Scaling All-to-All Multicast on Fat-tree Networks

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

Abstract

In this paper, we study the all-to-all multicast operation. These strategies need to be different for small and large messages. For small messages, the major issue is the minimization of software overhead, where as for large messages, the issue is network contention. Many modern large parallel computers use the fat-tree interconnection topology. We therefore analyze network contention on fat-tree networks and develop strategies to optimize collective multicast using known contention free communication schedules on fat-tree networks in the design of two novel strategies. We evaluate performance of these strategies with up to 256 nodes (1024 processors) on an alpha cluster. We present schemes that perform well when a contiguous chunk of nodes is not available. For large messages, many of our strategies have two times better throughput than native MPI. We also demonstrate that the software overhead of a collective operation is a small fraction of the total completion time in the presence of the communication co-processor. We therefore compare the performance of the studied strategies using both metrics (i) Completion time, and (ii) Computation overhead.

1 Introduction

In the all-to-all *multicast* [22, 21, 7, 2] collective operation each processor sends the same message to every other processor in the system. MPI defines the primitives *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 [22]. Molecular dynamics applications like NAMD[17] (a well known production level application) and computational quantum chemistry applications like CPAIMD[20] have the *many-to-many multicast* pattern, which is a close cousin of the all-to-all multicast pattern. In many-to-many multicast many (not all) nodes send the same message to many (not all) other nodes. Although we restrict our study to the all-to-all multicast problem, the strategies we develop can be extended to optimize many-to-many multicast. We shall use *collective multicast* as a generic term, that refers to both all-to-all and many-to-many multicast operations.

Collective multicast is commonly used with both small and large messages. For example, each processor in NAMD [17] sends relatively short messages (about 2-4KB) during its collective multicast operation. In CPAIMD[20], processors send large messages (160 KB) during the all-to-all multicast operation. However, different techniques are needed to optimize collective multicast 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 the performance results of our strategies on Pittsburgh Supercomputing Center's Lemieux [13]. Nodes on Lemieux are interconnected by QsNet[14] from Quadrics [19]. Lemieux has over 750 Compaq ES45 nodes (with 3,000 processors) with a peak performance of over 6TF. QsNet uses a fat-tree network topology. Fat-tree networks, as described in detail in [12, 15, 5], are easy to extend and have a high bisection bandwidth. Hence, they are the preferred communication networks for many modern parallel clusters. Network contention on fat-trees has been extensively studied in [5] for the CM5 data network. We use this analysis to design two new all-to-all multicast optimization strategies namely, *kShift* and *kPrefix*. The optimization strategies we describe are general, as they can be applied to any fat-tree network and do not restrict the number of processors to powers of two.

Further, in this paper, we also describe additional optimizations specific to QsNet. Specifically, sending messages directly from NIC memory substantially increases the network bandwidth, as it avoids DMA and PCI contention. We use this feature in our strategies. As the performance results in this paper show, the strategies scale to 256 nodes (1024 processors) of Lemieux with an effective bandwidth of 511

MB/s per node(255.5 MB/s each way), which is more than twice better than Lemieux Native MPI. Our strategies also perform well in the presence of *missing nodes* in the fat-tree (i.e. when a contiguous set of nodes are not available).

In this paper, we also present *Computation Overhead* as an important metric to evaluate collective communication strategies. Most related work has studied collective communication operations from the point of view of completion time. We believe that computation overhead is an equally important factor. Network interfaces like Elan have a communication co-processor, which reduces main processor participation in message management. This allows the main processor to compute while the collective operation is in progress. Collective communication operations on Lemieux can take tens to hundreds of milliseconds. Leaving processors idle while the multicast is in progress can lead to serious wastage. Hence, we define a split-phase interface for the all-to-all multicast operation. First each processor deposits its data and 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. We have implemented our strategies on top of the Converse/Charm++ [8, 9] runtime system. MPI programs can take advantage of this split-phase interface through the AMPI framework [6]. We begin by first presenting a communication model to describe the performance of our strategies.

2 Communication Model

We use a simple model for the time to send a point to point message, denoted as T_{ptp}.

$$T_{ptp} = \alpha + m\beta + Cm + L \tag{1}$$

Here, α is the total processor and network software overhead for sending each message, while β is the per byte network transfer time. The byte here is being sent out from main memory. The parameters L and C represent the network latency and the per byte network contention respectively. Further, when dealing with machine specific, and network-interface (Elan) specific costs, we use the parameter β_{em} to represent the per byte network transfer time from Elan memory. Parameter γ is the per byte memory copying overhead for message combining and δ is the per byte cost of copying data from main memory to Elan memory.

The *Elan network* can support a peak bandwidth of 400 Million Bytes/s (or 382MB/s) [15] each way. However, we found that the *processor-to-processor* (with data being transferred from source processor memory to destination processor memory) achievable bandwidth with one way traffic, is only 290MB/s. This is mainly due to PCI contention in the ES45 Alpha server. Further, when the two processors are simultaneously sending and receiving, this bi-directional traffic brings the each-way bandwidth down to 128 MB/s. We will use the term *effective bandwidth* to represent the combined bi-directional bandwidth of a node in both directions. In the above case the effective bandwidth is 256 MB/s.

