An Optimal Scheduling Algorithm for an Agent-Based Multicast Strategy on Irregular Networks

Pangfeng Liu[1] Yi-Fang Lin[1,2] Jan-Jan Wu[2] Zhe-Hao Kang[1]
Department of Computer Science and Information Engineering,
National Taiwan University, Taipei, Taiwan.
pangfeng@csie.ntu.edu.tw[1]

Institute of Information Science, Academia Sinica, Taipei, Taiwan.
{wuj,ice}@iis.sinica.edu.tw[2]

Abstract

This paper describes an agent-based approach for scheduling multiple multicast on wormhole switch-based networks with irregular topologies. Multicast/broadcast is an important communication pattern, with applications in collective communication operations such as barrier synchronization and global combining. Our approach assigns an agent to each subtree of switches such that the agents can exchange information efficiently and independently. The entire multicast problem is then recursively solved with each agent sending message to those switches that it is responsible for. In this way, communication is localized by the assignment of agents to subtrees. This idea can be easily generalized to multiple multicast since the order of message passing among agents can be interleaved for different multicasts. The key to the performance of this agent-based approach is the message-passing scheduling between agents and the destination processors. We propose an optimal scheduling algorithm, called *ForwardInSwitch* to solve this problem. We conduct extensive experiments to demonstrate the efficiency of our approach by comparing our results with SPCCO, a highly efficient multicast algorithm reported in literature. We found that SPCCO suffers link contention when the number of simultaneous multiple multicast becomes large. On the other hand, our agent-based approach achieves better performance in large cases.

1 Introduction

In recent years, with the speed of microprocessors increasing and cost decreasing and the availability of high bandwidth, low latency switches (such as Fast Ethernet switches, Myrinet switches, ATM switches, and GigaBit Ethernet) at a reasonable cost, it is popular to interconnect workstations/PCs together with commodity switches. This makes clusters of workstations/PCs an appealing vehicle for cost-effective parallel computing.

To reduce communication latency and buffer requirement, wormhole switching technique [2, 14] is often used in these switches. Systems with wormhole routing provide a very small buffer space at each hop and divide a message into small flits that travel through the network in a pipeline fashion. The main drawback of wormhole switching is that blocked messages hold up the links, prohibiting other messages from using the occupied links and buffers.

Multicast/broadcast is commonly used in many scientific, industrial, and commercial applications. Distributed-memory parallel systems, such as cluster systems, require efficient implementations of multicast and broadcast operations in order to support various applications. In a multicast, the source node sends the same data to an arbitrary number of destination nodes. When multiple multicast operations occur at the same time, it is very likely that some messages may travel through the same network link at the same time and thus content with each other, if they are not scheduled properly.

Minimizing contention in multicast/broadcast has been extensively studied for systems with regular network topologies, such as mesh, tori and hypercubes [3, 4, 5, 9, 8, 10, 15]. Cluster networks, especially switch-based clusters, on the other hand, typically have irregular topologies to allow the construction of scalable systems with incremental expansion capability. These irregular topologies lack many of the attractive mathematical properties of the regular topologies. This makes routing on such systems quite complicated. In the past few years, several routing algorithms have been proposed in the literature for irregular networks [1, 6, 11, 16]. These routing algorithms are quite complex and thus make implementation of contention-free multicast operations very difficult.

The goal of this paper is to develop efficient (multiple) multicast algorithms for irregular switch-based networks. In [7], Fan and King proposed an unicast-based implementation of single multicast operation based on *Eulerian trail* routing. In this paper, we consider the widely used, commercially available routing strategy called "up-down" routing.

The best known results on multicast on irregular networks are the *Partial-Order-Chain*-based algorithms proposed by Kesavan and Panda [12]. The basic idea is to order the destination processors into a sequence, then apply a binomial tree-based multicast [13] on these destinations. The chain concatenation ordering (CCO) algorithm first constructs as many partial order chains (POC) as possible from the network. A partial order chain is a sequence of destinations such that we can apply a binomial multicast on it without any contention. The CCO algorithm then concatenates these POCs into sequence where a binomial multicast is performed [12]. The sequence consists of fragments of processor sequences in which messages within the same fragment can be sent independently, therefore congestion is reduced. Based on the CCO algorithm, the source-partitioned CCO (called SPCCO) performs multiple multicasts simultaneously. Each multicast produces its own sequence (consisting of POCs), and each resulting sequence is shifted until the source appears at the beginning of the sequence. By shifting these sequence, the communication is "interleaved" according to the source, and communication hot-spots are avoided. However, both CCO and SPCCO use the idea of POC to reduce contention. Within a single POC different messages do not interfere with one another as long as they are from different sections within a POC. However, this POC structure may not always be preserved since the later binomial multicast is not aware of it. To solve this problem, we propose an agent-based multicast algorithm, which avoids network contention by localizing and interleaving message passings in multicast.

For a single multicast, our algorithm uses a recursive construct to localize communication. We then generalize it to multiple multicast by interleaving the communication tasks among different subnetworks. Figure 1 compares the numbers of contented links per multicast from SPCCO and our recursive agent-based multicast (RAM) under different number of simultaneous multicasts. The system consists of sixteen switches. Each switch has 16 ports, with half of which connected to processors, and the other half connected to other switches. Each multicast has 101 and 408 destinations on the left and the right side of Figure 1 respectively. As indicated in Figure 1, our multicast algorithm incur less contention. More experimental data will be presented in Section 6.

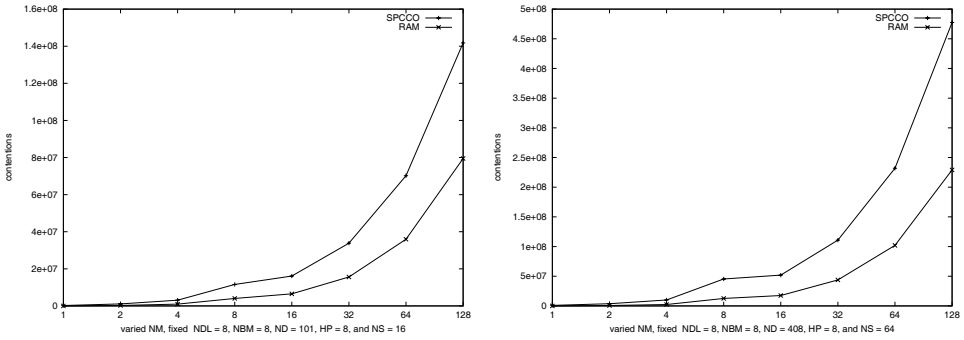

Figure 1: Number of contented links under different numbers of multicasts.

Our agent-based approach starts with a recursive multicast algorithm. An agent for a multicast

is chosen for each subtree of the routing tree. An agent is responsible for relaying (forwarding) the multicast messages to all the destinations in that subtree. This task is divided into subtasks for each subtree, where they are performed recursively. We generalize this algorithm to multiple multicast by choosing a *primary agent* for each multicast. The primary agent are chosen from the subtrees of the root of the routing tree, and are properly interleaved so that the tasks are distributed evenly. The primary agents for different multicasts exchange messages and then use the multicast algorithm to forward messages.

