, i_3) =$
$4 + 5 + 3 + 3 = 15.$
$L_L(v0, 1, \{\}, \mathbf{I}) = \max\{L_d(v0, 1, \{\}, \mathbf{I}), L_u(v0, 1, \{\}, \mathbf{I})\} =$
$\max\{29, 15\} = 29.$

$L_d(v0, 2, \{\}, \mathbf{I}) = R_r(v2, i_0) + R_r(v2, i_1) + R_r(v2, i_2) + R_r(v2, i_3) =$
$3 + 3 + 2 + 4 = 12.$
$L_u(v0, 2, \{\}, \mathbf{I}) = R_w(v2, i_0) + R_w(v2, i_1) + R_w(v2, i_2) + R_w(v2, i_3) =$
$3 + 2 + 0 + 3 = 8.$
$L_L(v0, 2, \{\}, \mathbf{I}) = \max\{L_d(v0, 2, \{\}, \mathbf{I}), L_u(v0, 2, \{\}, \mathbf{I})\} =$
$\max\{12, 8\} = 12.$

When a new I/O node x is assigned to tree $\mathbf{T}(v)$, the subtree-link loads will change accordingly. We define $M_L(v, i, x, \{\mathbf{I}_1, ..., \mathbf{I}_{nch(v)}\}, \mathbf{I})$ to be the maximum load among the $nch(v)$ subtree links when a new I/O node x is assigned to the ith subtree of v, as shown in Equation 4.

$$M_L(v, i, x, \{\mathbf{I}_1, ..., \mathbf{I}_{nch(v)}\}, \mathbf{I}) = \max\{L_L(v, i, \mathbf{I_i} + \{\mathbf{x}\}, \mathbf{I}), \max_{\forall j \neq i}\{L_L(v, j, \mathbf{I_j}, \mathbf{I})\}\} \quad (4)$$

Note that $M_L(v, 0, x, \{\mathbf{I}_1, ..., \mathbf{I}_{nch(v)}\}, \mathbf{I})$ is the maximum load among the $nch(v)$ subtree links when a new I/O node x is assigned to cluster v, the root of tree $\mathbf{T}(v)$, as shown in Equation 5.

$$M_L(v, 0, x, \{\mathbf{I}_1, ..., \mathbf{I}_{nch(v)}\}, \mathbf{I}) = \max_{\forall i \in \mathbf{Ch}(v)}\{L_L(v, i, \mathbf{I_i}, \mathbf{I})\} \quad (5)$$

Returning to Figure 5, before we decide the location of the first I/O node, $i0$, the maximum subtree link load if $i0$ is assigned to $v0$, $v1$, or $v2$ can be computed as follows.

$M_L(v0, 0, i0, \{\mathbf{I}_1, ..., \mathbf{I}_{nch(v0)}\}, \mathbf{I}) = \max\{L_L(v0, 1, \{\}, \mathbf{I}), L_L(v0, 2, \{\}, \mathbf{I})\} =$
$\max\{29, 12\} = 29.$

$L_d(v0, 1, \{\mathbf{i0}\}, \mathbf{I}) = R'_w(v1, i0) + R_r(v1, i1) + R_r(v1, i2) + R_r(v1, i3) =$
$4 + 6 + 10 + 5 = 25.$
$L_u(v0, 1, \{\mathbf{i0}\}, \mathbf{I}) = R'_r(v1, i0) + R_w(v1, i1) + R_w(v1, i2) + R_w(v1, i3) =$
$4 + 5 + 3 + 3 = 15.$
$L_L(v0, 1, \{\mathbf{i0}\}, \mathbf{I}) = \max\{L_d(v0, 1, \{\mathbf{i0}\}, \mathbf{I}), L_u(v0, 1, \{\mathbf{i0}\}, \mathbf{I})\} = \max\{25, 15\} = 25.$
$M_L(v0, 1, i0, \{\mathbf{I}_1, ..., \mathbf{I}_{nch(v0)}\}, \mathbf{I}) = \max\{L_L(v0, 1, \{\mathbf{i0}\}, \mathbf{I}), L_L(v0, 2, \{\}, \mathbf{I})\} =$
$\max\{25, 12\} = 25.$

$L_d(v0, 2, \{\mathbf{i0}\}, \mathbf{I}) = R'_w(v2, i0) + R_r(v2, i1) + R_r(v2, i2) + R_r(v2, i3) =$
$5 + 3 + 2 + 4 = 14.$
$L_u(v0, 2, \{\mathbf{i0}\}, \mathbf{I}) = R'_r(v2, i0) + R_w(v2, i1) + R_w(v2, i2) + R_w(v2, i3) =$
$9 + 2 + 0 + 3 = 14.$
$L_L(v0, 2, \{\mathbf{i0}\}, \mathbf{I}) = \max\{L_d(v0, 2, \{\mathbf{i0}\}, \mathbf{I}), L_u(v0, 2, \{\mathbf{i0}\}, \mathbf{I})\} = \max\{14, 14\} = 14.$
$M_L(v0, 2, i0, \{\mathbf{I}_1, ..., \mathbf{I}_{nch(v0)}\}, \mathbf{I}) = \max\{L_L(v0, 2, \{\mathbf{i0}\}, \mathbf{I}), L_L(v0, 1, \{\}, \mathbf{I})\} =$
$\max\{14, 29\} = 29.$

Clearly, i_0 should be placed in cluster v1, since the value of $M_L(v0, 1, i0, \{\mathbf{I}_1, ..., \mathbf{I}_{nch(v0)}\}, \mathbf{I}) = \max\{L_L(v0, 1, \{\mathbf{i0}\}, \mathbf{I}), L_L(v0, 2, \{\}, \mathbf{I})\} = 25$ is the smallest of the three M_L values.

Given a routing tree $\mathbf{T}(r)$, a set of I/O nodes $\mathbf{I}$, and the request functions, the *Load Balance Traversing* algorithm assigns the I/O nodes in $\mathbf{I}$ to the root or the subtrees of the root. The same dispatch-selection process is applied recursively to each subtree to determine the I/O node assignment for the next tree levels. The dispatch procedure iterates over the $|I|$ I/O nodes. In each iteration, the I/O node is assigned to the root or the subtree that minimizes the maximum subtree-link loads. When the iteration process terminates, the I/O node assignment for each cluster at the current tree level can be determined. The I/O node assignment for the processors in each cluster is then determined by calling the *Load Balance Match* algorithm. The pseudo code of **LBT** is outlined in Figure 9. For the routing tree of the general networked cluster shown in Figure 5(a) and the read/write requests in Tables 1 and 2, the final assignment of the I/O nodes by `LBT` is depicted in Figure 8.

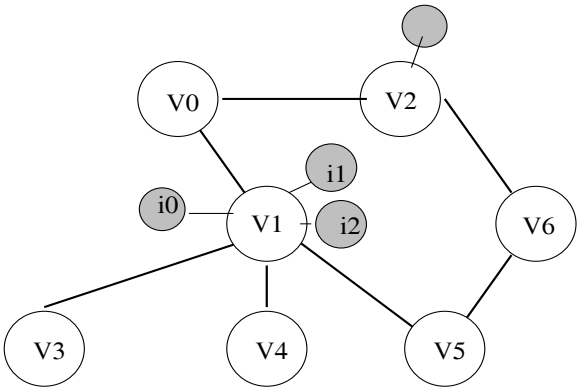


Figure 8: I/O assignment for the general networked cluster in Figure 5(a).