Heavy contention for the DMA by simultaneous send and receive operations is responsible for this drop in throughput. If the message is sent from Elan NIC [15] memory to the destination processor's memory, the effective bandwidth can be raised to 610MB/s (305MB/s each way). This is because messages in one direction will not require DMA or PCI intervention. Normally, the cost (δ) of copying messages into NIC memory nullifies the advantage of this optimization. In the next section we present the scenario where we can take advantage of this feature of Elan.

On PSC Lemieux, our measurements show that the cost of copying the message into the network interface (δ) is the same as the per byte network transmission time from Elan memory (β_{em}). Specifically, $\delta \approx \beta_{em} \approx 3.13ns/byte$ and $\beta = 7.5ns/byte$ (corresponding to effective bandwidths of 610MB/s and 256MB/s respectively).

3 Strategies for Short Messages

The cost of implementing a multicast by each processor directly sending messages to all (P-1) destinations is given by equation 2.

$$T_{all-to-all} = (P-1)\alpha + (P-1)m(\beta + C) + L \tag{2}$$

As stated in the previous section, $\alpha = 7.4\mu s$ and $\beta = 7.5ns/byte$. With short messages, this cost is dominated by the software overhead (α) term. Message combining reduces the total number of messages, making each node send fewer messages of larger size. Combining strategies route messages along a virtual topology, in multiple phases. In each phase, the messages received in the previous phases are combined into one large message before being sent out to the next set of destinations in the virtual topology. After the final phase, each node has received every other node's data. With these strategies, the number of messages sent out by each node is typically much smaller than P, thus reducing the total software overhead. We present two combining strategies: 2-D Mesh [3, 10] and Hypercube [11]. Although these schemes have been presented before, their adaptation to fat-trees and Quadrics, and the performance equations are our contributions.

3.1 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 multicasts its message to all the nodes in its row. In the second phase, the nodes combine all the messages they received in the previous round and send the combined message to the nodes in their respective columns. Thus each message travels two hops before reaching its destination. In the first phase, each node sends $\sqrt{P}-1$ messages of size m bytes. In the second phase, each node sends the same number of messages but of size $\sqrt{P} \times m$ bytes. Both the above steps are multicasts along rows and columns. Hence the messages are copied into the network interface before being sent out, taking advantage of the lower per-byte network transmission time β_{em} [1].

The completion time for the multicast with mesh strategy, T_{mesh} is shown in equation 3. Here, C_{mesh} and L_{mesh} represent the network contention and network latency experienced by the messages. The equation also includes, (i) memory copying overhead from message combining (the γ terms), (ii) the cost of copying the message into the network interface (the δ term).

$$T_{mesh} \approx 2\sqrt{P}\alpha + Pm(\beta_{em} + \gamma + C_{mesh}) + \sqrt{P}m\delta \quad (3)$$

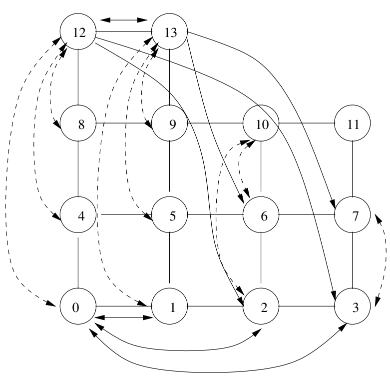


Figure 1. 2-D Mesh Topology

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 3.1, illustrates our scheme for handling holes in a mesh with two holes. The dotted arrows in Figure 3.1 show the second stage. The role assigned to each hole is mapped uniformly to the remaining nodes in its column. So if node (i, j) needs to send a message to column k and node (i, k) is a hole, it sends that message to node $(j \bmod (nrows-1), k)$ instead. 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 rows with no nodes in them. Dummy messages are used in case nodes have no data to send. By analysis similar to that in [10], the presence of holes does not affect equation 3.

[1] The Elan network interface has about 64 MB of SDRAM memory. This should be sufficient for the Mesh strategy (which sends messages of size $\sqrt{P} \times m$ bytes) for messages up to 4 MB on 256 nodes. Larger messages would have to be packetized, but with minimal additional cost

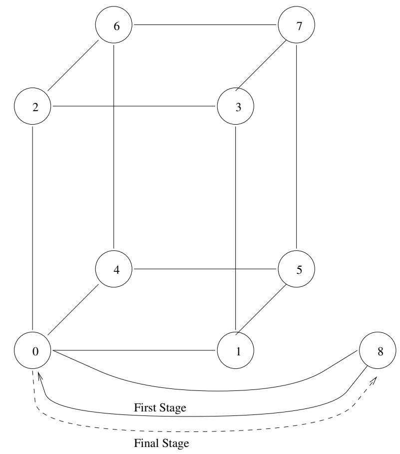


Figure 2. Hypercube Topology

3.2 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. So in the first phase, each node sends its multicast message of size m bytes to its neighbor. In the second phase, each node combines the message it received in the previous round with its message and sends $2m$ bytes to its neighbor. In the third phase the messages from the previous rounds are combined with the local message leading to a message size of $(m+m+2m = 4m)$. In round i the message size is $2^{i-1}m$. The overall cost is given by the equation 4.

