Scaling Collective Multicast on High Performance Clusters

Laxmikant V. Kalé, Sameer Kumar
Department of Computer Science
University of Illinois at Urbana-Champaign
{kale,skumar2}@cs.uiuc.edu

Krishnan Varadarajan
Microsoft Inc.
krishvar@windows.microsoft.com

Abstract

Collective communication operations often involve massive data movement over the entire network. A bad implementation of these operations can affect the scalability of an application to a large number of processors. In this paper we study the collective multicast operation. The extreme case of collective multicast is all-to-all multicast *(MPI_Allgather). Here each processor multicasts a message to all the other processors. We present optimizations and performance studies for all-to-all multicast. These optimizations need to be different for small and large messages. For small messages, the major issue is minimization of software overhead. This can be achieved by message combining. For large messages it is network contention that can be reduced by intelligent message sequencing. Modern NIC's have a communication co-processor that performs message management through zero copy remote DMA operations. We present an asynchronous non blocking collective multicast framework that allows the processor do other computation while the collective operation is in progress. We will also present performance comparisons of the various algorithms implemented by our framework with many relevant applications and benchmarks.*

1 Introduction

Many scientific applications require collective communication operations. These operations often involve all the processors in the system (all-to-all communication) making them complex and long drawn. All-to-all communication has been extensively studied in literature. Depending on the nature of the message sent, all-to-all communication can be classified as all-to-all *personalized* exchange[4, 11, 19, 2, 17, 21, 5, 20, 23] or all-to-all *multicast* [25, 24, 6, 3]. In all-to-all *personalized* exchange each processor sends distinct messages to all the other processors in the system while in all-to-all *multicast* each processor sends the same message to every other processor in the system. All-to-all multicast can be assumed to be a special case of all-to-all personalized exchange. But as it is a simpler problem, faster implementations of it are possible. MPI defines the primitives *MPI_Alltoall* for all-to-all personalized communication and *MPI_Allgather* for all-to-all multicast.

All-to-all multicast is needed by many applications such as Matrix Multiplication, LU-factorization and linear algebra operations [25]. We have studied collective personalized communication in [9] and in this paper we focus on collective multicast.

Collective multicast optimization strategies need to be different for small and large messages. For small messages the cost of the multicast is dominated by the software overhead of sending the messages. This can be reduced by message combining. For large messages the cost is dominated by network contention. Network contention can be minimized by smart sequencing of messages based on the underlying network topology. In this paper we present strategies to optimize collective multicast for both small and large messages.

We will also be presenting two metrics for analyzing the performance of our strategies namely, (i) Completion time, (ii) Computation Overhead. Most related work has studied collective communication operations from the point of view of completion time. We believe that the computation overhead is an equally important factor. A lower computation overhead would allow the processor to do more computation while the collective operation is in progress. This is very useful for applications that can process messages as soon as they arrive. They do not have to wait for all messages to arrive before starting computation.

We have tested our strategies on PSC Lemieux[12] with up to 512 nodes of Lemieux[12]. Nodes on Lemieux are interconnected by the Quadrics[18] Elan high performance[14] interconnect. Elan has a communication co-processor that can assist in message management. The processor is thus relieved from message management and can compute while the message is in flight. Hence we define asynchronous and non blocking primitives for our collective operations unlike MPI. We define a split phase interface for the collective multicast operation. In the first phase each processor deposits its data and it is returned a handle. The processor can then poll on that handle while doing other computation and retrieve the data when the operation has been completed. Collective communication operations on Lemieux can take 10's to 100's of milliseconds. Leaving processors idle while the multicast is in progress would lead to serious wastage. We have also taken advantage of the hardware features of Elan. The topology of Lemieux is a radix-4 fat tree and our strategies are fine-tuned for that.

In the next few sections we first present a communication model to model the performance of our strategies. We then present the indirect strategies Mesh, 3d Grid, Hypercube and direct strategies (iv) spanning tree, (v)k-direct. Section 3 presents a performance comparisons and analysis of our strategies. Section 4 presents the related work and we finally conclude in section 5.

2 All to All Multicast

In all-to-all multicast each processor sends the same message to all the other processors. Hence each processor ends up receiving $(P - 1) * m$ bytes of data, where P is the number of processors and m is the size of each message. If no hardware multicast is available the total number of bytes transfered on the network would be $P * (P - 1) * m$ bytes. So each processor on an average would deposit $(P - 1)*$m bytes on the network. This is similar to overhead of *all-to-all personalized* communication. But, since each processor has the same message to send to all the other processors, we can combine and reroute messages through intermediate processors to reduce the software overhead and the network contention of sending the messages.

In this paper we will be presenting two types of strategies to optimize all-to-all multicast, (i) Direct Strategies, (ii) Indirect strategies. Direct strategies just sequence the sends and receives to minimize contention in the network. Indirect strategies combine and reroute messages through other processors.

First we will present a communication model to analyze the performance of the strategies and then we will present the following strategies, (i) Mesh, (ii) 3d Grid, (iii) Hypercube, (iv) Spanning Tree, (v) k-Direct, (vi) Ring.