4 Experiment Results

In this section, we evaluate our Load-Balance-Matching (`LBM`) algorithm and Load-Balance-Traversing (`LBT`) algorithm on same-cost clusters and general-networked clusters respectively.

One of the pivotal factors to be considered is the data transfer patterns between compute nodes and I/O servers. Although "uniform" data transfers are frequently seen in scientific applications (where each client requests roughly the same amount of data, which is evenly distributed among the I/O servers), there are also many applications that exhibit hot spots in specific parts of the data and therefore in specific I/O servers. Thus, the actual effect of I/O assignment algorithms depends very much on the workload applied by the system. Since very few studies have analyzed the workload of parallel I/O and we can not obtain real parallel I/O trace data, we generate a synthetic workload in the following way. We do a one-to-one mapping of a geometric sequence of m items to the m I/O servers, with a common ratio r, where $0 < r <= 1.0$ (that is, the first item is 1.0, the second is r, the third is r^2, and so on.) Let the sum of the geometric sequence be S, and divide each item by S. In addition, let the resulting sequence be $p_1, p_2, \ldots, p_m$. It is clear that, for all p_i, $0 < p_i <= 1.0$ and $\sum_1^m p_i = 1$. Note that p_i represents the probability that a data transfer will be assigned to I/O server i. We choose an I/O server for each data request by picking a number between 0 and 1 at random, and decide the server's location by comparing the random number with the prefix sums of the probability sequence. With a large number of samples, our synthetically generated workload emulates a normal distribution function. The advantage of this method is that, by choosing different common ratios r, we can experiment with a wide range of workloads, ranging from a uniform workload ($r = 1.0$) to a workload with hot spots (with a very small r). Multiple hot spots can also be generated by mapping multiple sets of geometric sequences to the I/O servers.

Algorithm Load_Balance_Traversing

Input :

v : a cluster, and $\mathbf{Ch}(v) = \{c_1, c_2, \ldots, c_{nch(v)}\}$.

$\mathbf{I}'$: the set of I/O nodes to be assigned to the processors in tree $\mathbf{T}(v)$.

Output : The mapping from set $\mathbf{I}'$ to the set of processors P in the network.

Description :

[**step 1**] /* initialize variables */

Let $\mathbf{A} = \{\}$ and $\mathbf{I}'_k = \{\}$, for each k, $0 \leq k \leq nch(v)$.

[**step 2**] /* determine the location of the I/O node in $\mathbf{I}'$ with the highest load */

Pick an I/O node x from $\mathbf{I}'$ with the highest $R(root, x)$, and remove it from $\mathbf{I}'$.

Find i such that $M_L(v, i, x, \{\mathbf{I}_1, ..., \mathbf{I}_{nch(v)}\}, \mathbf{I})$ is maximized for $0 \leq i \leq nch(v)$

[**step 3**] /* assign the I/O node to either the root or the subtrees depending on the value of i */

If i is zero, assign x to $\mathbf{A}$ to dispatch I/O node x to cluster v.

If i is not zero, assign x to $\mathbf{I}'_i$ to dispatch I/O node x to $\mathbf{T}'(v, i)$.

[**step 4**] Repeat [step2] and [step3] until $\mathbf{I}'$ is empty.

[**step 5**] /* perform intra-cluster I/O assignment for the root cluster v */

Let $\mathbf{P}_v = \{p_1, p_2, ..., p_n\}$ be the set of processors in cluster v, and $\mathbf{A} = \{i_{a_1}, i_{a_2}, ..., i_{a_m}\}$ be the set of I/O nodes dispatched to cluster v.

/* Construct the I/O matrix W */

Let $w_{f,g} = req_r(v, p_g, i_{a_f}) + req_w(v, p_g, i_{a_f})$.

$$W = \begin{pmatrix} w_{1,1} & w_{1,2} & \ldots & \ldots & w_{1,n} \\ w_{2,1} & w_{2,2} & \ldots & \ldots & w_{2,n} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ w_{m,1} & w_{m,2} & \ldots & \ldots & w_{m,n} \end{pmatrix}$$

Call LBM(P,A,E,W) to assign the I/O nodes in $\mathbf{A}$ to the processors in cluster v.

[**step 6**] /* Recursively call Load_Balance_Traversing on the children of v */

Call **LBT(c_k, $\mathbf{I}'_k$)** for each child c_k of v.

Note :

$\mathbf{I}$ is the set of all I/O nodes in the network.

$req_r(u, p, i)$ computes the number of read requests sent from processor p in cluster u to I/O node i.

$req_w(u, p, i)$ computes the number of write requests sent from processor p in cluster u to I/O node i.

$r_r(u, i) = \sum_{\forall p \in \mathbf{P}_v} req_r(u, p, i)$.

$r_w(u, i) = \sum_{\forall p \in \mathbf{P}_v} req_w(u, p, i)$.

$R(v, i) = \sum_{\forall k \in \mathbf{N}(v)} (r_r(k, i) + r_w(k, i))$.

Figure 9: The Load Balance Traversing algorithm for assigning I/O nodes based on recursive traversal of a tree

4.1 Experiments on Same-Cost Clusters

We conduct experiments on same-cost, intra-cluster I/O assignments on a 64-node Pentium-4 cluster, with two alternative interconnects: a Fast Ethernet and a Myrinet. Each node is equipped with a 40GB IDE disk. The network bandwidth and disk bandwidth are measured as follows: 40.36MB/sec on Fast Ethernet, 462.56MB/sec on Myrinet, and 66.92 MB/sec on an IDE disk.

We implement four algorithms for comparison: a baseline algorithm (Baseline), which picks the first m processing nodes as I/O servers; a randomized algorithm (Random), which selects I/O nodes randomly; the *maximum weight matching* algorithm (MWM), which chooses I/O nodes by minimizing the total amount of remote data transferred and our `LBM` algorithm.

We consider two factors in the experiments: the length of messages (i.e., the edge weights in the bipartite graph) and the connection pattern of the edges in the bipartite graph. We only report the results of the read operations, since the results of the write operations are similar.

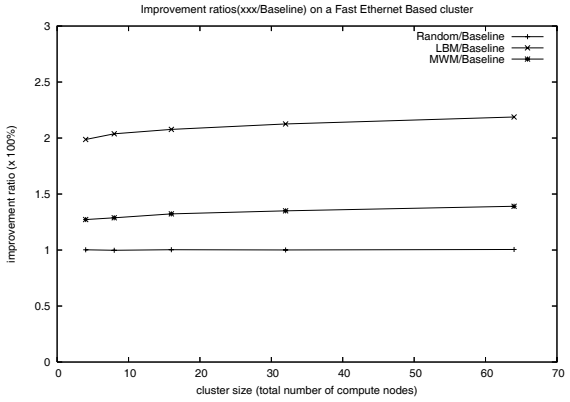


Figure 10: Improvement in I/O performance of Random, LBM, and MWM over the baseline algorithm on a 64-node Fast Ethernet-based cluster with different numbers of servers.