$$T_{hcube} = log_2 P\alpha + (P-1)m(\beta + \gamma + C_{hcube}) \quad (4)$$

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 3.1, node 8 sends it messages to node 0 in the first stage. Next, dimensional exchange of messages happens in the smaller 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 them to node 8. If there are holes, many nodes will have twice the data to send. The cost of hypercube with holes is shown in equation 5. Here $\lambda_h = 1$ if there are holes and 0 otherwise.

$$T_{hcube} = (log_2 P + \lambda_h)\alpha + (1+\lambda_h)(P-1)m(\beta + C_{hcube}) \quad (5)$$

Notice that in the hypercube strategy, a distinct message is exchanged in each phase. Hence we cannot take advantage of the lower transmission overhead β_{em}, which depends on the same message being sent several times. However, we can use a *hybrid* approach with hypercube exchange for $log_2 P - \zeta$ stages and then direct exchange on ζ-dimension sub cubes. For $\zeta = 2$, the number of messages would only increase by 1, and for $\zeta = 3$ it would increase by 4. However the per byte term would be reduced substantially, as most of the data is sent in the last few stages. We will be describing the performance of this optimization in the next section. The new cost equation is given by equation 6. For simplicity we have not included the holes term in this equation. Equation 6 has three parts to it, (i) hypercube cost for $log_2 P - \zeta$ stages, (ii) direct cost within the ζ-subcube with messages of size $(P/2^{\zeta})m$ bytes, (iii) cost of copying this message into the network interface. The optimal value of ζ depends on the number of nodes P and the size of the message m. The term P' represents $P/2^{\zeta}$ in equation 6.

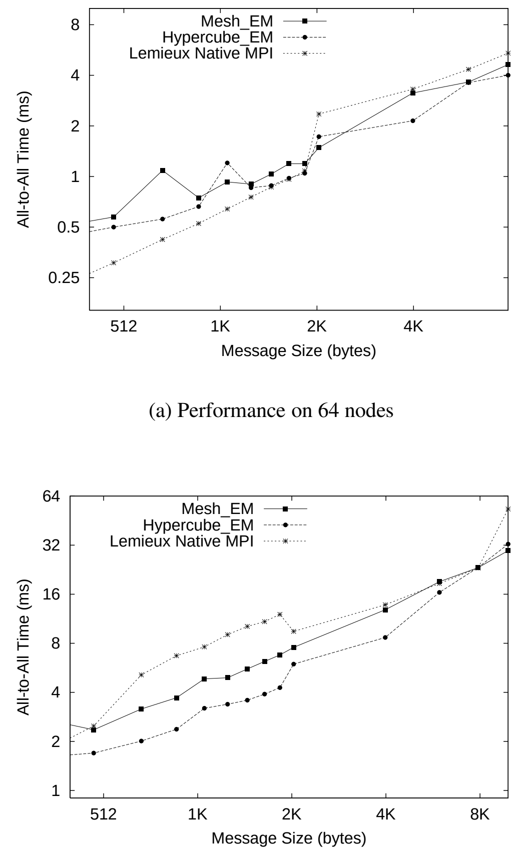
$$T_{hcube} = log_2 P\alpha + P'm(\beta+\gamma+\delta) + (P-P')m\beta_{em} \quad (6)$$

3.3 Performance

Figures 3(a) and 3(b) show the short message performance (completion time) of the strategies (combining strategies and Lemieux MPI), on 64 and 256 nodes respectively. The mesh strategy presented in these graphs (Mesh_EM) sends all the messages from Elan memory (EM). Hypercube_EM, shown in the plots, directly sends messages from Elan memory in the last three stages, i.e. parameter $\zeta = 3$. Observe that, MPI does better than our strategies for very short messages, because of scheduling and timer overheads in the Charm runtime system. But for messages larger than 2KB on 64 nodes and 400 bytes on 256 nodes, Hypercube_EM starts doing better.

Figure 4(a) shows the advantage of copying the message into Elan memory, on 256 nodes. Here, Mesh_MM shows the performance of the mesh strategy sending all its messages from main memory. For Hypercube_MM, there are no direct stages, i.e. $\zeta = 0$. Sending messages from Elan memory (EM) substantially improves the performance of the mesh strategy on 256 nodes. Hypercube also benefits from direct stages that send messages from Elan memory.

Figure 4(b) shows the computation overhead of Hypercube and Mesh strategies. Notice that the computation overhead is much less than the completion time. This suggests the need for an asynchronous split-phase interface, as provided by our framework.



