Dynamically Adaptive Binomial Trees for Broadcasting in Heterogeneous Networks of Workstations

Silvia M. Figueira[1] and Christine Mendes[2]

[1] Department of Computer Engineering, Santa Clara University
Santa Clara, CA 95053-0566, USA, sfigueira@scu.edu

[2] Applied Signal Technology, 400 West California Ave.
Sunnyvale, CA 94086, USA, christine_mendes@appsig.com

Abstract. Binomial trees have been used extensively for broadcasting in clusters of workstations. In the case of heterogeneous nondedicated clusters and grid environments, the broadcasting occurs over a heterogeneous network, and the performance obtained by the broadcast algorithm will depend on the organization of the nodes onto the binomial tree. The organization of the nodes should take into account the network topology, i.e., the communication cost between each pair of nodes. Since the network traffic, and consequently the latency and bandwidth available between each pair of nodes, is constantly changing, it is important to update the binomial tree so that it always reflects the most current network traffic condition. This paper presents and compares strategies to dynamically adapt a binomial tree used for broadcasting to the ever-changing traffic condition of the network, including accommodating nodes joining the network at any time and nodes suddenly leaving.

1 Introduction

Broadcasting is an important communication strategy that is used in a variety of parallel and distributed linear algebra algorithms [7]. In fact, algorithms for broadcasting in different kinds of environments have been discussed extensively in the literature [1, 2, 3, 5, 6, 8, 9, 10, 11]. Different structures, such as binomial trees, hypercubes, and rings are used for broadcasting. Binomial trees are particularly useful for broadcasting in clusters [12] and grid environments, in which they are used in each local group of machines [8, 9]. When used in heterogeneous networks, the binomial tree needs to reflect the network topology by taking into account the communication cost between each pair of nodes [4]. It also needs to evolve according to the dynamic changes in the network traffic conditions. In particular, the binomial tree needs to dynamically adapt to changes in latency and bandwidth, to nodes going down, and to nodes wishing to join the tree. Dynamically adapting the binomial tree is particularly important for applications that execute for a long time, facing many changes in the traffic condition of the network.

In this paper, we propose strategies that can be used to update a topology-based binomial tree as communication costs increase between nodes. We also propose strategies to deal with nodes joining the network at any time and nodes suddenly leaving the system. While rebuilding the tree from scratch would provide the best organization of nodes on the tree, allowing the broadcast operation to be completed in the smallest amount of time, the overhead associated with it is significant. Dynamically adapting the tree consistently by changing the placement of nodes on the tree according to the changes in the network condition is far less expensive and still helps decrease the cost of broadcasting.

The strategies presented have not been implemented in connection with any specific application. They are general strategies to enable binomial trees to adapt to changes in the topology of the underlying network and can be implemented as part of any message passing library, such as MPICH or PVM.

This paper is organized as follows. Section 2 explain how to build a topology-based binomial tree. Section 3 presents strategies to update topology-based binomial trees dynamically according to changes in the communication costs. Section 4 deals with nodes entering and leaving the binomial tree. Section 5 presents a representative set of experiments. Section 6 summarizes the results obtained.

2 Topology-Based Binomial Trees

In [4], the author presents the Balanced-Path algorithm to provide a performance-efficient binomial tree. This algorithm traverses the tree, choosing children for the nodes and serving the nodes with the highest number of undefined children first. When choosing a child, the algorithm picks the node, from those that are left, that is closest to the parent. This strategy balances the paths from the root to the leaves by giving priority to the parents that have the largest number of children, since parents with more children lie along longer paths. In a tie, the algorithm gives priority to the longer path. At the end, the total distance in the paths is balanced, and the communication costs will be distributed roughly evenly. For *N* nodes, the Balanced-Path algorithm takes $O(N^2)$ steps to generate an improved binomial tree.

Table 1: Distances between the nodes

	0	1	2	3	4	5	6	7
0	0	2	2	0	3	3	0	0
1	2	0	0	2	5	5	2	2
2	2	0	0	2	5	5	2	2
3	0	2	2	0	3	3	0	0
4	3	5	5	3	0	0	3	3
5	3	5	5	3	0	0	3	3
6	0	2	2	0	3	3	0	0
7	0	2	2	0	3	3	0	0

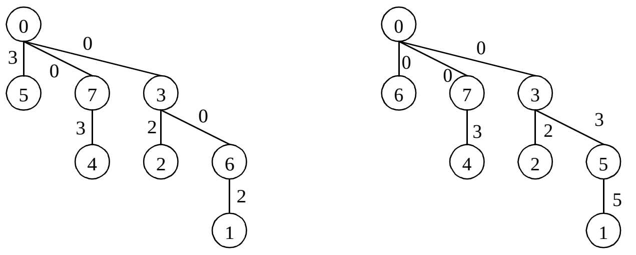


Fig. 1. Left: Balanced-Path tree formed according to the distances in Table 1.
Right: Paths affected if nodes 5 and 6 are swapped.

In Figure 1, the tree on the left was obtained using the Balanced-Path algorithm. The number on each edge represents the distance between the two nodes, or machines, connected via the edge. The distances were obtained from Table 1. Distances are specified in number of hops, and a distance of zero between two nodes means that the two nodes are directly connected.