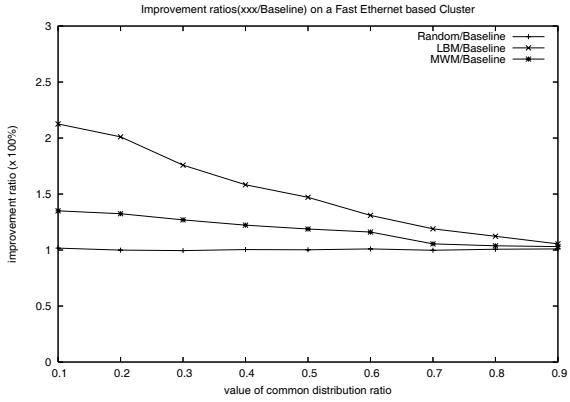


Figure 11: Improvement in I/O performance of Random, LBM, and MWM over the baseline algorithm on a 64-node Fast Ethernet-based cluster with different common distribution ratios r.

Figure 10 compares the I/O performances of the four algorithms on Fast Ethernet-based clusters with up to 64 nodes, and a fixed common distribution ratio $r = 0.5$. We observe that the improvement ratios of `LBM` and `MWM` increase as the cluster size increases. This is because the number of remote data transfers increases in proportion to the cluster size, which also increases the chance of load imbalance among I/O nodes when a naive strategy like `Baseline` is used.

Figure 11 shows the impact of data transfer patterns. We fix the number of I/O nodes at

eight, and vary the common distribution ratio r between 0.1 and 0.9. As shown in the figure, `LBM` performs better than the other algorithms when the workload is not balanced (i.e., when r is small). However, `LBM`'s improvement ratio drops rapidly when the common distribution ratio increases, whereas the improvement ratio of `MWM` only drops slightly. The larger the value of r, the closer the performances of `LBM` and `MWM` will be. A possible reason is that, when the value of r increases, the load imbalance among I/O nodes becomes less significant, which in turn reduces the optimization benefit derived by `LBM`.

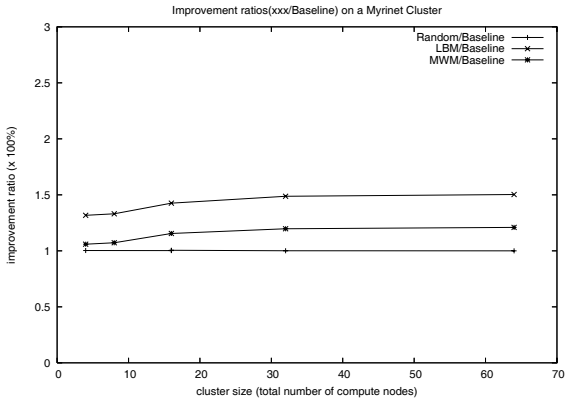


Figure 12: Improvement in I/O performance of the Random, LBM, and MWM algorithms over the baseline algorithm on a 64-node Myrinet-based cluster with different numbers of servers.

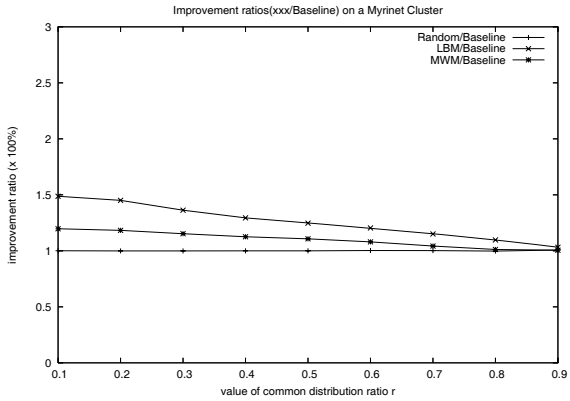


Figure 13: Improvement in I/O performance of the Random, LBM, and MWM algorithms over the baseline algorithm on a 64-node Myrinet-based cluster with different common distribution ratios r.

Figure 12 compares the performance of the algorithms on the same cluster with faster interconnects (Myrinet). As expected, in a faster network the benefit of optimization is less siginificant (compared with the results in Figure 10). This is because both `LBM` and `MWM` try to reduce the cost of remote data transfers in slower networks.

Figure 13 shows the improvement ratios under different values of the common distribution ratio r. Similar to the results in Figure 11, the improvement ratios of `MWM` and `LBM` also decline as the value of r increases.

4.2 Experiments on General-Networked Clusters

We evaluate our algorithms by analytical computation (i.e., computing the maximum link contention) and by simulation. The simulation parameters are network latency and bandwidth, disk latency and bandwidth, synchronization cost, and buffer size. These parameters were obtained experimentally from a 64-node Pentium-4 cluster with Myrinet interconnects and IDE disks.

In all the experiments, we fix the request size at 4MB. There are 37 clusters with the number of processors in each cluster ranging from 14 to 32, and there are up to 14 I/O servers to be placed. In total, one million data requests are generated by the client processors in the clusters. The distribution of the data requests among the clusters and the workload among the servers is determined by two *common distribution ratios* (r_c for client-side distribution, and r_s for server-side distribution) as described earlier.

We compare five algorithms: the Even algorithm, which assigns I/O servers evenly among the clusters; the Random algorithm, which assigns I/O servers randomly; the Shortest Path algorithm, which ensures the average distance between the I/O servers and the clusters is the shortest; the Request Volume algorithm, which assigns the I/O servers to the clients with the maximum number of data requests; and our Load Balancing Traversing algorithm.

Specifically, we investigate the impact of the following factors on the performance of the above-mentioned algorithms: the number of I/O servers (denoted as NIS), the distribution ratio of data requests among clients (denoted as DRC), and the distribution ratio of the work load among I/O servers (denoted as DRS). The lower the distribution ratios DRC and DRS, the more significant the hot spots between client requests and between server workloads.

4.2.1 Effect of the Number of I/O Servers

In the first set of experiments, we fix the values of DRC and DRS, and vary the value of NIS. Figure 14 compares the maximum link contention. Since we fix the total number of data requests, the workload on the servers generally decreases as the number of servers increases. As a result, link contention also decreases. Among the five algorithms, `LBT` causes the least contention.

Figure 15 compares the improvement ratio of I/O throughput over the baseline algorithm (i.e., the EVEN algorithm). As the number of I/O servers increases, the time required to process all the data requests decreases; thus, the I/O throughput increases. Consistent with the maximum contention result in Figure 14, `LBT` achieves the highest I/O throughput.

4.2.2 Effect of the Server Workload

In this set of experiments, we fix the values of NIS and DRC and vary the value of DRS. Note that a large DRS value implies a more evenly distributed workload among the servers. Conversely, a small DRS value indicates that the workload is badly balanced, which leads to the development of hot spots.