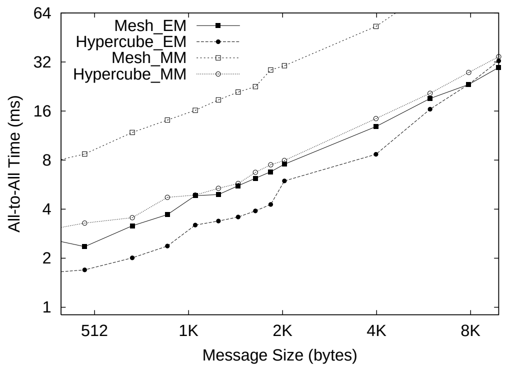
(a) Performance on 64 nodes

(b) Performance on 256 nodes

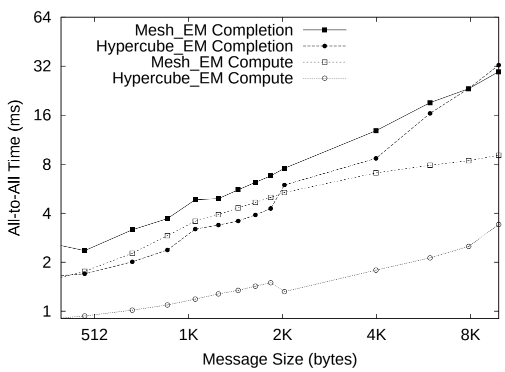
Figure 3. Performance for short messages

4 Direct Strategies

Direct strategies do not combine messages. These strategies are designed to optimize all-to-all multicast for large messages. Network contention is the main limiting factor here, so these strategies need to be topology specific. In this section, we first present an analysis on fat-tree networks (used in QsNet) from literature [5] and describe contention free communication schedules on fat-tree networks. The direct strategies that we present next take advantage of such communication schedules.



(a) Effect of sending data from Elan memory (256 nodes)



(b) CPU ovhd. vs completion time (256 nodes)

Figure 4. All-to-all Multicast on 256 nodes

4.1 Fat Tree Networks

QsNet uses a *fat-tree* (more specifically, 4-ary n-tree) interconnection topology. The graph k-ary n-tree has been defined in [16]. It is a type of fat-tree which can be defined as follows:

Definition: *A k-ary n-tree is a fat-tree that is composed of two types of vertices:* $P = k^n$ *processing nodes and* nk^{n-1} *switches. The switches are organized hierarchically with n levels that have* k^{n-1} *switches at each level. Each node can be represented by the n-tuple* $\{0, 1, ..., k-1\}^n$*, while each switch is defined as an ordered pair* $\langle w, l \rangle$ *where* $w \epsilon \{0, 1, ..., k-1\}^{n-1}$ *and* $l \epsilon \{0, 1, ..., n-1\}$. Here the parameter l represents the level of each switch and w identifies a switch at that level. The root switches are at level $l = n-1$, while the switches connected to the processing nodes are at level 0.

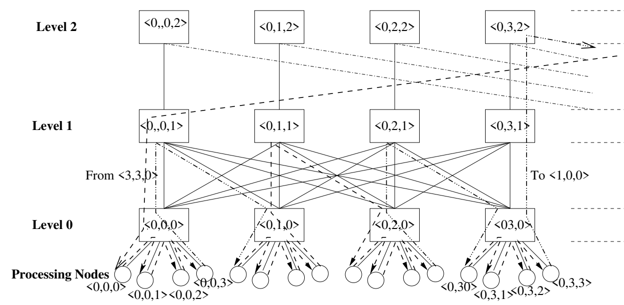


Figure 5. First Quarter of a 4-ary 3-tree

- *Two switches,* $\langle w_0, w_1, ..., w_{n-2}, l \rangle$ *and* $\langle w'_0, w'_1, ..., w'_{n-2}, l' \rangle$ *are connected by an edge iff* $l' = l + 1$ *and* $w_i = w_{i'}$ *for all* $i \neq n - 2 - l$
- *There is an edge between the switch* $\langle w_0, w_1, ..., w_{n-2}, 0 \rangle$ *and the processing node* $\{p_0, p_1, ..., p_{n-1}\}$ *iff*

$$w_i = p_i \ for \ all \ i \ \epsilon \ \{0, 1, ..., n-2\}$$

The bisection bandwidth of fat-trees is $O(k^n)$ or $O(P)$. Figures 4.1 shows the first quarter of a 64 node fat-tree, with nodes and switches labeled using the above definition. The switches $\langle w_0, w_1, 2 \rangle$ are the root nodes while the switches $\langle w_0, w_1, 0 \rangle$ are connected to the processing nodes $\langle w_0, w_1, i \rangle$.

Routing on a fat-tree has two phases: (i) Ascending phase: here the message is routed to one of the common ancestors of the source and the destination, (ii) Descending phase: here the message is routed through a fixed path from the common ancestor to the destination node. Network contention happens mainly in the downward descending phase[5]. However, many communication schedules on fat-trees are *congestion free*, i.e. they have no contention during the downward descending phase. The following lemmas present congestion free permutations, where each node sends a message to a distinct destination node. Proofs of these Lemmas have been presented in detail in [5]. We only briefly restate the Lemmas and the outlines of the proofs here.

Lemma 1 *Cyclic shift by 1, where. each processor* P_i *sends a message to the processor* $P_{(i+1) \bmod P}$*, is congestion free.*

The proof is straightforward. Only 1/4th of the traffic at the lowest level will go up to the next level and the rest will remain at the lowest level. The traffic that goes up will never compete for the same output link at any level of switches. Figure 4.1 also shows the congestion free schedule of the *cyclic-shift-by-1* operation on a 64 node fat-tree.

