

# Cluster Communication Protocols for Parallel-Programming Systems

KEES VERSTOEP, RAOUL A. F. BHOEDJANG, TIM RÜHL,  
HENRI E. BAL, and RUTGER F. H. HOFMAN

Vrije Universiteit

---

Clusters of workstations are a popular platform for high-performance computing. For many parallel applications, efficient use of a fast interconnection network is essential for good performance. Several modern System Area Networks include programmable network interfaces that can be tailored to perform protocol tasks that otherwise would need to be done by the host processors. Finding the right trade-off between protocol processing at the host and the network interface is difficult in general. In this work, we systematically evaluate the performance of different implementations of a single, user-level communication interface. The implementations make different architectural assumptions about the reliability of the network and the capabilities of the network interface. The implementations differ accordingly in their division of protocol tasks between host software, network-interface firmware, and network hardware. Also, we investigate the effects of alternative data-transfer methods and multicast implementations, and we evaluate the influence of packet size. Using microbenchmarks, parallel-programming systems, and parallel applications, we assess the performance of the different implementations at multiple levels. We use two hardware platforms with different performance characteristics to validate our conclusions. We show how moving protocol tasks to a relatively slow network interface can yield both performance advantages and disadvantages, depending on specific characteristics of the application and the underlying parallel-programming system.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*network communications*; C.4 [**Performance of Systems**]: *design studies; performance attributes*; D.1.3 [**Programming techniques**]: Concurrent Programming—*parallel programming*

General Terms: Performance, Design, Experimentation

Additional Key Words and Phrases: Clusters, parallel-programming systems, system area networks

---

## 1. INTRODUCTION

Modern custom network hardware allows latencies of only a few microseconds and throughputs of over a Gigabit per second, but such performance is rarely

---

Part of this research was performed while R. A. F. Shoedjang was at Cornell University. Authors' address: Vrije Universiteit, Faculty of Sciences, Department of Computer Science, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands. Authors' email addresses: versto@cs.vu.nl; raoul@holmes.nl; t.ruhl@datadistilleries.com; bal@cs.vu.nl; rutger@cs.vu.nl.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).

© 2004 ACM 0734-2071/04/0800-0281 \$5.00

attained on top of standard Application Programming Interfaces (APIs) such as TCP/IP, MPI, SunRPC or Java RMI. To close the gap between hardware capabilities and measured performance, many “lean” communication systems have been developed that provide fast low-level packet-based communication between user processes, avoiding the operating system kernel on the critical path. These systems are typically used on clusters, often in a parallel-programming setting. They differ widely in how they are implemented, especially on networks that have programmable interfaces (e.g., Myrinet [Boden et al. 1995]). The design and implementation choices in which the communication systems differ include the communication protocols (e.g., for reliability or multicast), optimizations (e.g., zero-copy), and parameter settings (e.g., maximum transfer unit).

Unfortunately, it is difficult to determine the performance impact of these implementation decisions. The communication systems usually implement incompatible low-level APIs, making it impossible to compare them using identical microbenchmarks. Even worse, the relation between low-level decisions and high-level application performance is poorly understood. The goal in this paper is to do a systematic performance evaluation and comparison of these implementation techniques. A key idea is to use a single low-level communication API that supports a variety of parallel-programming systems, each with a different interface to the programmer. Since the low-level API can be *implemented* in different ways, we are able to accurately study the impact of these implementation decisions both on low-level communication performance and on high-level application performance (speedup).

Our results confirm that the performance indicated by commonly used low-level benchmarks can be a poor predictor for application performance. On the one hand, programming systems add significant layering overheads that can hide low-level differences, especially for latency. On the other hand, application performance often depends on issues such as flow control or the load (i.e., occupancy, or overhead in LogP [Culler et al. 1996] terms) that a protocol puts on the host or the network interface. These performance aspects are usually not directly visible from the most commonly used low-level point-to-point benchmarks. Also, application-level performance is influenced by the communication patterns induced by parallel-programming systems, and these patterns can be very different for, say, message-passing libraries, Distributed Shared Memory (DSM) systems, or object-based languages. We therefore study multiple programming paradigms when examining the high-level impact of low-level changes.

This article focuses on four implementation choices for communication systems:

- reliability guarantees;
- the data-transfer mechanism (message passing and remote-memory copy);
- maximum transfer unit (MTU);
- multicast.

Reliable communication is required in all our parallel-programming systems and most of them rely on a low-level reliable communication service. We study

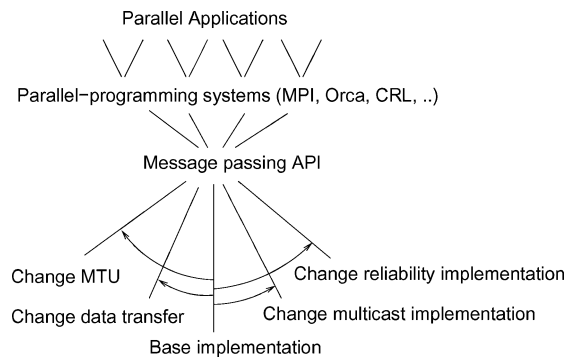


Fig. 1. Impact of modifying the Base implementation.

several implementation choices for reliability, including simple flow control (which assumes reliable hardware) and retransmission protocols on either the host or network interface. Message passing can be implemented using DMA or programmed I/O (PIO), each of which has different performance characteristics. A “zero-copy” remote-memory copy (RCPY) primitive transfers data between the virtual address spaces of processes on different machines. Various zero-copy implementation schemes have been suggested, but little application-level evaluation of such schemes has been performed. The maximum transfer unit partly determines throughput, but also buffer space requirements. Multicasting is important in systems that replicate data or that provide collective-communication operations, which are common in many parallel algorithms. Multicast, however, is often implemented as an afterthought by layering it on top of point-to-point primitives. We compare this strategy with more efficient alternatives.

Our methodology is as follows: We start from a basic, aggressive implementation of a low-level message-passing API. We next vary the implementation aspects outlined above, thus obtaining implementations that use alternative reliability protocols, data-transfer paradigms, packet sizes, and multicast forwarding schemes (see Figure 1). All but one of these implementations provide the same API, so we can run the same benchmarks and applications in almost all cases. In addition, all implementations run on two generations of cluster hardware, each supplied with a high-speed interconnect: 64 Pentium Pros connected by 1.2 Gb/s Myrinet and 64 Pentium IIIs connected by 2.0 Gb/s Myrinet-2000.

Next, we evaluate the performance of all implementations of the message-passing API at multiple levels. We use microbenchmarks to obtain the low-level performance characteristics (LogP parameters or variations thereof) of all implementations. In addition, we use runtime-system specific benchmarks for four parallel-programming systems (MPI, CRL, Orca and a domain-specific language) and eight parallel applications written in these systems. We study the overhead on latency and bandwidth of each system and we try to correlate the low-level and high-level performance based on the characteristics of the programming system and application.

Our study yields several insights:

- (1) *There is no single best low-level communication protocol.* Runtime-system specific communication patterns partly determine the sensitivity of an application to low-level implementation decisions. For example, a reliability protocol that is efficient for a runtime system designed to tolerate high communication latencies can be inefficient for a runtime system that performs mostly round-trip communication.
- (2) *The data-transfer method between the host and the Network Interface (NI) has the largest overall impact on the application performance.* The optimal data-transfer method is shown to be both platform and application dependent; we find application-level performance differences up to 30% in our study. Contemporary computer architectures generally favor the use of DMA over PIO. Nevertheless, we show that PIO-based message passing is still more efficient in a number of cases.
- (3) *Even when hidden beneath multiple software layers, low-level implementation issues can still be relevant.* Although the low-level performance differences between two implementations may be small, the differences at the application layer may be much more significant. This can, for instance, be due to flow control or network interface congestion effects that are only partially visible in microbenchmarks.
- (4) *Finding the right balance when dividing the required functionality over the host and the network interface is difficult.* Implementing reliability in software reduces performance at all levels, but adds robustness that is usually needed. Software reliability can be implemented on the host or (mostly) on the NI. We show that these alternatives yield different performance characteristics (e.g., host and NI occupancy) that play out differently depending on the runtime system and application. Neither alternative performs best under all circumstances. Nevertheless, we show that application slowdown due to an additional reliability protocol is relatively moderate on both our platforms.
- (5) *Zero-copy communication can help performance, but should not be overrated.* We show that zero-copy support can indeed improve performance for certain applications, especially if the communication can be made asynchronous. Most nontrivial (fine-grained) parallel applications, however, are not throughput-bound. We show this to be the case for all our eight applications, which are written using four different parallel-programming systems and which exhibit a wide variety of communication patterns.
- (6) *Large MTUs are not necessarily beneficial.* Although larger MTUs usually result in better low-level throughputs, the receive copy typically made by runtime systems may easily ruin this advantage, as it does on one of our evaluation platforms. Also, conservative protocols that are based on buffer reservation may show inferior performance for small messages when using large MTUs, due to the higher acknowledgment rate.
- (7) *Multicast forwarding should be performed at the lowest level possible.* Forwarding multicast packets on the network interface rather than on the

host reduces multicast latency, reduces host occupancy, and potentially increases multicast throughput. These factors can significantly improve application performance. The impact on performance is highest for applications that also use multicast for synchronization. However, applications that can pipeline multicasts are still able to get good speedups using a host-based multicast implementation. Multicast tree topology is also shown to have a large impact on performance.

- (8) *Performance results on two different hardware generations suggest implications for future work on parallel algorithm design.* Even though on the most modern of our two platforms the communication speed has increased much less than the processor speed, we show that the majority of our parallel applications still achieve a respectable efficiency. Applications that are sensitive to round-trip latency suffer most. Application restructuring, for example, to reduce the number of synchronizations, will therefore become increasingly more important.