The key to the performance of the agent-based multicast strategy is the scheduling of message forwarding between agents as well as between an agent and the destination processors within each subtree. For this purpose, we develop an optimal scheduling algorithm, called *ForwardInSwitch*, for message forwarding. We provide theoretical analysis for the optimality and time complexity of *ForwardInSwitch*. Our experimental results also demonstrate significant performance improvement of our multicast algorithms in comparison with the CCO and SPCCO multicast algorithms.

The rest of the paper is organized as follows: Section 2 formally describes the communication model in this paper. Section 3 first describes our multicast algorithm, and then describes the generalization to multiple multicast. Section 4 describes the communication model behind our forwarding scheme. Section 5 presents the *ForwardInSwitch* optimal scheduling algorithm in details. Section 6 reports our experimental results. Finally, we conclude with Section 7.

2 Model

We assume that the system consists of switches and processors. Each switch has a set of ports, which can be used to connect to processors or ports of other switches. The connectivity of switches in the network can be represented by a graph $G = (V, E)$, where the set of nodes V represents switches, and the set of edges E represents the bidirectional connection channels among switches. The graph G can be highly irregular. In addition, each processor is connected to a unique switch. Figure 2 illustrates an irregular network consisting of 4 switches (each has 8 ports) and 15 processors, and the corresponding graph.

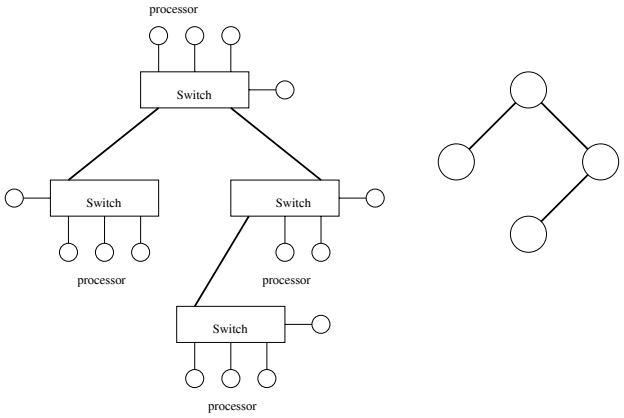

Figure 2: An irregular network of 4 switches and 15 processors and the corresponding connection graph.

The communication between two processors proceeds as follows. The source processor prepares a message and sends it to the switch it is connected to. Then dictated by the routing mechanism the message is routed in G from switch to switch. Finally the message is routed to the switch where the destination processor is connected to, and then delivered to the destination processor.

2.1 Routing Mechanism

We now describe the up-down routing [6] used in our multiple multicast algorithm. The up-down routing mechanism first uses a breadth-first search to build a spanning tree T for the switch connection graph $G = (V, E)$. Since T is a spanning tree of G, E is partitioned into two subsets – T and $E - T$. Those edges in T are referred to as *tree edges* and those in $E - T$ as *cross edges* [12]. Since the tree is built with a BFS, the cross edges can only connect switches whose levels in the T differ by at most 1. A tree edge going up the tree, or a cross edge going from a processor with a higher processor id to a processor with a lower one, are referred to as *up links*. The communication channels going the other direction are *down links*. In up-down routing a message must travel all the up links before it travels any down links. Due to the acyclic nature of how the direction of links are defined, the up-down routing is deadlock-free.

2.2 Contention

We assume that a switch can deliver multiple messages simultaneously from ports to ports, as long as the messages are delivered from different source and destination ports. This assumption is consistent with current routing hardware technology. As a result, congestion on the communication links becomes the major bottleneck. To avoid this problem, the routing method should fully utilize the communication capability, as suggested by the following cases.

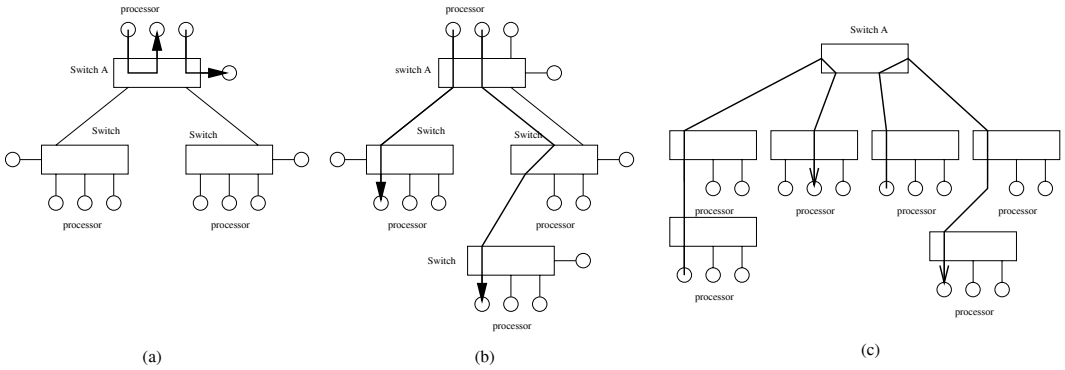

Figure 3: Example cases that avoid contention on the inter-switch channels.

We consider three cases where link contention can be avoided. We will focus on a particular switch A. In the first case, as shown in Figure 3(a), all source/destination processors are connected to the same switch A. In this case, there will be no contention since the messages travel through different paths within the switch. In the second case, as shown in Figure 3(b), both source processors reside on A. In this case, both can send messages to destinations in different subtrees of A simultaneously. Note that a destination node could be any processor in these two subtrees. In the third case two messages travel through four subtrees of switch A, as indicated in Figure 3 (c). If the two messages both go through switch A, there will be no link contention between them. Note that the source and destination processors may appear anywhere in the four subtrees.

3 Agent-Based Algorithms

This section describes our framework of a agent-based multiple multicast algorithm. We first introduce the algorithm for single multicast, then generalize the idea to multiple multicast. The algorithms specify how to perform a single/multiple multicast by determining the source and destination of all the intermediate communications, but the actual route from source to destination is determined by the underlying up-down routing strategy.

3.1 Single Multicast

For a given irregular network, we first construct a routing tree as in up-down routing [6]. The routing tree has all the switches as the tree nodes, and the inter-switch communication channels as the tree edges. Every tree node is the root of a unique subtree in this routing tree, and for ease of notation we will not distinguish a tree node (a switch in the network) from the subtree where it is the root.

For a given multicast message m and a switch v, we will define two functions – an agent function $A(m, v)$ that returns a processor within the subtree rooted at v and will be responsible for relaying multicast message m, and a cost function $C(m, v)$ that estimates the total cost of sending m to all of its specified destinations within the subtree rooted at v. Note that all these tree nodes here represent switches, not processors.

We define these agent and cost functions recursively. Let $D(m, v)$ be the set of destination processors of message m that are connected to switch v. For a leaf switch v, $A(m, v)$ is defined to be an arbitrary destination processor in $D(m, v)$, and $C(m, v)$ is $\log |D(m, v)|$. If $|D(m, v)| = 0$, that is, m does not have any destination processor connected to switch v, we define $A(m, v)$ to be an empty set and $C(m, v) = 0$. For an internal node (switch) v, if $|D(m, v)| > 0$, we pick an arbitrary destination of m in $D(m, v)$ to be the agent $A(m, v)$. Otherwise we consider all the children of v that m must be sent to, and set $A(m, v)$ to be the agent from these subtrees that has the highest cost. Formally, let $S(m, v)$ be the set of children of switch v that have destinations of m in their subtrees, then $A(m, v) = w$ such that $w \in S(v)$ and $C(m, w) \geq w'$ for all $w' \in S(v)$. Note that from this definition, the agent of a switch is not necessarily connected to the switch itself.