3 Dynamically Adapting the Binomial Tree

After building the tree, it is necessary to evolve the tree according to the changes in the network conditions. The strategies used to evolve the tree need to be cost effective, otherwise it would be more efficient to rebuild the tree instead of simply updating it. Four strategies were developed to update the tree as the network conditions change. The cost of the tree is used as a measure to compare the different strategies. The cost is defined as the cost of the path with the highest cost, and the cost of each path is calculated as the sum of all the edges (i.e. distances) from the root to the respective leaf. This measure assumes that all paths are covered roughly in parallel, and it uses the costliest path for comparison, since this is the path that determines the time to execute a broadcast operation.

When the tree is first built, the cost of all the paths from each leaf to the root needs to be computed in order to determine the path with the highest cost. The cost obtained for each path in the tree shown in Figure 1 (left) is listed in Table 2. Since the highest cost is 3, the cost of the tree is 3.

Table 2: Cost of each path for the tree in Figure 1 (left).

Leaf	5	4	2	1
Cost	3	3	2	2

The network conditions may cause the communication cost of a link in the tree to increase and the four strategies described below may be used to swap nodes in the tree in order to obtain a configuration that uses communication links that are of lower costs. When we swap two nodes in the tree, the cost of the tree needs to be recomputed in order to determine if the swap reduces the cost of the tree. Rather than compute the cost of all the paths in the tree to determine the cost of the tree as we have to do initially, we only compute the cost of those paths affected by the swap. This is more efficient since previously computed values for many paths in the tree are still valid. For example, if we swap node 5 and node 6 in Figure 1 (left), then node 6 is now a leaf and the paths from the root to node 6 and from the root to node 1 have been affected by the swap. Figure 1 (right) shows the tree after the swap. It is evident that the paths leading to leaves 4 and 2 have not been impacted by the swap, and the costs for these two paths do not need to be re-computed. The cost of the affected paths to nodes 6 and 1 are re-computed as shown in Table 3, and we are able to determine that the new cost of the tree is 8.

Table 3: Cost of each path for the tree in Figure 1 (right).

Leaf	6	4	2	1
Cost	0	3	2	8

The cost or number of steps taken to re-compute the cost of the tree is $O(N)$ when the binomial tree has N nodes. The actual number of leaves whose cost needs to be recomputed depends on how high a node is located in the tree. The higher the node is in the tree, the greater the number of leaves affected by a swap with that node. On average, however, when nodes are swapped, many of the nodes affected will be located lower in the tree thus reducing the number of leaves whose cost needs to be re-computed.

Network conditions may change causing the link cost between two neighboring nodes in the tree to go down. Although at this point we do not take any action to modify the tree when link costs decrease, the cost of the binomial tree needs to be re-computed to reflect the change. Since a tree with *N* nodes will have *N*/2 leaves, the maximum number of leaves whose paths will have to be traversed to calculate their cost is *N*/2. The cost associated with updating the cost data structure for the tree therefore has an upper bound of *O*(*N*). In many cases, the cost for fewer than *N*/2 leaves will have to be recalculated.

The main goal of the proposed swapping strategies is to provide a tree with a cost that is at least as low as the cost of the original tree before the link cost between two nodes in the tree increased. Suppose that the cost (i.e., latency and/or bandwidth) between two nodes that are neighbors in the binomial tree increases, then each of the strategies will try to exchange one of these nodes with some other node in the tree in order to keep the cost of the tree at least as low as it was before the change in the network condition. As it tries each swap, the algorithms remember the lowest cost of the tree obtained so far, as well as the nodes whose swap produced that cost. This information is needed in case the algorithm cannot reduce the cost to what it was before the link cost increased. When the cost cannot be reduced to its original value, then the algorithms will perform the swap that leads to the lowest possible cost. On the other hand, if the algorithms are able to find a swap that results in a tree cost that is at least as low as what it was before the link cost increased, then the algorithms stop searching.

3.1 Family_Swapping Algorithm

This approach involves swapping one of the nodes connected by the link whose cost increased with another node in the family. If the link with an increased cost is the link that connects nodes a and b, where b is the child, node b will be the one to be swapped. In this case, the algorithm searches for a better position for node b by trying to swap it with its children, its parent (node a), and with its siblings, i.e., the other children of node a. Note that the algorithm does not swap node zero, which acts as the root of the tree. The upper bound for the number of steps in this algorithm is *O*(*logN*), where *N* is the number of nodes in the tree.

Figure 2 (left) illustrates the algorithm's behavior. The number on the left side of each node represents the order in which that node is checked to see if it can be swapped with the node of the affected link. Suppose the cost of the link between nodes 8 and 12 has been increased. Node 8 is the parent, and node 12 is the child. The Family_Swapping algorithm will try to first exchange node 12 with its children, nodes 13 and 14. Then it will try to exchange node 12 with node 8, its parent, and then it will try to exchange node 12 with its siblings, which are nodes 9 and 10.

