LA-UR- 01-4692
c.1



*Title:* HARDWARE-AND-SOFTWARE-BASED COLLECTIVE COMMUNICATION ON THE QUADRICS NETWORK

*Author(s):* Eitan Frachtenberg, CCS-3
Fabrizio Petrini, CCS-3

*Submitted to:* NCA 2001



**Los Alamos**
NATIONAL LABORATORY



Form 836 (8/00)

# Hardware- and Software-Based Collective Communication on the Quadrics Network *

Fabrizio Petrini, Salvador Coll, Eitan Frachtenberg and Adolfy Hoisie

CCS-3 Modeling, Algorithms, & Informatics
Computer & Computational Sciences Division
Los Alamos National Laboratory
{fabrizio,scoll,eitanf,hoisie}@lanl.gov

## Abstract

*The efficient implementation of collective communication patterns in a parallel machine is a challenging design effort, that requires the solution of many problems. In this paper we present an in-depth description of how the Quadrics network supports both hardware- and software-based collectives. We describe the main features of the two building blocks of this network, a network interface that can perform zero-copy user-level communication and a wormhole switch. We also focus our attention on the routing and flow control algorithms, deadlock avoidance and on how the processing nodes are integrated in a global, virtual shared memory.*

*Experimental results conducted on 64-node AlphaServer cluster indicate that the time to complete the hardware-based barrier synchronization on the whole network is as low as 6 μs, with very good scalability. Good latency and scalability are also achieved with the software-based synchronization, which takes about 15 μs. With the broadcast, similar performance is achieved by the hardware- and software-based implementations, which can deliver messages of up to 256 bytes in 13 μs and can get a sustained bandwidth of 288 Mbytes/sec on all the nodes, with messages larger than 64KB.*

*The hardware-based barrier is almost insensitive to the network congestion, with 93% of the synchronizations taking less than 20 μs. On the other hand, the software based implementation suffers from a significant performance degradation. In high load environments the hardware broadcast maintains a reasonably good performance, delivering messages up to 2KB in 200 μs, while the software broadcast suffers from slightly higher latencies inherited by the synchronization mechanism.*

*The work was supported by the U.S. Department of Energy through Los Alamos National Laboratory contract W-7405-ENG-36

## 1 Introduction

Many scientific applications exhibit the need of communication patterns which involve global data movement and global control [4]. Barrier synchronization, broadcast, gather, scatter, reduce and total exchange are typical examples of collective communication patterns.

Hardware or software support for multicast communication can substantially improve the performance and the resource utilization of a parallel computer. Software overhead accounts for a high percentage of the communication latency, and replacing several point-to-point primitives with a single multicast operation may substantially decrease the communication latency. Furthermore, when a node sends the same message towards several destinations, some of these replicated messages may traverse the same communication channels, generating more traffic then needed.

A common network design trend is to place a communication processor in the network interface [1]. This processor can quickly handle incoming messages and perform simple computations without interacting with the host node [2]. The close integration of these network processors with the capability of performing multicast communication is likely to play an important role in the near future. In fact, the multicast can be enhanced to perform some type of active-message [16] computation on the set of destinations. This creates the opportunity of executing system-level operations to enhance fault-tolerance, for example to check the status of the processing nodes, perform distributed algorithms to balance the load, or to synchronize the local clocks. More generally, these mechanisms can help to integrate the resources in a parallel machine, as if they were a single seamless system.

Hardware support for multicast communication requires many functionalities, that are dependent on the network topology, the routing algorithm and the flow control strategy. For example, in a wormhole network, switches must be capable of forwarding flits from one input channel to multiple output channels at the same time in a tree-like fashion [13]. Unfortunately, these tree-based algorithms can suffer from blocking problems in the presence of congestion [14]. Also, the packets must be able to encode the set of destinations in easy-to-decode, compact manner, in order to reduce the packet size and to guarantee fast routing times in the switches.

Software multicasts, based on unicast messages, are simpler to implement, do not require dedicated hardware and are not constrained by the network topology and routing algorithms, but they can be much slower than the hardware ones.

In this paper we analyze in depth how hardware- and software-based multicasts are designed and implemented in the Quadrics network (QsNET).

The paper is logically divided into two parts. In the first part we analyze the relevant design issues of the network. The list include the main characteristics of the network interface, the communication libraries, how local memories are integrated in a global shared memory, the topology of the interconnection network, the routing algorithm, and the link-level and end-to-end flow control algorithms. This initial part introduces the mechanisms at the base of the hardware and software multicast primitives that, on their turn are at the base of more sophisticated collective communication patterns as broadcasts, barriers, scatter, gather, reduce, etc.

In the second part we provide an extensive performance evaluation of two user-level collective communication patterns, barrier and broadcast, implemented using both hardware and software multicast algorithms. One important contribution of this paper, is the performance evaluation of these algorithms under network congestion.

The rest of this paper is organized as follows. Section 2 provides an overview of the QsNET hardware building blocks and their collective communication capabilities. Section 3 discusses the hierarchy of communication libraries, while Section 4 gives a detailed description of the main collective communication services. The experimental methodology is described in Section 5 and Section 6 presents the experimental results and performance analysis. Finally, in Section 7, some conclusions are drawn.

# 2 The QsNET

The QsNET is based on two building blocks, a programmable network interface called Elan [11] and a low-latency high-bandwidth communication switch called Elite [12]. Elites can be interconnected in a fat-tree topology [6]. The network has several layers of communication libraries which provide trade-offs between performance and ease of use. Other important features are hardware support for collective communication patterns and fault-tolerance.

## 2.1 Elan

The Elan[1] network interface links the high-performance, multi-stage Quadrics network to a processing node containing one or more CPUs. In addition to generating and accepting packets to and from the network, the Elan also provides substantial local processing power to implement high-level message-passing protocols such as MPI. The internal functional structure of the Elan, shown in Figure 1, centers around two primary processing engines: the microcode processor and the thread processor.

The 32-bit microcode processor supports four separate threads of execution, where each thread can independently issue pipelined memory requests to the memory system. Up to eight requests can be outstanding at any given time. The scheduling for the microcode processor is extraordinarily lightweight, enabling a thread to wake up, schedule a new memory access on the result of a previous memory access, and then go back to sleep in as few as two system-clock cycles.

The four microcode threads are described below: (1) *inputter thread:* Handles input transactions from the network. (2) *DMA thread:* Generates DMA packets to be written to the network, prioritizes outstanding DMAs, and time-slices large DMAs so that small DMAs are not adversely blocked. (3) *processor-scheduling thread:* Prioritizes and controls the scheduling and descheduling of the thread processor. (4) *command-processor thread:* Handles operations requested by the host processor at user level.

The thread processor is a 32-bit RISC processor used to aid the implementation of higher-level messaging libraries without explicit intervention from the main CPU. In order to better support this implementation, the thread processor's instruction set was augmented with extra instructions that construct network packets, manipulate events, efficiently schedule threads, and block save and restore a thread's state when scheduling.

The MMU translates 32-bit virtual addresses into either 28-bit local SDRAM physical addresses or 48-bit PCI physical addresses. To translate these addresses, the MMU contains a 16-entry, fully-associative, translation lookaside buffer (TLB) and a small data-path and state machine used to perform table walks to fill the TLB and save trap information when the MMU faults.

The Elan contains routing tables that translate every virtual process number into a sequence of tags that determine

[1]This paper refers to the Elan3 version of the Elan. We will use Elan and Elan3 interchangeably throughout the paper.