The cost function for an internal node is defined as follows: For the purpose of recursion we assume that the agent of switch v knows the message m. If $|D(m, v)| = 0$, the agents of tree nodes from $S(v)$ will first perform a multicast among themselves using a binomial multicast [13], then as soon as an agent a from $S(m, v)$ finishes receiving m, it recursively performs a multicast to all the destinations in the subtree where it is defined as the agent. The total communication cost is then defined as $C(m, v)$. When $|D(m, v) > 0|$, the situation is more complicated, because the agent of v, now a processor connected to v, can send m to other destinations in $D(m, v)$, or to the agents of $S(m, v)$. We apply a procedure *ForwardInSwitch* that determines the order for those in $D(m, v)$ and $S(m, v)$ to receive messages. After the schedule is determined, we compute the total cost $C(m, v)$ for v. The algorithm *ForwardInSwitch* takes $D(m, v)$ and $C(w, m)$ for all $w \in S(m, v)$ as inputs, then computes an optimal schedule and the total cost. The details of *ForwardInSwitch* will be given in Section 5. The pseudo code of our recursive agent-based multicast (RAM) is given in Figure 4.

```
RAM(v, m)
{
  if (|D(m, v)| == 0)
    A(m, v) sends message m to all agnets in S(m, v)
    by a binomial multicast;
  else
    call ForwardInSwitch to determine the order for A(v, m)
    to send messgaes to elements in D(m, v) and
    agents of S(m, v);
  For all switch s in S(m, v)
    call RAM(s, m);
}
```

Figure 4: The pseudo code of RAM, which recursively performs a multicast for each subtree of switch v that has destinations of message m.

When $|D(m, v)| > 0$, v does have some destination processors for message m and one of them is the agent of v. When the agent sends messages to those destinations in $D(m, v)$ (Figure 3 (a)), the messages will not interfere with each other. Also when the agent of v sends messages to those agents in

$S(m, v)$ (Figure 3 (b)), no contention is possible if no cross edges are involved. In addition, the message passing from one category (Figure 3 (a)) will not contend with those in the other category (Figure 3 (b)). When $|D(m, v)| = 0$, we use a single multicast to send the messages among all the agents of $S(m, v)$, with one of them now being assigned as the agent of v. From Figure 3 (c) we conclude that these messages will not contend with each other unless cross edges are involved, since the agents of different subtrees in $S(m, v)$ will not be in the same subtree.

To sum up, we expect a very low level of congestion from this scheduling algorithm. After guaranteeing low congestion, the algorithm *ForwardInSwitch*, which determines the optimal schedule of message-passing between agents as well as between an agent and the destinations within its subtree, computes the total cost. The complete details of the optimal *ForwardInSwitch* scheduling will be given in Section 4 and section 5.

3.2 Multiple Multicast

Let r be the root of the up-down routing tree. The agent-based multiple multicast is carried out in three steps as described below. First for each message m we choose a *primary agent* among the agents of $S(m, r)$ - the set of subtrees of root r. Each source processor then sends its message to its primary agent. Second, the primary agent sends its message m to a destination d in $D(m, r)$ if any, and to the agents of $S(m, v)$. Finally, each agent a of $S(m, r)$ sends messages to its destinations by calling RAM, and a sends m to $D(m, r)$ with a binomial multicast.

We consider several alternatives in the first two steps of our multiple multicast algorithm. First we consider two alternatives in choosing the primary agent. It is now clear that if different multicasts select different primary agents, we can "interleave" the traffic in the second step and achieve good performance. In fact this *interleaving* is very important for multiple multicasts to use the network resources without hot-spots. On the other hand, we do not want to place the primary agents away from the original multicast source very often, which may cause large traffic through the root of the routing tree. As a result, there is a tradeoff between good locality and interleaving. In our implementation we experimented two methods – we either choose the primary agent that is in the same subtree as the multicast source, or any agent of switches in $S(m, v)$ at random. These two approaches will be denoted as *SameTree* and *Random* respectively.

Secondly, we consider two alternatives in implementing the second step of multiple multicast. After the primary agent is chosen, it has to send the message to a processor in $D(m, r)$ and all the agents of switch in $S(m, r)$. This can be implemented in two different methods – the primary agent can either send m to all the others with a binomial multicast, or work with all the other primary agents to propagate information cyclicly. In the second approach, we arrange the chosen processor in $D(m, v)$ and all the primary agents as a ring. Each processor in the ring is responsible for relaying the information to the right side neighbor in the ring. Initially every primary agent places its message into this "circular track" and the message will be relayed to all the primary agents. We refer to these two approaches as *Binomial* and *Cyclic* respectively. Therefore, we have four multiple multicast algorithms as follows – *SameTree-Binomial*, *SameTree-Cyclic*, *Random-Binomial* and *Random-Cyclic*. These four algorithms will be denoted as STB, STC, RB and RC, and their performances will be reported in the Section 6.

4 Message Forward Model

This section describes our optimal forwarding mechanism (*ForwardInSwitch*) in details. Consider a switch v and a message m. Recall that $D(m, v)$ is the set of destination processors of m which connect to v. From previous discussion $|D(m, v)| > 0$. One of the processors in $D(m, v)$, denoted by $s = A(m, v)$, is the agent for switch v, and s has the message m. Also recall that $S(m, v)$ denotes the set of switches, which are children of v that contain destination processors of m in their subtrees. The procedure *ForwardInSwitch* should determine in what order, s (the agent of v), should send m to those in $D(m, v)$ and agents of $S(m, v)$. See Figure 5 for an illustration. For ease of notation, we will use the *source* to denote $s = A(m, v)$, *local nodes* for those processors in $D(m, v)$, and *remote nodes*

for agents of switches in $S(m, v)$. For example, in Figure 5 there are two local nodes and three remote nodes.

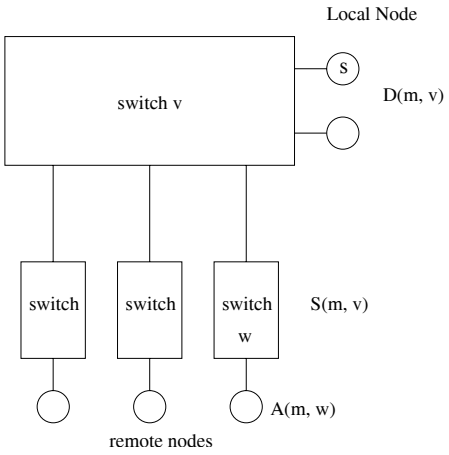

Figure 5: The switch with two local nodes and three remote nodes.

A remote node $g = A(w, m)$ is an agent of a switch $w \in S(m, v)$, and is responsible for sending m to all destinations in the subtree rooted at w. Recall that $C(m, w)$ denotes the total time for the agent of switch w to finish its task. We denote this multicast as the *task* for the agent g, so the *task cost* of g is $C(m, w)$. An agent cannot do its task until it receives m from other nodes. However, an agent g can choose to help other agents by forwarding m to them, if g thinks that its task can wait. Without lose of generality, we assume that once the agent g (the agent of w) starts its task, it will finish it in $C(m, w)$ time and will not help forward m during the process. As a result, all the forwarding by an agent will proceed its task.