2.1 Communication Model

Many models for communication on parallel architectures have been presented [13, 1, 22, 4, 21]. We will use a communication model similar to [13, 1]. In our model the time to send a message is given by T_{ptp}.

$$T_{ptp} = \alpha_s + \alpha_n + \alpha_r + m\beta + C + L \tag{1}$$

- α_s is the send overhead for the source processor.
- α_n is the overhead in the NIC for sending and receiving the message. This includes routing the message, scheduling the DMA etc. This similar to the gap g in the LogGP model[1].
- α_r is the overhead of the receiving processor.
- β is the cost of depositing a byte of data on the network. It is the same as G in the LogGP model.
- C is the network contention experienced by the message.
- L is the network latency.
- m is the size of the message.
- P will be used to represent the number of processors in the system. It is however not present in the above equation

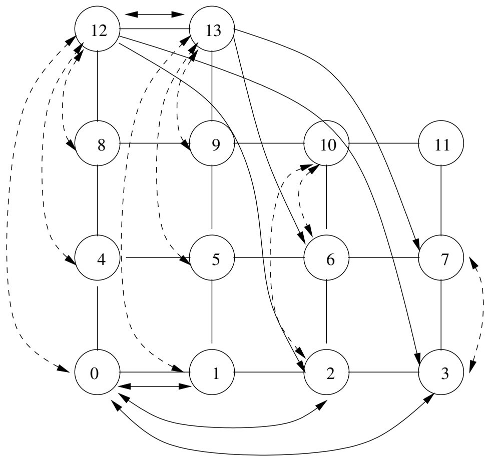

Figure 1. Mesh based All-to-all multicast, with holes.

On a network like Elan, L is very small and can be ignored. In a burst of messages, the messages will be sent at times $\alpha_s + \alpha_n$, $\alpha_s + 2\alpha_n + m\beta$, $\alpha_s + 3\alpha_n + 2m\beta$, ..., $\alpha_s + k\alpha_n + (k-1)m\beta$. So the completion time for a burst of k messages is given by

$$T_{burst} = \alpha_s + k(\alpha_n + m\beta) + \alpha_r + C(k, m) \tag{2}$$

In a burst of messages sometimes the α_s, α_r are dominated by the other factors and can be ignored.

2.2 Indirect Strategies

We now present three indirect strategies namely, (i) Mesh, (ii) 3D Grid, (iii) Hypercube. Indirect strategies reroute messages through intermediate nodes. At these intermediate nodes messages are received are combined into a larger message before being sent to their destinations. This reduces the total number of messages being sent by each processor and hence the software overhead.

2.2.1 2-D Mesh Strategy

In this scheme, the messages are routed along a 2-D mesh. In the first phase of the algorithm, each node sends its multicast message to all the nodes in its row. In the second phase, the nodes group these messages and send them to the nodes in their respective columns. Thus all the messages travel at most two hops before reaching the destination. In the first phase each processor sends a burst of $\sqrt{P} - 1$ messages of size m bytes. In the second phase each processor sends the same number of messages but of size $\sqrt{P} \times m$ bytes. The completion time for the multicast with mesh strategy, T_{mesh} is shown in equation 3. Here C_1 and C_2 are the contentions in each of the phases.

$$T_{mesh} \approx (\sqrt{P} - 1) \times (\alpha_n + m\beta + C_1 + \alpha_n + \sqrt{P}m\beta + C_2)$$

$$T_{mesh} \approx 2\sqrt{P}\alpha_n + (P - 1)m\beta + C_1 + C_2 \tag{3}$$

When the number of nodes is not a perfect square, the mesh is constructed using the next higher perfect square. This gives rise to *holes* in the mesh. Figure 1 illustrates our scheme for handling holes, in a mesh with two holes. (The dotted arrows show the second stage.) The role assigned to each hole (which are always in the top row) is mapped uniformly to the remaining processors in its column. So if a node (i, j) needs to send a message to column k and node (i, k) is a hole, it sends the message to node $(j\%(NROWS - 1), k)$. Here NROWS is the number of rows in the mesh. Thus in the first round node 12 sends messages to nodes 2 and 3. No messages are sent to a row with no processors in it. Dummy messages are used in case a processor has no data to send. Similar to the analysis presented in [9] the presence of holes does not affect equation 3.

2.2.2 3D Grid

We also implemented the 3d grid strategy. In this strategy messages are sent in three phases along the X, Y and Z dimensions respectively. In the first phase, each processor sends its multicast message of size m bytes to its

$\sqrt[3]{(P)} - 1$ neighbors along the X dimension. In the second phase these messages are combined and sent along the Y dimension. In the third phase all the data received in the second phase is data is combined and communicated to all the Z axis neighbors. The sizes of the messages exchanged in each of the phases are m, $\sqrt[3]{(P)}m$, $(\sqrt[3]{(P)})^2m$ bytes respectively. The total time for this strategy is given by equation 4.

$$T_{grid} = (\sqrt[3]{P} - 1) \times (3\alpha_n + m\beta + \sqrt[3]{P}m\beta + (\sqrt[3]{P})^2 m\beta) + C_1 + C_2 + C_3$$
$$T_{grid} \approx 3\sqrt[3]{P}\alpha_n + (P-1)m\beta + C_1 + C_2 + C_3 \tag{4}$$

When the number of processors is not a perfect cube, the next larger perfect cube is chosen. Here we use a simpler strategy to map holes for ease of implementation. Holes are mapped to the corresponding processor in the penultimate plane. So all messages destined to a hole are received by the corresponding processor in the penultimate plane. If there is only one plane the holes are mapped like they are in the mesh strategy. No messages are sent to planes with no processors in them. Here each processor can receive messages for a hole in the first and second phases. We also assume that the receiving α_n cost is half the total α_n cost. The cost for grid strategy with holes is given by equation 5. Here λ_h is 1 if there are holes in the grid, 0 otherwise.

$$T_{grid} \approx (3 + \lambda_h) \times \sqrt[3]{P}\alpha_n + (P-1)m\beta + C_1 + C_2 + C_3 \tag{5}$$

2.2.3 Hypercube

The hypercube (Dimensional Exchange) scheme consists of $log_2(P)$ stages. In each stage, the neighboring nodes in one dimension exchange messages. In the next stage, these messages are combined and exchanged between the neighbors in the next dimension. This continues until all the dimensions are exhausted. Thus in phase 1, each processor sends its multicast message of size m bytes to its neighbor. In the second phase, each processor combines the message it received in the previous round with its message and sends it to its neighbor. So the size of the combined message is 2m. In the third phase the messages from the previous rounds will be combined with