As shown in Figures 16 and 17, as the value of DRS decreases, data traffic tends to focus on a specific server or servers. This increases the possibility of link contention (Figure 16), which

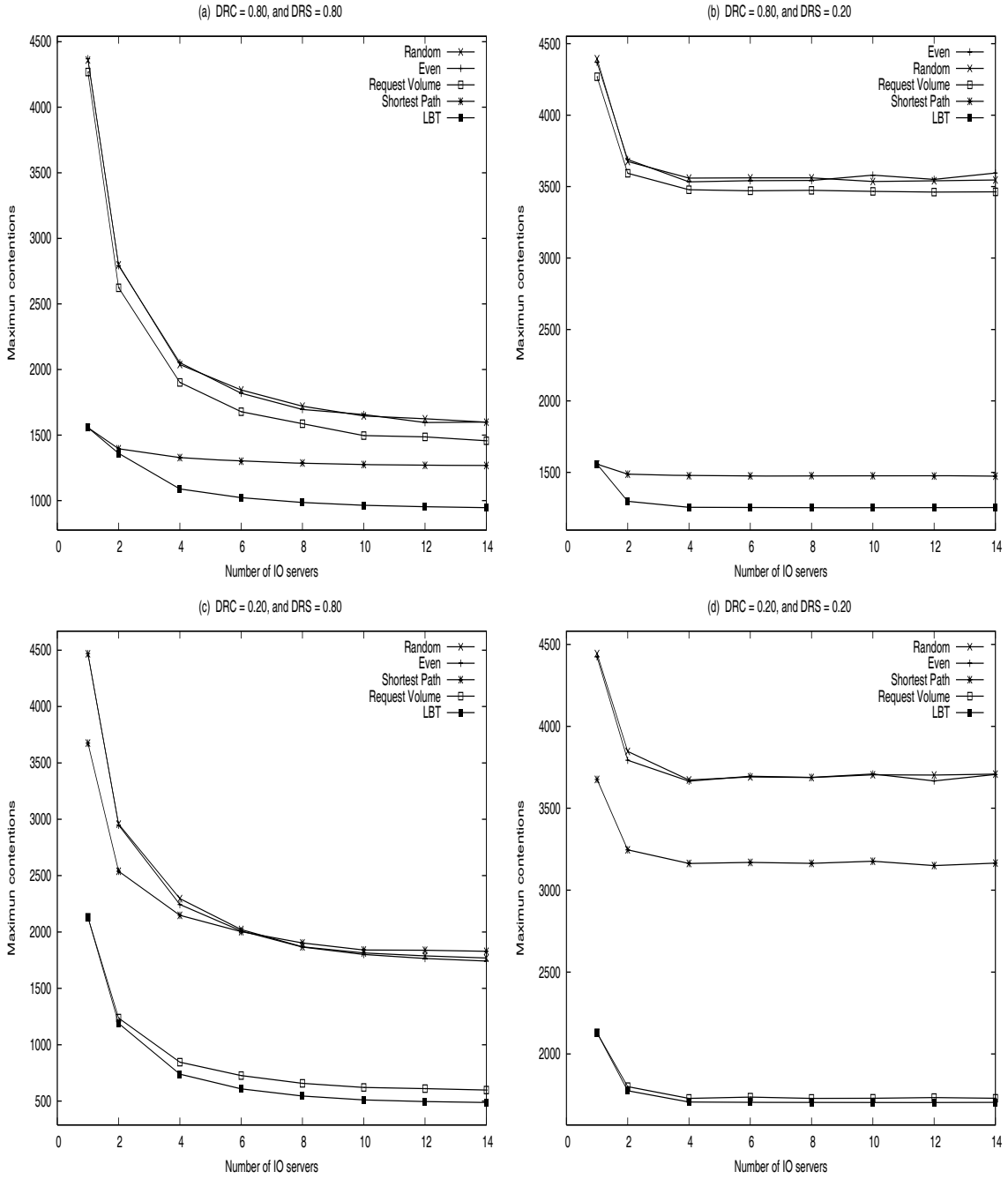


Figure 14: The effect of the number of I/O servers, measured by the maximum amount of contention.

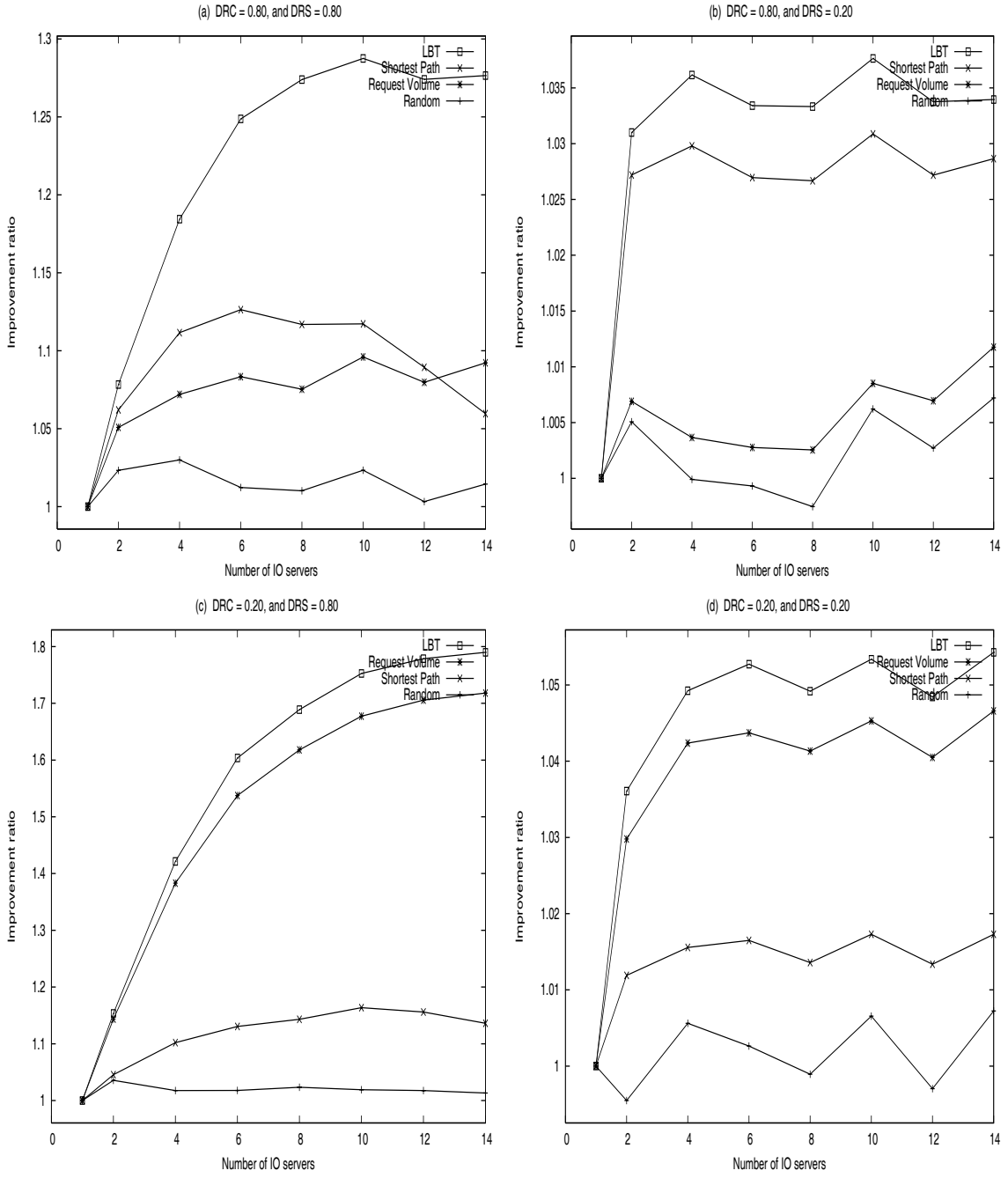


Figure 15: The effect of the number of I/O servers, measured by the improvement in I/O throughput over the EVEN algorithm.

in turn degrades I/O throughput (Figure 17). Among the five algorithms, `LBT` causes the least link contention and achieves the highest I/O throughput.

Furthermore, Figures 16(a), (b), (c), and (d) show the combined effect of DRC (the workload distribution among clients) and DRS. When DRC is small, data transfers focus on certain clients and servers, which causes higher link contention (Figures 16(b) and 16(d)) and lower I/O throughput (Figures 17(b) and 17(d)).