Lemma 2 *All quarter permutations that preserve the order of messages within a quarter are congestion free.*

In a quarter permutation, all messages from a source quarter go to the same destination quarter. The destination quarter for each quarter is also distinct. For example, Q1, Q2, Q3, Q4 sending messages to Q3, Q4, Q1, Q2 is a quarter permutation. In a quarter permutation, all messages go to the top of the fat tree. At the topmost level switches, each incoming packet is destined to a different quarter and hence a different output port. So, there will be no contention at the topmost level switches.

Message order is preserved if processor $P_{i,l}$ in quarter Q_i only sends a messages to the corresponding processor $P_{j,l}$ in quarter Q_j. In this scenario the message from $P_{i,l}$ at the topmost switch will use the path used by the message from $P_{j,l}$ (or a translated path) to the topmost switch, hence there will be no network congestion. In fact [5] describes shuffle and exchange quarter permutations that are all congestion free.

Definition: *A permutation is said to map a tree to itself when hierarchical groupings are preserved: siblings remain siblings, first cousins remain first cousins, k^{th} cousins remain k^{th} cousins.*

Lemma 3 *If a permutation maps a fat-tree into itself it is congestion free.*

This is the generalization of Lemma 2 where at each level of switches the traffic is a quarter permutation preserving the order of messages. So it is congestion free. The following Lemmas present communication schedules that map the fat-tree into itself. Hence they are also congestion free.

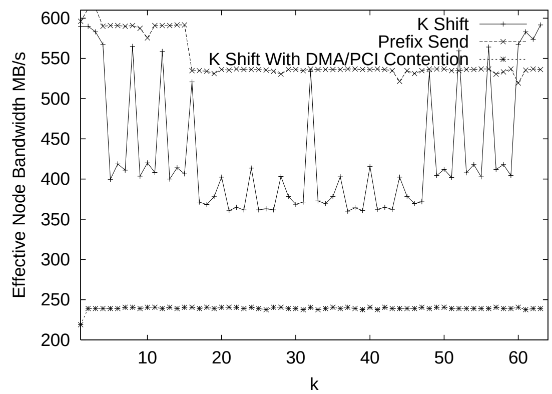
Lemma 4 *Hypercube dimension exchange is congestion free.*

Lemma 5 *Prefix-Send, where each processor P_i in stage j sends a message to the processor $P_{i\oplus j}$, is congestion free iff the total number of processors P is a power of two.*

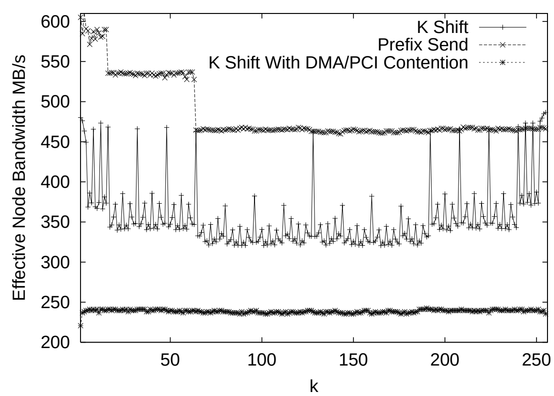
Lemma 6 *Cyclic shift by $k = a{*}4^j$ (i.e. processor P_i sends a message to either of the processors $P_{i\pm k}$), is congestion free iff a=1,2,3 and $k \leq P$.*

The permutations Hypercube and Prefix-Send require that the number of processors P, is a power of two. The communication pattern *cyclic-shift-by-k* only requires that the number of nodes and the start node be multiples of k.

Since the performance of these permutations is important for their use as steps of our multicast strategies, we analyzed them empirically. Figures 6(a) and 6(b) show the performance of the cyclic-shift and the prefix-send permutations on 64 and 256 nodes of Lemieux. In both permutations, in the k^{th} step each node P_i sends a message to its neighbor at distance k. For cyclic shift this neighbor is $P_{(i+k) \bmod P}$, while it is $P_{i\oplus k}$ for prefix-send. In the figures, the x-axis shows the distance k and the y-axis displays the effective bandwidth. The bandwidth for prefix-send is more stable than cyclic shift. Observe that for $k > 16$ the bandwidth of prefix-send drops from 610 MB/s to 534 MB/s and for $k > 64$ it drops to 470 MB/s. Large wire/switch delays to far away nodes [4] is responsible for this loss of performance.



(a) 64 nodes



(b) 256 nodes

Figure 6. Effective Bandwidth(MB/s) vs k

In the figures, observe that the peaks for cyclic shift occur only at the values of k given by Lemma 6. For other values of k network contention impairs throughput. On 64 nodes, the effective bandwidth at the peaks in the plots varies between 560 and 580 MB/s. However, on 256 nodes the peak bandwidth drops and varies between 460 and 485

MB/s. This is because in the *cyclic-shift-by-k* operation nodes at the boundaries send messages to distant nodes, restricting the network throughput at at the peaks. Again wire and switch delays are responsible for this loss of network throughput.