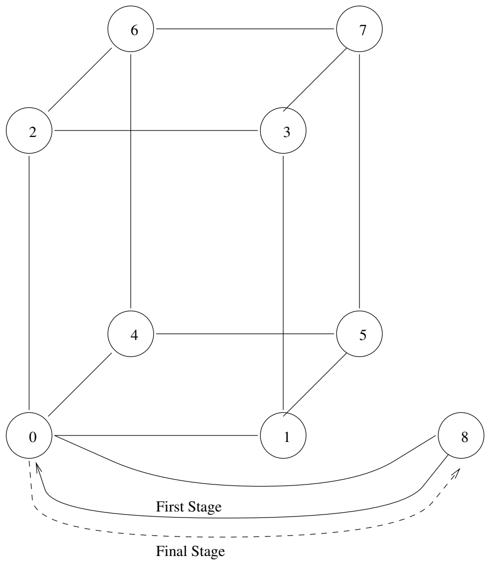

Figure 2. Hypercube strategy with an imperfect hypercube.

the local message leading to a message size of (m + m + 2m = 4m). In round i the message size will be $2^{i-1}m$. The overall cost is given by the equation 6.

$$T_{hypercube} = log_2 P \times (\alpha_s + \alpha_n + \alpha_r) + \sum_{i=1}^{log_2 P} (2^{i-1} m\beta + C_i) \tag{6}$$

In the case of an imperfect hypercube (when the number of nodes is not a power of 2), the next lower hypercube is formed. In the first step, the nodes that are outside this smaller hypercube send their message to their corresponding neighbor in the hypercube. For example, in Figure 2, node 8 sends it messages to node 0 in the first stage. As in the usual scheme, dimensional exchange of messages happens in this hypercube. All the messages for node 8 are sent to node 0. In the final stage, node 0 combines all the messages for node 8 and sends it to node 8. If there are few holes many processors will have twice the data to send. The cost of hypercube with holes is shown in equation 7

$$T_{hypercube} = (log_2 P + \lambda_h) \times (\alpha_s + \alpha_n + \alpha_r) + (1 + \lambda_h) \sum_{i=1}^{log_2 P} (2^{i-1} m\beta + C_i) \tag{7}$$

2.3 Direct Strategies

Direct strategies do not route messages through other processors but rather send them directly. These strategies tend to have higher software overheads as they send more messages. They also have no copying overheads as they do not combine messages.

In this section we will describe two direct strategies, (i) Spanning tree strategy, (ii) k-Direct Strategy.

2.3.1 Spanning Tree Strategy

In the tree strategy each processor sends messages along a multicast spanning tree rooted at that processor. Intermediate nodes just forward the message along that tree. If the processors are loadbalanced in the all-to-all multicast operation each of them will send $P-1$ messages. The sends will go in bursts of δ messages where δ is the degree of the spanning tree. The cost for the all-to-all operation with the tree strategy is given by equation 8. The parameter to tune here is δ.

$$\begin{aligned} T_{tree} &= \frac{(P-1)}{\delta} \times (\delta\alpha_n + \alpha_s + \alpha_r + \delta m\beta + C(\delta, m)) \\ T_{tree} &= (P-1)(\alpha_n + m\beta) + \frac{(P-1)}{\delta}(\alpha_s + \alpha_r + C(\delta, m)) \end{aligned} \tag{8}$$

2.3.2 k-Direct Strategy

In this strategy each processor sends k messages to the next k neighbors. So in iteration 'i' processor 'p' sends messages to the processors $\{p+i, p+i+1, p+i+2, ..., (p+i+k)\%P\}$. Then it *waits* for k messages and repeats this process again till it has sent P-1 messages. The cost of k-direct is given by equation 10. The parameter k controls how many messages will be submitted to the NIC. If the NIC can have multiple messages in flight at a time then a $k > 1$ would be more efficient. Also pre scheduling k messages would reduce α_s and α_r costs as shown in the equation.

$$T_{k-Direct} = (P-1)(\alpha_n + m\beta) + \frac{(P-1)}{k}(\alpha_s + \alpha_r + C(k,m)) \tag{9}$$

2.4 Ring Strategy

Here messages are sent along a ring formed by the leaves of fat tree. In stage i, on receiving a message the node sends that message to its neighbor. If the nodes are the successive hardware nodes then this will minimize contention. Only nodes at the edges of clusters will send messages to the higher levels of the switches. The cost is given by

$$T_{k-Direct} = (P-1)(\alpha_s + \alpha_n + \alpha_r + m\beta + C) \tag{10}$$

2.5 Collective Multicast Interface

MPI defines the blocking calls MPI_Allgather and MPI_Allgatherv for collective multicast. These calls will return only after the collective operation has been completed.

```
MPI_Allgather(sndbuf, msg_size, MPI_CHAR, recvbuf, msg_size,
             MPI_CHAR, MPI_COMM_WORLD);
MPI_Allgatherv(sndbuf, sndcnts, sdispls, MPI_CHAR, recvbuf, recvcnts,
               rdispls, MPI_CHAR, MPI_COMM_WORLD);
```

We plan to make interfaces split-phase. Now the collective multicast operation is going to be first scheduled and then polled for completion.

```
MPI_Iallgather(sndbuf, msg_size, MPI_CHAR, recvbuf, msg_size, MPI_CHAR,
               MPI_COMM_WORLD, &req);
MPI_Iallgatherv(sndbuf, sndcnts, sdispls, MPI_CHAR, recvbuf, recvcnts,
```

```
                rdispls, MPI_CHAR, MPI_COMM_WORLD, &req);
// User Code

while(!MPI_Test(&req, ...)){
  // Do computation;
}
```

For languages which provide asynchronous method invocation such as Charm++ all the multicast messages will invoke handlers on the remote processors when they are received there.