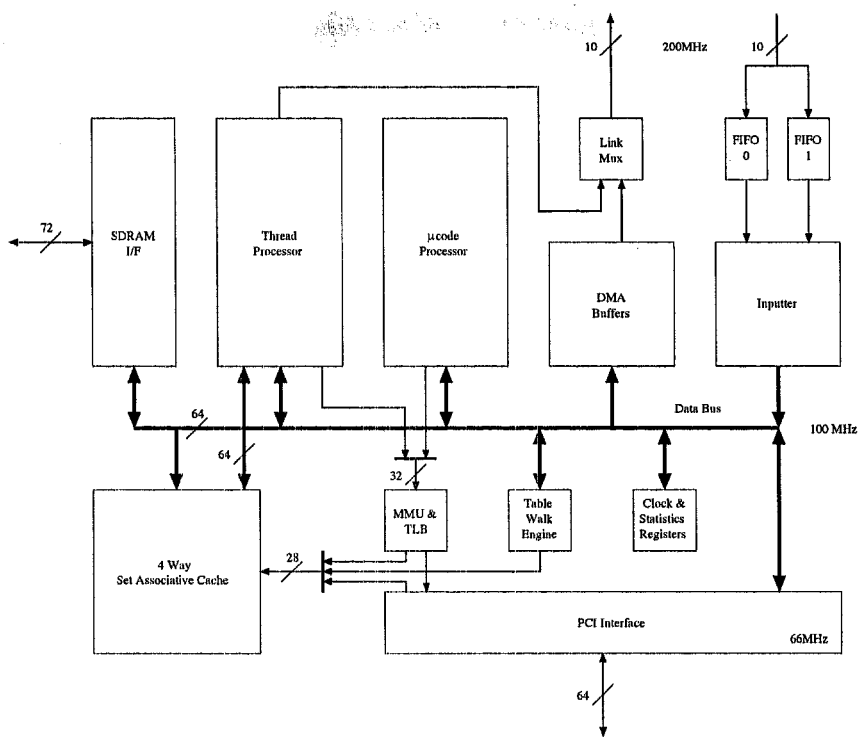



**Figure 1. Elan Functional Units**

the network route. Several routing tables can be loaded in order to have different routing strategies.

The Elan has 8KB of cache memory, organized as 4 sets of 2KB, and 64MB of SDRAM memory. The cache line size is 32 bytes. The cache performs pipelined fills from the SDRAM and is able to issue a number of cache fills and write backs for different units while still being able to service accesses for units that hit on the cache. The interface to the SDRAM has 64 bits and there are 8 check bits added to provide Error Code Correction. The memory interface also contains a 32 byte write buffer and a 32 byte read buffer.

The link logic transmits and receives data from the network and outputs 9 bits and a clock signal on each half of the clock cycle. The flit encoding scheme allows data and command tokens to be interleaved on the link and prevents a corrupted data word being interpreted as a token or a token being interpreted as another token. Each link provides buffer space for two virtual channels with a 128 entry, 16 bit FIFO RAM for flow control.

## 2.2 Elite

The other building block of the QsNET is the Elite switch. The Elite provides the following features: (1) 8 bidirectional links supporting two virtual channels in each direction, (2) an internal 16 × 8 full crossbar switch[2], (3) a nominal transmission bandwidth of 400 MB/s on each link direction and a flow through latency of 35 ns, (4) packet error detection and recovery, with routing and data transactions CRC protected, (5) two priority levels combined with an aging mechanism to ensure a fair delivery of packets in the same priority level, (6) hardware support for broadcasts, (7) and adaptive routing.

The Elite switches are interconnected in a quaternary fat-tree topology, which belongs to the more general class of the $k$-ary $n$-trees [8] [7]. A quaternary fat-tree of dimension $n$ is composed of $4^n$ processing nodes and $n * 4^{n-1}$ switches interconnected as a delta network, and can be recursively build by connecting 4 quaternary fat trees of dimension $n-1$.

Quaternary fat trees of dimension 1, 2 and 3 are shown in Figure 2.

### 2.2.1 Packet Routing and Flow Control

Each user- and system-level message is chunked in a sequence of packets by the Elan. An Elan packet contains three main components. The packet starts with the (1) routing information, that determines how the packet will reach the destination. This information is followed by (2) one or

[2]The crossbar has two input ports for each input link, to accommodate the two virtual channels.

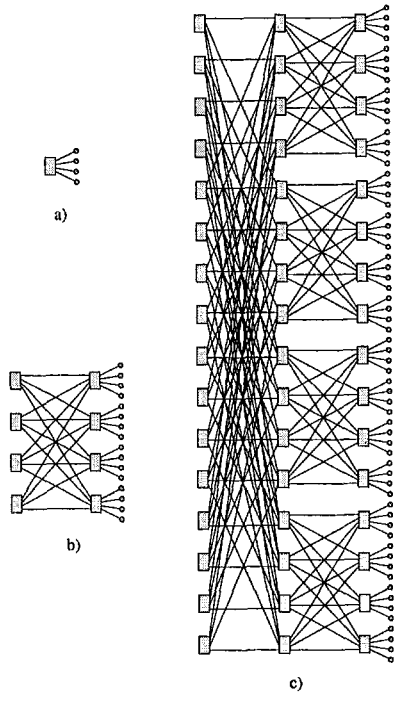



**Figure 2. 4-ary $n$-trees of dimension 1, 2 and 3.**

more transactions consisting of some header information, a remote memory address, the context identifier and a chunk of data, which can be up to 64 bytes in the current implementation. The packet is terminated by (3) an end of packet (EOP) token, as shown in Figure 3.

Transactions fall into two categories: write block transactions and non-write block transactions.

The purpose of a write block transaction is to write a block of data from the source node to the destination node, using the destination address contained in the transaction immediately before the data. A DMA operation is implemented as a sequence of write block transactions, partitioned into one or more packets (a packet normally contains 5 write block transactions of 64 bytes each, for a total of 320 bytes of data payload per packet).

The non-write block transactions implement a family of relatively low level communication and synchronization primitives. For example, non-write block transactions can atomically perform remote test-and-write or fetch-and-add and return the result of the remote operation to the source, and can be used as building blocks for more sophisticated distributed algorithms.

Elite networks are source routed. The routing information is attached to the header before injecting the packet into the network and is composed by a sequence of Elite link tags. As the packet moves inside the network, each Elite

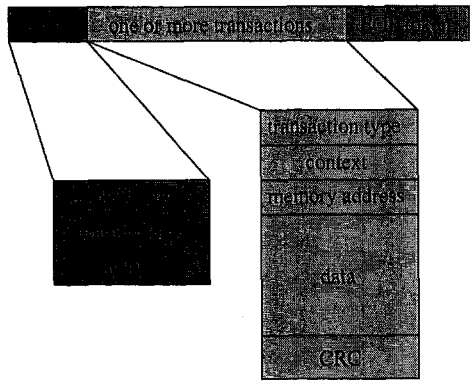



**Figure 3. Packet Transaction Format**

removes the first routing tag from the header, and forwards the packet to the next Elite in the route or to the final destination. The routing tag can identify either a single output link or a group of adjacent links.

The transmission of each packet is pipelined into the network using wormhole flow control. At link level, each packet is partitioned in smaller units called flits (flow control digits) [3] of 16 bits. The header flit opens a circuit between source and destination, and this path stays in place until the destination sends an acknowledgement to the source. At this point, the circuit is closed by sending and End Of Packet (EOP) token. It is worth noting that both acknowledgement and EOP can be tagged to communicate control information. So, for example, the destination can notify the successful completion of a remote non-write block transaction without explicitly sending an extra packet.

Minimal routing between any pair nodes can be accomplished by sending the message to one of the nearest common ancestors and from there to the destination. That is, each packet experiences two routing phases, an adaptive ascending phase to get to a nearest common ancestor, followed by a deterministic descending phase. The Elite switches can adaptively route a packet picking the least loaded link.

## 2.3 Collective Communication

Packets can be sent to multiple destinations using either the *hardware* multicast capability of the network or a *software* tree implemented with point-to-point communication between the Elan thread processors.