The remainder of this article is structured as follows: Section 2 describes LCI, our Low-level Communication Interface. Section 3 describes the cluster hardware on which all LCI implementations run. Section 4 gives an overview of the internal design and implementation of the base version of LCI. Section 5 discusses the dimensions of the design space along which we experimented and compares different design choices by means of microbenchmarks. Section 6 summarizes the performance of the parallel-programming systems used in our evaluation. Section 7 analyzes application performance. Section 8 discusses related work and Section 9 concludes.

## 2. COMMUNICATION INTERFACE

LCI is the low-level communication interface that we use in our evaluations. It was designed for use on a Myrinet cluster with the explicit goal to ease the implementation of a variety of runtime systems. LCI provides a programming interface for reliable, packet-based point-to-point and multicast communication. Packets are delivered in FIFO order and can be received using polling or interrupts. For distributed synchronization (e.g., to implement totally ordered multicast communication), LCI provides an atomic, remote fetch-and-add (F&A) primitive.

LCI is lean: addressing is simple, and the communication is packet-based with a message-passing interface comparable to Active Messages [von Eicken et al. 1992]. The interface can be implemented efficiently and is sufficiently low-level that different types of runtime systems can be layered on top of it in a fairly efficient way.

Figure 2 illustrates LCI's architecture for message passing. To send data, an LCI client allocates a send buffer in network interface (NI) memory (step 1) and it writes a descriptor telling the NI to use DMA to transfer the data, or (2) uses programmed I/O (PIO) to copy data into this packet buffer directly. Next, the client hands the packet to a point-to-point or multicast routine which passes a send request to the NI (3). The NI transmits the packet to the destination NI (4), which copies the packet to pinned host memory by means of a DMA transfer

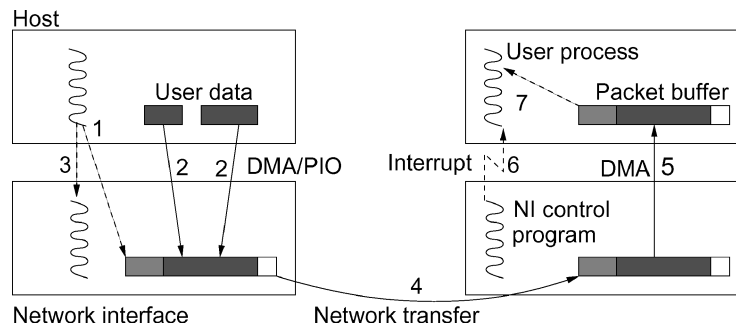


Fig. 2. LCI's packet-based communication architecture.

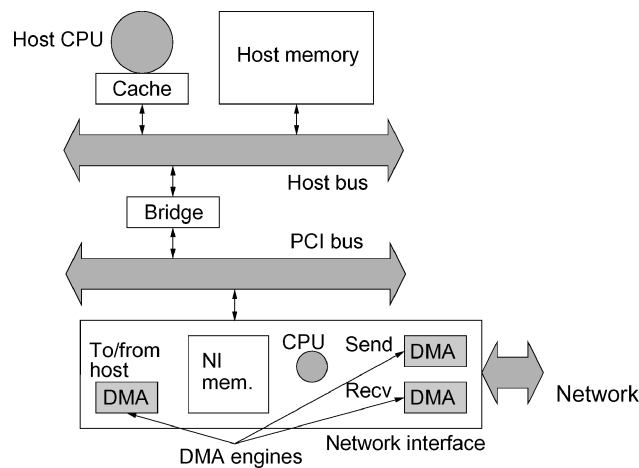


Fig. 3. Myrinet architecture.

(5). If network interrupts are enabled, the NI generates an interrupt (6). The user process detects the packet through an explicit poll by the LCI client or in response to an interrupt. In both cases the packet is passed to a client-supplied upcall routine (7), which processes the packet. If necessary, the client can retain ownership of the packet after the upcall has returned; in that case, the client must later release the packet explicitly.

With one exception, all LCI implementations implement this programming interface. The exception, described in Section 5.2, is an implementation that adds a remote-memory copy primitive.

### 3. HARDWARE ENVIRONMENT

All LCI implementations run on Myrinet [Boden et al. 1995] network hardware. Figure 3 illustrates the architecture of Myrinet network cards (common to both types used in this paper) and shows how they connect to a host system. The programmable NI has a custom RISC processor, a modest amount of SRAM memory, and three DMA engines. The processor runs approximately an order of magnitude more slowly than the host processor. The NI's memory holds the

Table I. PPro and P3 Platform Details

	PPro platform	P3 platform
Cluster size	64 nodes	64 nodes
Processor	Intel Pentium Pro, 200 MHz	Intel Pentium-III, 1 GHz
L1 cache	8 KB I + 8 KB D	16 KB I + 16 KB D
L2 cache	256 KB I+D	256 KB I+D
Memory	128 MB EDO RAM (66 MHz)	1 GB SDRAM (133 MHz)
IO-bus	32 bit, 33 MHz PCI	64 bit, 33 MHz PCI
Peak DMA	127.2 Mbyte/s	254.3 Mbyte/s
Network	Myrinet, SAN cabling	Myrinet-2000, optical fiber
Bandwidth	1.28 Gbit/s full duplex	1.92 Gbit/s full duplex
NI processor	LANai 4.1, 33 MHz	LANai 9.2, 133 MHz
NI memory	1 MB SRAM	2 MB SRAM

code and data for a control program. All inbound and outbound network packets must also be staged through this memory. One DMA engine transfers data between host memory and NI memory; the two others transfer packets from NI memory to the network and vice versa. Packets on Myrinet are protected by means of CRCs which are generated and checked by the NI in hardware.

Network packets are cut-through-routed through switches and full-duplex links with hardware flow control. Its hardware flow-control protocol and very low error rate make Myrinet highly reliable. No packets are dropped if all NIs agree on a deadlock-free routing scheme and remove incoming packets in a timely manner.

LCI implementations abstract from the network hardware in two ways: they provide *reliable* and *multicast* communication services. The programmable network interfaces and the highly reliable network links allow us to investigate different design options. In particular, we can divide protocol tasks such as reliability in different ways over the host processor, the network-interface firmware, and the network hardware. Although the network hardware is very reliable, at least a minimal flow control protocol is needed to prevent buffer overruns in the NI firmware and the host software. Since the Myrinet hardware does not support multicast, multicast services must be provided either by the NI firmware or by the host software.

We use two hardware generations of the host systems and the network; the details are shown in Table I. The first generation is a 64-node Pentium Pro cluster with a Myrinet [Boden et al. 1995] interconnect. The Myrinet NIs are connected via a 3D grid of 8-port switches using Myrinet SAN (flat) cables. The switch delay is approximately 100 ns and the maximum distance between two NIs is 10 hops, so the total switch delay is at most 1  $\mu$ s. We refer to this system as the PPro platform.

The second, more recent, generation is a 64-node Pentium-III cluster with a Myrinet-2000 interconnect. We refer to this system as the P3 platform. The Myrinet-2000 NIs are connected via a large, modular Myrinet-2000 switch supporting over 64 nodes. The switch has a topology allowing full bisection bandwidth. Due to the larger switch, the maximum distance between NIs is lower than on the PPro system (3 hops), but the fiber cabling used between the NIs and switch causes some additional latency due to (de)serialization at fiber/SAN



NI-resident Send Buffer (see Figure 4), and it writes an address and size field, instructing the NI to transfer the user data to the buffer using DMA. Next, the sending process invokes a send call that adds a packet header and enqueues a Send Descriptor in NI memory, which triggers the send. The send descriptor specifies the destination and contains a pointer to the send buffer. The user data should not be modified until the host is sure the NI has transferred the data to NI memory; this is made known to the host by means of Send Completions. Send completions are reported back via a DMA to the host receive queue. Sends to different destinations can complete out of order, so a queue rather than a counter is needed.

To avoid buffer overflow at a receiving NI, the base implementation uses a credit-based sliding-window flow control protocol between each pair of NIs. Each NI reserves a small, fixed number  $W$  of NI Receive Buffers for each sender. (In contrast, retransmitting implementation variants let all senders share one NI receive buffer pool, see Section 5.1.) The total number of receive buffers is limited by the amount of NI memory available: the PPro platform has space for 256 buffers of 2 Kbyte; on the P3 platform the number of buffers is doubled. Each NI transmits a packet only if it knows that the receiving NI has free buffers. When the NI's control program finds a send descriptor in module Host Send Queue Mgt, it first transfers the user data to the corresponding Send Buffer using DMA. It then checks if its send window allows it to transmit the packet; if not, the NI queues the packet on a per-destination queue (modules Send Queue Mgt and Reliability / Flow Control). Blocked packets are dequeued and transmitted during acknowledgment processing. In the base implementation, an NI send buffer can be reused immediately after it has been transmitted. (The retransmitting implementations instead must wait until the packet's receipt has been acknowledged.)

The receiving NI copies each incoming unicast packet from its NI Receive Buffer to a free packet buffer in host memory using DMA and then releases the receive buffer (using modules Recv Queue Mgt, Recv Dispatch, Unicast, Host Recv Queue Mgmt and DMA to Host in Figure 4). The number of newly released buffers is piggybacked on each packet that flows back to the sender. In the absence of return traffic, the receiving NI will send an explicit half-window acknowledgment after releasing  $W/2$  buffers (module Reliability / Flow Control).

If network interrupts are enabled, the NI starts a timer after receiving a packet. The idea is to delay the generation of an expensive network interrupt in the hope that the host will detect the packet through polling. An interrupt is generated only by the Timer module if the timer expires before the host polls. The default timeout value is  $70 \mu s$  on the PPro platform and  $30 \mu s$  on the P3 platform. (Implementation details of this polling-watchdog mechanism [Maquelin et al. 1996] are described in a separate paper [Bhoedjang et al. 2000a].) The host passes network interrupts as Unix signals to the application. The signal handler in the LCI library responds to the signal by polling the host receive queue, as described below.