3 Performance

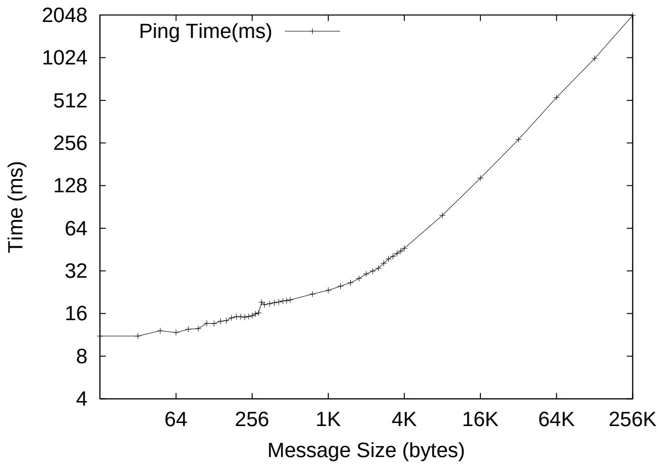

Figure 3. Bi-directional ping times for Charm on Lemieux

In this section we will present the performance of our all-to-all framework. First we describe the architecture of Lemieux where we did our performance runs and then we will analyze the performance of each strategy.

3.1 Lemieux Basics

Lemieux uses the high performance clustering technology QsNet[15, 14] from Quadrics[18]. QsNet has two building blocks, (i) programmable network interface Elan, (ii) high bandwidth communication switch Elite. Elites are connected by a radix 4 fat tree topology. Each of the Elite links is bi-directional and has a bandwidth of 400MB/s in each direction [14]. However due to contention for DMA's and a shared bus in the Elan NIC the bi directional bandwidth achievable is only around 160MB/s [14] in each direction. The MPI node to node message latency is around $5\mu s$ for uni-directional traffic and around $9\mu s$ for bi-directional traffic.

A plot of the bi-directional ping times on Lemieux with two processors and two nodes for our Charm/Converse[8] framework is presented in Figure 3. Small ($m < 288b$) messages on Elan are sent inline with the rendezvous packet. These messages have a smaller $\alpha_n = 8.5\mu s$. This time is spent in scheduling the DMA and waiting for the the local NIC processor which might be servicing other receives. Overheads $\alpha_s = 0.5\mu s$ and $\alpha_r = 2.5\mu s + 2\mu s = 4.5\mu s$. Here the $2.5\mu s$ is because of posting a receive and the $2\mu s$ is due to timer and scheduling overheads of Converse[7]. This gives a total ping latency of $0.5 + 8.5 + 2 = 11\mu s$ and a software overhead of $5\mu s$. Notice that the receive overhead does not appear because receives are posted before sends and posted again after while the acknowledgment is in flight.

For messages with ($m > 288b$) the rendezvous is sent first. After it has been acknowledged the DMA is scheduled for sending the message thus incurring a larger $\alpha_n = 14\mu s$. The bi directional bandwidth is around 130MB/s each-way. This is less than the bandwidth claimed in [14] and we are investigating this.

We now present the performance of our strategies on Lemieux using 1 processor per node. We will later show the performance of these libraries on 4 processor per node.

3.2 Direct Strategies

The performance of the k-Direct strategy on 256 nodes for different values of k is presented in Figure 4. Notice that the best performance is when k is 8 or 16. Hence in the rest of our experiments we will be using k values of 8

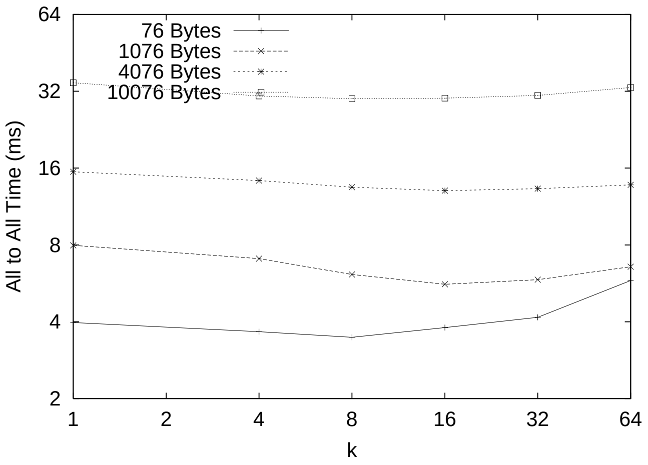

Figure 4. k-Direct all-to-all multicast performance comparison on 256 nodes

and 16.

As k increases the performance of the strategy improves because the α_s and α_r overheads are further minimized in a larger burst. But as k increases further, too many messages get scheduled in the NIC which hampers cache performance of the NIC. The Elan NIC shares the instruction and data cache [14].

Since spanning tree has the same behavior as k-direct we will be using a $\delta = 8$ in our performance runs. A performance comparison of the different direct strategies is presented in figure 5. Spanning tree has a similar performance as k-Direct. Ring does slightly better than k-Direct because it sends messages only to its nearby neighbors leading to a lower network contention. The above mentioned results were during a system drain when the load of the system was minimal. This advantage is however lost on on a regularly loaded system with non contiguous nodes (Figure 6). Lack of contiguity of the nodes makes more messages go to the higher layers of switches in the fat tree where there will be more interference from other jobs. This delays the completion time of the ring strategy. Ring also has a spike at around 2KB and we are investigating this.