### 2.3.1 Hardware Multicast

A multicast packet can only take a pre-determined path, in order to avoid deadlocks. In Figure 4 a) it is shown that the top leftmost switch is chosen as the logical root for the collective communication, and every request, in the ascending

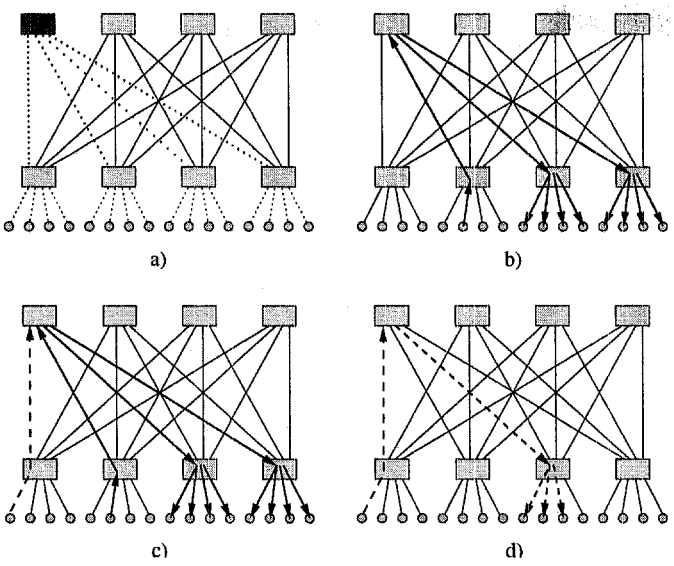



**Figure 4. Hardware Multicast**

phase, must pass through one of the dotted paths until it gets to the root switch. In Figure 4 b) we can see how a multicast packet reaches the root node; the multiple branches are then propagated in parallel. If another collective is issued while the first one is still in progress, it is serialized in the root switch. The second multicast packet will be able to proceed only after an EOP token cleans the circuit of the first communication. All nodes connected to the network are capable of receiving the multicast packet, as long as the multicast set is physically contiguous.

For a multicast packet to be successfully delivered, a positive acknowledgement must be received from all the recipients of the multicast group. The Elite switches combine the acknowledgements [9] returning a single one to the source. Acknowledgements are combined in a way that the "worst" ack wins (a network error wins over an unsuccessful transaction, which on its turn wins over a successful one), returning a positive ack only when all the partners in the collective communication complete the distributed transaction with success.

### 2.3.2 Software Tree

The Elan thread processor can receive an incoming packet, do some basic processing (such as an atomic increment of a variable) and send one or more replies in few $\mu$s, without any interaction with the main processors. Software collectives can be implemented using the communication and computation capability of the Elan thread processor, for example multicast trees. Software collectives can be based on trees with programmable arity, depth and regularity, and do not suffer from the limitation that the destination set must be composed of adjacent nodes.

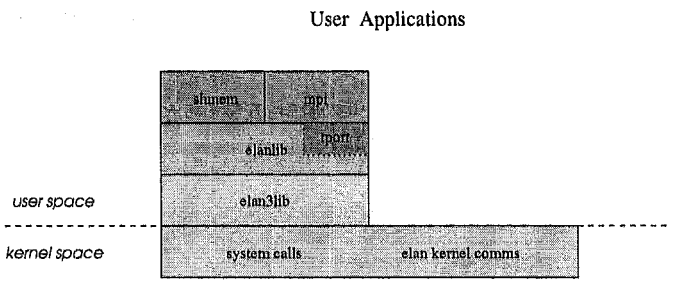



**Figure 5. Elan3 programming library hierarchy**

## 3 Programming libraries

The Elan network interface can be programmed using several programming libraries [10], as outlined in Figure 5. These libraries trade speed with machine independence and programmability. Starting from the bottom, Elan3lib is the lowest programming level available in user space which allows the access to the low level features of the Elan3. At this level, processes in a parallel job can communicate with each other through an abstraction of distributed virtual shared memory. Each process in a parallel job is allocated a virtual process id (VPID) and can map a portion of its address space into the Elan. These address spaces, taken in combination, constitute a distributed virtual shared memory. Remote memory (i.e., memory on another processing node) can be addressed by a combination of a VPID and a virtual address. Since the Elan has its own MMU, a process can select which part of its address space should be visible across the network, determine specific access rights (e.g. write- or read-only) and select the set of potential communication partners.

Elanlib is a higher level layer that frees the programmer from the revision-dependent details of the Elan, and extends Elan3lib with point-to-point, tagged message passing primitives (called Tagged Message Ports or Tports) and support for collective communication. Standard communication libraries as such MPI-2 [5] or Cray Shmem are implemented on top of Elanlib.

### 3.1 Elan3lib

The Elan3lib library supports a programming environment where groups of cooperating processes can transfer data directly, while protecting process groups from each other in hardware. The communication takes place at user level, with no data copying, bypassing the operating system. The main features of Elan3lib are: (1) event notification, (2) the memory mapping and allocation scheme and (3) remote DMA transfers.

### 3.1.1 Event Notification

Events provide a general purpose mechanism for processes to synchronize their actions. The mechanism can be used by threads running on the Elan and processes running on the main processor. Events can be accessed both locally and remotely. Thus, processes can be synchronized across the network, and events can be used to indicate the end of a communication operation, such as a completion of a remote DMA. Events are stored in Elan memory, in order to guarantee the atomic execution of the synchronization primitives[3]. Processes can wait for an event to be triggered by blocking or polling. In addition, an event can be tagged as being a block copy event. The block copy mechanism works as follows. A block of data in Elan memory is initialized to hold a pre-defined value. An equivalent sized block is located in main memory, and both are in the user's virtual address space. When the specified event is set, for example when a DMA transfer has completed, a block copy takes place. That is, the block in Elan memory is copied to the block in main memory. The user process polls the block in main memory to check its value, (for example, bringing a copy of the corresponding memory block into the L2 cache) without having to poll for this information across the PCI bus. When the value is the same as that initialized in the source block, the process knows that the specified event has happened.

### 3.1.2 Memory Mapping and Allocation

The MMU in the Elan can translate between virtual addresses written in the format of the main processor (for example, a 64-bit word, big Endian architecture as the AlphaServer) and virtual addresses written in the Elan format (a 32-bit word, little Endian architecture). For a processor with a 32-bit architecture (for example an Intel Pentium), a one-to-one mapping is all that is required.

In Figure 6 the mapping for a 64-bit processor is shown. The 64-bit addresses starting at 0x1FF0C808000 are mapped to Elan's 32 bit addresses starting at 0xC808000. This means that virtual addresses in the range 0x1FF0C808000 to 0x1FFFFFFFFFF can be accessed directly by the main processor while the Elan can access the same memory by using addresses in the range 0xC808000 to 0xFFFFFFFF. In our example, the user may allocate main memory using malloc and the process heap may grow outside the region directly accessible by the Elan delimited by 0x1FFFFFFFFFF. In order to avoid this problem, both main and Elan memory can be allocated using a consistent memory allocation mechanism. As shown in Figure 6 the MMU tables can be set up to map a common region of virtual memory called *memory allocator heap*. The allocator maps physical pages, of either main or Elan memory into this virtual address range on demand. Thus, using allocation functions provided by the Elan library, portions of virtual memory (1) can be allocated either from main or Elan memory, and (2) the MMUs of both main processor and Elan can be kept consistent.

For efficiency reasons, some objects can be located on the Elan, for example communication buffers or DMA descriptors which the Elan can process independently of the main processor.

[3]The current PCI bus implementations cannot guarantee atomic execution, so it is not possible to store events in main memory.