Each host maintains a queue of free receive buffers and writes the buffers' addresses into a Recv Completions queue in NI memory. The host, in response to a user's poll or a signal, detects newly arrived packets by checking a full/empty

flag in the buffer at the head of the queue. When this buffer is marked full, it is passed to the application using an upcall. Received packets are explicitly released by the application, to allow temporary queuing (rather than copying) of individual packets when they cannot be handled immediately. Packets may therefore be released in a different order than they arrive.

#### 4.2 Multicast Implementation

Since multicast is not directly supported by Myrinet, multicast packets are forwarded in software along a spanning tree rooted at the sender. Each node has one spanning tree for each multicast group that it is a member of. To prevent store-and-forward deadlocks, we by default use binary forwarding trees and do not allow overlapping multicast groups. In this paper, systems layered on LCI use only a single multicast group that contains all nodes (i.e., the broadcast group).

In the base implementation, multicast packets are recognized and forwarded by the NI firmware, which avoids unnecessary host-to-NI data transfers and also reduces multicast latency [Bhoedjang et al. 1998a; Verstoep et al. 1996]. (Section 5.4 discusses host-level multicast forwarding.) Each NI stores a multicast forwarding table that is created by the host at initialization time. When a multicast packet arrives at an NI, the Multicast module copies it to the host, just like a unicast packet, and in addition creates a new Send Descriptor for each forwarding destination. These descriptors are enqueued just like send descriptors created by the host for unicast packets. Multicast packets are thus subject to the same NI-to-NI flow control as unicast packets. NI receive buffers holding multicast packets are not released until the packet has been copied to the host and forwarded to all children in the multicast tree.

#### 4.3 Fetch-and-Add Implementation

In the base implementation, each NI maintains a single fetch-and-add variable that can be accessed by any host by sending a fetch-and-add request to the NI. The NI Fetch&Add module replies with the current value of the variable and then increments the variable.

### 5. ALTERNATIVE IMPLEMENTATIONS

The base implementation occupies only one point in the design space. Sections 5.1 through 5.4 discuss alternatives in four areas: reliability, data-transfer method, maximum transfer unit (MTU), and multicast. For each area, we discuss alternative designs and present corresponding implementations. Table II summarizes the characteristics of all implementations. For brevity, we will frequently refer to an implementation by its mnemonic name. The base implementation is referred to as *l-sdma*; the names of the other implementations are listed in Table II.

All alternative implementations are variations of the base implementation, with which they share a fair amount of code. All implementations implement the API described in Section 2 (in addition, *l-rcpy* has an API extension for

Table II. Key Characteristics of LCI Implementations

	l-sdma	Alternatives	Names
Reliability scheme	No retransmission	– NI retransmits – Host retransmits	l-nirx l-hrx
Data-transfer scheme	Message passing (DMA-based)	– Message passing (PIO-based) – Remote-memory copy (+ message passing)	l-pio l-rcpy
Maximum Transfer Unit (MTU)	2 Kbyte	– 1 Kbyte – 4 Kbyte	l-1k l-4k
Multicast forwarding scheme	NI forwards	Host forwards	l-hmc

remote-memory copy). This allows us to run the same benchmarks, the same programming systems, and the same applications on all implementations.

All implementations currently allow at most one user process at a time to access the Myrinet network and NI access is unprotected. Techniques for protected user-level network access are well known [Druschel et al. 1994; von Eicken et al. 1995], but not the subject of this paper. Adding protection to the LCI implementations would mainly involve appropriately protecting NI memory pages and checking the validity of buffer pointers. No system calls would have to be added to the critical path, but the extra checks and indirections would make the critical path slightly more expensive. In this article, we ignore that effect.

In the next sections, we will discuss the alternatives along the dimensions of Table II in detail, and analyze their performance impacts by means of microbenchmarks. Since in the sequel we sometimes vary the LCI implementation in multiple dimensions simultaneously, the implementation names used may be a composition of the names shown in Table II. For example, l-pio-1k refers to an LCI implementation that uses PIO-based message passing with the a packet size of 1 Kbyte, but with default NI-level multicast forwarding, and without retransmission on the host or NI. Furthermore, l-hmc is really short for l-sdma-hmc, etc.

### 5.1 Reliability

The base implementation assumes reliable network hardware and uses flow control to preserve that reliability. Although several Myrinet-based communication systems make this assumption, treating all communication errors as an unrecoverable failure may not be acceptable in environments where robustness is essential. (Also, Myrinet’s manufacturer, Myricom, currently does advise the use of an NI-level reliability protocol for its latest fiber-based hardware.)

Below, we consider two alternative implementations that assume unreliable network hardware and recover from lost, corrupted, and dropped packets by means of time-outs, retransmissions, and hardware-supported CRC checks. We consider only transient network failures and ignore permanent link failures, which require the discovery of new routes. The first alternative implementation is based on a traditional design that employs retransmission by the host

processor. In the second alternative implementation, the retransmission protocol is run on the NI.

**5.1.1 Host-Level Retransmission.** The implementation of the host-level retransmission design, *l-hrx*, uses a go-back-N sliding-window protocol. This protocol is efficient if packets are rarely dropped or corrupted, as is the case on Myrinet. After transmitting a packet, the sender starts a retransmission timer by reading the CPU's timestamp register (which has very little overhead) and storing the timestamp in the sliding-window control data structure. The receiving NI drops packets with CRC errors and packets that cannot be stored due to a shortage of NI receive buffers. The receiving host drops all out-of-sequence packets. If the sender's timer expires before an acknowledgment is received, it retransmits all unacknowledged packets whose timestamps have become too old. Timeout management is optimized in *l-hrx* to use the CPU's timestamp register as much as possible, falling back to OS-generated timer signals only when needed.

*l-hrx* uses NI memory to buffer outbound packets for retransmission. Since the Myrinet hardware can transmit only packets that reside in NI memory, all LCI implementations must copy data from host memory to NI memory. By using the NI buffer for retransmission, the implementation does not need to make more memory copies than the base implementation. A disadvantage is that NI send buffers cannot be reused until they have been acknowledged. Given the small size of NI memories, a sender that communicates with many receivers may run out of send buffers before acknowledgments flow back. One solution is to acknowledge each data packet, but this increases network, NI, and host occupancy. Instead, *l-hrx* uses piggybacked and half-window acknowledgments, just like the base version. To ensure that all send buffers can be freed eventually, each receiver maintains in addition an acknowledgment timer per sender  $S$ . The timer for  $S$  is started when a data packet from  $S$  arrives and the timer is not yet running. The timer is canceled when an acknowledgment, possibly piggybacked, travels back to  $S$ . If the timer expires, the receiver sends an explicit *delayed* acknowledgment. Another solution, which we did not implement, is to tag outgoing packets with an explicit acknowledgment request when the number of free send buffers drops below a threshold [Tang and Bilas 2002].

Finally, the reliability protocol that *l-hrx* implements for unicast is also used in its host-level multicast implementation. This should be contrasted with *l-hmc* (discussed in Section 5.4), which for its host-level multicast implementation depends on the reliability of the NI-level unicast implementation.

**5.1.2 NI-Level Retransmission.** In the second alternative implementation, *l-nirx*, the NI runs the retransmission protocol. The protocol is almost identical to the host-level protocol *l-hrx* described above, but sliding-window and timer management, acknowledgment processing, etc., are now all performed on the NI. This increases NI occupancy, but reduces host overhead.

*l-nirx* forwards multicast packets directly from the NI receive queue, like the base implementation *l-sdma*. While multicasting, the NI receive queue may fill up with packets that have been delivered locally but still have to be forwarded.

Table III. LogP Values and F&A Latencies in Microseconds for SDMA-Based Message Transfers on Different LCI Implementations and GM. The End-to-End Latency Is Equal to  $o_s + L + o_r$ 

LogP parameter	PPro platform				P3 platform			
	l-sdma	l-nirx	l-hrx	GM	l-sdma	l-nirx	l-hrx	GM
Send overhead ( $o_s$ )	1.8	1.8	3.3	1.1	0.5	0.5	0.9	0.7
Recv overhead ( $o_r$ )	2.6	2.5	5.9	1.6	0.5	0.5	1.2	0.5
Latency ( $L$ )	10.7	13.7	9.3	22.2	7.8	8.3	7.8	9.5
Gap ( $g$ )	11.1	12.6	10.2	31.1	4.5	5.2	4.6	12.6
End-to-end latency	15.1	18.0	18.5	24.9	8.8	9.4	9.9	10.7
F&A latency	18.0	24.4	32.5	N/A	7.4	9.0	13.4	N/A

Since NI memories are small compared to host memory, the NI-level retransmission variant is more likely to be forced to drop packets than the host-level variant under such conditions (l-hrx does its multicast forwarding from the host-level receive queue which can be extended on demand, unlike the NI-level receive queue that l-nirx uses). In l-nirx, NIs therefore keep track of available receive buffer space. When a packet must be dropped due to buffer space shortage, the NI requests retransmissions from the sender as soon as buffers become available again.

**5.1.3 Microbenchmarks.** Table III shows the LogP parameters [Culler et al. 1996] and fetch-and-add latencies on both platforms for l-sdma, l-nirx, and l-hrx. For comparison, we have also included in this table the values for GM (version 1.5), which is the standard software package supplied by the manufacturer of Myrinet. GM uses DMA to send messages and implements reliable communication at the NI, like LCI's l-nirx.