Analyzing the performance results, we realized that most significant source of contention is in the Elan NIC. The Elite network is a low latency network which allows only one message to be in flight at a time. The Elites are connected by links with bi-directional bandwidths of 400MB/s. But the achievable node to node bandwidth

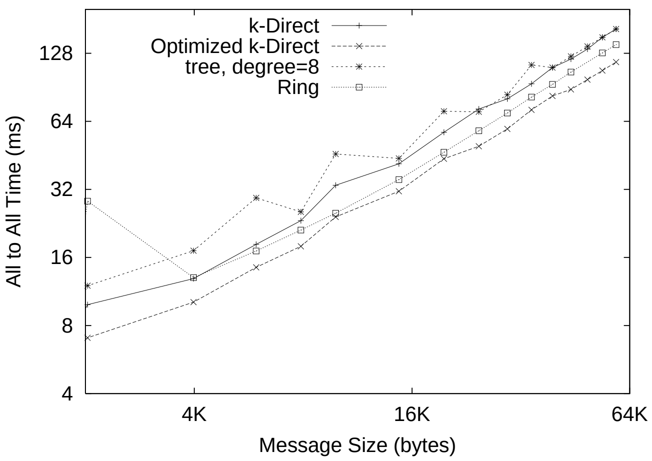

Figure 5. Direct Strategies on 256 nodes

by our Charm/Converse[8] framework is only around 130MB/s though. The effective bandwidth further drops for the collective multicast strategies due to other overheads. It is mentioned in [14] that node to node bandwidth is low because of contention for DMA while sending and receiving messages simultaneously. This bandwidth can be improved to if the source is in the elan memory and the receiver buffer is in main memory leading to the DMA being used only for receives. This perfectly suits the k-Direct strategy which sends the same message to all the processors. Now, that message can be copied once into an elan NIC buffer and sent to all the other processors from there. The *Optimized k-Direct* in figure 5 shows the performance of this optimization. The performance improvement is around 25%!

3.3 Indirect Strategies

Indirect strategies aim to optimize collective multicast for small messages. As mentioned earlier this is achieved by message combining. The performance of the indirect strategies on 256 nodes is presented in Figure 7. Notice that hypercube does the best. This is because hypercube sends only $log_2 P$ messages and hence has the least α overhead. Hypercube also takes advantage of the communication topology being a fat tree and minimizes network contention. Routing on a fat tree [16] has two stages the ascending stage and a descending stage. If the source

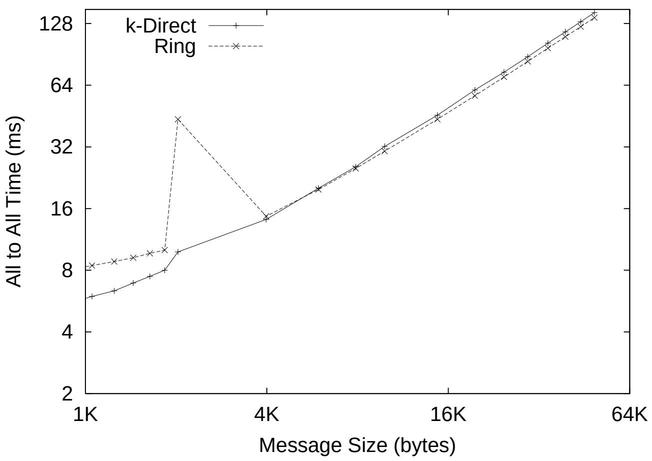

Figure 6. Ring performance on a loaded system

to destination mapping maps the fat tree onto itself the mapping does not generate any contention [16]. The hypercube goes through $log_2(P)$ phases where the mapping between sources and destinations is one to one and hence it should have no contention. Mesh can be thought of having $2\lceil\sqrt(P)\rceil$ phases with distinct destinations. But the messages are actually sent in a burst so there will be contention. The same is the case with 3d-grid. Moreover in our implementation of Mesh and 3d Grid nodes are arranged in row major order. So in the first phase the Mesh strategy will make processors send messages of size m bytes to nearby processors. In the second phase $\sqrt(P)m$ byte messages will be sent to far away processors. This will lead to more contention. The same is the case with 3d Grid. Hypercube on the other hand sends smaller messages to distant nodes and larger messages to nearby nodes. The message size keeps doubling as destinations get closer. This is also responsible for the scalable performance of hypercube.

To compute the contention in the hypercube strategy we did the following experiment: Only one node in the hypercube sends and receives messages with all its neighbors and the rest of the nodes in the system idle thus eliminating the effect of network contention. The results are shown in Figure 8. The figure also plots the performance of the hypercube using equation 6 with ($\alpha = 20us, \beta = 7.69ns/byte$). Notice that hypercube with 'no network contention' matches very closely with the theoretically predicted values. The rest can thus be

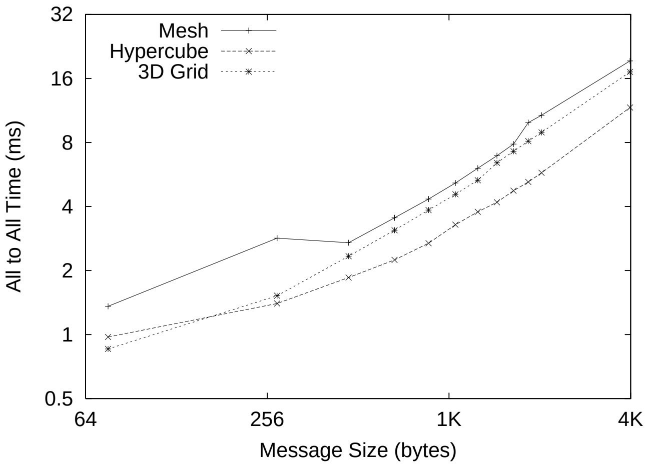

Figure 7. Indirect Strategy Performance on 256 nodes

attributed to network contention. Network contention may be due to other jobs running in the system which also share the communication network. Moreover the nodes were not fully contiguous because some nodes were down or being used for system management daemons.

3.4 Analysis

Table 1 shows the performance of all our strategies on 256 nodes. Hypercube does better than MPI_Allgather for most message sizes! Mesh and 3d Grid do well for small message sizes ($m < 4KB$) and k-Direct and spanning tree do well for larger message sizes. In fact for very small messages 3d Grid performs best because it sends $3\sqrt[3]{P}$ messages in a burst leading to a lower α overhead. Hypercube, which has small number of sends and lower network contention does well till about 4KB. Optimized k-Direct starts doing better than hypercube for messages larger than 4KB.