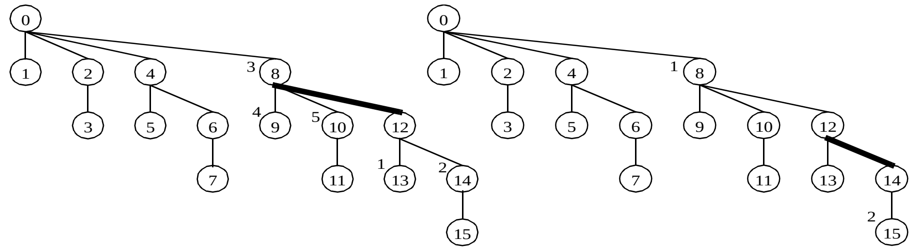


Fig. 2. Left: Using the Family_Swapping strategy.
Right: Using the Path_Swapping strategy.

In the example above, if swapping node 12 with its children does not reduce the cost of the tree, then the algorithm tries to swap node 12 (the child) with node 8 (the parent). Note that the link with the increased cost is still retained but now node 12 becomes the parent of node 8 and the child of node 0, and node 8 becomes the parent of nodes 13 and 14. This swap can lead to a lower tree cost if the communicating cost between node 12 and node 0 is less than that between nodes 8 and 0, or if the cost of communication between node 8 and its new children, nodes 13 and 14 are lower than that between node 12 and nodes 13 and 14.

The algorithm may not be able to lower the cost of the tree to a value at least as low as the cost before the link cost increased. In this case, the algorithm will swap node 12 with the child, parent, or sibling that results in the lowest possible cost. Each time the algorithm performs a swap, it has to re-compute the cost of the tree by examining the cost of each path to the leaves affected by the swap. The cost to perform each swap has $O(N)$, where N is the number of nodes in the tree since the number of leaves present in the tree is $N/2$. The swap between nodes closer to the root node will have more paths leading to leaves than the swap between nodes closer to the edge of the tree. In the tree shown in Figure 2 (left), if nodes 8 and 12 are swapped, then the cost to the leaves 9, 11, 13 and 15 need to be computed. However, had the link cost increased between nodes 12 and 14 and had the two nodes been swapped, then the leaves affected are only 13 and 15. Since the link cost randomly increases between nodes in different locations in the tree, in the average case fewer than $N/2$ leaves will be affected by a swap. Also, when a parent and a child node are swapped, the cost to compute the new cost of the binomial tree is less because all the leaves with paths leading to the child are also connected by a path to the parent. After swapping the parent and the child, the child will be in the parent's position and the parent will be in the child's position. The cost of only those leaves with paths to the child node need to be calculated because the leaves impacted by the parent node are a subset of the leaves impacted by the child node. However, when a node is swapped with its sibling there is no overlap in the leaves affected by the swap, so the cost of the leaves impacted by the new position of the node and its sibling has to be calculated independently, which is costlier.

3.2 Path_Swapping Algorithm

This approach involves swapping one of the two nodes connected by the link with an increased cost with another node in the same path. If the link with an increased cost is the link between nodes a and b, where a is the parent and b is the child, the algorithm alternates between trying to find a better position for node a by following its path to the root and trying to find a better position for node b by following its longest path down to a leaf. Note that the algorithm does not try to swap with the root. The upper bound for the number of steps in this algorithm is $O(logN)$, where N is the number of nodes in the tree.

Figure 2 (right) illustrates the algorithm's behavior. The number on the left side of each node represents the order in which that node will be checked. Suppose the cost of the link between nodes 12 and 14 has been increased. Node 12 is the parent, and node 14 is the child. The Path_Swapping algorithm will try to exchange node 12 with node 8, then it will try to exchange node 14 with node 15. Since the algorithm does not swap with node 0, it would stop trying to swap node 12 with any other node. Since node 14 does not have any children, other than 15, with which it can be swapped, the algorithm stops. If additional nodes were still available for swapping along the longest path from node 14 down to a leaf, then the algorithm would continue in that direction. Similarly, if additional nodes were available along the path up the tree leading to the root, the algorithm would continue trying to swap the parent node with one of those nodes. The algorithm stops as soon as a swap results in a tree whose cost is at least as low as the cost of the tree before the link cost increased. If the algorithm cannot reduce the cost to what it was before the change in the network condition, it performs a swap that leads to a tree with the lowest possible cost.

Since the Path_Swapping algorithm swaps nodes along the same path in the tree, it takes the least amount of steps to compute the cost of the tree on average compared to the other algorithms. Each time the algorithm performs a swap between two nodes, one of the nodes will be higher in the tree or closer to the root, and the other node will be located lower in the tree or closer to a leaf. The leaves affected by the new position of the node lower in the tree will be a subset of the leaves affected by the new position of the node higher in the tree. Hence, when the cost of the tree is re-computed, only calculating the cost of the paths to the leaves forming the larger set is sufficient. So, although the cost associated with re-computing the cost of the tree after a swap is still $O(N)$, where N is the number of nodes, the cost for only one set of leaves needs to be recalculated, even though two nodes have changed positions.