4.2.3 Effect of the Client Workload

In this set of experiments, we fix the values of NIS and DRS and vary the value of DRC. Note that a large DRC value implies more evenly distributed data requests among clients, while a small value indicates that the distribution of data requests is badly balanced, which causes hot spots.

Figures 18 and 19 show that, as the value of DRC decreases, data requests focus on a certain client or clients, which increases the possibility of link contention. The `Even`, `Random` and `Shortest Path` do not consider the distribution of data requests, tend to cause more link contention. In contrast, the `Request Volume` algorithm and our `LBT` algorithm consider the workload on both the servers and the clients; thus, they are able to place the servers such that the distribution of data traffic is better, which reduces the chances of link contention.

5 Related Work

Although I/O server placement has been extensively studied in multimedia research [10, 11, 26, 29, 31, 32], unfortunately the results of the research cannot be applied to parallel I/O. The above works assume that a client's I/O request can be satisfied entirely by one I/O server, and the goal is to place multiple copies of the server over the network such that each client is within a certain distance of at least one copy of the data. Parallel I/O, however, is more complicated in that the data is distributed over multiple I/O servers and each parallel I/O operation involves multiple data transfer requests to multiple I/O servers.

Next, we discuss some related works in the area of parallel I/O. The problem of I/O placement in traditional parallel machines with regular network topologies, such as mesh, tori, hypercubes, and ring topologies is addressed in [4, 8, 24, 30]. However, switch-based clusters of workstations/PCs typically adopt irregular topologies to allow the construction of scalable systems with an incremental expansion capability. Irregular topologies lack many of the attractive mathematical properties of regular topologies, which makes optimizing server placement in such networks a difficult task.

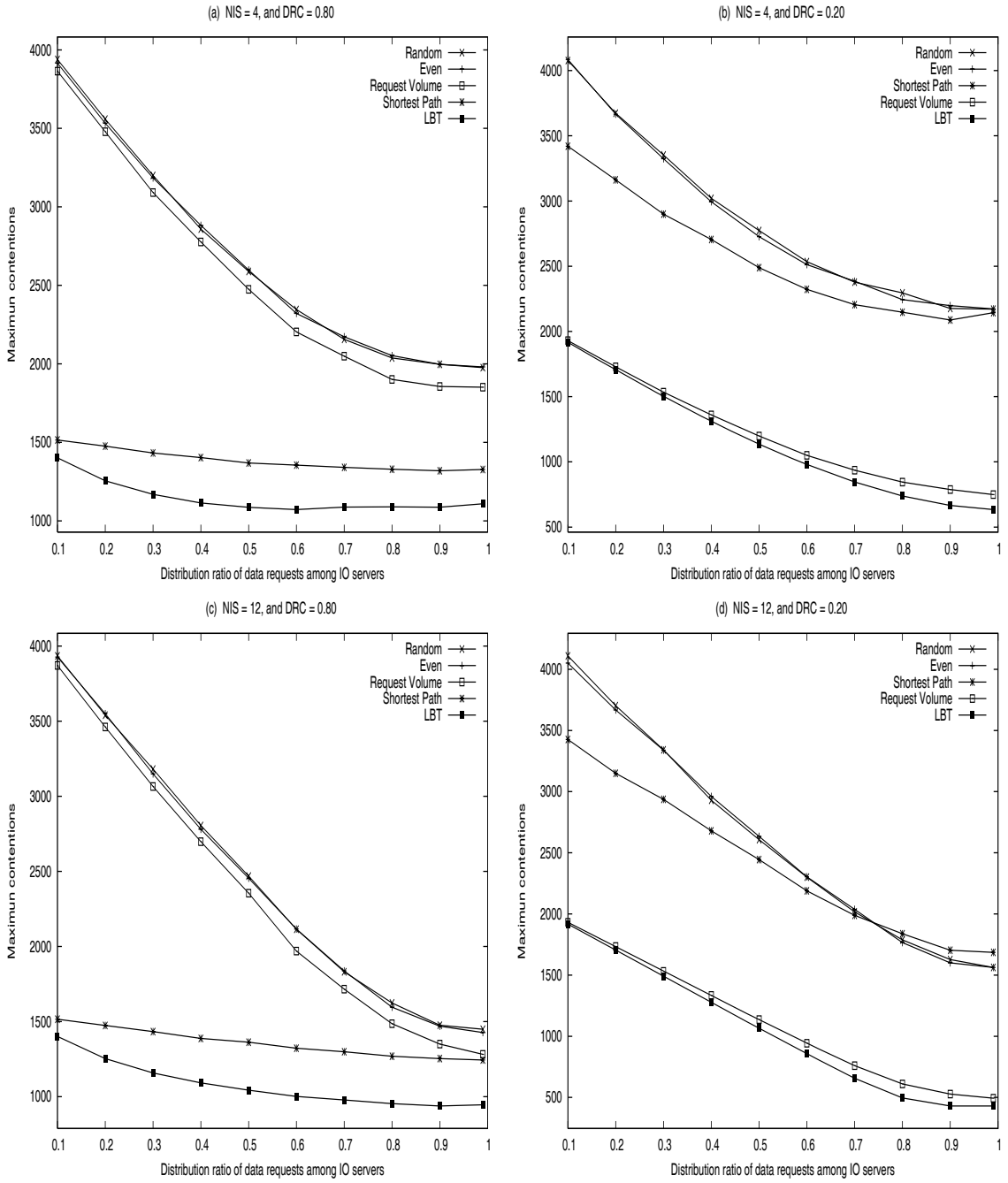


Figure 16: Effect of the distribution ratio of data requests among I/O servers, measured by the maximum amount of contention.