### 3.1.3 Remote DMA

The Elan supports remote DMA (Direct Memory Access) transfers across the network, without any copying, buffering or operating system intervention. The process that initiates the DMA fills out a DMA descriptor, which is typically allocated on the Elan memory for efficiency reasons. The DMA descriptor contains the VPIDs of both source and destination, the amount of data, the source and destination addresses, two event locations (one for the source and the other for the destination process) and other information used to enhance fault tolerance. The typical steps of remote DMA are outlined in Figure 7.

## 3.2 Elanlib and Tports

Elanlib is a machine independent library that integrates the main features of Elan3lib with the Tports. Tports provide basic mechanisms for point-to-point message passing. Senders can label each message with a tag, the sender identity and the size of the message. This is known as the *envelope*. Receivers can receive their messages selectively, filtering them according to the identity of the sender and/or a tag on the envelope. The Tport layer handles communication via shared memory for processes on the same node. It is worth noting that the Tports programming interface is very similar to MPI [15].

Elanlib provides support for collective communication operations (those that involve a group of processes). The most important collective communication primitives implemented in Elanlib are: (1) the barrier synchronization and (2) the broadcast.

# 4 Barrier Synchronization and Broadcast

## 4.1 Barrier Synchronization

A synchronization barrier is a logical point in the control flow of a parallel program at which all processes in a group must arrive before any of the processes in the group are allowed to proceed. Typically, a barrier synchronization

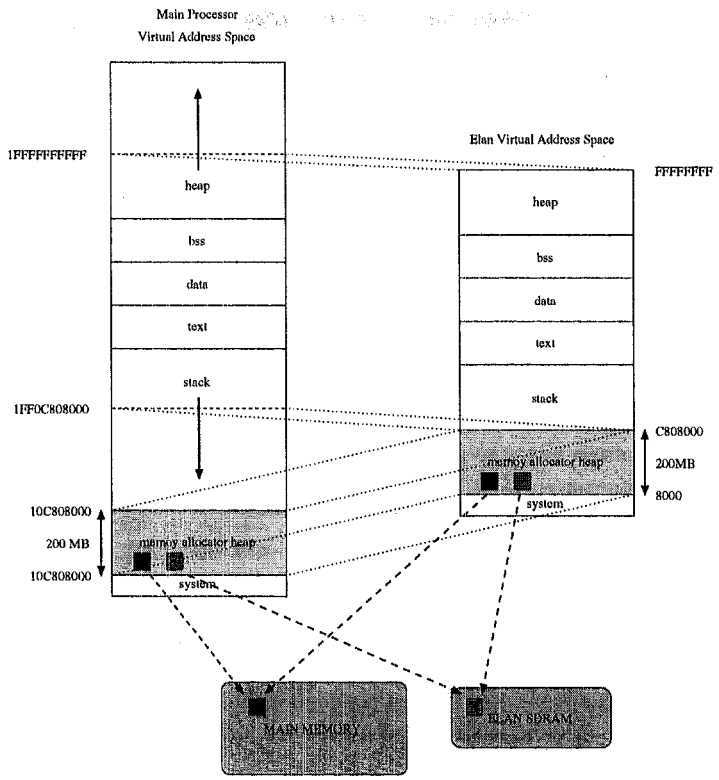



**Figure 6. Virtual Address Translation**

involves a logical reduce operation followed by a broadcast operation.

QsNET implements two different synchronization mechanisms in Elanlib, a mixed software and hardware barrier called `elan_gsync()` and a purely hardware one called `elan_hgsync()`.

The algorithm implemented with `elan_gsync()` uses a balanced tree to send the 'ready' signal to the process with virtual process identifier 0. Each process in the tree waits for 'ready' signals from its children, and when it receives all of them sends its own signal up to the parent process. This phase of the barrier is illustrated in Figure 8. When the root process receives its 'ready' signals it performs a hardware broadcast 'go' packet which either sets an event (which all processes are waiting for) or writes a single word in a given memory location (which all processes are polling). If the destination nodes are not adjacent the same tree structure is used to distribute the data using point-to-point messages.

The `elan_hgysnc()` implementation (used by MPI_Barrier [15]) uses an Elan thread to send a special test-and-set broadcast packet. This packet spans all the processes and compares a barrier sequence value with a given memory location (which is updated by every process in the group when the process reaches the barrier) to see if it matches; if it does, the packet can then set an event

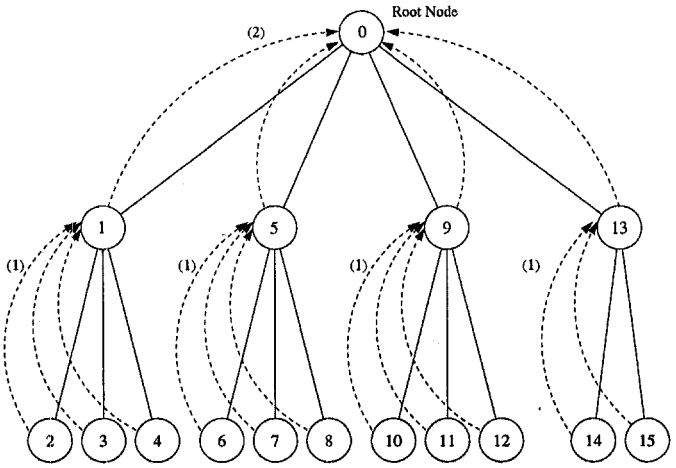



**Figure 8. Implementation of the first phase of the elan_gsync() barrier for a group of 16 processes. Each process in the group (1) waits for the 'ready' packet from its children processes and, then (2) sends its own 'ready' signal up to its parent.**

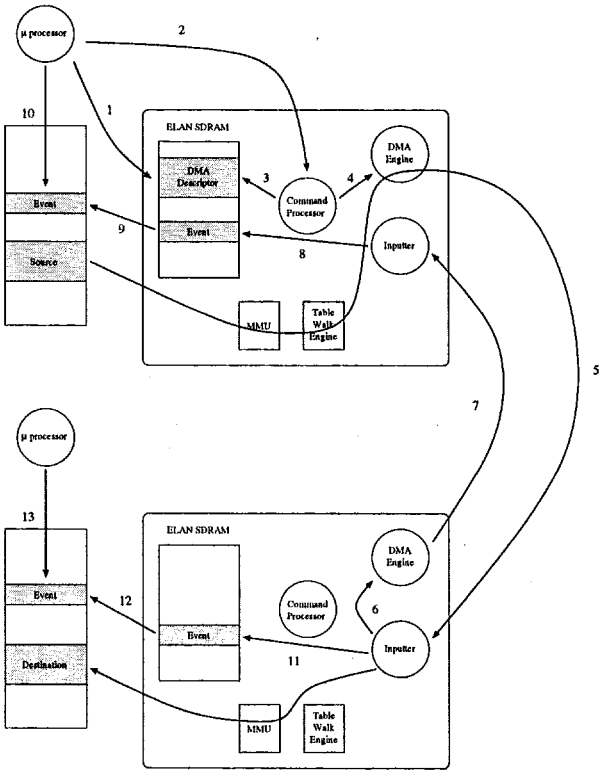



**Figure 7. Execution of a Remote DMA. The sending process (1) initializes the DMA descriptor in the Elan memory and (2) communicates the address of the DMA descriptor to the command processor. The command processor (3) checks the correctness of the DMA descriptor and (4) adds it to the DMA queue. The DMA engine (5) performs the remote DMA transaction. Upon completion the remote inputter (6) notifies the DMA engine which (7) sends an ack to the source Elan. Source (8-10) and destination (11-13) events can be notified, if needed.**