Figures 6(a) and 6(b) also show the performance of the *cyclic-shift-by-k* permutation with messages sent from main memory. DMA and PCI contention bring the effective bandwidth down to a steady 240MB/s. Observe that network contention makes no difference to this bandwidth, as the node bandwidth here is much lower than the network capacity. This shows the usefulness of sending messages from NIC memory, as node bandwidth is more than doubled.

The analysis presented so far brings out three major performance bottlenecks on Lemieux, (i) Network contention, (ii) Wire/Switch delays to distant nodes, (iii) Low network bandwidth for messages sent from main memory.

Our direct multicast strategies handle each of these three issues. Network contention can be avoided by using prefix-send and cyclic-shift permutations. Collective multicast can implemented by P-1 such permutations. Wire/switch delays are addressed by the kPrefix strategy, which minimizes data exchange with far away nodes, enabling it to scale to 256 nodes. Finally, our direct strategies copy messages into the network interface and send it from there with the lower transmission time β_{em}. We now describe four direct strategies, (i) Ring, (ii) Prefix-Send (iii) kShift, and (iv) kPrefix.

4.2 Ring Strategy

In this strategy, messages are sent along a ring formed by all the nodes in the system. In all stages of the ring strategy, on receiving a message node p forwards that message to its neighbor $((p+1) \bmod P)$ in the ring. This strategy is the same as *cyclic-shift-by-1* operation, repeated P-1 times. So by Lemma 1 it is congestion free. The cost of the ring strategy is given by

$$T_{ring} = (P-1)(\alpha + m\beta) + L_{ring} \tag{7}$$

Even though the ring strategy is congestion free it cannot take advantage of lower network transmission time β_{em}. This is because in each iteration every node sends a different message.

4.3 Prefix-Send Strategy

In this strategy each node exchanges its message with its prefix neighbor $p \oplus i$, in the i^{th} step. Here, the messages to all neighbors is send from Elan memory. Equation 8 shows the performance of the prefix-send strategy. The cost of copying the message into Elan memory is also included.

$$T_{Prefix} = (P-1)(\alpha + m\beta_{em}) + m\delta + L_{Prefix} \tag{8}$$

From Lemma 5, the prefix-send strategy is congestion free if P is a power of 2. Prefix-Send strategy has two main disadvantages, (i) it forces the number of nodes P to be a power of 2, (ii) it sends data to distant nodes. As mentioned earlier, wire and switch delays would limit the throughput of the prefix-send strategy. The kPrefix strategy has been designed to address both these problems.

4.4 kPrefix Strategy

The kPrefix strategy is a hybrid of the *ring* and the *prefix-send* strategies. Here, k is a power of two and P is a multiple of k. We divide the fat-tree into partitions of size k. Prefix send is used to send multicast messages within the partition, while ring strategy is used to exchange messages between neighbor partitions. Each node in the partition is involved in a different ring across all the partitions.

In the first $k-1$ phases, each node exchanges its message with its $k-1$ prefix neighbors within the partition. So, in phase i (where $0 \leq i \leq k-1$) node p exchanges a message with the node $p \oplus (i+1)$. In the k^{th} phase, node p sends a message to the node $(p+k) \bmod P$ forming the ring across partitions.

In the next iteration, the message from node $p-k$ (in the previous iteration) is multicast to the same k neighbors. This is repeated P/k times until all the messages have been exchanged. Moreover, multicast to k neighbors is implemented by first copying the message into the network interface and sending it from there.

By Lemma 5 the first $k-1$ phases are congestion free. Since k is a power of two, by Lemma 6 the last phase is also congestion free. Hence, kPrefix is congestion free. The cost of the kPrefix strategy is given by equation 9.

$$T_{kPrefix} = (P-1)(\alpha + m\beta_{em}) + (P/k)m\delta + L_{kPrefix} \tag{9}$$

In $k-1$ out of k phases of this strategy, messages are sent to nearby nodes (at most k away). This makes the kPrefix strategy have a high throughput and scale to a large number of nodes.

4.5 kShift Strategy

Both prefix-send and kPrefix are very sensitive to missing nodes (holes) in the system. On Lemieux, it is often hard to find contiguous nodes. Moreover, the Elan hardware skips these holes, while assigning virtual processor ids to the programs running on the nodes, confusing the optimization strategies and their performance drops. We now describe the kShift strategy which performs better in the presence of holes in the system.

The kShift strategy takes advantage of Lemma 6. In kShift strategy each node p sends messages to k nodes $\{(p-$

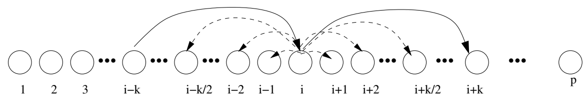


Figure 7. kShift Strategy

$\lceil(k-1)/2\rceil, ..., p-2, p-1, p+1, p+2, ..., p+\lfloor(k-1)/2\rfloor),$ $p+k\}$. Each message that node p gets from node $p-k$, can be copied into its NIC before it is sent to the k neighbors. This is repeated for P/k iterations to complete the collective operation.