3.3 Leaf_Swapping Algorithm

This approach involves swapping one of the nodes connected by the link with an increased cost with one of the leaves in the tree. If the link with an increased cost is the link between nodes a and b, where a is the parent and b is the child, the algorithm alternates between looking for a better position for node a and looking for a better position for node b by trying to replace each of them by a leaf in the tree. The upper bound for the number of steps in this algorithm is $O(N)$, where N is the number of nodes in the tree.

Figure 3 (left) illustrates how the algorithm works. The number on the left hand side of each node represents the order in which that node will be checked. Suppose the cost of the link between nodes 8 and 12 increases. Node 8 is the parent of node 12. The Leaf_Swapping algorithm will try to exchange nodes 8 and 12 with node 1, the first leaf. Then, it will try to exchange nodes 8 and 12 with node 3, the second leaf. The algorithm will follow the leaves as shown in Figure 3 (left), trying to exchange each of them with nodes 8 and 12. The algorithm will stop when it finds a swap that leads to a tree with a cost at least as low as the cost of the tree before the link cost increased. If the cost cannot be reduced to what it was before the change in the network condition, then it will swap node 8 or 12 with a leaf that leads to the smallest cost possible.

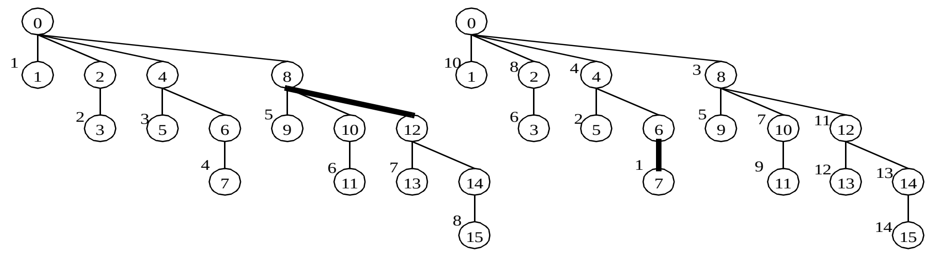


Fig. 3. Left: Using the Leaf_Swapping strategy.
Right: Using the Position_Swapping strategy.

The advantage of exchanging a node with a leaf node is that fewer links are affected by the exchange since the leaves are at the edge of the tree and do not have any children. However, since nodes closer to the root have paths leading to more leaves, swapping one of these nodes with a leaf takes $O(N)$ steps to recalculate the cost of the tree. In the worst case scenario, we have to re-compute the cost for $N/4$ leaves when we swap a child of the root node with a leaf. However, since most of the time we will be swapping nodes from different positions in the tree, the number of leaves affected by a swap will be on average less than $N/4$.

3.4 Position_Swapping Algorithm

This approach involves trying to swap one of the nodes connected by the link with an increased cost with a nearby node. If the link with an increased latency is the link between nodes a and b, where a is the parent and b is the child, node a will be the one to be swapped. In this case, the algorithm searches for a better position for node a by trying to swap it with nodes in nearby positions first. This is accomplished by choosing the nodes with increasing and decreasing positions in the tree. For example, if the node to be swapped is in position x, then the nodes will be checked according to their position in the following order: node in position x+1, x-1, x+2, x-2, and so on. Note that the algorithm does not swap the root node. The upper bound for the number of steps in this algorithm is $O(N)$, where N is the number of nodes in the tree.

Figure 3 (right) illustrates the algorithm's behavior. The number on the left side of each node represents the order in which that node will be checked. Suppose that the link between nodes 6 and 7 has been increased. Node 6 is the parent, and node 7 is the child. The Position_Swapping algorithm will try to exchange node 6 with nodes 7 and 5 (distance 1 from position 6). Then it will try to exchange node 6 with nodes 8 and 4 (distance 2 from 6). Then it will continue increasing the distance and checking the nodes as shown in Figure 3 (right). Note that the node in each position x may or may not be node x, since the nodes were rearranged to reflect the network topology. In this example, there are fewer nodes available for swapping whose position is less than that of position 6. When the algorithm exhausts the nodes available for swapping on one side of the tree, it will continue trying to swap with nodes on the other side of the tree, in this case, with nodes whose positions are greater than that of node 6.

The algorithm will try to swap until it finds a swap that results in a tree whose cost is at least as low as the cost of the tree before the latency was increased or until it runs out of nodes with which it can swap the parent. If the algorithm runs out of nodes without finding a swap that reduces the cost of the tree to at least what it was before the network change, then it will swap with the node that leads to the tree with the smallest possible cost. The Position_Swapping algorithm may run for a longer time than the Family_Swapping, Leaf_Swapping, or Path_Swapping algorithm, because it can potentially attempt a swap with all the nodes in the tree until it realizes that the cost of the tree cannot be reduced to its original value. However, because it has a greater number of potential candidates for a swap it is more successful in reducing the cost of the tree than the other three algorithms.

The number of steps taken to compute the cost of the tree after each swap has an upper bound of $O(N)$, where N is the number of nodes in the tree and the maximum number of leaves in the tree is $N/2$. This algorithm is the most comprehensive, and it generally does more work to compute the cost of the tree because the nodes used to swap could come from any position in the tree and may not lie in the same path.