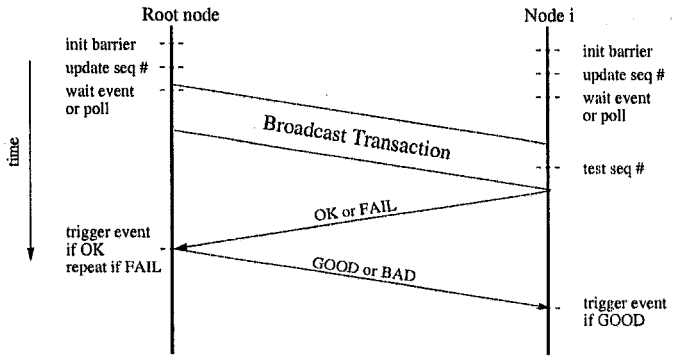



**Figure 9. elan_hgsync() barrier implementation. Each node sets the sequence number of the barrier and either waits for an event or polls a memory location waiting for the correct sequence number. The root node of the group sends a broadcast packet which contains a conditional transaction that tests if each node has already set the sequence number (reached the barrier). All the replies are combined by the Elites on the way back to the root node which receives a single ACK token. If all the nodes are ready an EOP token is sent to the group to complete the barrier.**

or write a word to wake up the processes waiting in the barrier. This gives the best figures as long as the processes enter the barrier fairly close together, otherwise it backs off exponentially (to stop flooding the network with broadcasts). Figure 9 shows a detailed timing diagram of this barrier mechanism. The waiting and waking up operation can be configured to use a busy polling of each process on a memory location or to use an event mechanism (using a different function called `elan_hgysncEvent()`).

## 4.2 Broadcast

The major network communication primitive of the Quadrics interconnection network is the remote DMA. A DMA operation transfers data between the local and remote address space (including Elan memory). In addition to providing point-to-point communication, DMAs can also be used to perform group-wide operations such as broadcast and flood DMAs (a flood is similar to a broadcast but the operation completes as soon as any of the destinations accepts the DMA). A group of destination processes is defined by specifying a virtual group identifier. The effect of a write broadcast DMA is to copy the data from the source to the destination buffers in all of the processes in the group. The basic implementation of the broadcast DMAs relies on all receiving processes having the destination buffer at the same virtual address, to obtain good performance.

QsNET provides broadcast hardware support that should send a broadcast message in the same time required to send a point-to-point message. The network can be considered as a tree of Elite switch chips that connect an array of Elan network interface cards. Broadcasts are propagated into the network by sending a packet to the top of the tree and then copying the packet to more than one switch output as the packet is sent down the tree. Deadlocks might occur on the way down when multiple broadcasts are sent simultaneously. This situation is avoided by sending broadcast packets always to a fixed top tree switch, thus serializing all broadcasts (Section 2.3). All the Elans connected to the network are capable of receiving the broadcast packet, but the hardware mechanism can only be used with a contiguous subset of Elans.

Two different broadcast implementations are provided by the Elanlib library: `elan_bcast()` and `elan_hbcast()`. Both must be called by all the processes in the group involved in the broadcast operation to guarantee that the receivers have allocated the buffers by the time the transaction is performed by the sender process. As a result, the broadcast is composed of two transactions: first, a barrier synchronization and, second, the broadcast itself. In both implementations, two types of memory resources can be used. On the one hand a global destination buffer, which has the same virtual address in all the processes (Elanlib libraries provide special memory allocation functions that guarantee this property), allows to perform a DMA transaction directly from source to destination. On the other hand, if this memory allocation is not used, system buffers are utilized as intermediate copy space from which the DMA is performed (this approach implies one copy at the source, and another copy at the destination).

The `elan_bcast()` implementation uses a software-based synchronization for the first phase similar to that utilized by the first phase of `elan_gsync()` (Section 4.1). The second phase is triggered by an event set in the source node and is done using the hardware broadcast mechanism (if all the destination Elans are contiguous) or by means of a software-based broadcast from the source node (if the destination Elans are not). This transaction distributes the data and wakes up the processes waiting in the barrier performed during the first phase. This implementation provides better performance than a call to `elan_gsync()` (which involves a software-based synchronization and a broadcast) and a later broadcast to send the data.

The `elan_hbcast()` primitive calls `elan_bcast()` if the hardware broadcast mechanism is not available, for example when the nodes are not contiguous. If this mechanism is available, it performs a barrier to synchronize all the nodes using `elan_hgsync()` (Section 4.1) and a hardware broadcast to distribute the

data.

The Elan hardware broadcast can only write to the memory space of a single process per node since there is only a single context specified by the virtual process identifier. Hence, with multiple processes per node, the only way to use the hardware broadcast facility is to broadcast into an area of shared memory and then get the processes to copy from there. This has been optimized by using a FIFO like scheme that tries to overlap the broadcast with the copies.

# 5 Experimental Framework

The main features of the QsNET have been tested on a 64-node cluster of Compaq AlphaServer ES40s, running Tru64 Unix. Each AlphaServer node is equipped with 4 Alpha 667MHz 21264 processors, 8GB of SDRAM and two 64 bits, 33MHz PCI I/O buses. The Elan3 PCI card QM-400 is attached to one of them and links the SMP to a quaternary fat tree of dimension three, like the one shown in Figure 2 c).

## 5.1 Unidirectional Ping

We analyze the latency and the bandwidth of the network by sending messages of increasing size from a source to a destination SMP. In order to identify different bottlenecks, the communication buffers are placed either in main or in Elan memory. The alternatives include main memory to main memory and Elan memory to Elan memory. These buffers are placed in the desired type of memory using the allocation mechanisms provided by Elan3lib, as described in Section 3.1.

The latency is measured as the elapsed time between the posting of the remote DMA request and the notification of the successful completion at the destination (steps 2 through 5 and 11 through 13 in Figure 7). The unidirectional ping tests for MPI are implemented using matching pairs of blocking sends and receives. These tests provide a performance reference to consistently analyze the results on collective communication.

## 5.2 Collective Communication

The barrier synchronization and broadcast primitives provided by the QsNET system software have been tested using configurations ranging from 4 to 64 nodes. Results have been obtained by averaging the results over 10000 consecutive tests. Average latency results and latency distribution are reported for the barrier synchronization tests. For the broadcast tests bandwidth and latency are reported.

In addition, tests with background traffic have been performed to analyze the behavior of the collective communications under network contention. This traffic is generated by 64 processes running in 64 nodes, with all nodes injecting messages into the network at maximum load. These tests can identify performance limitations due to the interference by other applications. Two different traffic patterns were used to generate background traffic:

- Complement. The node with binary coordinates $a_{n-1}, a_{n-2}, \ldots, a_1, a_0$ communicates with the node $\overline{a_{n-1}}, \overline{a_{n-2}}, \ldots, \overline{a_1}, \overline{a_0}$. This pattern uses all the network links at the same time.

- Uniform. Each node selects randomly its destination for every single transaction.

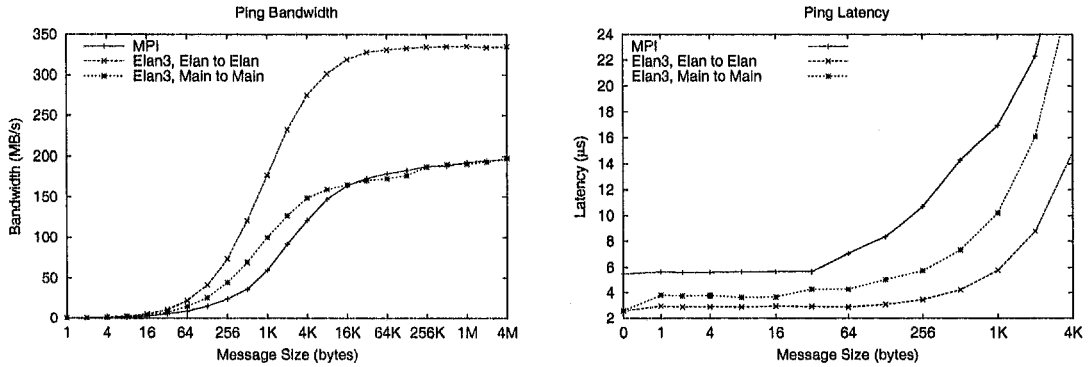
To guarantee that the performance degradation of the collective communication is only due to the network contention and not to scheduling issues, the background traffic generation and the collective communication benchmark were run in different processors.