For $k = \{1, 2, 3, 4, 8\}$ we get a contention free schedule if P is a multiple of k. Other values of k will have contention in some or all stages of the strategy. Figure 7 shows the kShift communication schedule. The cost of kShift strategy is shown by the following equation.

$$T_{kShift} = (P-1)(\alpha + m\beta_{em}) + (P/k)m\delta + L_{kShift} \quad (10)$$

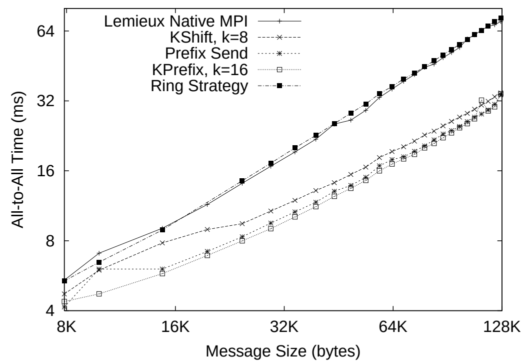
The equation also includes the cost of copying the message into the Elan NIC. Due to this additional overhead, larger values of k would have a better performance. So we use $k = 8$ in all our performance runs.

In the kShift strategy, most messages are sent to successive nodes. So having a few non-contiguous nodes, results in network contention only in some (not all) phases of the strategy.

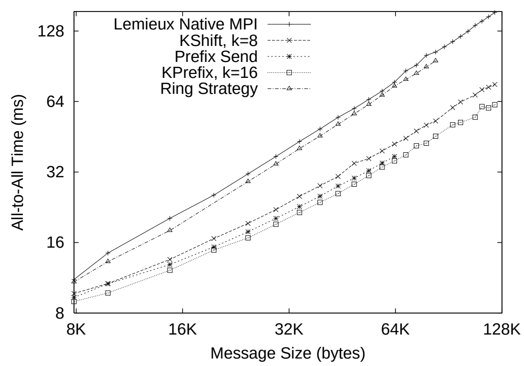
4.6 Performance

The performance results of the direct strategies are shown in Figures 8(a) and 8(b). Our results (not shown) indicate that combining strategies do better than direct strategies for messages smaller than 8KB. Therefore, we present performance data for direct strategies with message sizes greater than 8KB. The kPrefix strategy has the best performance. For messages larger than 40KB, kPrefix performs twice better than Lemieux MPI. The ring strategy, which only sends messages from main memory, has a performance very close to that of MPI.

To keep the nodes synchronized during the collective multicast operation, we have inserted global barriers in our direct strategies after every message is sent. However, these barriers make the computation overhead the same as the completion time. But, kShift and kPrefix can be altered to perform barriers after k messages have been sent and received. The altered strategies return control to the Charm++ scheduler after sending k messages. The scheduler can schedule useful computation until k messages have been received from the node's neighbors. Control is returned to the strategy, which first performs a barrier and then executes its next step. The performance of the altered kPrefix strategy is



(a) Performance on 64 nodes

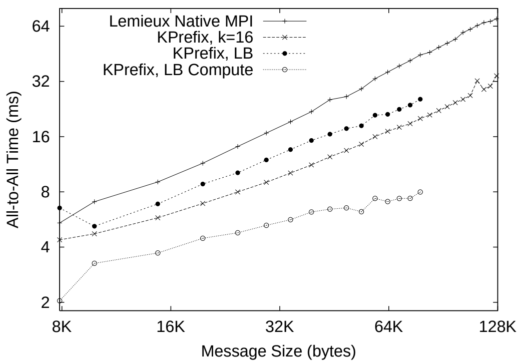


(b) Performance on 128 nodes

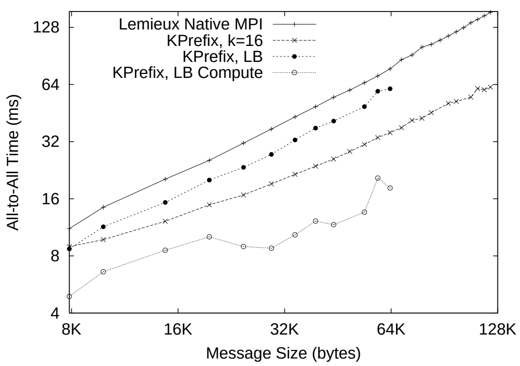
Figure 8. All-to-All with large messages

shown by *kprefix-lb* in figures 9(a) and 9(b). Here *lb* represents *less barriers*. This modification causes a drop in performance, because nodes are not completely synchronized any more. But the computation overhead is improved a lot. On 64 nodes, the performance of kprefix-lb is comparable to that of kPrefix. However on 128 nodes this performance lower than expected and we are investigating it.

Table 1 shows the bandwidth of the collective multicast operation for 256 KB messages. In the absence of missing nodes kPrefix scales best among all the strategies, with an effective bandwidth of 511MB/s on 256 nodes. As mentioned earlier wire and switch delays to distant nodes results in kShift and prefix-send not performing as well as kPrefix. But in the presence of missing nodes kShift performs best. This is because missing nodes cause only some (not all) phases of the kShift strategy to have congestion, resulting in better performance. On Lemieux, a large number of contiguous nodes are often hard to find. The kShift strategy can be used in such a scenario.