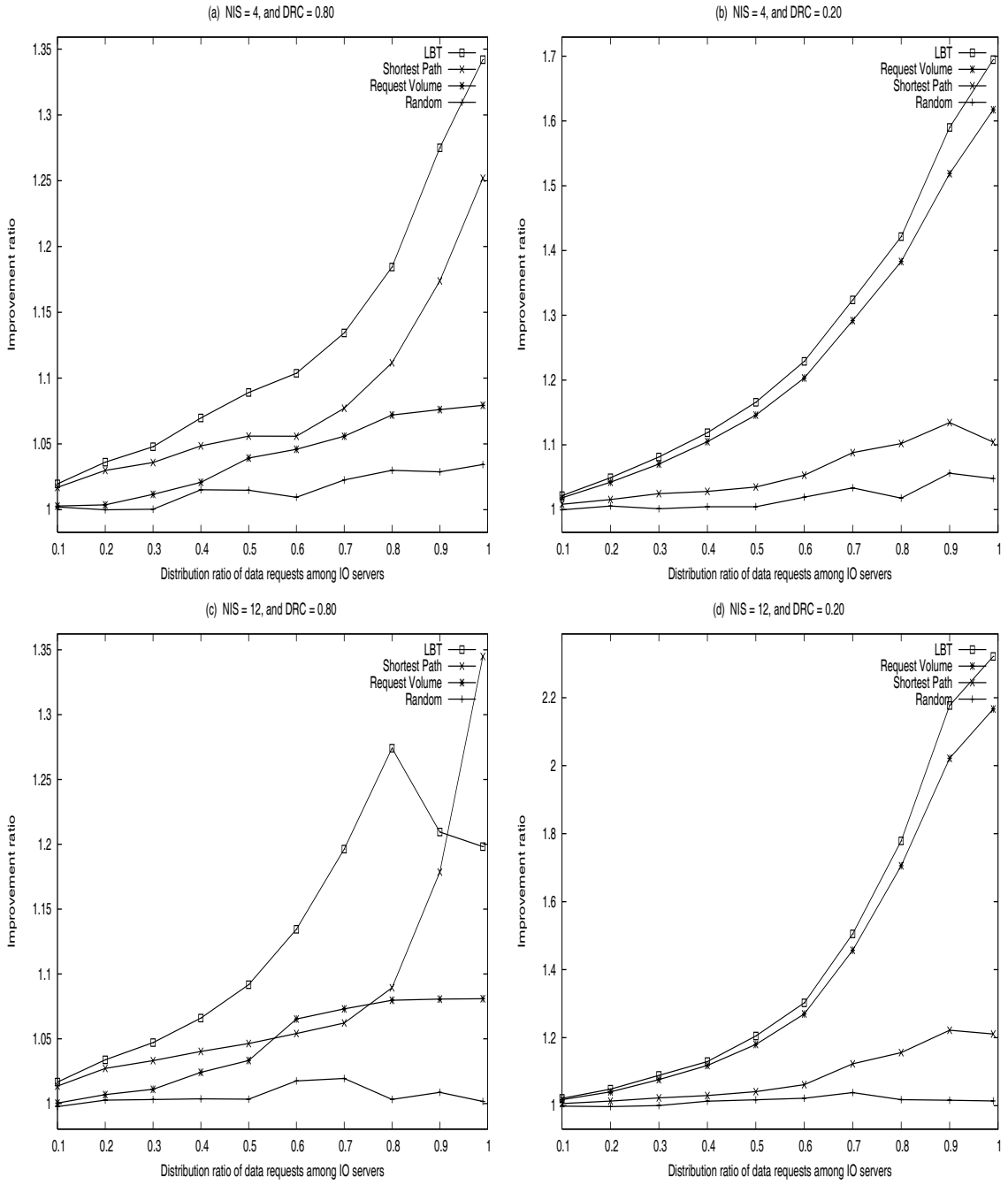


Figure 17: Effect of the distribution ratio of data requests among I/O servers, measured by the improvement in I/O throughput over the EVEN algorithm.

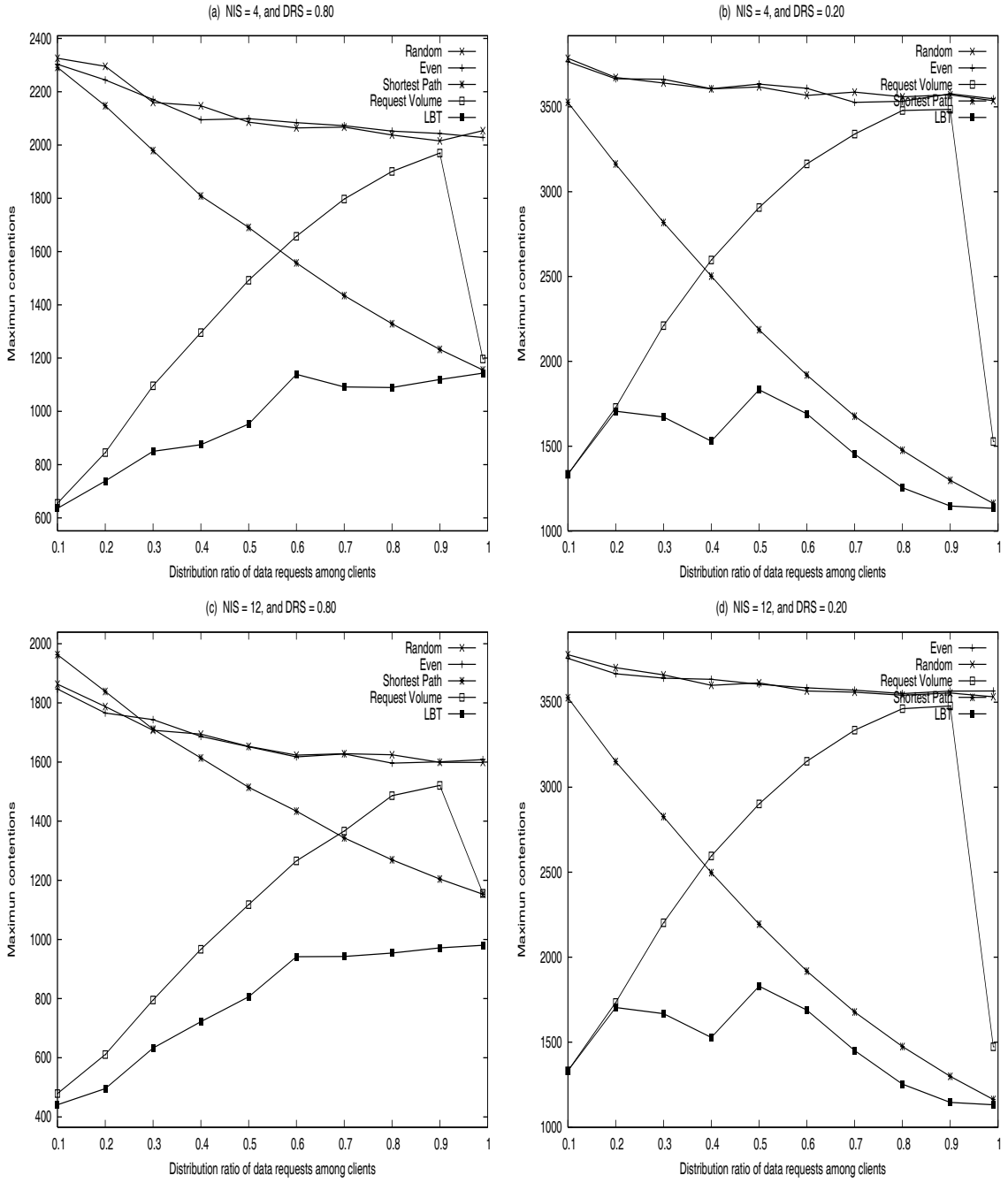


Figure 18: Effect of the distribution ratio of data requests among clients, measured by the maximum amount of contention.