Now we describe the forwarding mechanism among local and remote nodes. A local node can send m to remote nodes and other local nodes. A remote node, after receiving message m, can help forward m to other remote nodes. We first consider the case where the sender is a local node. Here we assume that a local node can finish sending m to another local node in one unit of time, since both of them are connected to the same switch. For example, if a local node s sends m to another local node r at time t, the communication will complete at time $t + 1$, and both s and r will be ready to send m. That is, we will get one more local node to help forward m right from time $t + 1$. See Figure 6 for an illustration.

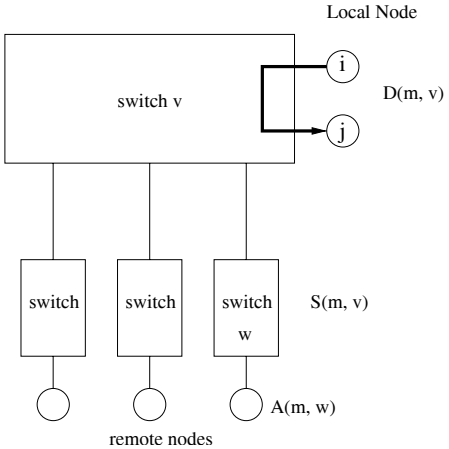

Figure 6: Local node s sends message to another local node r. The latency is 1.

If a local node s sends message m to a remote node g, we assume that there is a latency for the remote node to receive the data. We denote this latency by L_l, so that the remote node g will not complete receiving m until after L_l unit of time after i sends m. Here we use an integer value L_l to model the latency among different switches, and $L_l >> 1$ since it takes much longer time for a remote node to receive m than a local node does. We also assume that after one unit of time, the local node s is ready to send another message, since it does not need to wait for m to reach the remote destination g. See Figure 7 for an illustration.

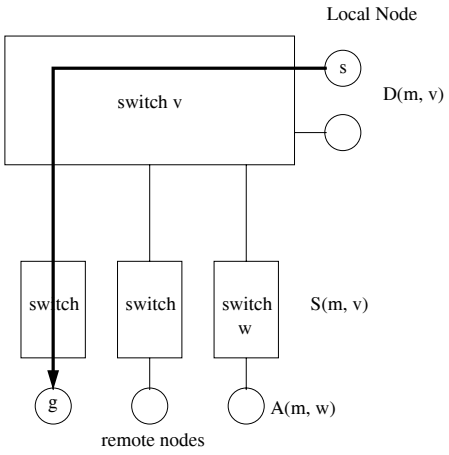

Figure 7: Local node s sends message to a remote node g. The latency is L_l.

If a remote node g helps forward message m to another remote node h, the latency is $L_r > L_l$, since the message must traverse through the root switch between them (Figure 8). Similarly, the source remote node g can start the next forwarding, should it decide to do so, after one unit of time since it doe not need to wait for y to receive m. As a result, if an agent g forwards the message n number of times to other remote nodes, it will be delayed by n time unites before it can work on its own task.

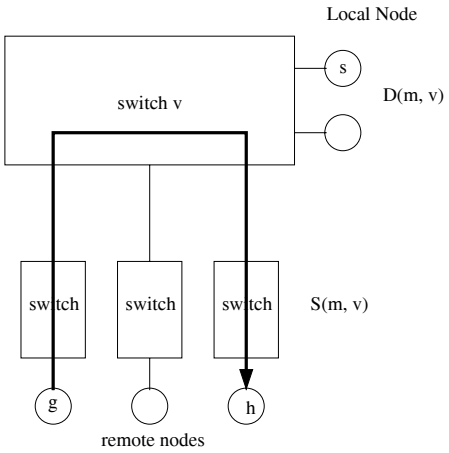

Figure 8: Remote node g forwards a message to another remote node h. The latency is L_r.

Consider a switch v. Initially we have a local node i that has message m, and the goal is to multicast m to all local nodes, and finish all tasks of all remote nodes. We denote this time as the *total multicast time*, and the problem is to, given L, $|D(v,m)|$, and all $C(w,m)$, where $w \in S(v,m)$, find a schedule that minimizes the total multicast time.

5 Forwarding Algorithm

To find the optimal total multicast time, we derive a testing algorithm to test if a given problem instance can finish within a given total multicast time T. We then perform a binary search on T to determine the optimal total multicast time. As a result the focus of the forwarding mechanism is to determine whether a given problem instance can finish in a given time T.

5.1 Criticality

Based on an expected total multicast time T, we will define *criticality* of a remote node g. There are two levels of criticality for a remote node g, depending on which kind of source node sending m to g. First, if a remote node starts sending m to g at time $T - L_r - C(g, m)$, g will be able to finish its task in time T. Second, if a local node starts sending m to g at time $T - L_l - C(g, m)$, g will also be able to finish its task in time T. As a result, we define that g is *r-critical* at time $d_r(g) = T - L_r - C(g, m)$, and *l-critical* at time $t = d_l(g) = T - L_l - C(g, m)$. Recall that $C(g, m)$ is the time for agent g to finish its task, L_l is the latency between a local and a remote node, and L_r is the latency between two remote nodes. For ease of notation we also define that a remote node g is *non-l-critical* at time t if $t < d_l(g)$, and *non-r-critical* at time t if $t < d_r(g)$.

Now for a remote node g we can divide the time T into three segments according to these two deadlines. When $t < d_r(g)$, we do not need to worry about g since it can wait. When $d_l(g) > t > d_r(g)$, g has already missed the first deadline for a remote node to send m to it, so its only hope to finish in time T is for a local node to send m to it. When $t > d_l(g)$ agent g misses both deadlines, the entire multicast will not finish in time T.

5.2 The Testing Algorithm

We now describe our testing algorithm which determines whether it is possible to finish the multicast within time T. At every time step, both local and remote node can send message, and we consider the first case first. A local node that has message m selects the destination according to the following priority.

1. l-critical remote node
2. local node
3. non-l-critical remote node, with the heaviest remote node first.

Similarly, an agent that has already received m chooses destination according to the following priority. If an agent cannot send messages in time for any other agent to complete its task, it simply starts its own task.

1. r-critical remote node
2. non-r-critical remote node, with the heaviest remote node first.

We first show that there exists an optimal schedule in which every local node tries to send message to local nodes before remote nodes, unless the remote nodes are l-critical.

Lemma 1 *There exists an optimal schedule in which the local nodes send message to remote node only when the remote nodes are l-critical, or there is no local nodes that have not yet received the message.*

Proof. If a remote node is l-critical, there is no way to match the deadline unless a local nodes starts sending message to it immediately. If there is no available local node, the total broadcast time T cannot be met since other remote nodes will not be able to send the message in time either. As a result we only need to show that there exists an optimal schedule in which local nodes send message to non-l-critical remote nodes only when there is no local node to send to.