4 Nodes Entering and Leaving the Binomial Tree

4.1 Nodes Entering the Tree

When a new node needs to be added, we need to add a new position to the binomial tree. This new position will be the next position in the binomial order. For example, in Figure 4 (left) we have an 8-node binomial tree composed of nodes 0 through 7. We add node 8 to the tree by creating position 8, which is a new sub-tree, with just one node, under node 0. The sub-tree rooted at node 8 is incomplete and can accommodate 7 more nodes.

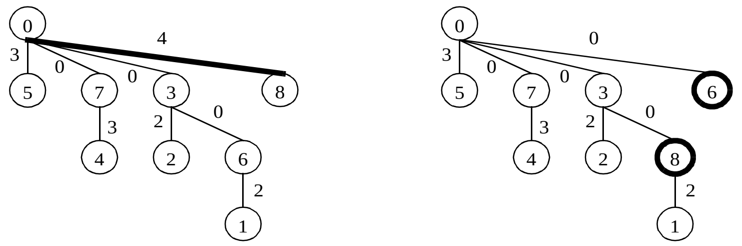


Fig. 4. Left: Adding a ninth node to a complete 8-node tree.
Right: Result of running Position_Swapping algorithm on the tree.

Once a new node is added to the tree, the cost of the tree may be affected if the communication cost between the newly added node and its parent is expensive. In fact, even if adding the new node did not impact the tree's cost, swapping this node with another node in the tree may help reduce the communication cost of the tree. In order to determine if there is a better position for the new node, we first re-compute the cost of the tree. Using the example in Figure 4 (left), since we added a new branch to the tree and since node 8 is a leaf, we update our cost data structure by adding the cost of the path from the root node to the leaf node 8 as shown in Table 4. We compute the cost by obtaining the latency from the latency matrix shown in Table 5, which reflects the distance in number of hops between node 8 and the other nodes. The latency matrix is updated with values each time a new node needs to be inserted into the tree.

Table 4: Updated costs using the latencies from Table 5.

Leaf	5	4	2	1	8
Cost	3	3	2	2	4

Table 5: Distance between the nodes (in number of hops) after adding node 8.

	0	1	2	3	4	5	6	7	8
0	0	2	2	0	3	3	0	0	4
1	2	0	0	2	5	5	2	2	2
2	2	0	0	2	5	5	2	2	5
3	0	2	2	0	3	3	0	0	0
4	3	5	5	3	0	0	3	3	0
5	3	5	5	3	0	0	3	3	4
6	0	2	2	0	3	3	0	0	4
7	0	2	2	0	3	3	0	0	3
8	4	2	5	0	0	4	4	3	0

After computing the cost of the tree, we then try to find a better position for node 8. A better position for node 8 would be one that reduces the cost of the binomial by making it equal or less than the cost of the tree before the new node was added. We do so by using the Position_Swapping algorithm, that will swap node 8 with nodes in different positions located around position 8. If we cannot reduce the cost of the tree to what it was before the node was added to the tree, the algorithm swaps node 8 with a node that produces the minimal cost possible.

In the example presented in Figure 4 (left), the cost of the tree before adding node 8 was 3. After adding node 8, the cost of the tree increases to 4 since the communication cost between node 0 and node 8 is 4, the highest cost path in the tree. The Position_Swapping algorithm tries to improve the cost of the tree by swapping node 8 with node 1. The algorithm works by taking the position of node 8, which is 8 and each time it determines which node to swap with by subtracting one and adding one to the position of node 8. Since there are no nodes in positions greater than 8, the algorithm can only look at positions whose values are less than 8. A swap with the node in position 7, node 1 is first attempted. If the swap does not decrease the cost of the tree, the algorithm subtracts 2 from 8 to get position 6 and attempts a swap with the node in position 6 and so on. Given the latency table in Table 5, we can reduce the cost of the tree from 4 to 3 by swapping node 8 with node 6 in position 6. The new tree formed is shown in Figure 4 (right). The algorithm stops at this point since it has performed a swap that reduced the cost of the tree to what it was before node 8 was added.

After running the Position_Swapping algorithm on the tree, the cost data structure is updated to reflect the cost of the path from node 0 to node 6, which is now a leaf node and the cost of the path from node 0 to node 1. The cost of the paths unaffected by the swap is not re-computed. Table 6 shows the cost of all the paths in the tree after the swap.

Table 6: Updated costs after running Position_Swapping algorithm.

Leaf	5	4	2	1	6
Cost	3	3	2	2	0

4.2 Nodes Leaving the Tree

Machines attached to a network sometimes go down, and machines that are part of a binomial tree structure are no exception. If a machine that is a node in the tree is no longer running, all of its children and their children in turn are disconnected from the tree. Even if the node that goes down does not have any children, the tree is still impacted. In order to keep the structure of the tree consistent with that of a binomial tree and in order to keep all nodes connected to the tree, we replace a node that fails with one that is functioning. The replacement node is always the one at the last position in the tree. In the special case when the last node in the tree goes down, there is no need to find another node to replace it.