As is usual in the literature, we distinguish between send overhead ( $o_s$ ) and receive overhead ( $o_r$ ) on the host. Since l-sdma and l-nirx perform the same work on the host, they have almost identical send and receive overheads (the  $0.1 \mu\text{s}$  difference in Table III is presumably due to a caching effect). l-nirx, however, runs a retransmission protocol on the NI, which is reflected in its larger gap (i.e., small-message bottleneck) and latency. l-hrx runs a retransmission protocol on the host and therefore has larger send and receive overheads than l-sdma and l-nirx. However, due to its host-level retransmission protocol, the window size for l-hrx can be kept largely independent of the cluster size (unlike l-sdma, which uses a conservative NI-level buffer reservation scheme). As a result, fewer acknowledgments need to be sent for large sequences of one-way transfers, explaining the lower gap of l-hrx on the PPro platform.

l-sdma has the best end-to-end latency. As expected, the retransmission support in l-nirx and l-hrx increases end-to-end latency due to timer management and sender-side buffering. Host-level retransmission increases send and receive overhead ( $o_s$  and  $o_r$ ), while NI-level retransmission increases NI occupancy (reflected in  $g$ ) and latency ( $L$ ). l-nirx increases the end-to-end latency slightly less than l-hrx. In l-nirx the host and the NI can operate in parallel, because the NI has its own copy of each incoming packet's header. In l-hrx, the host-level library must complete packet processing before passing the packet to the application. The increased host performance on the P3 platform (relative to the PPro

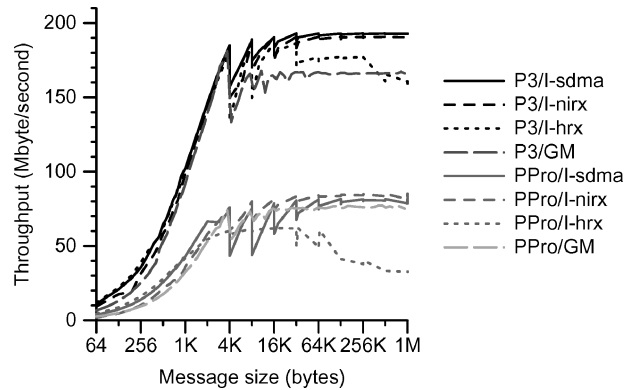


Fig. 5. Throughput for different reliability implementations.

platform) works out favorably for l-hrx's receive overhead:  $o_r$  decreases from 5.9 to 1.2  $\mu$ s. The end-to-end latency of l-sdma, l-nirx, and l-hrx on the P3 platform are much closer than on the PPro platform. The reason is that host overheads scale quite well with the increased host CPU speed, but  $L$  does not: it contains several components like DMA overhead and network delay that show only limited improvement on the P3 platform.

Compared to LCI, GM's host overhead is quite low on the PPro platform (LCI's receive overhead is higher mainly due to its upcall mechanism and additional receive queue management). On the P3 platform GM's host overhead is similar to LCI's, since there the number of PCI bus operations is relatively more important than the amount of host CPU cycles. However, on both platforms GM's  $L$  and  $g$  are noticeably higher than l-nirx's, due to additional overheads in the NI software (e.g., support for multiprogramming).

l-sdma has the fastest F&A implementation, while l-hrx, which handles F&A requests on the host processor, is the slowest. For all implementations, the F&A performance on the P3 platform is consistently about 2.5 times better than on the PPro platform.

Figure 5 compares the throughput of l-sdma, l-nirx, l-hrx, and GM on both the PPro and the P3 platforms. With the default 2 Kbyte MTU, the higher per-packet overhead in l-nirx (relative to l-sdma) does not reduce the throughput for medium-size and large messages (in the figure the lines for l-sdma and l-nirx largely overlap). The reason is that for this packet size, the bottleneck is at the host rather than at the NI. As shown in an earlier paper [Bhoedjang et al. 2000b], if l-nirx is configured with a smaller packet size the balance can be different, causing a lower throughput at the PPro platform. The jumps in the throughput graphs are due to fragmentation at packet-size boundaries; the low-level benchmarks used make no special effort to reduce this effect. (For a detailed discussion on the influence of fragment size on the *latency* of medium-size messages, see Wang et al. [1998]).

On both the PPro and P3 platforms, l-hrx shows a lower throughput than l-sdma due to the higher host-level overhead. Although GM's  $g$  is quite high, GM's throughput is only slightly lower than that of l-nirx. This is because  $g$  in

LogP model represents the throughput bottleneck for *small* messages. For the large messages used in the throughput benchmark this overhead is spread over many bytes, so there the impact is only minor.

## 5.2 Data-Transfer Alternatives

The base implementation, l-sdma, uses DMA to transfer the user data to the NI at the sender. Though for medium-size and large packets this reduces host overhead, PIO data transfers, as used in l-pio, can in fact be more efficient for small messages.<sup>1</sup>

Throughput in l-sdma is in practice limited by the fact that data is delivered to preallocated receive buffers in host memory. Most LCI clients will have to copy data from the receive buffer to another location (e.g., into some data structure). This extra copy increases latency and, on the PPro platform, reduces throughput significantly (see Section 5.2.4). We also discuss an alternative implementation, l-rcpy, that avoids the copy at the receiver by providing a DMA-based remote-memory copy primitive. Note that the main difference between l-sdma and l-rcpy is at the receiver: l-sdma delivers its packets in a receive queue, whereas l-rcpy tries to DMA the data directly to the proper destination address (possibly inside a user-level data structure).

Below we will first discuss a number of implementation aspects of l-pio and l-rcpy. We will then compare these alternative implementations with l-sdma by means of microbenchmarks.

**5.2.1 Message Passing using Programmed I/O.** l-pio only differs from l-sdma at the sender: it delivers its packets in the same receive queue as l-sdma. Since in l-pio the user data is directly written by the host to the NI send buffer (see Figure 2), overhead is shifted from the NI to the host. Depending on the platform and the size of the message, this might improve performance. Another advantage of l-pio is that the host completely controls the data transfer, so it knows when user data may be modified again, without having to synchronize with the NI.

At the receiving side, all LCI implementations use DMA. PIO reads are typically too slow; much more so than PIO writes, which often can be pipelined over the PCI bus. Furthermore, receiving packets via PIO would also make asynchronous delivery of packets in host memory impossible, further reducing performance.

**5.2.2 The Remote-Memory Copy Primitive.** l-rcpy's remote-memory copy (RCPY) primitive allows a sender to copy a contiguous block of memory from its virtual address space to a specific destination address in a receiver's virtual address space. (This is sometimes referred to as *sender-based communication* [Buzzard et al. 1996], *virtual memory-mapped communication* [Dubnicki et al. 1997a], or *remote deposit* [Bilas et al. 1999b].) The new primitive allows only point-to-point transfers to and from 4-byte aligned addresses. It is

<sup>1</sup>Several previous papers [Bhoedjang et al. 1998a; Bhoedjang et al. 1998b] refer to this implementation as LFC.

nonblocking so that the sender is free to do other work while the transfer is in progress. A separate API call allows the sender to check for completion of a pending transfer. By default, no notification about a completed transfer is triggered at the receiver, since in some parallel-programming systems this upcall is not needed, and would only introduce extra host overheads which RCPY is trying to avoid. If needed, in addition the regular message-passing primitives can be used to implement control transfers. To use RCPY transfers efficiently, the sender needs to have detailed knowledge about the placement and layout of the data structures at the receiver. This is a responsibility of the higher software layers, however.

*5.2.3 Implementation.* The host library splits each RCPY transfer into a series of page transfers; a page transfer spans at most one 4-Kbyte page. Using an RCPY MTU larger than the system page size is difficult. Since the NI's DMA engine operates on physical addresses, it would require that multipage source and destination buffers be stored in consecutive physical page frames. This is difficult to combine with paging.

One of the key implementation problems for this type of primitive is to preclude the situation in which a host memory page is simultaneously being paged in or out by the operating system *and* used as the source or target of a RCPY DMA transfer. Pages can be pinned into memory to prevent the operating system from replacing them, but it is not always possible or acceptable to pin every virtual page (in general the operating system limits the number of pages that an application may pin to ensure that it does not monopolize these resources). Most RCPY implementations therefore use a software TLB to keep track of a limited set of pages that are currently pinned and of the physical addresses of these pages [Dubnicki et al. 1997b; Tezuka et al. 1998; Welsh et al. 1997]. The NI control program needs to know the pages' physical addresses, because its DMA engine requires a *physical* host address. Since pinning takes place on the host and DMA transfers are started by the NI, the host and the NI need to share the TLB. One of the main differences between RCPY implementations is how they implement this sharing. Our implementation uses a novel scheme that resolves both send and receive TLB misses in a nonblocking way. The scheme requires a small kernel extension that allows a user process to request virtual-to-physical address translations.

In `l-rcpy`, the host library maintains a Host TLB (HTLB) and the NI maintains an NI TLB (NTLB). The TLBs are direct-mapped and contain 1024 entries. Host-level and NI-level state machines keep corresponding NTLB and HTLB entries consistent; the host-level state machine acts as master and can cause an NI-level state change using PIO. The TLB update protocol ensures that NTLB entries which are marked valid always contain the right virtual-to-physical mappings and refer to pinned host memory pages. Since both the host and the NI have direct access to their copy of the TLB (i.e., without having to cross the I/O bus), both the send and receive path can be implemented efficiently.

*Optimal Zero-Copy Transfer.* If the sending node has valid HTLB entries for all the pages in the source buffer and the receiving node has valid NTLB entries

for the destination buffer, then a RCPY transfer proceeds as follows. The sending host computes the virtual addresses of the pages contained in the source buffer; given that they are in the HTLB, (and, hence, NTLB), TLB *miss* processing (see below) is unnecessary. For each source buffer page, a send descriptor is filled in NI memory. Each descriptor contains a virtual source address, a size, a destination, and a virtual destination address. For each descriptor, the NI looks up the virtual source address in its NTLB and performs a DMA transfer from the corresponding physical host address to a free NI buffer. Next, the send buffer is transmitted to the destination NI. The receiving NI extracts the virtual destination address from the packet header, looks it up in its NTLB, and performs a DMA transfer to copy the data to host memory.