# 6 Experimental Results

## 6.1 Unidirectional Ping

Figure 10 a) shows the performance of the unidirectional ping. The peak bandwidth of 335 MB/s is reached when both source and destination buffers are placed in the Elan memory. The maximum amount of data payload that can be sent by the current Elan implementation is 320 bytes, partitioned in five low-level write-block transactions of 64 bytes. For this message format, the overhead is 58 bytes, for the message header, CRCs, routing info, etc. This implies that the peak bandwidth delivered by the network is approximately 396 MB/s, or 99% of the nominal bandwidth (400 MB/s). The asymptotic bandwidth for main memory to main memory communication is only 200MB/s for both Elanlib and MPI. These results also show that the PCI interface running at 33MHz is the bottleneck for this type of communication.

Figure 10 b) shows the latency in the range $[0 \ldots 4KB]$. With Elan3lib the basic latency for 0-byte messages is only 2.2 $\mu$s and is almost constant at 2.4 $\mu$s for messages up to 64 bytes, because these messages can be packed as a single write-block transaction. We note an increase in the latency at MPI level, compared to the latency at the Elan3lib level, from approximately 2 $\mu$s to 5.5 $\mu$s. While at Elan3lib level the latency is mostly hardware, MPI needs to run a thread in the Elan microprocessor in order to match the message tags: this introduces the extra overhead responsible for the higher latency value.




(a) (b)

**Figure 10. Unidirectional ping**

## 6.2 Collective Communications

### 6.2.1 Barrier Synchronization

Figure 11 shows the average time required to perform a barrier synchronization when there is no other traffic in the network. Results for the three Elanlib primitives (Section 4.1) are shown versus the number of nodes. We can see that the hardware-based implementations of the barrier (`elan_hgsync()` and `elan_hgsyncEvent()`) provide the best results when compared to the software-based implementation (`elan_gsync()`), both in absolute performance and in scalability. The latency of the software-based implementation grows as the logarithm of the number of nodes (approximately 2.5$\mu$s each time the number of nodes is quadrupled). In this case the average latency to synchronize 64 nodes is 14.8$\mu$s. On the other hand, the `elan_hgsync()` barrier provides an average latency of 5$\mu$s below 16 nodes and 5.5$\mu$s and 6$\mu$s for 32 and 64 nodes respectively. The `elan_hgsyncEvent()` synchronization gives latencies on average 0.7$\mu$s above those obtained with `elan_hgsync()`. This is due to the additional delay associated with the event notification. In both thread based barriers, the latency increase above 16 nodes is probably due to scheduling issues on the OS (Tru64 Unix). Better scalability could be obtained by synchronizing all the Unix schedulers against the Elan clock.

The behavior of the barrier synchronization has been analyzed by performing tests with uniform and complement background traffic. The results depicted in Figure 12 show that the performance of the various barrier implementations is affected by the network traffic with higher degradation when uniform background traffic, which produces higher network contention, is used. In fact, with complement traffic there is always one virtual channel available in each link.

Figure 13 shows the latency distribution of

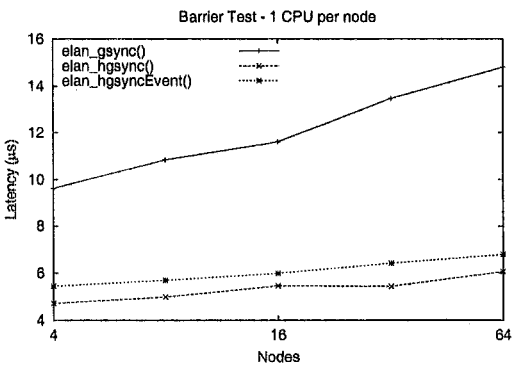



**Figure 11. Barrier Synchronization**

`elan_hgsync()` in a 64-node configuration. Only 2% of the operations take more than 20$\mu$s and 94% less than 9$\mu$s when there is no background traffic. In the worst case the average latency for 64 nodes is 13$\mu$s, with more than 93% of the barriers taking less than 20$\mu$s. Similar results were obtained with the `elan_hgsyncEvent()` primitive.

The latency distribution for the software-based implementation of the barrier synchronization is shown in Figure 14. Only 1% of the barriers take more than 30$\mu$s when there is no network contention. In the presence of network contention, `elan_gsync()` suffers a significant degradation in performance. In the worst case (uniform background traffic) an average latency of 595$\mu$s is obtained and 93% of the synchronizations complete with latencies below 605$\mu$s.

### 6.2.2 Broadcast

Figure 15 shows the results obtained with broadcast over 64 nodes using both algorithms supported by Elanlib (Sec-