A plot comparing the performance of the hypercube and optimized k-direct and the 3d grid strategies is presented in Figure 9. This presents the need for a *learning framework* that can switch to the best strategy given the message size.

The computation time for each strategy on 256 nodes is presented in Figure 10. The compute time is a the sum

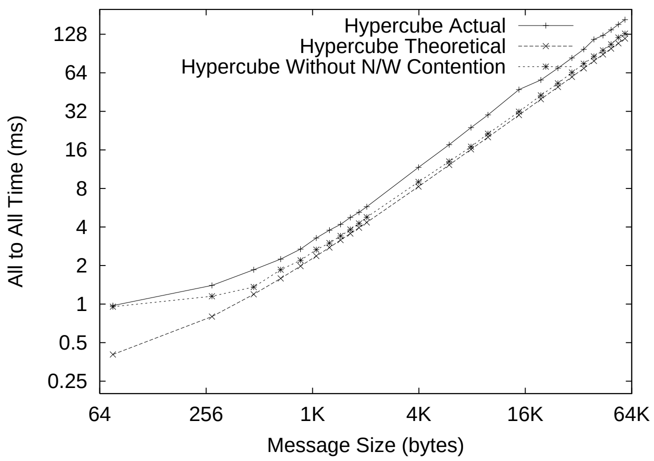

Figure 8. Hypercube performance comparison on 256 nodes

of the software message overhead and the combining/copying overhead. As can be noticed from the figure the compute overhead is a small fraction of the total completion time particularly for the indirect strategies. For small messages indirect strategies have lower compute overhead because they send fewer messages. As message size increase the copying overhead from combining starts to dominate and compute cost increases. Direct strategies have similar compute overheads for all messages sizes.

The compute overhead may determine the choice of strategy for the application developer. For example for a collective multicast on 256 nodes with 4KB messages one may choose hypercube strategy over Optimized k-Direct despite the fact that the latter has a better completion time by 1.4ms. If the completion time of the collective operation is not very critical then using hypercube would reduce the compute overhead by around 0.6ms thus letting all the 256 nodes compute for 0.6ms more. By a similar analogy the ring strategy may be preferred to the optimized k-direct strategy for 60KB messages because that would let the processors compute for 2ms more.

The scalability of our strategies to 512 nodes of Lemieux presented in Figure 11. For short message (276b) hypercube does well and for the larger message k-Direct does best.

Size(b)	Native MPI	MESH	HYP	GRID	k-Direct	Opt. k-Dir	Tree	Ring
76	1.11	1.35	0.97	0.85	4.63	-	8.80	9.57
276	3.56	2.84	1.39	1.52	5.21	7.18	9.81	7.16
1076	13.36	5.16	3.28	4.56	5.50	5.45	7.95	7.34
4076	14.19	19.33	11.68	17.2	12.9	10.2	17.1	13.0
10076	131.6	50.41	30.04	50.5	33.4	24.2	45.9	25.2
20076	250.4	101.3	56.19	101.1	57.3	43.7	70.9	46.7
40076	501.6	204.6	116.7	201.4	110.7	82.9	110.6	93.2
60076	747.1	287.9	166.3	289.4	160.3	117	163.9	139.8

Table 1. All strategies performance on 256 nodes (ms)

3.5 Performance using all the processors in each node

Lemieux has four processors in each node which share the network interface. With four processors per node the critical factor is out of node bandwidth per processor. Since all processors share the network the bandwidth drops down to around 33MB/s per processor each way. The latency also shoots up to around $45\mu s/msg$. The performance of the strategies on 4 processors per node is shown in Figure 12. The hypercube strategy will perform well because the last two steps with message sizes of $P/4 * m$ and $P/2 * m$ bytes respectively will take place within the node. Ring also deposits only $P/4 * m$ bytes of data on the network and hence performs well for large messages. The intra-node performance is low for small messages on Lemieux. This affects the perform ace of the ring strategy for small messages. Hypercube combines and exchanges large messages within the node in the last two stages. Hence it has a good performance for small message sizes. MPI however does best for very small messages. Hypercube is the best strategy small and intermediate sized messages and for large messages ring does the best.

Mesh and 3d Grid currently transfer $(P-1) * m$ bytes of data on the network and do not perform very well. To

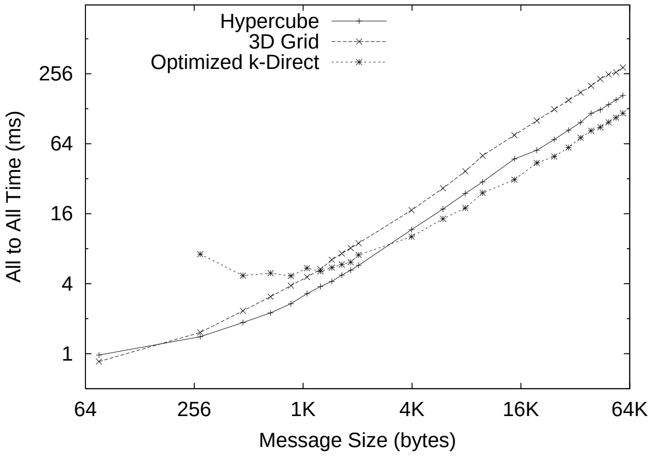

Figure 9. All to all multicast performance on 256 nodes

optimize this, the programmer will have to create 4 node level meshes each using the processors of the same rank. Collective multicast will be done independently on each of them. All global data acquired would then have to be broadcast within each node so that each processor has all the data. k-Direct should be extended to k-NodeDirect where each processor sends only one message to each to node to be handled by its rank processor. After the messages have been received on the nodes they can be broadcast internally and all the processors will have the data. We are currently in the process of doing the above.