To illustrate, suppose node 7, a child of node 0, goes down as shown in the tree in Figure 5 (left). As a result of node 7 going down, node 4 is now no longer connected to the tree. Any information broadcast by node 0 will not reach node 4. We fill in the empty position, caused by node 7’s failing, with node 1, the node in the last position in the tree. We pick the last node in the tree, because by doing so we avoid creating additional holes in the tree and we preserve the binomial tree structure in all the paths except for the last one. After replacing node 7 with node 1, the binomial tree will now have the structure shown in Figure 5 (right). It is clear that node 6 loses its child, and the tree is no longer complete. However, by replacing node 7, the tree remains connected and any broadcast operation will reach all the functional nodes.

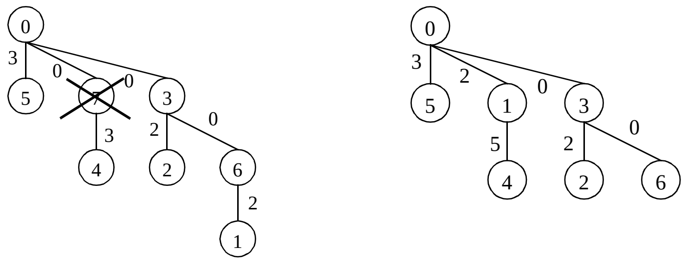


Fig. 5. Left: Binomial tree after node 7 goes down.
Right: Binomial tree after node 7 is replaced by node 1.

Since the position of two nodes in the tree was altered, the cost of some paths in the tree and, consequently, the cost of the tree need to be updated. In this example, replacing node 7 with node 1, increases the cost of the tree from 3 to 7, since node 1 is not close to either node 0 or node 4. The cost data structure is updated to reflect the new costs for all the paths in the tree as shown in Table 7. Since node 6 lost its child, the cost of the path from node 0 to node 6 is now 0.

Table 7: Updated costs after replacing node 7 with node 1.

Leaf	5	4	2	6
Cost	3	7	2	0

After updating the cost, we try to reduce the cost of the tree by using the Path_Swapping algorithm or the Position_Swapping algorithm. If the Path_Swapping algorithm is used in this example, we are unable to improve the cost of the tree by swapping node 1 with a node along its path. We do not swap node 1 with node 0, since node 0 is the root of the tree and the only other available option is performing a swap with node 4 as illustrated in Figure 6 (left). The expensive link between nodes 1 and 4 is still in the tree and the communication cost between node 0 and 4 is 3 hops, increasing the cost of the tree even further to 8 hops. In this scenario, the Path_Swapping algorithm was unable to reduce the cost of the tree.

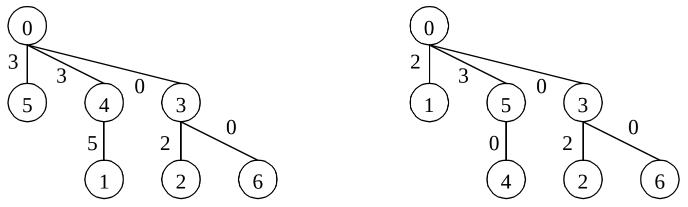


Fig. 6. Left: Binomial tree after swapping nodes 1 and 4 using Path_Swapping algorithm.
Right: Binomial tree after swapping nodes 1 and 5 using Position_Swapping algorithm.

The Position_Swapping algorithm can swap with a greater number of nodes since it is more flexible than the Path_Swapping algorithm. Hence, it has a better chance of improving the cost of the tree after a node that goes down is replaced with the last node in the tree. In the example above, the Position_Swapping algorithm determines that node 1 needs to be swapped with another node in the tree in order to have a tree with a reduced cost. Node 1 is at position 2 in the tree and the algorithm tries to swap node 1 with the nodes that are one position away from it. The algorithm swaps node 1 with node 4, which is at position 3. As demonstrated above, swapping nodes 1 and 4 does not reduce the tree's cost. So the Position_Swapping algorithm tries swapping node 1 with node 5, which is at position 1. This swap does succeed in reducing the cost of the tree down to 3, its cost before node 7 was replaced by node 1. Figure 6 (right) shows the tree after swapping nodes 1 and 5.

The improved cost of the tree is reflected in the cost data structure that maintains the cost of the paths from the root to each leaf. Since a node went down and was replaced with a node from the edge of the tree, there is one less node in the tree and the last leaf of the tree has also changed as Table 8 shows.

Table 8: Updated costs after running Position_Swapping algorithm

Leaf	1	4	2	6
Cost	2	3	2	0

5 Experiments

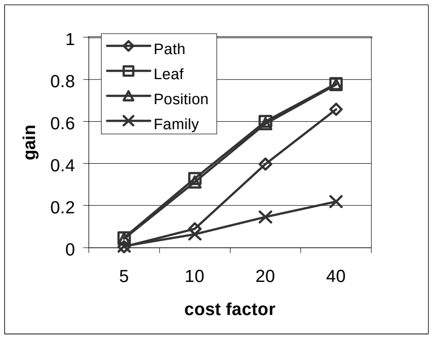
5.1 Comparison of Swapping Algorithms

We compare the efficiency of the four strategies proposed to adapt the binomial tree by simulating them on randomly generated networks. We have executed experiments on networks with different number of nodes, and we show below a representative subset of these experiments. In each set of experiments, we vary the cost factor, which is the amount by which a link cost is increased. For each cost factor, we report the average gain, the average number of steps to achieve the gain, and the average benefit obtained for 1000 different randomly-generated topologies, determined by the distance between each pair of nodes. For each topology, a randomly chosen link is increased by the cost factor specified.