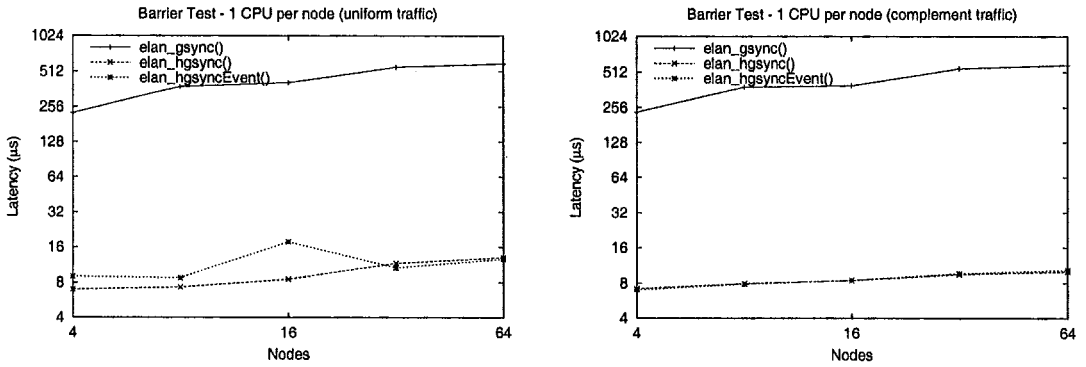



**Figure 12. Barrier Synchronization with Contention**

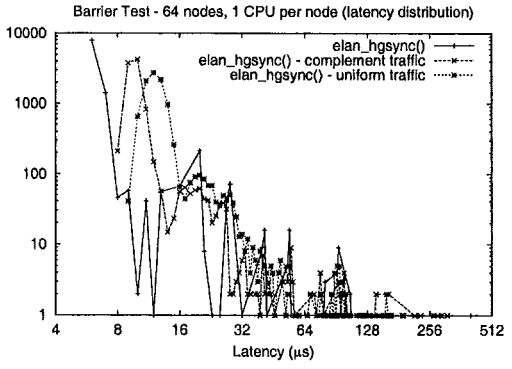



**Figure 13. elan_hgsync() Latency Distribution with Contention**

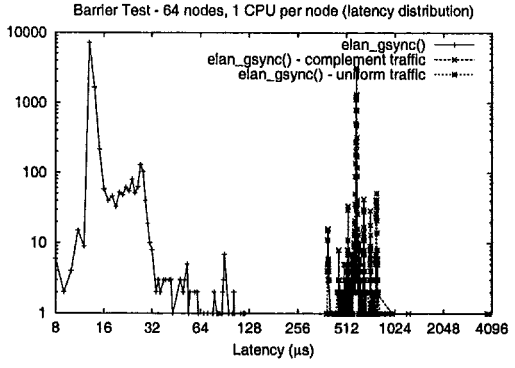
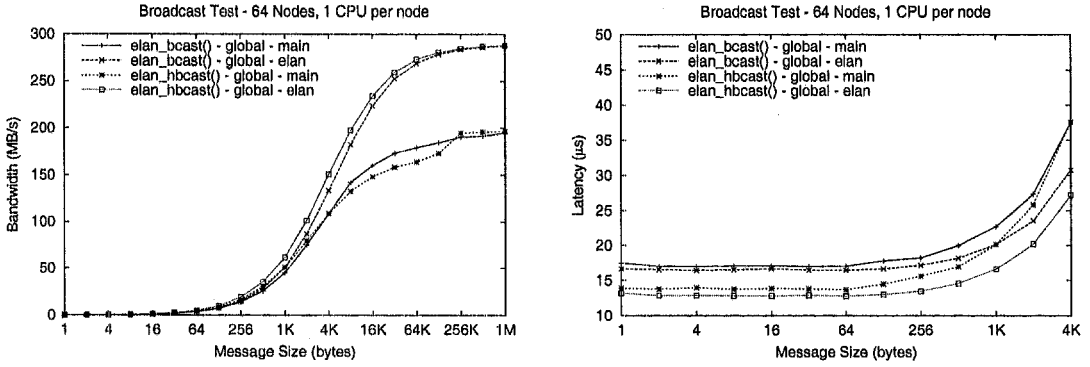



**Figure 14. elan_gsync() Latency Distribution with Contention**

tion 4.2) with buffers globally allocated in main and Elan memory, that is with the same virtual address in all processes. The best performance is obtained, as expected, with Elan memory. In this case the measured bandwidth for 1MB messages is 288MB/s for both `elan_bcast()` and `elan_hbcast()`. The `elan_hbcast()` primitive provides lower latencies (3.5$\mu$s) because it uses hardware-based synchronization rather than a software-based one, used by `elan_bcast()`. For this reason the bandwidth for shorter messages is slightly higher with `elan_hbcast()`. For messages up to 256 bytes the latency is constant and approximately 13$\mu$s for `elan_hbcast()` and 16.5$\mu$s for `elan_bcast()`. This is due to the fact that messages shorter than 320 bytes are sent using a single packet [11].

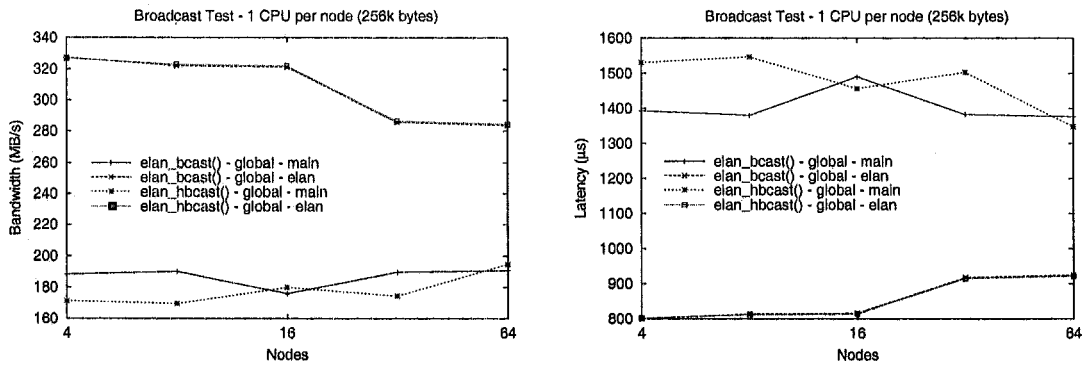
Bandwidth and latency versus the number of nodes for 256KB messages are depicted in Figure 16. Both performance metrics are insensitive to the number of nodes when the buffers are allocated in main memory, because the PCI bus is the bottleneck in this case. On the other hand, when Elan memory is used, a performance degradation occurs when the number of nodes increases above 16 (8% decrease in bandwidth and 12% increase in latency). A similar, albeit lower, effect is experienced when the number of nodes is increased above 4 (1% differences in bandwidth and latency).

In the presence of network contention (Figure 17) the performance of the broadcast decreases significantly. The maximum bandwidth is obtained using main memory allocation. This is caused by the job running in the background which allocates its communication buffers in Elan memory. This configuration gives 36MB/s with complement background traffic and 24MB/s with uniform background traffic using 1MB messages. Although both broadcast implementations provide approximately the same maximum bandwidth (with 1MB messages), the `elan_hbcast()` primi-




(a) (b)

**Figure 15. Broadcast**




(a) (b)

**Figure 16. Broadcast Scalability**

tive obtains better performance for smaller messages thanks to the hardware synchronization mechanism (Figures 17 (b) and (d)), which is less sensitive to additional network traffic (Section 6.2.1).

In terms of scalability (Figure 18) the four alternatives suffer from the same performance degradation as the number of nodes increases. This effect slows down with the increase in the number of nodes. This behavior suggests that no additional significant performance decrease should be produced by networks larger than 64 nodes.

## 7 Conclusion

In this paper, we presented an in-depth description of the Quadrics interconnection network (QsNET) with special emphasis on the support for collective communication and its integration with the system software. We focused our attention on two basic communication patterns: barrier synchronization and broadcast. An experimental evaluation of hardware-based and software-based implementations of these services has been performed on a 64-node AlphaServer cluster.

Our experiments show that the time to complete a hardware-based barrier synchronization on the whole set of nodes is as low as $6\mu s$, with very good scalability for the network configurations tested. Good latency and scalability are also achieved with the software-based synchronization, which completes in $15\mu s$.

Another important contribution of this paper is the analysis of the collectives in the presence of network contention. In this case, the average latency for the hardware-based barrier is $13\mu s$, with 93% of the synchronizations taking less than $20\mu s$. On the other hand, the software-based implementation is shown to suffer a significant performance degradation. From a practical point of view the hardware-based barrier can be considered insensitive to the network contention.

With the broadcast, similar results have been obtained for the hardware-based and the software-based implementations in the absence of additional network traffic. These results show that without contention the two algorithms can be used interchangeably. The broadcast latency for messages up to 256 bytes is $13\mu s$ and the bandwidth is 288MB/s. Contention tests, done in the presence of extremely high network load, show that the broadcast maintains reasonably good performance (i.e. less than $200\mu s$ to deliver messages up to 2KB). In this case the hardware-based broadcast outperforms the software-based broadcast thanks to its hardware-based synchronization mechanism.

Overall, our analysis shows the potential of the interconnect to efficiently support large scale collective communication patterns, even in the presence of high network contention.

As future work, we plan to address the problem of the serialization of the hardware broadcasts on the root node and to study how collective communication can be furtherly integrated with the network processor.

## Acknowledgements

The authors would like to thank the Quadrics team, David Addison, Jon Beecroft, Robin Crook, Moray McLaren, David Hewson, Duncan Roweth and John Taylor, for their invaluable support.

## References

[1] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawick, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, January 1995.

[2] Darius Buntinas, Dhabaleswar Panda, and P. Sadayappan. Performance Benefits of NIC-Based Barrier on Myrinet/GM. In *Workshop on Communication Architecture for Clusters (CAC '01)*, San Francisco, CA, April 2001.

[3] William J. Dally and Charles L. Seitz. Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.

[4] José Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks: an Engineering Approach*. IEEE Computer Society Press, 1997.

[5] Al Geist, William Gropp, Steve Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, William Saphir, Tony Skjellum, and Marc Snir. MPI-2: Extending the Message Passing Interface. In *Second International Euro-Par Conference, Volume I*, number 1123 in LNCS, pages 128–135, Lyon, France, August 1996.