(a) Performance on 64 nodes



(b) Performance on 128 nodes

Figure 9. CPU Overhead vs Completion Time

5 Related Work

All-to-all multicast has been extensively studied in literature. Much of the related work has been specific to architectures. All-to-all multicast on 2d Meshes and Tori has been studied in [22, 21] and cluster of workstations in [7]. The LS strategy presented in [7] is best suited for small clusters of work stations. This paper also presents the ring algorithm for multicast which also used by us in [1]. General mechanisms for all-to-all multicasts are presented in [2]. 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 to this node at the same time. Collective communication on the CM5 network is presented in [18], but the paper mainly deals with collective personalized communication. The Mesh and hypercube strategies (without analysis of holes) have been presented in [3, 11].

Contention free permutations on fat-tree networks and are also presented in [16, 15]. Prefix send is presented in [5], as a contention free solution for all-to-all personalized communication. The analysis presented in [5] requires that the entire fat tree is available for the application, forcing P to be a power of 2. In a large machine like Lemieux, several nodes are often down and powers of two nodes may not be available. Hence such restrictions may be hard to meet.

Nodes	NativeMPI	kShift	kPrefix	PrefixSnd
64	225	507	531	520
128	198	432	519	428
144	187	433	521	-
192	-	416	516	-
256	190	405	511	429
128 (1 hole)	143	392	338	316
128 (2 hole)	112	399	373	-
240 (1 hole)	138	394	346	-

Table 1. Effective bandwidth (MB/s/node)

In contrast, our all-to-all multicast strategies do not restrict the number of nodes to be powers of two or perfect squares. We also present two new strategies kPrefix and kShift to optimize collective multicast on fat-tree networks. Our strategies take advantage of the higher bandwidth available by sending messages from Elan memory. We also analyze collective communication from the point of view for *completion time* and *computation overhead*.

6 Summary and Future Work

We present optimization strategies for both small and large messages. We use Mesh and Hypercube combining strategies to optimize all-to-all multicast for short messages. These strategies use virtual topologies and can be applied to any network. We evaluate their performance on fat-tree networks, by equations and experiments. We then present contention free lemmas for fat-tree networks from literature. The direct strategies presented in this paper, optimize collective multicast for large messages by taking advantage of contention free permutations. We also present two new strategies *kPrefix* and *kShift*. These direct strategies can be applied to any fat-tree network. The strategy kShift has good performance even with missing (non-contiguous) nodes present in the fat-tree. As nodes often go down in large production systems (which leads to non-contiguous nodes), this is a very desirable feature.

We also optimize the direct and combining strategies in the context of Quadrics QsNet. We first present a performance evaluation of QsNet. The network bandwidth from Elan memory is much more than the bandwidth from main memory. We quantify the affects of network contention on system throughput, showing the utility of congestion free permutations. Moreover, bandwidth to far-away nodes is lower than bandwidth to nearby nodes. Our strategies take advantage of these QsNet optimizations. They copy the messages into Elan SDRAM buffers and send the messages from there, leading to a significant performance improvement. The hypercube strategy is actually a hybrid as it sends messages directly in the last 3 stages, which lets it send messages from Elan memory. Mesh, prefix send, kPrefix and kShift also send messages from Elan memory. As an additional optimization for QsNet, the kPrefix strategy minimizes message sends to far-away nodes. Hence, it scales well to 256 nodes with an effective per-node bandwidth of 511 MB/s. This corresponds to 64GB/s of data being sent out on the network, which is 80% of the achievable bisection bandwidth of a 256 node QsNet fat-tree.

Another point we made is that the time that the CPU has to spend on the collective operation (computation overhead), is much lower than the completion time of the operation. So, if the application can take advantage of the split-phase asynchronous interface provided by our framework, (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. For example, kprefix-lb may be a better choice over kPrefix, if the application can compute while the collective multicast is in progress.

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* system in the Charm++ framework.

Moreover, on a future machine such as IBM's Blue Gene/C, the physical interconnect is a 3D grid. Hence there will be 6 ports out of a node which 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.

References

[1] Amitabh B. Sinha and Klaus Schulten and Helmut Heller. Performance analysis of a parallel molecular dynamics program. *Computational Physics Communications*, 78:265–278, 1994.

[2] 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.

[3] 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.

[4] W. J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6):775–785, 1990.

[5] S. Heller. Congestion-free routing on the cm-5 data router. *LNCS*, 853:176–184, 1994.

[6] C. Huang, O. Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, College Station, Texas, October 2003.

[7] 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.

[8] 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.

[9] 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.

[10] L. V. Kale, S. Kumar, and K. Vardarajan. A Framework for Collective Personalized Communication. In *Proceedings of IPDPS'03*, Nice, France, April 2003.

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

[12] C. Leiserson. Fat-trees: Universal networks for hardware efficient supercomputing. *IEEE Transactions on Computers*, 35(10):892–901, 1985.

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

[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. *Journal of Cluster Computing*, 6(2):125–142, April 2003.

[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] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.

[18] 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.

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

[20] R. Vadali, L. V. Kale, G. Martyna, and M. Tuckerman. Scalable parallelization of ab initio molecular dynamics. Technical report, UIUC, Dept. of Computer Science, 2003.

[21] 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.

[22] 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.