Note that each algorithm stops when it finds a swap leading to a tree with a cost that is less than or equal to the cost of the original tree. Therefore, the average number of steps reported indicates how long it takes the respective algorithm to reach that level of optimization. The average gain reported is calculated as (|increased_cost – new_cost| / increased_cost), where increased_cost is the cost of the tree after the link had its cost increased, and new_cost is the cost of the tree updated by the respective algorithm. The average benefit is given by the ratio (average_gain / average_steps), which will help determine which approach is more cost-effective.

It is important to note that the gain may be as low as zero, if no swapping leads to an improved tree, i.e., a tree with a cost that is equal to or lower than the cost of the original tree. In this case, the number of steps will be the maximum allowed by the algorithm, and the gain will be the lowest one obtained during the search.

Figures 7 and 8 show the results obtained for 1024 nodes, which are apart by at most 10 hops. This is a large network, in which the nodes are close together. The graphs show that the Family_Swapping algorithm provides the smallest gain, but the Leaf_Swapping and Position_Swapping algorithms take more steps. The success of the four algorithms depends on the ratio of benefit to the cost, and the Path_Swapping algorithm has the best results when the gain and steps are examined together.



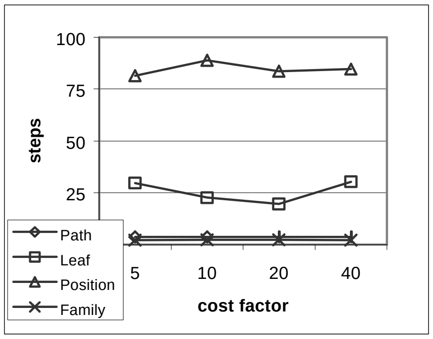


Fig. 7. Sets of 1024 nodes, maximum distance 10 hops.

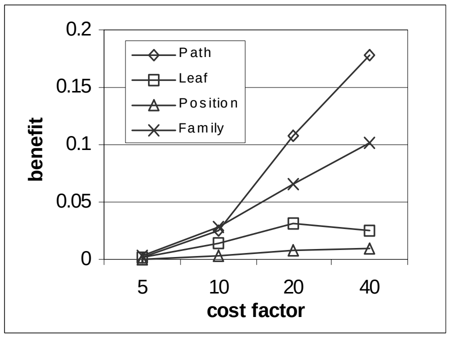


Fig. 8. Sets of 1024 nodes, maximum distance 10 hops.

These results lead to the conclusion that the Path_Swapping algorithm is the best approach. Even though the gain obtained by this approach may not be the largest, the number of steps is always low and the benefit to cost ratio obtained is always the largest. In addition, the cost associated with computing the new cost of the tree after performing a swap with the Path_Swapping algorithm is the lowest since the leaves affected by the swapped node located lower in the tree is a subset of the leaves affected by the swapped node higher in the tree or closer to the root. Therefore, when two nodes are swapped with the Path_Swapping algorithm, the cost of the tree can be computed by calculating the cost of the leaves linked by a path to the swapped node located closer to the root. It is important to note that the gain grows proportionately to the cost factor, and can be as high as 80% when the cost factor is 40. This shows the importance of keeping the binomial tree updated, especially because the overhead for the update with the Path_Swapping algorithm is quite low.

5.2 Analysis of Nodes Entering and Leaving Tree

A set of experiments were conducted to explore the effects of dynamically adding and removing nodes from a binomial tree in order to simulate nodes entering the cluster and nodes going down. The experiment presented was conducted on 100 trees, starting with 1024 nodes. The initial maximum distance for each set of 100 trees was set to 10, 30 and 50 hops. 1000 nodes were then randomly added to the tree or removed from the tree. We first obtain the average cost of the 100 trees after adding and removing the nodes without using any algorithm to improve the tree's cost after each addition or deletion operation. Then we perform the experiment of adding and removing nodes but we use the Position_Swapping algorithm to improve the tree's cost after a node is added and use the Path_Swapping algorithm to improve the tree when a node is deleted. The same experiment is run a third time for each set of trees but this time the Position_Swapping algorithm is used to improve the tree after adding a node and the Position_Swapping algorithm is again used to reduce the tree's cost after removing a node.

Figure 9 illustrates the outcome of adding and deleting 1000 nodes randomly from a tree with 1024 nodes. The Position/Path combination and the Position/Position combination of algorithms are both able to cut the cost of the tree to nearly half of what it is when no algorithms are used. Further more, the Position/Position combination is able to reduce the cost of the tree even more than the Position/Path combination. This is because the Position_Swapping algorithm is able to attempt a swap with each node in the tree and is not restricted to nodes along the same path as the Path_Swapping algorithm is. As a result, it is able to improve the cost of the tree in more cases than the Path_Swapping algorithm.