We prove the lemma by showing that if the priority is violated, we can transform it into a new schedule that obeys the priority without increasing the total time. We assume that there is an optimal

schedule in which a non-l-critical remote node is scheduled at time t (to be sent by a local node p) and a local node is scheduled at time at time $t + k$ (to be sent by a local node q). In other words, the priority is violated. Since there is no distinction among these local nodes that have received the message (p and q), we may switch the destinations p and q send at time t. Therefore it suffices just to show that when a local node sends messages to remote nodes before local nodes, we can always switch the order so that the total time will not increase. In the following discussion, we assume that there is an optimal schedule in which a local node s sends a message to a non-l-critical remote node g_1 at time t, and at time $t + k$ sends a message to a local node r. In other words, the priority is violated. We also assume that g_1 is the *last* non-critical remote node that s send message to before the local node r, therefore s send messages to critical remote nodes $g_2, \ldots, g_k$ at time step $t + 1, \ldots, t + k - 1$.

Recall that agent g_1 starts it task at time $K(g)$, which means that between time $t + L_l$ and $K(g_1)$ agent g_1 helped forward $K(g_1) - t - L_l$ messages to other agents. We will use $n(g_1)$ to denote this number of other agents helped by g_1, and $h_1, \ldots h_{n(g)}$ are those agents.

Case $k = 1$: We first consider the ease case that $k = 1$, that is, there is no critical remote nodes sent between g_1 and r. In this case we simply switch the order of g_1 and r, so that s sends message to r and g_1 at t and $t + 1$ respectively. See Figure 9 for illustrations.

Since g_1 is not critical, its delay of one time step will not be fatal. In addition only h_1, we schedule r, who is ready to send messages at time $t + 1$, to send the message to h_1, and let g_1 to take care of the rest of $h_2, \ldots, h_{n(g_1)}$. r is able to send this additional message since it is moved earlier one time step. Also since g_1 is delayed by exactly one time step, it is able to send messages to $h_2, \ldots, h_{n(g_1)}$ as in the original schedule.

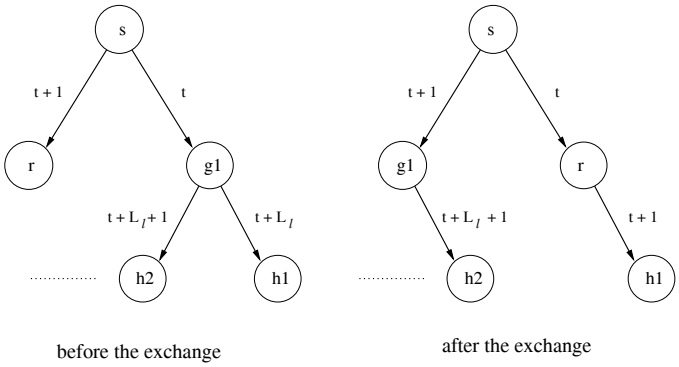

Figure 9: The schedule before and after the switch. The number next to the edge is the time when the communication starts.

Case $k > 1$: Now we consider the other case when k is greater than 1, that is, s did send messages to l-critical remote nodes between g_1 and r. Now we would like to change the schedule. The processor s will send a message to r (instead of g_1) at time t, and let r to send the message to g_1 at time $t + 1$. We first consider the effects on g_1 in two possibilities. The first case is when $n(g)$ is 0, i.e., g starts its work immediately when it complete receiving the message. Because g is not l-critical at time t, delaying g by one time step will not be fatal and g simply starts its task one time step later at time $t + L_l + 1$. g will be able to finish before T, by the definition of l-criticality.

Now consider the second case when g did help forwarding messages. In the new schedule if g still still wants to help forward messages, it will be able to do so except for h_1, since in the new schedule g_1 is delayed by one time step. See Figure 10 for illustrations. Fortunately in the new schedule r completes receiving data from s at time $t + 1$. Now r can send the message to h_1 (the first agent g helped forwarding to), at time $t + 2$. In the new schedule h_1 completes receiving message at time $t + 2 + L_l$, which is earlier than the completion time $t + L_l + L_r$ in the original schedule. As a result h_1 will not be delayed in the new schedule. Also in the new schedule g_1 is able to take care of the

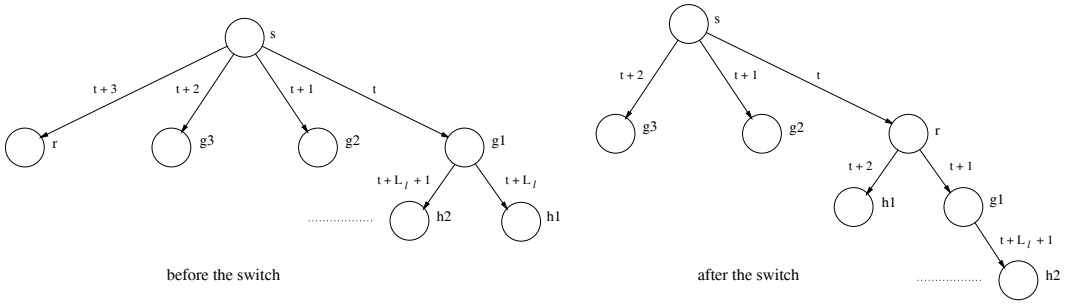

Figure 10: The schedule before and after the switch. The number next to the edge is the time when the communication starts.

remaining agents $(h_2, \ldots, h_{n(g)})$ it helped forwarding in the original schedule, because g_1 is delayed by exactly one time step and can still send messages to them at $t+L_l+1, \ldots, t+L_l+n(g)-1$ respectively.

Now consider the effects on r by the change of schedule. In the new schedule r completes receiving the message from s at time $t+1$, and sends its first message to g_1 at time $t+1$, and the second message to h_1 at time $t+2$. Recall that in the original schedule r starts sending messages at time $t+k+1$, so when $k \geq 2$ there will be no conflict in the schedule of r. ■

By Lemma 1 we conclude that there exists an optimal schedule in which local nodes sends message to local nodes before any non-l-critical remote nodes. Now we show that there exists an optimal schedule with the additional property that it will send messages to non-l-critical remote nodes according to their workloads.

Lemma 2 *There exists an optimal schedule in which the local nodes send messages to remote nodes according to their workloads, with the heaviest remote node being sent first.*

Proof. The proof is similar to the proof of Lemma 2. The only difference is that now both g and r are remote nodes, and $C(g,m) < C(r,m)$. Again if the remote node g did not help any other remote node, delaying it by one time step is not fatal since it is not l-critical at time t. If the remote node g did help other agents $h_1, h_2, \ldots h_{n(g)}$ in the old schedule, we schedule the remote node r to help the first h_1, so that g can take care of the rest $h_2, h_3, \ldots h_{n(g)}$. This is always feasible since in both schedules h_1 starts receiving message at time $t+L_l$ and no delay occurs. The remote nodes $h_2, \ldots h_{n(g)}$ will not be delayed either since g is delayed by exactly one time step. In addition, since $C(g,m) < C(r,m)$, the total completion time is not delayed. ■

After establishing the priority for local senders, now we argue that the remote nodes should follow the same priority. We show that there exists an optimal schedule that remote nodes will help forwarding messages to non-r-critical remote nodes according to their workloads.