*Transfers with TLB Misses.* Depending on the number of pages that may be pinned, the size and associativity of the TLBs, and the memory access pattern for message transfers, a true zero-copy transfer as just described is not always possible on the send or the receive path. If the sender supplies a source buffer that spans a page without a valid HTLB entry, then the LCI library will pin that page and obtain its physical address. The library cannot, in general, safely update the HTLB entry until it knows that the NI has no DMA transfer pending to the address in the corresponding NTLB entry (pointing to a different page). To hide latency due to synchronization with the NI, the library copies the page segment that caused the miss directly to a send buffer in NI memory. In addition, the library saves the new mapping and indicates in its send descriptor for the page segment that a miss occurred. When the NI finds the miss notification, it can be sure that all pending hits for the NTLB entry have been processed, so it can be updated. The host will learn that the NTLB has changed state while processing the corresponding send completion, or while initiating a new transfer for the entry.

At the receiving end, TLB misses occur on the NI. When the NI receives an RCPY packet and finds that it has no valid translation for a virtual destination address, it transfers the data to a so-called *anonymous* page, which is picked from a set of pages reserved in host memory. (This is identical to *transfer redirection* in VMMC-2 [Dubnicki et al. 1997a].) It then invalidates and locks the TLB entry. The RCPY notification message indicates that a miss occurred, supplies the address of the anonymous page, and also serves as a TLB replacement request. When the host receives the notification, it copies the data from the anonymous page to the sender-specified destination, pins the destination page, updates both its own and the NI's TLB, and finally unlocks the NI's TLB entry.

5.2.4 *Microbenchmarks.* Figure 6 compares the unicast throughput of l-sdma, l-pio, and l-rcpy. l-rcpy can deliver message data directly to its final destination in the receiver's host memory, provided that no TLB misses occur. The PIO- and SDMA-based transfer methods, in contrast, deposit message data in a separate receive queue, and the receiver will usually have to copy the data from that queue to another location. The figure shows that this extra memory copy noticeably decreases performance on the PPro platform. This effect is

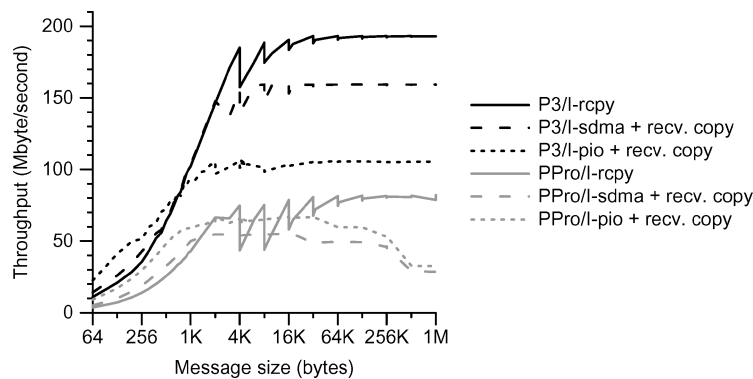


Fig. 6. Throughput using PIO- and DMA-based message transfers.

particularly strong when the destination buffer does not fit into the L2 cache; in that case, the Pentium Pro's low *memcpy* speed for large buffers (52 Mbyte/s) becomes a bottleneck.

The P3 platform has enough spare memory bandwidth to accommodate the additional copying required without influencing the performance in this microbenchmark. For this platform, however, the difference in PIO and DMA performance is much more pronounced, due to the 64 bits/33 MHz PCI bus. This bus has a peak throughput (254.3 Mbyte/s) that is very close to the Myrinet-2000 link bandwidth (250 Mbyte/s). Writing packets over this bus using PIO is less effective, even though the processor's write-combining feature is used. The PIO-write bandwidth on the P3 platform is limited to about 120 Mbyte/s, so the packet transfer to the NI becomes a clear bottleneck for l-pio (note that PIO reads over the PCI bus have such a low throughput that they are never used by LCI in packet transfers).

The efficient scheduling of message transfers, acknowledgments, and DMAs is very important to achieve good throughput performance. A convenient way to analyze the sequencing and efficient overlap of the relevant operations is to visualize the corresponding events happening at the NI. For optimizing the NI software, we created an instrumented version of the NI firmware, collecting traces of timestamped events. The events were then asynchronously transferred to the host to be analyzed off-line. It turns out that in LCI's current version, on the PPro platform not all NI-level protocol handling and message transfers are hidden behind DMA transfers, causing the throughput to be lower than what is theoretically possible. Although on the P3 platform, the NI's local memory bandwidth is much higher than the unidirectional network bandwidth, the PCI DMA bandwidth matches the network bandwidth so closely that attaining optimal performance is still not trivial. We checked that on a P3 system with a faster (64 bits/66 MHz) PCI bus, LCI does manage to use the full network bandwidth.

Table IV contains the LogP performance figures for SDMA-, PIO-, and RCPY-based communication, respectively. Several trends can be drawn from this table. For the small message size considered in the LogP model, the send overhead of the different data-transfer primitives is rather similar. For 16-byte messages

Table IV. LogP Values for LCI's SDMA-, PIO-, and RCPY-Based Primitives.  
The End-to-End Latency Is Equal to  $o_s + L + o_r$

LogP parameter	PPro platform			P3 platform		
	l-sdma	l-pio	l-rcpy	l-sdma	l-pio	l-rcpy
Send overhead ( $o_s$ )	1.8	1.5	1.8	0.5	0.5	0.6
Recv overhead ( $o_r$ )	2.6	2.3	1.2	0.5	0.4	0.1
Latency ( $L$ )	10.7	6.7	16.1	7.8	5.2	9.1
Gap ( $g$ )	11.1	6.7	13.9	4.5	2.8	5.1
End-to-end latency	15.1	10.4	19.1	8.8	6.2	9.8

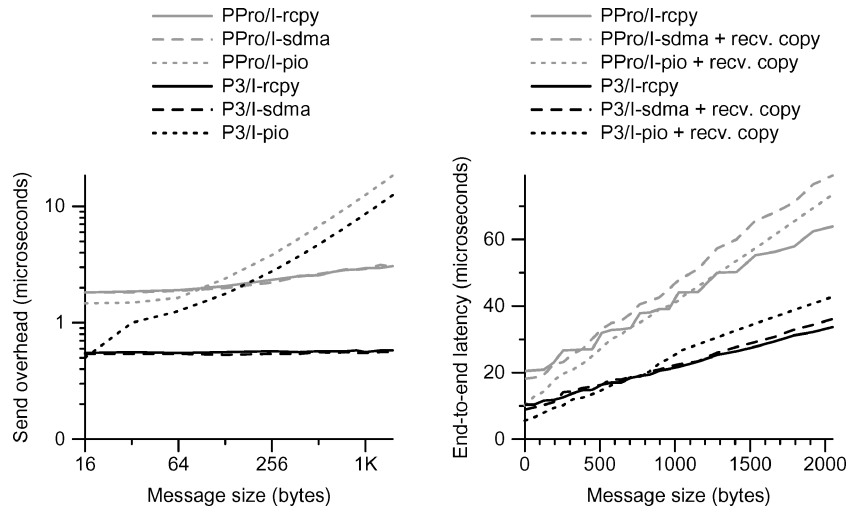


Fig. 7. Host send overhead (left) and end-to-end latency (right) using PIO- and DMA-based message transfers.

(which are commonly used), the PIO data transfers in l-pio are in fact no more costly than the host-level DMA preparation overhead in l-sdma and l-rcpy. The receive overhead for l-rcpy is smaller than for l-pio and l-sdma since l-rcpy does not have the upcall overhead (polling is still required in l-rcpy, to handle data transfers to addresses that are not in the NI TLB). The largest difference is in the LogP  $L$  and  $g$  parameters. For small messages, l-sdma and l-rcpy suffer from the NI overhead due to the DMA at the sender.

However, LogP parameters in general depend on message size. Figure 7 (left) shows how this works out in particular for host send overhead. As can be seen from the figure, l-sdma and l-rcpy succeed in their aim to reduce the host overhead at the sender for all but the smallest messages. The influence of message size on LogP's  $g$  can be observed in the unidirectional throughput results (shown in Figure 6).

The impact of message size on the remaining LogP parameters  $L$  and  $o_r$  can be seen most easily in an end-to-end latency graph (but note that this performance figure also includes  $o_s$ ). Figure 7 (right) compares the PIO-based transfer technique with the ones based on DMA with respect to end-to-end latency. The figure shows that the DMA-based transfer methods cause a distinct

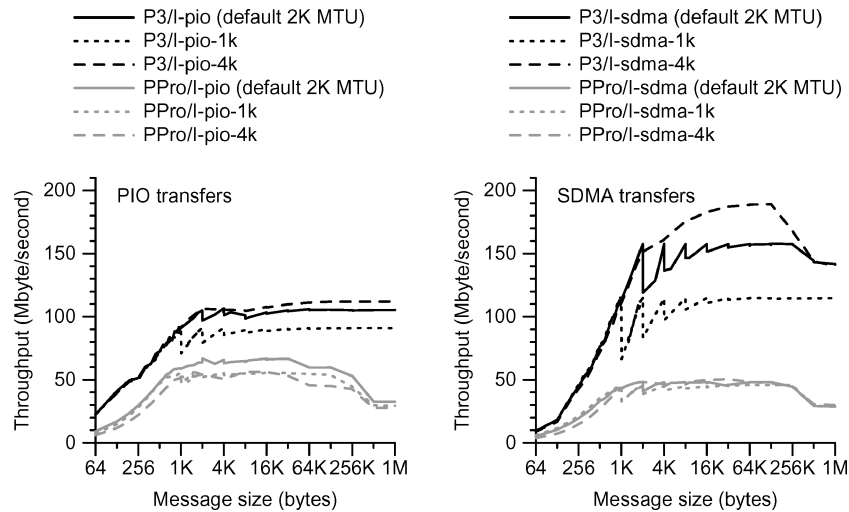


Fig. 8. Throughput for PIO- and SDMA-based message passing for different MTU sizes. The data is copied at the receiver.

increase of the latency for small messages. The cross-over point is at a message size of around 800 bytes on the P3 platform, and at around 1200 bytes on the PPro platform. Furthermore, on the P3 platform l-sdma and l-rcpy show a similar end-to-end latency, while on the PPro platform l-sdma's memory copy from the receive queue makes it more expensive than l-rcpy for messages larger than 350 bytes.