The results of these sets of experiments indicate that, when nodes are entering and leaving the tree, it is most beneficial to use the Position_Swapping algorithm to reduce the cost of the tree, if it is not important to perform the addition and deletion of nodes in the minimal number of steps. However, if our goal is to perform the operations as quickly as possible, then the Position_Swapping algorithm can be used when a node is added and the Path_Swapping algorithm can be used to improve the tree when a node is deleted since together the two algorithms take fewer steps and still provide performance-efficient trees.

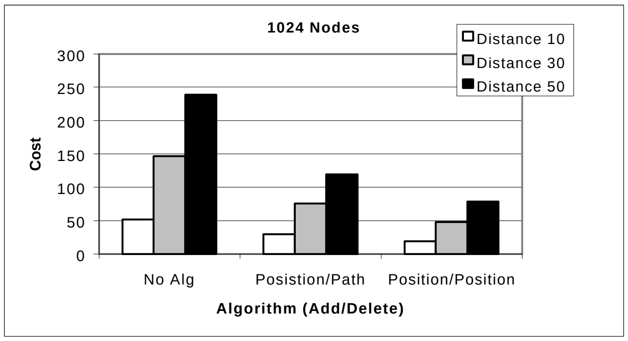


Fig. 9. 1024 nodes, 100 trees, 1000 nodes added or deleted

6 Conclusion

The four strategies, Path_Swapping, Position_Swapping, Family_Swapping and Leaf_Swapping, described above provide ways to update the binomial tree in response to the changing conditions of the network. Both the gains obtained and the number of steps used to reach an improved tree have been compared in order to determine the best approach. The efficiency of the strategies proposed is evaluated by comparing the benefit, i.e., the ratio of gain to steps, achieved by each approach. The experiments performed indicate that it is worthwhile to update the tree since the overhead involved in the operation is outweighed by the gain obtained. The experiments have also shown that the Path_Swapping algorithm achieves the best results by providing improved trees in a short number of steps.

Experiments were also conducted to show how the tree can be dynamically modified to handle new nodes wishing to join the tree or nodes suddenly leaving. Rather than rebuilding the tree from scratch to handle the entry or exit of nodes, we have implemented algorithms to allow nodes to enter or to be removed from the tree at any time. Running the Position_Swapping algorithm when a node is added to the tree, and the Path_Swapping or Position_Swapping algorithm when a node is removed from the tree, leads to a tree with a cost that is much lower compared to the cost of one in which there is no attempt to reduce costs.

In summary, the strategies presented enable the dynamic nature of the network to be accommodated in a structure that promotes performance-efficient communication.

References

1. M. Banikazemi, V. Moorthy, and D. K. Panda, “Efficient Collective Communication on Heterogeneous Networks of Workstations,” in Proceedings of the International Conference on Parallel Processing, August 1998.
2. M. Banikazemi, et al., “Communication Modeling of Heterogeneous Networks of Workstations for Performance Characterization of Collective Operations,” in Proceedings of the Heterogeneous Computing Workshop, April 1999.
3. M. Bernaschi and G. Iannello, “Collective Communication Operations: Experimental Results vs. Theory,” Concurrency: Practice and Experience, vol. 10, no. 5, pp. 359-386, 1998.
4. S. M. Figueira, “Improving Binomial Trees for Broadcasting in Local Networks of Workstations,” VECPAR’02, Porto, Portugal, June 2002.
5. I. Foster, et al., “Wide-Area Implementation of the Message Passing Interface,” *Parallel Computing*, vol. 24, no. 12, pp. 1735-1749, 1998.
6. I. Foster and N. Karonis, “A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems,” in *Electronic Proceedings of the IEEE/ACM Supercomputing Conference*, November 1998.
7. D. Gannon and J. Van Rosendale, “On the Impact of Communication Complexity in the Design of Parallel Numerical Algorithms,” IEEE Transactions on Computers, vol. C-33, pp. 1180-1194, December 1984.
8. N. Karonis, et al., “Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance,” proceedings of the 14th International Parallel Distributed Processing Symposium (IPDPS’00), pp. 377-384, May 2000.
9. T. Kielmann, H. E. Bal, and S. Gorlatch, “Bandwidth-Efficient Collective Communication for Clustered Wide Area Systems,” in Proceedings of the International Parallel and Distributed Processing Symposium, May 2000.
10. T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, “MagPIe: MPI’s Collective Communication Operations for Clustered Wide Area Systems,” in Proceedings of the Symposium on Principles and Practice of Parallel Programming, May 1999.
11. B. B. Lowekamp and A. Beguelin, “ECO: Efficient Collective Operations for Communication on Heterogeneous Networks,” in Proceedings of the 10th International Parallel Processing Symposium, April 1996.
12. C. Martin and O. Richard, “Parallel Launcher for Cluster of PC,” in ParCo 2001.