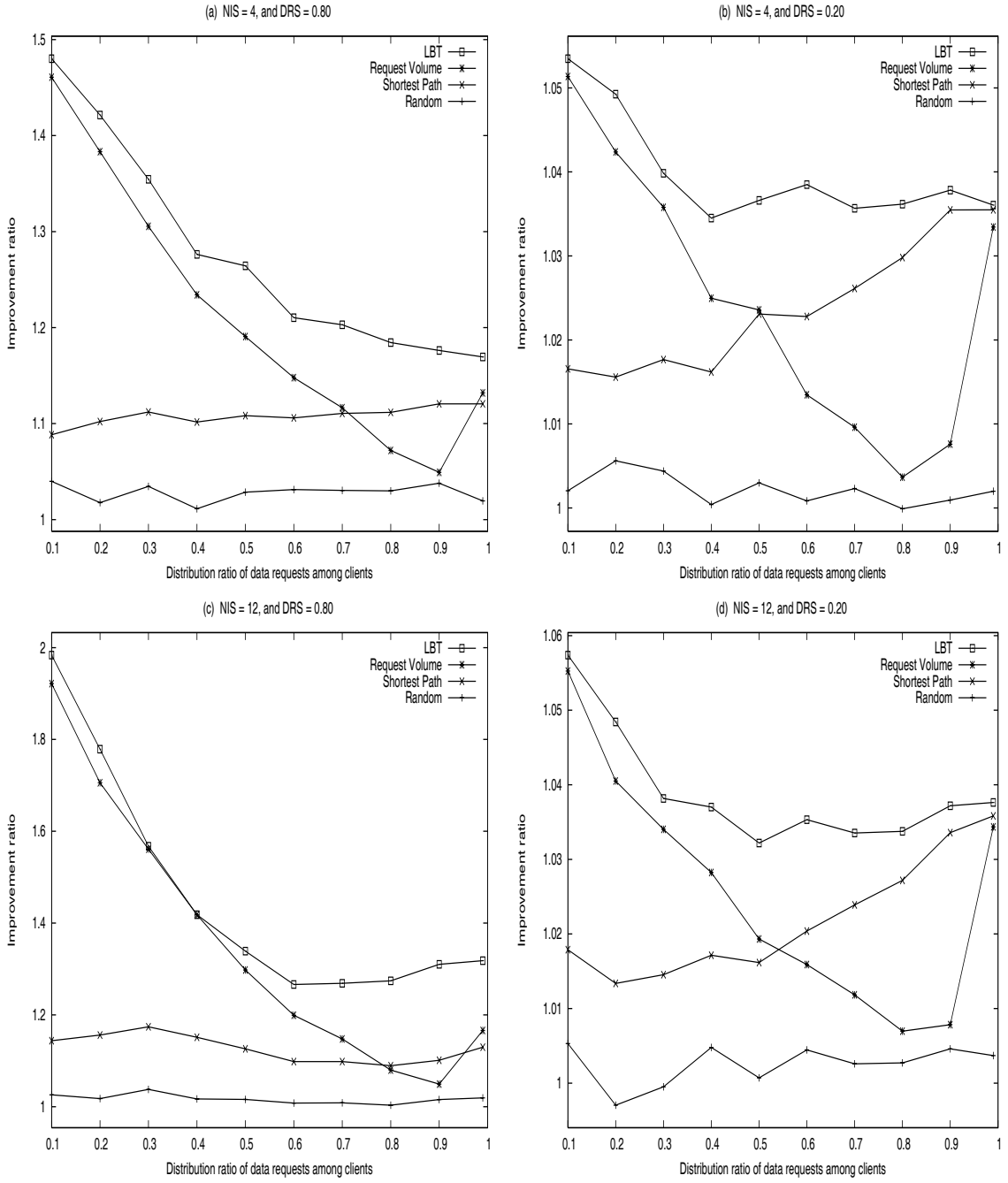


Figure 19: Effect of the distribution ratio of data requests among clients, measured by the improvement in I/O throughtput over the EVEN algorithm.

Subraman et al. [28] studied Panda 2.1's implementation of I/O server placement and reported basic performance results. Kuo et al.[18] experimented with I/O placement for a real application on SP2. They found that I/O servers can also be used efficiently for an application that requires the data to be offloaded to Unitree when the application is finished, if the compute nodes not involved in offloading the data can be released immediately. The same authors also propose an algorithm for server placement that guarantees minimal remote data access during I/O operations. In [8], Cho et. al. quantify the savings derived by careful placement of servers in clusters connected via a shared medium, and use an analytical model to explain other performance trends. Although our load-balance-based approach to the placement of I/O servers was inspired by the Panda system, it focuses on clusters connected by commodity switches. Our experiment results show that, in switch-based clusters, the approaches' performances are comparable for uniform data transfers; however, our approach outperforms Panda for non-uniform server workloads.

VIP-FS [14] is a parallel file system that provides a collective I/O interface for scientific applications run in parallel and distributed environments. This I/O strategy is designed for systems with a shared-medium network, which is a potential source of congestion. VIP-FS prevents network congestion by reducing the number of I/O requests made by all the compute nodes involved in a collective I/O operation. In parallel and distributed environments, careful placement of I/O servers can also reduce the total number of non-local data transfer operations and thus can help reduce network congestion.

Kotz, et. al. [17] exploited the use of I/O nodes for computation in an MIMD multiprocessor. By extensive simulation studies, they found that, except for some I/O-intensive applications, I/O nodes are mostly under-utilized such that 80-97% are available for computation at any one time. To maximize the use of the available computing power provided by the I/O nodes, as many of them as possible are also used for computation. Under this model, I/O nodes and compute nodes are separate, and I/O nodes are utilized by another application program. In contrast, our notion of part-time I/O assumes there is no distinction between I/O nodes and compute nodes and an application can perform both computation and I/O on the same node. This assumption gives us the flexibility to chose I/O nodes on a per application basis.

VIPIOS [6] is a parallel I/O system designed for use with High Performance Fortran and Vienna Fortran. It exploits two levels of data locality: the logical data's locality by mapping between servers and the application's processes, and the physical data's locality by mapping between servers and disks. It also exploits two sources of I/O parallelism: inter-server parallelism in writing from application processes to the servers, and intra-server parallelism in writing from a server to multiple disks attached to that server. The mapping strategy of VIPIOS is similar to the way the Panda system exploits local data; both methods are effective in reducing I/O time

for applications that have uniform data transfer patterns. Meta-VIPIOS [13] extends VIPIOS to harness I/O resources distributed over the Internet.

Jovian-2 [2] is a parallel I/O library developed at the University of Maryland. Similar to the notion of part-time I/O, the I/O nodes in Jovian-2 are non-dedicated. All nodes are allowed to run both an application thread and an I/O server thread in the system's peer-to-peer configuration, and each node can make an I/O request to any other node. To minimize the I/O time, Jovian-2 utilizes global information about I/O requests, which is available from the application, to prefetch and cache data.

Mache, et. al. [19] devised a processor allocation strategy that is sensitive to parallel I/O traffic and the resulting network contention. Their strategy improves the average response time of parallel I/O intensive jobs by up to a factor of 4.5. More recently, they proposed an allocation strategy that is sensitive to both communication and I/O operations [20]. Their results show that spatial layout is more critical for I/O-intensive applications at lower utilization levels and more critical for communication-intensive applications at higher utilization levels. The results also show that, in general, the impact of I/O traffic is dominant.

6 Conclusion

We have presented a novel two-level strategy for placing I/O servers in switch-based clusters. For server placement within a same-cost cluster, we formulated the problem as a weighted bipartite matching with the goal of balancing the workload on the I/O servers. We have also proposed an $O(n^{\frac{3}{2}}m(\log n + \log m))$ algorithm, called *Load Balancing Matching* (`LBM`), to find the optimal solution for this problem, where n is the number of compute nodes and m is the number of I/O nodes.

Our experiment results on a 64-node PC cluster (with Myrinet interconnects and Fast Ethernet interconnects) validate our approach. The results indicate that our method is comparable to three other methods for parallel I/O operations with an *uniform* server workload, and superior to these methods for parallel I/O operations with a *non-uniform* server workload.

In the second part of the paper, we investigate server placement for general networked clusters, which consist of a set of same-cost subclusters connected by high-speed networks. General networked clusters typically adopt irregular topologies to allow the construction of scalable systems with an incremental expansion capability. Also, due to limited number of network links between subclusters, link contention between subclusters is one of the dominant factors that determine parallel I/O performance in general networked clusters.