4 Related Work

All to all broadcast has been extensively studied in literature. Much of the related work has been specific to architectures. All to all broadcast on 2d Meshes and Tori has been studied in [25, 24], hypercubes[**?**] and cluster of workstations in [6]. The LS strategy presented in [6] is suited for clusters of where the number of nodes local to a switch is comparable to the number of external nodes. These optimizations will not scale to thousands of processors on Lemieux. The paper also presents the ring algorithm for multicast which is also dealt by us. General mechanisms for all-to-all broadcasts are presented in [3]. The algorithms presented in this paper could lead to bottlenecks on the nodes which are going to become experts as most of the other nodes could be sending messages

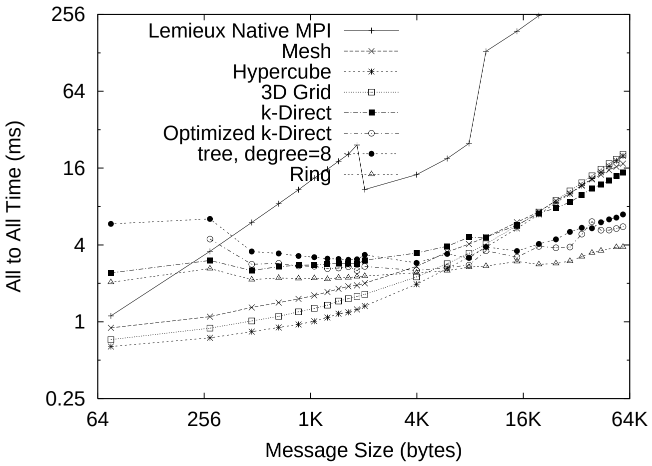

Figure 10. All to all multicast software overhead on 256 nodes

to this node at the same time. Collective communication on fat-trees is presented in [17] but the paper mainly deals with collective personalized communication. The Mesh, Grid and hypercube strategies have been presented in [4, 10].

All the above analyze collective communication only from the point of view of completion time. In this paper we analyze collective multicast from the point of view of both completion time and computation time. We also present and analyze the performance of our strategies on the fat-tree architecture of Lemieux.

5 Summary and Future Work

We presented an asynchronous interface for our collective multicast library, and implementations of several strategies aimed at optimizing its performance. The strategies include ”direct” strategies such as k-direct (that takes advantage of the fact that sending k messages in a burst may be faster than sending 1 message at a time, and controls contention by limiting k), and multiple-spanning-trees. The ”indirect” strategies we investigate multi cast messages in several stages, combining messages at intermediate stages in each phase. This is aimed at reducing the total number of messages sent/received by each processor(and so the software overhead incurred by the main processor). These strategies arrange processors in virtual topologies such as 2-D meshes, 3-D grids and

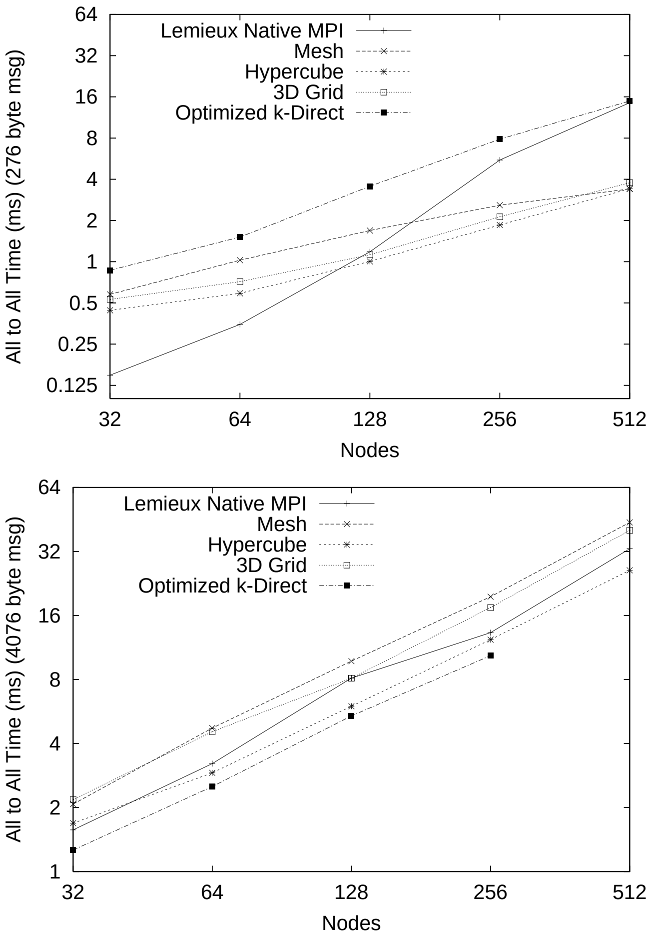

Figure 11. Scalability of our strategies

hypercubes.

We evaluated the strategies in the context of a modern communication architecture consisting of Quadrix switches and NIC's with the Elan communication library. We have also taken advantage of the special hardware features of the Elan NIC. The optimized k-direct strategy copies the message into an elan SDRAM buffer and sends the messages from there leading to a significant performance improvement. We are investigating similar mechanisms for the ring and spanning tree strategies. Here messages will be received in elan SDRAM buffers and stored temporarily. In the next stage the messages will be sent from these buffers. Combining algorithms may also benefit from this optimization if messages can be combined in the elan memory.