Lemma 3 *There exists an optimal schedule in which the remote nodes send messages to remote nodes according to their workloads, with the heaviest remote node being sent first.*

Proof. The proof is similar to the proof of Lemma 4. The only difference is that now the sender s is now a remote node. If the remote node g did not help any other remote node, delaying it by one time step is not fatal since it is not r-critical at time t. If the remote node g did help other agents $h_1, h_2, \ldots h_{n(g)}$ in the old schedule, we schedule the remote node r to help the first h_1, so that g can take care of the rest $h_2, h_3, \ldots h_{n(g)}$. This is always feasible since in both schedules h_1 starts receiving at time $t+L_r$ and no delay occurs. The remote nodes $h_2, \ldots h_{n(g)}$ will not be delayed since g is delayed by exactly one time step. In addition since $C(g,m) < C(r,m)$, the total completion time is not delayed. ■

Finally, we would like to establish the priority *between* local and remote senders. We assume that the local nodes will pick destinations *before* the remote nodes do.

Lemma 4 *There exists an optimal schedule in which the local nodes send messages to remote nodes before the remote nodes send messages.*

Proof. We consider an optimal schedule in which the remote nodes pick destination before the local nodes. As a result we can find a time step t a remote node r picks a destination g_2, a local node node picks a destination g_1, and $C(m, g_2) > C(m, g_1)$.

We now swap the role of l and r, and let l sends to g_2 and r sends to g_1. If $K(g_1)$ is at least $t + L_r$, we break the set of destinations g_1 helped forwarding to into two subsets. One subset contains those destination g_1 help forwarding from time $t + L_r$ to $K(g_1)$. This subset can still be handled by g_1 since in the new schedule the sender of g_1 is a remote node r and g_1 will complete receiving at time $t + L_r$. The other subset contains those destination g_1 help forwarding from time $t + L_l$ to $t + L_r$. This subset can be handled by g_2 in the new schedule.

Now consider the second case when $K(g_1)$ is less than $t + L_r$. Now we schedule g_1 to start its own task immediately. This will not violate the total time bound since $C(m, g_1) < C(m, g_2)$, and old schedule worked fine. Whatever remains to be done from $t + L_l$ to $K(g_1)$ in the old schedule will be handled by g_2 in the new schedule, and those handled by g_2 will still be handled by g_2. None of these destination will be delayed by this switch so the total time will not increase. ■

Now we have the complete algorithm. The pseudo code of the scheduling (and testing) algorithm is in Figure 11.

```
test_finish_time(Time T)
{
  repeat the following steps until time step T.
  {
    1. Schedule all l-critical nodes to receive messages from local nodes.
      if the number of local nodes is not enough, declare failure.

    2. If there are source local processors left from step 1,
      schedule them to send data to other local nodes that have not
      yet received the message.

    3. If there are source local processors left from step 2,
      schedule them to send data to non-l-critical remote nodes, with
      the heaviest remote nodes being sent first.

    4. Schedule all r-critical nodes to receive messages from remote nodes.

    5. If there are source remote nodes left from step 4,
      schedule them to send data to other remote nodes, with the heaviest
      remote nodes being sent first.
  }
}
```

Figure 11: The pseudo code of the testing algorithm `test_finish_time` for determining the feasibility of a proposed finishing time T.

Theorem 1 *The algorithm test_finish_time correctly determines if a total forwarding time T can be achieved.*

Proof. A direct result from Lemma 1, Lemma 2, and Lemma 3. ■

5.3 Optimal Schedule

We can use apply the testing algorithm `test_finish_time` to find the optimal total multicast time. Since `test_finish_time` can determine the feasibility of a proposed total multicast time T, we use a binary search on T to determine the optimal T, and the optimal schedule. The pseudo code of the binary search is Figure 12.

```
Optimal_Schedule(the number of local/remote nodes and their costs)
{
  Set lowbound and upperbound.

  while upperbound != lowerboun
    if check_finish_time((upperbound + lowerbound) / 2)
      upperbound = (upperbound + lowerbound)/2
    else
      lowerbound = (upperbound + lowerbound)/2
    return lowerboud;
}
```

Figure 12: The pseudo code for finding the optimal schedule by using `test_finish_time`.

6 Simulation Experiments and Results

In this section, we present results of simulation experiments to compare the algorithms proposed in Section 3 and the two order-chain-based algorithms (CCO, SPCCO) proposed in prior works [12].

We use OMNET++ [17], a discrete event-based simulator for our experiments. The simulator can model wormhole routing switches with arbitrary network topologies. We chose system parameters as follows. Communication start-up time is 5.0 microseconds, link transmission time between processors and switches is 10.5 nanoseconds, and routing delay at switch is 200 nanoseconds. The default buffer size at each port is one flit. The default numbers of input ports and output ports are 16. The network topologies are generated randomly. For each data point, the multicast statistics is averaged over 30 different network topologies.

For all experiments, we assumed a default system configuration of a 512-processor system connected by 64 sixteen-port switches in an irregular topology. 50% of the ports on a switch are connected to processors, and the other 50% of the ports are connected to other switches.

For our study, we varied each of the following parameters one at a time: the message length (NBM), the number of destinations in each multicast (ND), and the number of simultaneous multicast operations (NM). we use *throughput*, which is defined by M/T, where M is the total length of the messages and T is the parallel completion time of the (multiple) multicast operation, to measure the performance.

In the following we compare our proposed agent-based algorithms *Random-Binomial Random-Cyclic*, *SameTree-Binomial*, and *SameTree-Cyclic* (`RB`, `RC`, `STB`, and `STC` for short) with two ordered-chain-based algorithms, `CCO` and `SPCCO`. The first part of the name of our algorithms refers to how to choose the primary agent, and the second part refers to how the primary agent sends mesages to all the agents. Please refer to Section 3 for details.

Effects of the Number of Multicast Operations First we examine the effects of variation in the number of multicast operations on the performance of the proposed algorithms. Other parameters are assumed to be as follows. The number of switches NS is from 32 to 64. Each switch has 8 ports that connect to processors, and the number of destinations in each multicast ND is from 16 to 80. The destinations are generated randomly.

Figure 13 indicates that when the number of multicast is less than four, ordered-chain-based algorithms perform better than our agent-based algorithms do. This is because when the number of multicast operations is small, message contention is not significant and thus the importance of reducing number of communication stages outweighs that of reducing message contention. However, when the number of multicast operations increases, the impact of message contention becomes more important and the benefit of agent-based optimization becomes very significant.