We show that finding an optimal placement strategy that minimizes link contention is computationally intractable. To resolve the problem, we propose a tree-based heuristic algorithm,

called *Load Balance Traversing* (**LBT**), which balances the workload on the links by recursive traversal of the routing tree of the network. After the assignment of servers to subclusters has been determined, the location of each server within its assigned subcluster is determined by the *Load Balance Matching* (`LBM`) algorithm. The simulation results demonstrate that, under various experimental settings, namely, the number of servers, the distribution of the server workload, and the distribution of data requests on the clients, our method is superior to the four methods used for comparison.

Finally, we consider some possible directions for our future work. In this paper, we assume that the application contains only one dominant program block. The problem is simply to decide the optimal placement of I/O servers for this program block. For applications with several program blocks, each of which may prefer a different data distribution strategy, finding the optimal placement strategy for each block separately may result in excessive data migration between the blocks. In this case, global optimization techniques may be useful for finding a good overall solution.

Furthermore, since the assignment strategy decides the optimal location for the data storage, data must be migrated to that location if it is stored elsewhere in the system. In this paper, based on the evidence of several application programs we have studied, we assume that the dominant program block is likely to execute iteratively many times; thus, the cost of migrating data to the optimal location can be amortized. In the future, we will investigate the interplay between server placement and the cost of data migration under more general models.

Acknowledgements

The authors wish to thank Hsih-I Lu for many inspiring discussions and useful suggestions, which helped us improve our Load Balancing Matching algorithm. This work is supported in part by National Science Council of Taiwan under grant number NSC-95-2221-E-001-001.

References

[1] Guide to myrinet-2000 switches and switch networks. Technical report, Myricom, Inc., Aug. 2001.

[2] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. Hollingsworth, J. Saltz, and A. Sussman. Tuning the performnace of i/o-intensive parallel applications. In *Proc. the 4th Annual Workshop on I/O in Parallel and Distributed Systems (IOPADS'96)*, pages 15–27, 1996.

[3] S. Arora, F.T. Leighton, and B.M. Maggs. On-line algorithms for path selection in a nonblocking network. Technical report, Mathematics Dept. and Laboratory for Computer Science, Massachusetts Institute of Technology, 1989.

[4] M. Bae and B. Bose. Resource placement in torus-based networks. *IEEE Trans. Computers*, 46(10):1083–1092, October 1997.

[5] P. Brezany, T. A Mueck, and E. Schikuta. A software architecture for massively parallel input-output. In *Proc. 3rd International Workshop PARA'96, LNCS Springer Verlag*, 1996.

[6] P. Brezany, T. A Mueck, and E. Schikuta. A software architecture for massively parallel input-output. In *Proc. 3rd International Workshop PARA'96, LNCS Springer Verlag*, 1996.

[7] Y. Cho, M. Winslett, S.-W. Kuo, Y. Chen, J. Lee, and K. Motukuri. Parallel i/o on networks of workstations: Performance improvement by careful placement of i/o servers.

[8] Y. Cho, M. Winslett, M. Subramaniam, Y. Chen, S. W. Kuo, and K. E. Seamons. Exploiting local data in parallel array i/o on a practical network of workstations. In *Proc. fifth Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, pages 1–13, 1997.

[9] Y.E. Cho, M. Winslett, S.-W. Kuo, J. Lee, and Y. Chen. Parallel i/o for scientific applications on heterogeneous clusters: a resource-utilization approach. In *Proc. the International Conference on Supercomputing*, 1999.

[10] A. Dan and D. Sitaram. An on-line video placement policy based on bandwidth to space ratio. In *ACM SIGMOD International Conf. Management of Data*, pages 376–385, 1995.

[11] J. Dukes and J. Jones. Dynamic replication of content in the hammerhead multimedia server. Technical report, Department of Computer Science, Trinity College Dublin, Ireland, 2003.

[12] M. D. Schroeder et. al. Autonet: A high-speed, self-configuring local area network using point-to -point links. Technical Report SRC research report 59, DEC, April 1990.

[13] T. Fuerle, O. Jorns, E. Schikuta, and H. Wanek. Meta-VIPIOS: Harness distributed I/O resources with VIPIOS.

[14] M. Harry, J. Rosario, and A. Choudhary. VIP_FS: A virtual, parallel file system for high performance parallel and distributed computing. In *Proc. 9th International Parallel Processing Symposium*, April 1995.

[15] M. Harry, J. Rosario, and A. Choudhary. Vipfs: A virtual parallel file system for high performance parallel anddistributed computing. In *Proc. 9th International Parallel Processing Symposium*, 1995.

[16] J. Huber, C. L. Elford, D. A. Reed, A. A. Chien, and D. S. Blumenthal. Ppfs: A high performance portable parallel file system. In *Proc. 9th ACM International Conference on Supercomputing*, pages 485–394, 1995.

[17] D. Kotz and T. Cai. Exploring the use of I/O nodes for computation in a MIMD multiprocessor. In *Proc. third Workshop on I/O in Parallel and Distributed Systems*, pages 78–89, 1995.

[18] S. Kuo, M. Winslett, K.E. Seamons, Y. Chen, Y. Cho, and M. Subramaniam. Application experience with parallel input/output: Panda and the H3expresso Black Hole Simulation on the SP2. In *Proc. the 8th SIAM Conference on Parallel Processing for Scientific Computing*, March 1997.

[19] Jens Mache, Virginia Lo, and Sharad Garg. The impact of spatial layout of jobs on parallel i/o performance. In *IOPADS'99*, 1999.

[20] Jens Mache, Virginia Lo, and Sharad Garg. Job scheduling that minimizes network contention due to both communication and i/o. In *Parallel and Distributed Processing Symposium*, 2000.

[21] S. Moyer and V. Sunderam. Pious: A scalable parallel i/o system for distributed computing environments. Technical Report Computer Science Report CSTR-940302, Department of Math and Computer Science, Emory University, 1994.

[22] Nils Nieuwejaar. *Galley: A New Parallel File System for Scientific Workload*. PhD thesis, Dept. of Computer Science, Dartmouth College, 1996.

[23] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.

[24] P. Ramananthan and S. Chalasani. Resource placement with multiple adjacency constraints in k-ary n-cubes. *IEEE Trans. Parallel and Distributed Systems*, 6(5):511–519, May 1995.

[25] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective i/o in panda. In *Proc. of Supercomputing*, 1995.

[26] D. N. Serpanos, L. Georgiadis, and T. Bouloutas. MMPacking: A Load and Storage Balancing Algorithm for Distributed Multimedia Servers. *IEEE Trans. Circuits and Systems for Video Technology*, 8(1):13–17, 1998.

[27] F.-C. Shao and A. Y. Orug. Efficient nonblocking switching networks for interprocessor communications in multiprocessor systems. *IEEE Trans. Parallel and Distributed Systems*, 6(2):132–141, Feb. 1995.

[28] M. Subramaniam. High Performance Implementation of Server Directed I/O. Master's thesis, Dept. of Computer Science,University of Illinois, 1996.

[29] S.R. Subramany, B. Narahari, and R. Simha. Placement of storage nodes in a network. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, 1998.

[30] N. F. Tseng and G. L. Feng. Resource allocation in cube network systems based on the covering radius. *IEEE Trans. Parallel and Distributed Systems*, 7(4):323–342, April 1996.

[31] N. Venkatasubramanian and S. Ramanathan. Load management in distributed video servers. In *Inter. Conf. Distributed Computing Systems*, 1997.

[32] Y. Wang, J. Lin, D. Du, and J. Hsieh. Efficient video allocation for video-on-demand services. In *IEEE Multimedia Conference*, 1996.