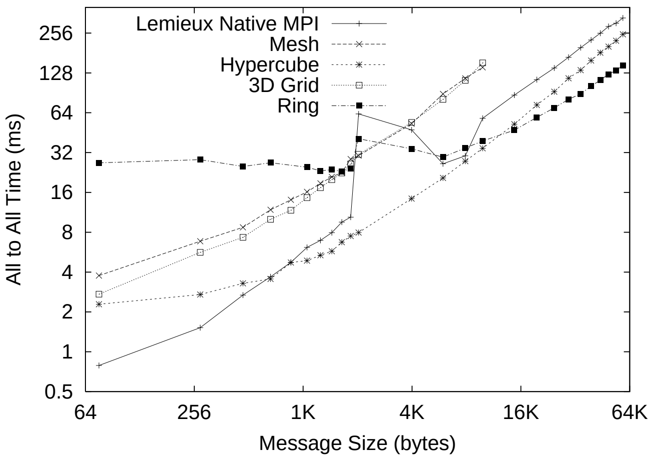

Figure 12. All to all multicast performance on 64 nodes and 256 processors

Our results also show that the best strategy to be used varies based on the context: For example, on 256 nodes, for small message sizes, hypercube generally performs better, whereas for message over 4 KB, k-direct strategy performs better. Another point we made is that the computation time that the CPU has to spend on the communication operation can be substantially lower than the completion time of the operation. So, if the application can take advantage of the split-phase asynchronous interface provided by our library, (and do useful computation while the multicast completes), it will gain substantially over a synchronous interface such as that provided by MPI. Moreover, the optimal strategy in such a context may be different. The examples in section 3 show that choice of strategy may determined by lower CPU overhead as opposed to completion time.

The above conclusions emphasize another theme in our research: the runtime system must be smart enough to observe the application behavior, and for the same interface, substitute the most appropriate strategy that the context indicates. We are engaged in development of such a "learning" strategy in the Charm++ framework.

Several new research issues for future work were identified during this work. For example, on a future machine such as IBM's Blue Gene/C, the physical interconnect is a 3D grid. Hence the number of connections out of a node is larger than the fat-tree, but the bisection bandwidth is limited. Evaluation of existing strategies, as well as development of new strategies in this context is necessary. Also, for short message sizes we believe that the hypercube may not be the best possible strategy: it uses only 1 message in each phase, whereas we have seen with k-Direct strategies that having multiple messages ”in flight” at a time is desirable. Strategies that combine best features of both need to be investigated.

References

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[2] S. Bokhari. Multiphase complete exchange: a theoretical analysis. *IEEE Trans. on Computers*, 45(2), 1996.

[3] M.-S. Chen, J.-C. Chen, and P. S. Yu. On general results for all-to-all broadcast. *IEEE Transactions on Parallel and Distributed Systems*, 7(4), 1996.

[4] C. Christara, X. Ding, and K. Jackson. An efficient transposition algorithm for distributed memory clusters. In *13th Annual International Symposium on High Performance Computing Systems and Applications*, 1999.

[5] V. V. Dimakopoulos and N. J. Dimopoulos. A theory for total exchange in multidimensional interconnection networks. *IEEE Transactions on Parallel and Distributed Systems*, 9(7):639–649, 1998.

[6] M. Jacunski, P. Sadayappan, and D. K. Panda. All-to-all broadcast on switch-based clusters of workstations. In *13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*, 1999.

[7] L. V. Kalé, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, Honolulu, Hawaii, April 1996.

[8] L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[9] L. V. Kale, S. Kumar, and K. Vardarajan. A framework for collective personalized communication, communicated to ipdps 2003. Technical Report 02-10, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2002.

[10] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings, 1994.

[11] C. C. Lam, C.-H. Huang, and P. Sadayappan. Optimal algorithms for all-to-all personalized communication on rings and two dimensional tori. *Journal of Parallel and Distributed Computing*, 43(1):3–13, 1997.

[12] Lemieux. http://www.psc.edu/machines/tcs/lemieux.html.

[13] C. A. Moritz and M. Frank. Logpc: Modeling network contention in message-passing programs. In *Measurement and Modeling of Computer Systems*, pages 254–263, 1998.

[14] F. Petrini, W. chun Feng, S. Hoisie, A.and Coll, and E. Frachtenberg. The quadrics network: high-performance clustering technology. *IEEE Micro*, 22(1):46 –57, 2002.

[15] F. Petrini, S. Coll, E. Frachtenberg, and A. Hoisie. Performance Evaluation of the Quadrics Interconnection Network. To appear, Journal of Cluster Computing, 2002.

[16] F. Petrini and M. Vanneschi. K-ary N-trees: High performance networks for massively parallel architectures. Technical Report TR-95-18, 15, 1995.

[17] R. Ponnusamy, R. Thakur, A. Chourdary, and G. Fox. Scheduling regular and irregular communication patterns on the CM-5. In *Supercomputing*, pages 394–402, 1992.

[18] Quadrics supercompters world ltd. http://www.quadrics.com.

[19] Y. J. Suh and S. Yalamanchili. All-to-all communication with minimum start-up costs in 2d and 3d tori. 9(5), 1998.

[20] N. S. Sundar, D. N. Jayasimha, D. K. Panda, and P. Sadayappan. Hybrid algorithms for complete exchange in 2d meshes. In *International Conference on Supercomputing*, pages 181–188, 1996.

[21] A. Tam and C. Wang. Efficient scheduling of complete exchange on clusters. In *ISCA 13th International Conference On Parallel And Distributed Computing Systems*, August 2000.

[22] A. T. Tam and C.-L. Wang. Realistic communication model for parallel computing on cluster. In *1st IEEE Computer Society International Workshop on Cluster Computing*, December 1999.

[23] Thakur and Choudhary. All-to-all communication on meshes with wormhole routing. In *IPPS: 8th International Parallel Processing Symposium*. IEEE Computer Society Press, 1994.

[24] Y. Yang and J. Wang. Efficient all-to-all broadcast in all-port mesh and torus networks. In *The Fifth International Symposium on High Performance Computer Architecture*, 1999.

[25] Y. Yang and J. Wang. Near-optimal all-to-all broadcast in multidimensional all-port meshes and tori. *IEEE Transactions on Parallel and Distributed Systems*, 13(2), 2002.