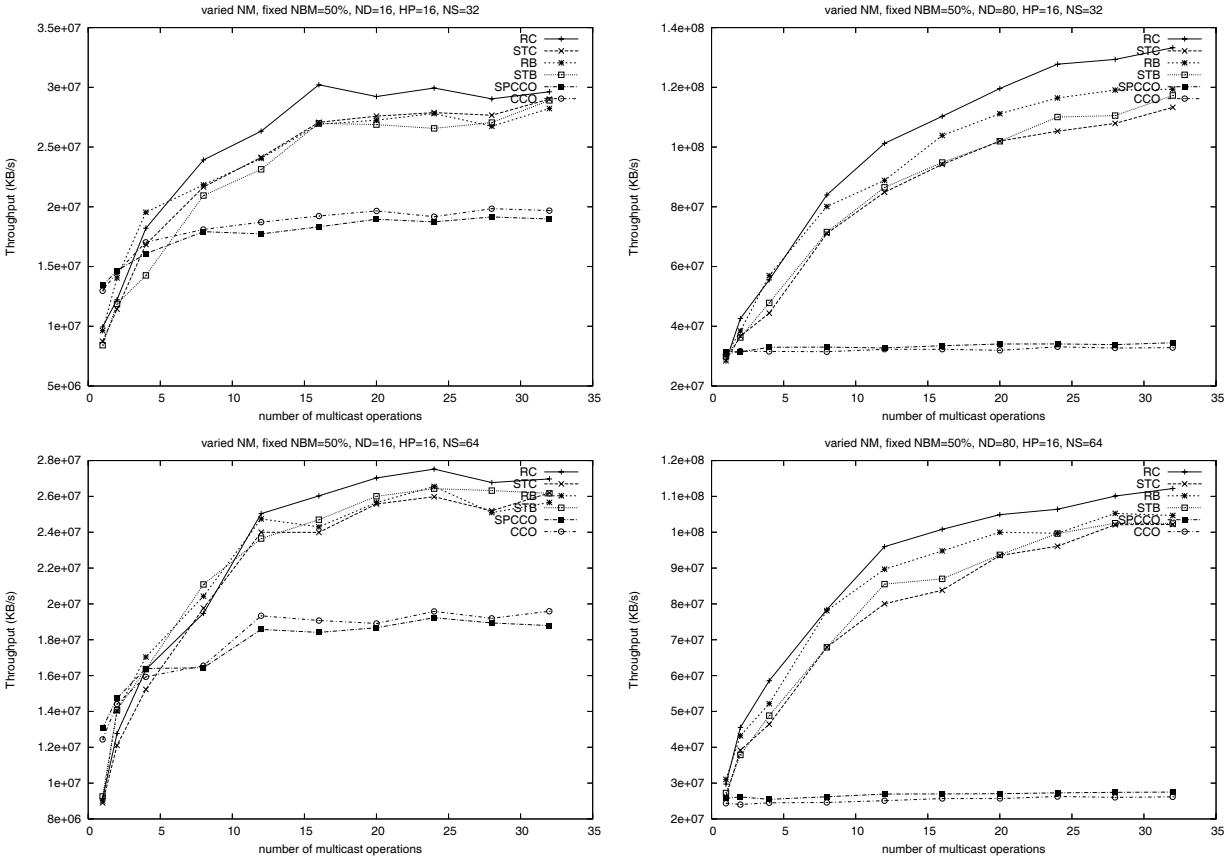

Figure 13: Throughput under different numbers of destinations

Effects of the Number of Destinations In this set of experiments we set the number of ports connected to processors to be 8. We choose two different numbers of switches, (32 and 40), and vary the number of destinations for each multicast from 16 to 300. Figure 14 illustrates the throughput of these algorithms. We observe that the throughputs of all the algorithms increase when the number of destinations increases, due to the increased amount of communication trafficts. In addition, the improvement ratio of the agent-based algorithms is higher than the improvement ratio of ordered-chain-based algorithms.

Effects of the Message Length We examine the effects of message length on the performance of proposed algorithms. We choose two message lengths – 128KB for short messages and 32MB for long messages, and vary the percentage of the multicast operations with long messages from 0 to 100. The source and destinations of a multicast are generated randomly.

Figure 15 indicates that the throughputs of agent-based algorithm is higher than those of the order-chain-based algorithms. In addition, we find that the throughput is minimized when the number of long-message is small, and is maximized when the percentage of the multicast operations with long

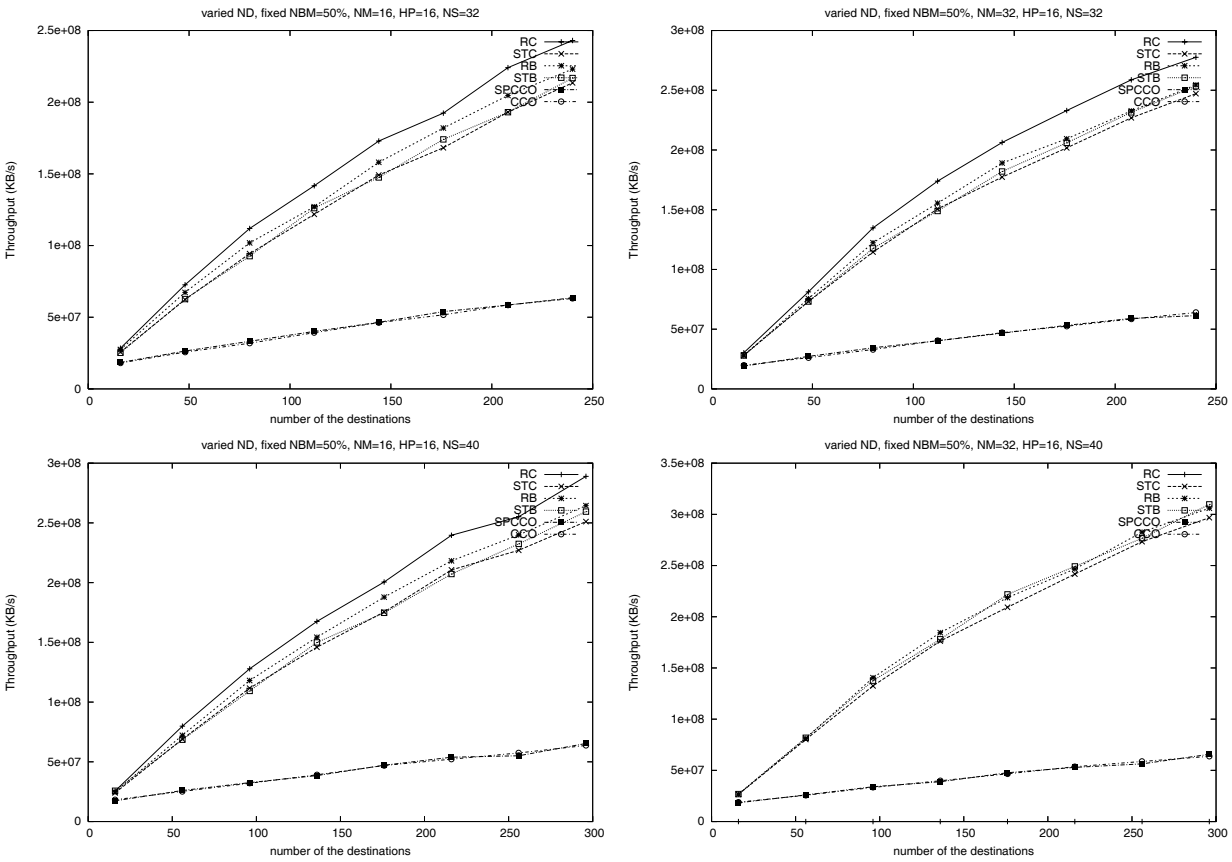

Figure 14: Throughput under different numbers of destinations

messages is 0 or 100. The possible reason is that long messages are likely to increase the chance of contention. When the number of long-messages is small, they will cause contention in particular regions and there is not enough of them to be evenly distributed in the system. This may cause hot-spots in communication so the throughput is reduced. When the percentage of long messages increeases, the throughput increases because the total amount of traffic increases, *and* the long messages can now be interleaved.