[6] Charles E. Leiserson. Fat-Trees: Universal Networks for Hardware Efficient Supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[7] Fabrizio Petrini and Marco Vanneschi. $k$-ary $n$-trees: High Performance Networks for Massively Parallel Architectures. In *Proceedings of the 11th International Parallel Processing Symposium, IPPS'97*, pages 87–93, Geneva, Switzerland, April 1997.

[8] Fabrizio Petrini and Marco Vanneschi. Performance Analysis of Wormhole Routed $k$-ary $n$-trees. *International Journal on Foundations of Computer Science*, 9(2):157–177, June 1998.

[9] G. F. Pfister and V. A. Norton. Hot-spot Contention and Combining in Multistage Interconnection Networks. *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[10] Quadrics Supercomputers World Ltd. *Elan Programming Manual*, January 1999.

[11] Quadrics Supercomputers World Ltd. *Elan Reference Manual*, January 1999.

[12] Quadrics Supercomputers World Ltd. *Elite Reference Manual*, November 1999.

[13] Rajeev Sivaram, Dhabaleswar Panda, and Craig Stunkel. Efficient Broadcast and Multicast on Multistage Interconnection Networks using Multiport Encoding. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*, New Orleans, LA, October 1996.

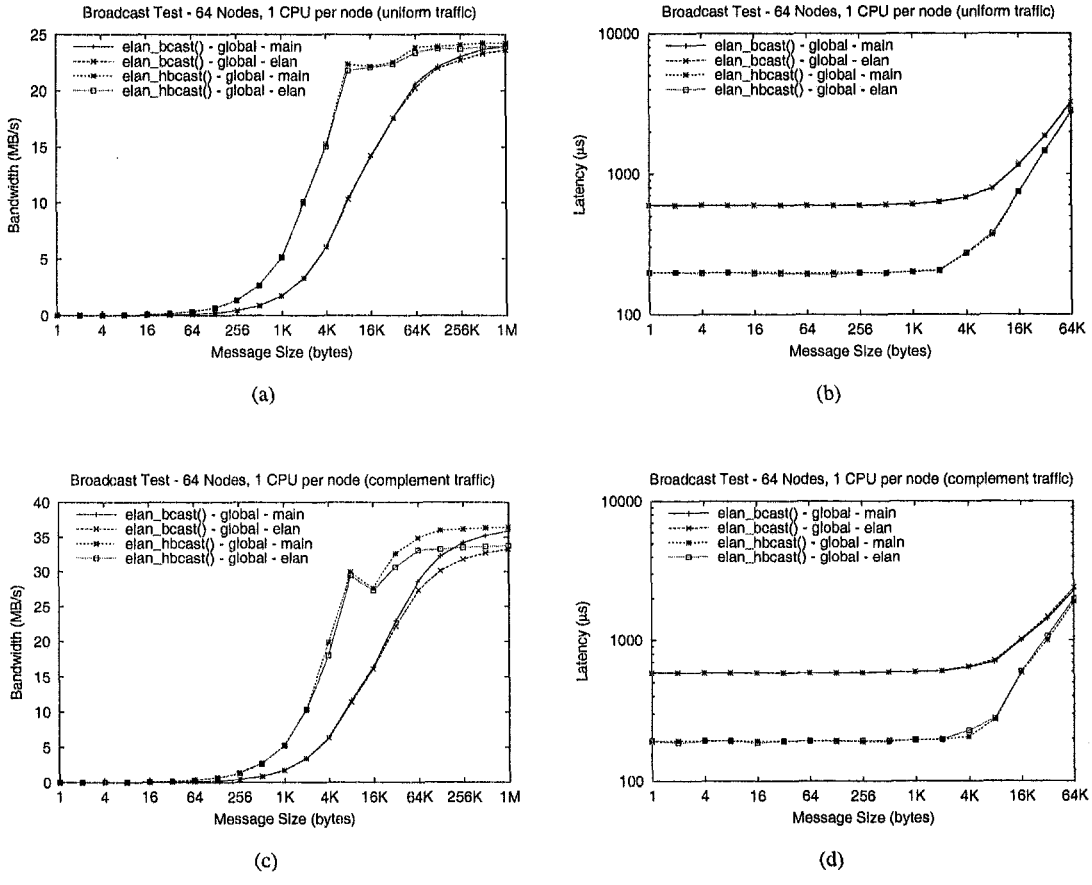



**Figure 17. Broadcast with Contention**

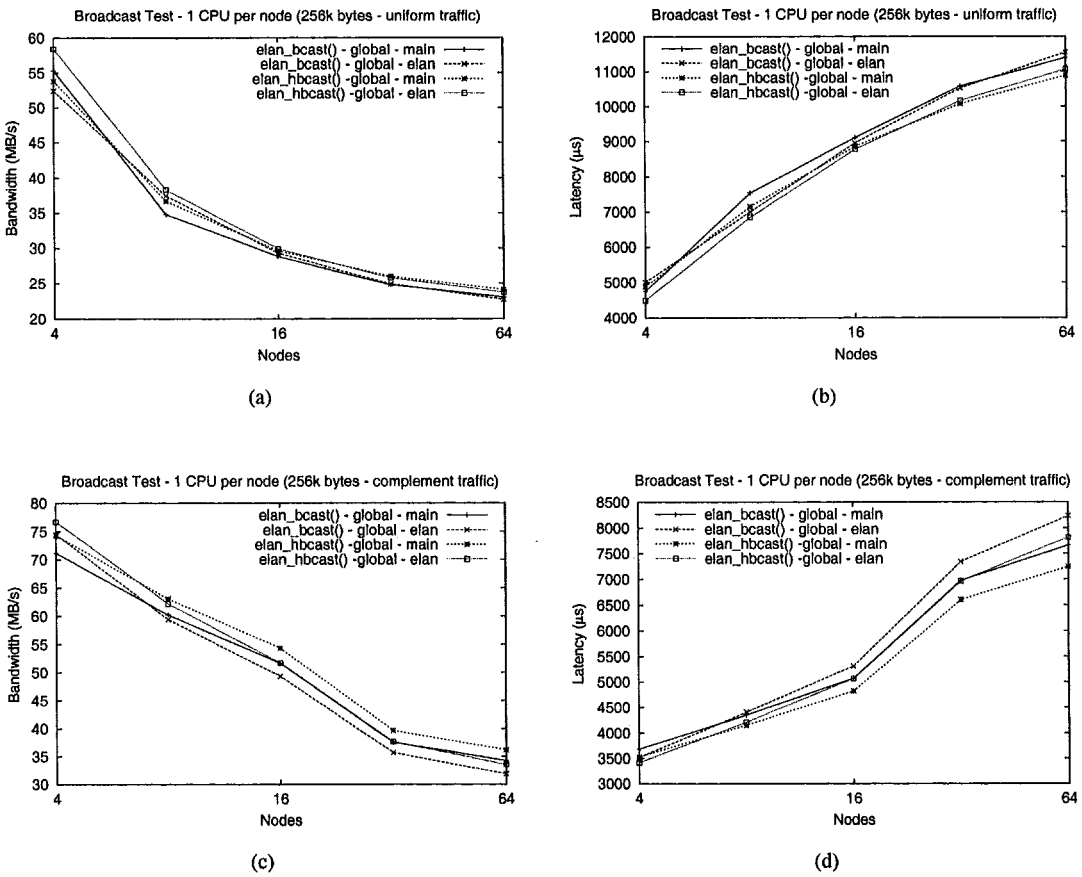



**Figure 18. Broadcast Scalability with Contention**

[14] Rajeev Sivaram, Dhabaleswar Panda, and Craig Stunkel. Multicasting in Irregular Networks with Cut-Through Switches using Tree-Based Multidestination Worms. In *Parallel Computing, Routing, and Communication Workshop, PCRCW'97*, Atlanta, GA, June 1997.

[15] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI - The Complete Reference*, volume 1, The MPI Core. The MIT Press, 1998.

[16] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.