### 5.3 Maximum Transfer Unit

In this section we analyze the impact of using different MTU sizes for PIO- and SDMA-based message passing. To simplify NI-level memory management, all LCI implementations use fixed-size packet buffers. The size of these buffers is also the maximum transfer unit (MTU) for all packets. A larger MTU requires fewer per-packet operations for large transfers and can therefore improve the throughput of such transfers. On the other hand, the MTU determines how many buffers can be stored in a given amount of memory. This is of particular importance for the NI, which has only a small amount of memory. A larger MTU results in fewer NI receive buffers and therefore in smaller NI-level send windows, which may be disadvantageous for applications that send many small messages.

The base implementation uses an MTU of 2 Kbyte; l-1k and l-4k use a 1-Kbyte MTU and a 4-Kbyte MTU, respectively. Figure 8 shows unicast throughput for all three MTU sizes on both hardware platforms. Since virtually all LCI clients will copy (or read) received packets out of the fixed receive area, we show the throughput with the inclusion of an additional copy at the receiver.

As expected, a larger MTU yields better throughput, because the constant per-packet overheads are incurred fewer times per message. Compared to the base implementation, a 4-Kbyte MTU yields a small throughput improvement,

but it does so at the cost of introducing more acknowledgment traffic for applications that are dominated by small messages.

Finally, in this benchmark, the influence of the copy at the receiver is noticeable on the PPro platform. For both l-pio and l-sdma, this memory copy causes the effective throughput to drop below 52 Mbyte/s for messages exceeding the L2 cache size. LCI is usually able to keep all received packets in its L2 cache by restricting the receive queue size, but not when all received packets are copied into a large linear array. On the P3 platform, however, sufficient memory bandwidth is available, so a copy at the receiver decreases the throughput much less.

#### 5.4 Multicast

The base implementation uses NI-level multicast forwarding. Most communication systems, however, use a host-level forwarding scheme in which the multicast implementation is layered over point-to-point primitives. This approach requires less implementation effort, but has several disadvantages:

- Host-level forwarding adds at least one host-to-NI data transfer to each forwarding hop in the multicast tree, increasing latency and bus occupancy.
- Host-level forwarding consumes extra CPU cycles, especially if PIO is used to forward packets to children.
- With host-level forwarding a packet or message is not forwarded unless the forwarding host polls or takes an interrupt. If MPI-like *collective* multicast primitives are used, the MPI library knows when it has to poll for the multicast message. However, in systems that use one-sided multicast primitives (e.g., Orca) it is unpredictable when a multicast arrives. Given sufficient buffer space, NI-level forwarding can then proceed without host-level intervention.

Host-level forwarding comes in several flavors. Systems such as MPICH [Gropp et al. 1996] use host-level *message* forwarding; a forwarding host waits to receive an entire message before it forwards it. With host-level *packet* forwarding [Kielmann et al. 2001], a forwarding host forwards individual packets. With packet forwarding, multipacket messages can be pipelined; with message forwarding, they cannot.

Our alternative, host-level multicast implementation, l-sdma-hmc, uses an aggressive form of host-level packet forwarding. When a host must forward a packet to a set of children, l-sdma-hmc issues a data transfer to NI memory only once and transmits the same NI buffer multiple times. (This strategy is also used in the PM communication system [Tezuka et al. 1997].)

Figure 9 shows broadcast latency and throughput on 64 nodes for l-pio, l-sdma, and l-sdma-hmc on the PPro and P3 platform. We define broadcast latency as the time it takes to reach the last receiver and we define throughput as the sender's outgoing data rate. We show results for the two most commonly used multicast tree topologies: binary trees (which have a constant fan-out of 2 per node) and binomial trees (in which nodes near the root of the tree have a larger fan-out to reduce overall latency). On the PPro platform using binary trees, the latency

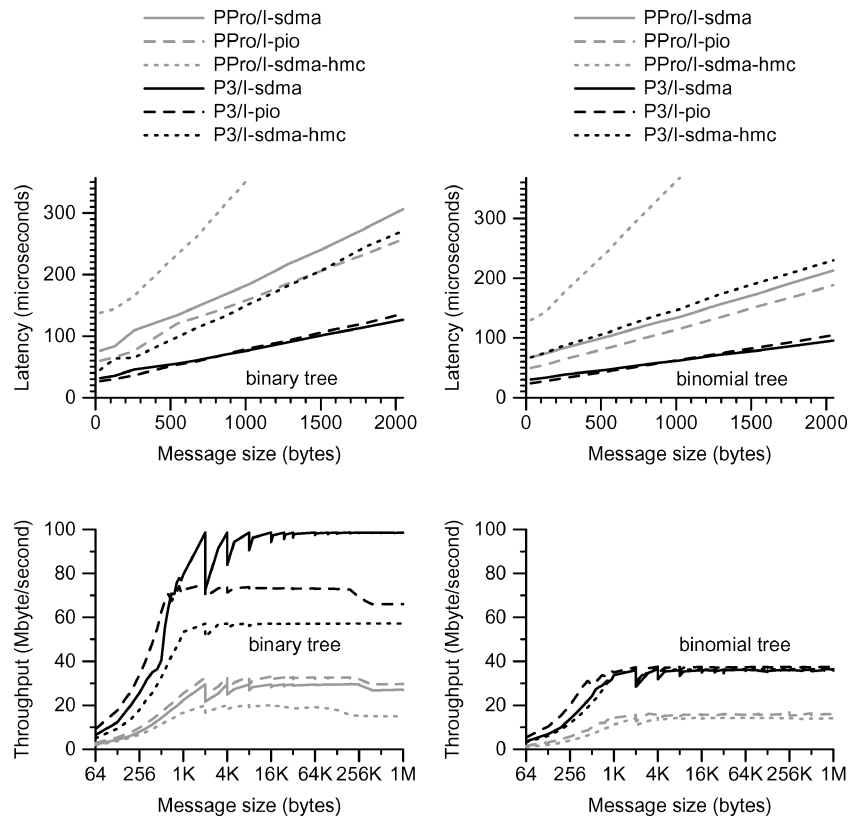


Fig. 9. Broadcast latency and throughput on 64 processors for different implementations with binary trees (left) and binomial trees (right). The throughput tests are with receiver copy, the latency tests without.

of a 16-byte message is  $58.5 \mu s$  for l-pio,  $70.4 \mu s$  for l-sdma and  $135.0 \mu s$  for l-sdma-hmc. On the P3 platform these latencies are  $25.7 \mu s$  for l-pio,  $28.9 \mu s$  for l-sdma and  $45.3 \mu s$  for l-sdma-hmc. As can be seen from Figure 9, the multicast latencies with binomial trees are consistently lower than for binary trees on both platforms.

l-sdma on the P3 platform is able to achieve a multicast throughput of almost 100 Mbyte/s with binary trees; for l-pio this peak throughput is 75 Mbyte/s. Since the top-level fan-out of the multicast tree is higher for binomial trees, the multicast throughput is limited by the NI's link bandwidth at the root. On 64 nodes the top-level fan-out increases from 2 to 6, which causes throughput to drop to around 37 Mbyte/s for all implementations (close to the theoretical maximum of about 41 Mbyte/s). On the PPro platform l-sdma has a lower throughput than l-pio, since the copying of the data to the host-level network packets at the sender is relatively more expensive.

l-sdma-hmc adds two data copies per internal tree node to the critical path: from the NI to the host and back. l-sdma-hmc's extra host-to-NI data transfer also increases the throughput bottleneck, even though it is performed in

parallel with the two data transfers from the NI to its children. As in the unicast throughput benchmark, large messages achieve lower throughput on the PPro platform due to the extra pressure on the L2 cache. The effect is more pronounced on l-sdma-hmc than on l-sdma as a result of the additional data transfer to the NI. Apart from showing up in this throughput benchmark, the extra data transfer also consumes processor cycles that an application might need (see Section 7.4).

### 5.5 Interactions Between Implementation Aspects

In the previous subsections, we have focused on four important implementation aspects mostly in isolation. However, in practice multiple implementation choices will be made together, and these choices will not always be completely independent. Rather than exhaustively analyzing how a particular implementation aspect influences (combinations of) other ones, we will here qualitatively discuss the most important interactions.

- Data-transfer method and reliability*: Assuming that packets may be kept on the NI for retransmission until an acknowledgment is received (as is the case in LCI), the choice of data-transfer method (PIO versus SDMA) can be made independent of the reliability implementation at the host or the NI. Otherwise, a host-based reliability layer may be forced to make an additional copy of the packets sent.
- Data-transfer method and MTU*: For DMA-based data transfers (e.g., LCI's SDMA and RCPY), a page-sized MTU should usually be supported to achieve maximum throughput. Making the MTU larger than a page does not make much sense since operating systems usually do not allow DMA transfers larger than a page. Also, making packets very large may in fact cause throughput degradation due to reduced pipelining. For PIO-based data transfers, a full page-sized MTU usually does not offer performance advantages, since the throughput bottleneck is typically in the host/NI transfer itself, and almost the same throughput can be attained using a smaller MTU (see Section 5.3).
- Data-transfer method and multicast*: Depending on a platform's characteristics and on how multicast is implemented (at the NI or on the host), the multicast throughput bottleneck may be in the network, at a forwarding node (NI or host) or at the sending host. If a host-to-NI transfer is causing the bottleneck, using DMA-based transfer of the multicast data may improve throughput and reduce host overhead as it does for unicast. In Section 5.4, it was shown that multicast throughput in fact could be increased using SDMA on the P3 platform, but not on the PPro platform. For latency, on the other hand, the trade-off may be different. The choice of implementing multicast on the host or the NI has typically more impact on multicast than the choice of data transfer. For small messages, a combination of NI-based message forwarding and PIO transfers will usually give the lowest latency, but for sufficiently large messages the use of DMA transfers may be preferable, depending both on the platform and multicast tree topology (see Section 5.4).