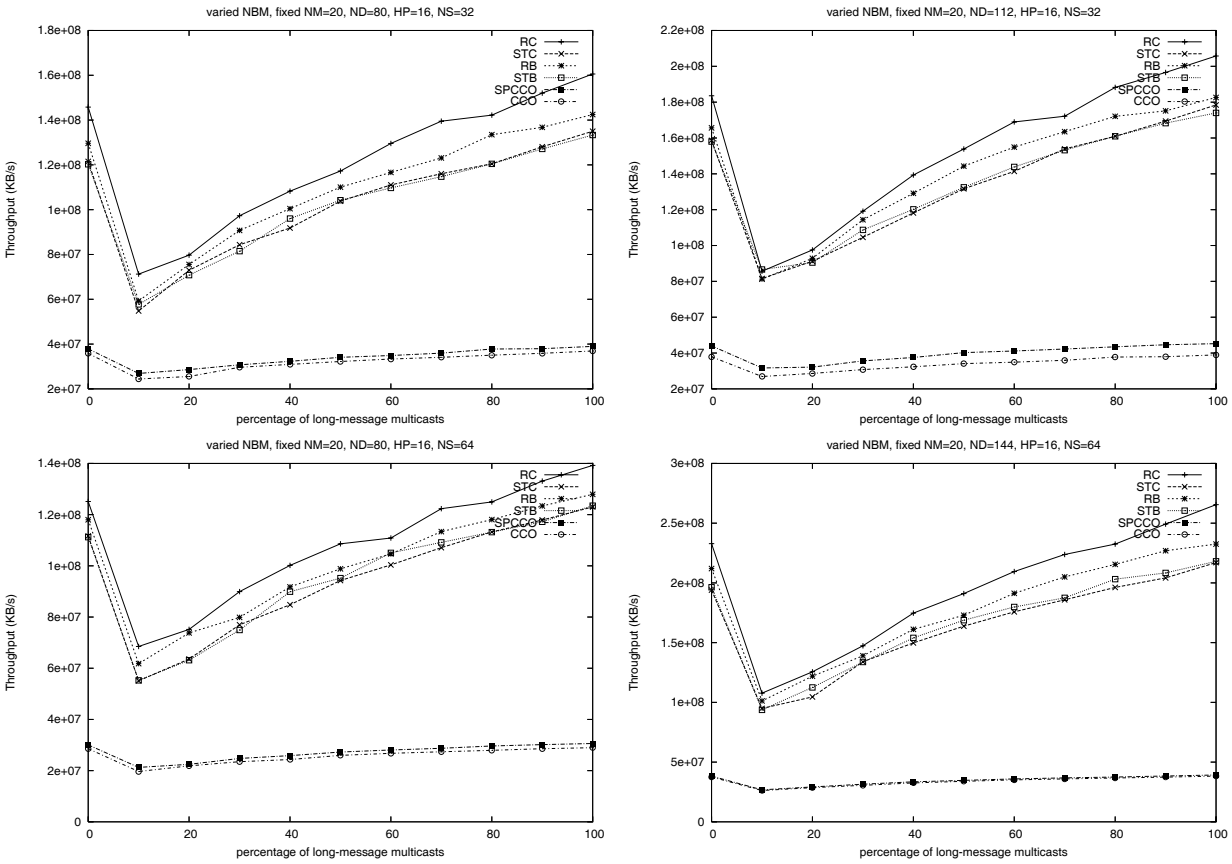

Figure 15: Throughput under different numbers of long-message multicasts

Summary of Results

In summary, the agent-based algorithms and the ordered-chain-based algorithms compliment each other. The ordered-chain-based algorithms are superior to the agent-based algorithms for small number of multicast operations, while the agent-based algorithms perform better than the ordered-chain-based algorithms for larger number of multicast operations (larger than 8 in our experiments). The difference in performance of these algorithms increases with number of multicast destinations, and number of processors in the system.

7 Conclusion

This paper describes an agent-based approach for scheduling multiple multicast on switch-based networks. Our approach assigns an agent to each subtree of switches such that the agents can exchange information efficiently and independently. The entire multicast problem is recursively solved with each agent sending message to those switches that it is responsible for. Communication is localized by the

assignment of agents to subtrees. In addition, the agent mechanism provides an easy mechanism in performing multiple multicasts simultaneously, with very low chances of network contention.

The key component of our recursive multiple algorithm is an optimal scheduling method that interleaves the local and remote message destinations. We showed that by properly setting the priorities of destinations we are able to find an optimal schedule to multicast messages within a subtree in the switch network.

We compare the results with SPCCO [12] and found that SPCCO, a highly efficient multicast algorithm based on *Partial Ordered Chains*, incurs high contention in large cases. Our agent-based approach minimizes contention by properly interleaving multiple multicast and optimally scheduling message passings between agents and destination processors to avoid hot spots.

References

[1] N. J. Boden, D. Cohen, R. F. Felderman, A. E. Kulawik, C. L. Seitz, J. Seizovic, and W. Su. Myrinet - a gigabit per second local area network. *IEEE Micro*, pages 29–36, Feb. 1995.

[2] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[3] W.J. Dally. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Trans. Comput.*, C-36(5):547–553, May 1987.

[4] J. Duato. On the design of deadlock-free adaptive routing algorithms for multicomputers. In *Proceedings of Parallel Architectures and Languages Europe 91*, June 1991.

[5] J. Duato. A necessary and sufficient condition for deadlock-free adaptive routing in wormhole networks. In *Proceedings of the 1994 International Conference on Parallel Proceeding*, August 1994.

[6] 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.

[7] K.-P. Fan and C.-T. King. Efficient multicast on wormhole switch-based irregular networks of workstations and processor clusters. In *Proceedings of the Internationl Conference on High Performance Computing Systems*, 1997.

[8] P. T. Gaughan and S. Yalamanchili. Adaptive routing protocols for hypercube interconnection networks. *IEEE Computer*, 26(5):12–23, May 1993.

[9] C.J. Glass and L.M. Ni. The turn model for adaptive routing. *J. ACM*, 41:847–902, Sept. 1994.

[10] G. Gravano, G. D. Pifarre, P. E. Berman, and J. L. C. Sanz. Adaptive deadlock- and livelock-free routing with all minimal paths in torus networks. *IEEE Trans. Parallel and Distributed Systems*, 5(12):1233–1251, Dec. 1994.

[11] R. Horst. Servernet deadlock avoidance and fractahedral topologies. In *Proceedings of the International Parallel Processing Symposium*, pages 274–280, April 1996.

[12] Ram Kesavan and Dhabaleswar K. Panda. Efficient multicast on irregular switch-based cut-through networks with up-down routing. In *IEEE Trans. Parallel and Distributed Systems*, volume 12, August 2001.

[13] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, hypercubes.* Morgan Kaufmann.

[14] L.M. Ni and P.K. McKinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, 26(2):62–76, February 1993.

[15] A.-H. Esfahanian P.K. McKinley, H. Xu and L.M. Ni. Unicast-based multicast communication in wormhole-routed networks. *IEEE Transactions on Parallel and Distributed Systems*, 5(12):1252–1265, December 1994.

[16] W. Qiao and L.M. Ni. Adaptive routing in irregular networks using cut-through switches. In *Proceedings of the 1996 International Conference on Parallel Proceeding*, pages I:52–60, August 1996.

[17] Andras Varga. The omnet++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference*, June. 2001.