- Multicast and reliability*: In Section 5.4 we considered host- or NI-based multicast implementations based on a reliable unicast implementation. However, building reliable multicast based on an unreliable unicast implementation is also possible. In this case, reliability and flow control for unicast and multicast traffic can be integrated in a similar way as for an NI-based implementation [Bhoedjang et al. 2000b]. Section 7 will discuss the impact of reliability and multicast implementation on application performance, both combined and in isolation.
- Multicast and MTU*: Like for unicast, increasing the MTU can improve multicast throughput since per-packet overheads can be spread over a larger packet size. However, depending on the performance characteristics of the platform, choosing the MTU too large may in fact cause a lower throughput due to a reduction in pipelining (an effect similar to message-based multicast forwarding discussed in Section 5.3).
- Reliability and MTU*: Due to the use of fixed-size packet buffers and sliding window protocols in all our message-passing implementations, larger MTUs translate into increased buffer-space requirements or smaller window sizes (and more frequent acknowledgments). This is particularly disadvantageous for credit-based protocols which use a per-sender NI receive-buffer reservation strategy. By using alternative buffering methods at the sender and receiver for non-full packets in combination with a modified acknowledgment scheme (based on transferred size rather than packet count, similar to TCP/IP) this issue might potentially be avoided. However, there is a clear danger that these alternative methods will be more costly to implement than the current straightforward implementation, increasing NI overhead (especially considering the relative slowness of the NI processor). An alternative is to modify the current (static) credit allocation policy and let it respond dynamically to actual traffic patterns [Canonico et al. 1999].

## 6. PARALLEL-PROGRAMMING SYSTEMS

Parallel applications are rarely developed directly on low-level communication systems such as LCI, but usually run on top of a parallel-programming system (PPS). The communication system is used in the implementation of the PPS's runtime system. Frequently, PPSs implement a small, fixed set of communication patterns and do not give applications full control over each message that is sent. As a result, the performance of applications depends partly on the match between a PPS's communication behavior and the properties of the communication system.

In this article, we consider four parallel-programming systems with different programming paradigms:

- [MPI Forum 1994]: a popular message-passing interface standard;
- CRL [Johnson et al. 1995]: an invalidation-based distributed shared-memory (DSM) system;
- Orca [Bal et al. 1998]: an update-based DSM system;
- Multigame [Romein et al. 2002]: a distributed-search system.

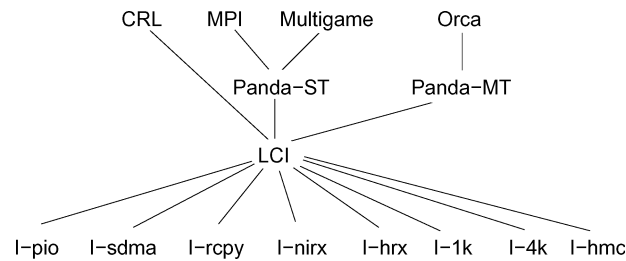


Fig. 10. Dependencies between software systems.

MPI, Orca, and Multigame use Panda, a communication library designed for the development of PPSs (see Figure 10). Panda provides an interface that can be implemented on a wide variety of networks; other projects have also encountered the need for a portable communication layer [Aumage et al. 2000]. CRL is simple enough to be implemented directly on top of LCI. We first describe Panda and then present the details of the PPSs.

### 6.1 Panda

Panda provides a message datatype, Remote Procedure Call (RPC), message passing, unordered broadcast, and totally ordered broadcast. To send a totally ordered broadcast message, the sender fetches a global sequence number from a central sequencer node using LCI's fetch-and-add. Next, the sender broadcasts the message with the sequence number, allowing the receivers to reorder messages when needed.

Panda copies data only when necessary. Senders supply I/O vectors; Panda copies the referenced client data directly into NI send buffers. At the receiving side, Panda queues incoming LCI packets until they can be copied to a destination specified by the receiver.

Panda can be configured with or without threads. MPI and Multigame use Panda-*SingleThreaded* (Panda-ST). Panda-ST performs no locking and is faster than Panda-*MultiThreaded* (Panda-MT). Also, Panda-ST disables network interrupts and relies on polling by its clients to receive messages. Panda-MT polls automatically when all threads are idle; otherwise messages are received by means of interrupts.

### 6.2 Parallel-Programming System Implementation

The four parallel-programming systems used in our evaluation represent a wide spectrum of programming interfaces. Their implementations are also distinctly different.

**MPI** is an elaborate message-passing standard. We ported MPICH [Gropp et al. 1996], a widely used MPI implementation, to Panda-ST. An MPI implementation usually generates one message for each send statement in an MPI program. Collective operations such as broadcasts and reductions, however, involve groups of processes and generate more complex communication patterns. All our MPI applications use such operations and all these operations use a

broadcast in their implementation. MPICH by default provides a broadcast implementation built on top of message-based unicast primitives. We replaced this broadcast with Panda's unordered broadcast which, in turn, uses LCI's broadcast. In contrast with the MPICH broadcast, all LCI broadcast implementations forward packets instead of messages and avoid repeated host-to-NI copying (see also Section 5.4).

**CRL** [Johnson et al. 1995] is a software DSM system, which we ported to LCI. Processes can share memory regions of a user-defined size. Processes enclose their accesses to shared regions by calls to the CRL runtime system, which implements coherent region caching. Most communication results from read and write misses, which generate a small request to a region's *home node*, zero or more invalidations from the home node to other sharers, and a reply from the home node to the node that missed. Whether a reply carries region data depends on the type of miss. This single-threaded CRL implementation blocks the application during a pending region miss. CRL uses both polling and interrupts to deliver packets. After sending a request, CRL polls until the reply arrives. Requests can arrive at any moment and may generate an interrupt. Consequently, CRL applications are sensitive to round-trip latency ( $o_s$ ,  $o_r$ , and  $L$ , in LogP terms). Peak throughput is usually not important, because many applications use small regions (which yields small data messages) and many messages carry no region at all.

**Orca** is an object-based DSM system. To communicate, processes invoke user-defined operations on shared objects [Bal et al. 1998]. Orca requires multithreading and therefore uses Panda-MT. Orca's runtime system stores each shared object either in a single processor's memory or replicates it in the memory of all processors that have a reference to the object. When a process performs an update operation on a shared object, the runtime system ships the operation's parameters to all processors that store the object; each receiving processor performs the operation. For nonreplicated objects Orca uses Panda RPC to ship the operation; for replicated objects Orca uses Panda's totally ordered broadcast.

**Multigame** (MG) is a parallel game-playing system. Given the rules of a board game and a board evaluation function, it plays the game by searching for good moves in a game tree. To avoid re-searching positions that have been investigated before, these positions are cached in a hash table that is partitioned among the compute nodes. To access a remote hash-table entry, a process sends its current position (a small, 32-byte job descriptor) to the entry's owner and starts working on another job [Romein et al. 2002]. The owner looks up the entry and continues to work on the job (in the class of games discussed here, no results have to be reported back to the sender). Table-access messages are small one-way messages to arbitrary destinations. Processes usually have enough work queued and need not wait for messages; latency in the LogP sense ( $L$ ) is therefore relatively unimportant. Instead, performance is dominated by send and receive overhead. To reduce the impact of these overheads, Multigame aggregates messages before transmitting them. Multigame uses polling to receive messages.

Table V. PPS Unicast Performance Summary for l-pio on the PPro Platform (Top) and l-sdma on the P3 Platform (Bottom). The Latency Numbers are for PPS-Specific Round-Trip Communication; the Throughput Figures are One-Way

Unicast performance on the PPro platform			
PPS	Latency ( $\mu$ s)		Throughput (Mbyte/s)
LCI/CRL	20.4/23.3	(+14%)	64/54 (-16%)
LCI/Panda-ST/MPI	20.4/27.0/42.3	(+107%)	68/66/64 (-5%)
LCI/Panda-MT/Orca	20.4/28.9/46.0	(+126%)	63/57/52 (-18%)

Unicast performance on the P3 platform			
PPS	Latency ( $\mu$ s)		Throughput (Mbyte/s)
LCI/CRL	18.1/18.6	(+3%)	158/150 (-5%)
LCI/Panda-ST/MPI	18.1/18.9/22.2	(+23%)	159/158/158 (-1%)
LCI/Panda-MT/Orca	18.1/19.8/22.2	(+23%)	157/147/145 (-8%)

Table VI. PPS Multicast Performance for l-pio on the PPro Platform (Top) and l-sdma on the P3 Platform (Bottom) with 64 Nodes. The LCI and Panda-MT Numbers for Orca also Include a Fetch-and-Add Request

Multicast performance on the PPro platform			
PPS	Latency ( $\mu$ s)		Throughput (Mbyte/s)
LCI/Panda-ST/MPI	58.5/68.3/77.6	(+33%)	33/31/31 (-6%)
LCI/Panda-MT/Orca	76.5/94.4/113.3	(+48%)	33/31/31 (-5%)

Multicast performance on the P3 platform			
PPS	Latency ( $\mu$ s)		Throughput (Mbyte/s)
LCI/Panda-ST/MPI	28.9/33.7/36.5	(+26%)	99/76/76 (-23%)
LCI/Panda-MT/Orca	37.9/43.4/47.1	(+24%)	99/76/76 (-23%)

### 6.3 Parallel-Programming System Performance

Table V summarizes the minimum latency and maximum throughput of PPS-specific unicast-based operations. The table also shows the cost of similar communication patterns when executed at lower software levels. These numbers were obtained using benchmarks that send messages of the same size as the PPS-level benchmark, but do not perform PPS-specific actions such as updating coherence-protocol state. The results for different levels are separated by slashes; performance differences between levels are caused by layer-specific overheads. The percentage between brackets is the performance loss between the LCI and the PPS level. The performance figures were measured using l-pio on the PPro platform and l-sdma on the P3 platform since these combinations in most cases offer the best match for applications (see Section 7.2).

For CRL, we show the cost of a clean write miss (a small control message to the home node followed by a reply with data). For MPI, we show the cost of a round-trip message using the *MPI\_Send/MPI\_Receive* primitives. For Orca, we show the cost of a null operation on a remote, non-replicated object, which results in an RPC. No results are shown for Multigame, because a Multigame rule set does not correspond to a simple communication pattern.

Table VI contains a performance summary of the multicast-based PPS primitives. For MPI, we show the cost of a broadcast. For Orca, we show the cost of

Table VII. Application Characteristics and Run Times for l-sdma on the PPro and P3 Platform.  $T_1$  is the Execution time (in Seconds) on One Processor;  $E_{64}$  is the Parallel Efficiency on 64 Processors (Speedup Divided by 64)

Appl.	PPS	Communication Pattern	PPro platform		P3 platform	
			$T_1$	$E_{64}$	$T_1$	$E_{64}$
ASP	MPI	Pipelined Broadcast	63.3	1.39	17.8	2.52
Awari	Orca	Arbitrary RPC	452.4	0.44	85.8	0.25
Barnes	CRL	Arbitrary RPC	80.5	0.25	15.5	0.14
LEQ	Orca	Simultaneous Broadcast	643.1	0.44	212.5	0.64
Puzzle-4	MG	Small Unicast	209.0	0.65	39.4	0.68
Puzzle-64	MG	Large Unicast	209.0	0.77	39.4	0.79
QR	MPI	Broadcast + Reduce	58.7	0.74	14.8	0.73
Radix	CRL	Arbitrary RPC	4.6	0.19	1.0	0.13
SOR	MPI	Neighbor exchange	27.9	0.64	13.7	1.05

an operation on a replicated object (which results in an F&A operation followed by a broadcast). Since CRL and Multigame use broadcasting only for operations that are not performance-critical in our applications (e.g., statistics gathering), there are no natural broadcast benchmarks for these PPSs.

A key observation is that on the PPro platform, unicast PPS-level latencies are up to 126% larger than LCI-level latencies (for multicast 48%). However, on the P3 platform the latency increase is relatively small: 23% for unicast and 26% for multicast. These overheads are due to demultiplexing, fragmentation and reassembly, locking, and procedure calls in Panda and the PPSs. Since these relatively high overheads are independent of the underlying LCI implementation, they reduce the relative differences in latency at the PPS level.

The impact of PPSs on throughput is much smaller than on latency: the PPSs decrease LCI-level unicast throughput by no more than 18% on the PPro platform and 8% on the P3 platform, because they all carefully avoid unnecessary copying. The broadcast throughput figures for the PPro platform show only a minor performance drop, since at the PPS level the bottleneck is not at the host but at the NIs: each broadcast fragment received at an NI has to be delivered locally and in addition must be forwarded up to two times, which in turn causes sliding window acknowledgments. Finally, the F&A operations which are used to implement totally ordered broadcast do not have a noticeable impact on throughput.

## 7. APPLICATION PERFORMANCE

This section studies the application-level impact of low-level implementation decisions. The applications used in this section were selected to cover a variety of parallel-programming systems and communication patterns (see Table VII). We will first discuss the applications used, as well as the application-specific details that play an important role in the performance results.

**ASP** (All-pairs Shortest Paths) computes the shortest path between all nodes in a graph; in this paper we use 1024 nodes. In each iteration, one processor broadcasts a 4-Kbyte matrix row. The algorithm iterates over all of this processor's rows before switching to another processor's rows.

**Awari** creates an endgame database for Awari, a two-player board game. In this article, we compute the endgame database for 13 stones. The program starts with the game's end positions and then makes reverse moves. When a processor updates a board's game-theoretical value it must recursively update the board's ancestors in the search graph. Since the boards are randomly distributed across all processors, a single update may result in several remote update operations. To reduce communication overhead, Awari aggregates remote updates. Awari's performance is determined by the RPCs that are used to transfer the accumulated updates. Only recently, all possible (48) endgame databases have been computed on our P3 cluster, thereby providing the best move in any possible position [Romein and Bal 2003], that is, effectively solving the game. To obtain better parallel efficiency than the implementation discussed in this article, Awari was restructured, using latency-hiding techniques as in the Puzzle application discussed below.

**Barnes** simulates a galaxy using the Barnes–Hut  $N$ -body algorithm. We use a configuration of 16,384 bodies. All simulated bodies are stored in a shared oct-tree. Tree nodes and bodies are represented as small CRL regions (88–108 bytes). Each processor owns a subset of the bodies. Most communication takes place during the force computation phase. In this phase, each processor traverses its slice of the shared tree to compute for each of its bodies the interaction with other bodies or subtrees. Due to its small regions, Barnes has a low data rate (see Figure 11). The packet rate, however, is high ( $\approx 10,000$  packets/s/processor). More efficient implementations of Barnes–Hut are known, but for this paper we rather focus on the relative implementation efficiencies for the given traffic pattern.

**LEQ** is an iterative linear equation solver. We use a configuration of 1000 equations. Each iteration refines a candidate solution vector  $x_i$ . To produce its part of  $x_{i+1}$ , each processor needs access to all elements of  $x_i$ . At the end of each iteration all processors therefore simultaneously broadcast their 128-byte partial solution vectors. Next, they synchronize to decide on convergence.

**Puzzle-4 and Puzzle-64** are two instances of the same application, Puzzle. Puzzle performs a parallel IDA\* search to solve the 15-puzzle, a single-player sliding-tile puzzle. Puzzle-4 aggregates at most 4 jobs before pushing them to another processor; Puzzle-64 accumulates up to 64 jobs. Both programs solve the same problem, but Puzzle-4 sends many more messages than Puzzle-64. Since jobs are pushed rather than pulled, little time is spent waiting for new jobs, and good speedups can be attained.

**QR** is a parallel implementation of QR matrix factorization. For the measurements in this article, a matrix of  $1024 \times 1024$  doubles is used. In each iteration, one column, the *Householder vector*  $H$ , is broadcast to all processors, which update their columns using  $H$ . The current upper row and  $H$  are then deleted from the data set so that the size of  $H$  decreases by 1 in each iteration. The vector with maximum norm becomes the Householder vector for the next iteration. This is decided with a reduce-to-all collective operation to which each processor contributes two integers and two doubles.

**Radix** sorts an array of random integers using a parallel radix sort algorithm. We use a configuration of 3,000,000 integers, with a radix of 128. Each

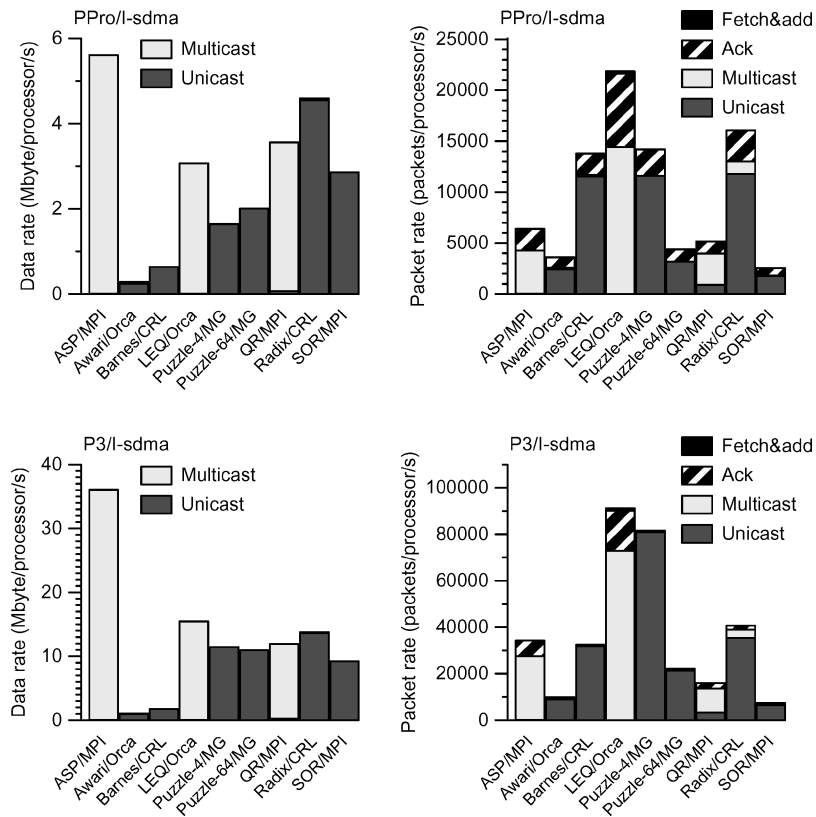


Fig. 11. Application data and packet rates for l-sdma on 64 processors, on the PPro platform (top) and P3 platform (bottom). Note that the scales are different.

processor owns a contiguous part of the array. Each part is subdivided into 1-Kbyte CRL regions, which act as software cache lines. Communication is dominated by the permutation phase, in which each processor moves integers in its own array partition to other partitions. This phase is very communication-intensive and leads to Radix’s high data rate. After the permutation phase, each processor reads the new values in its partition and starts the next sorting iteration.

**SOR** (Successive over-relaxation) solves discretized Laplace equations. The program uses red-black iterations to update a  $1536 \times 1536$  matrix. Each processor owns an equal number of contiguous matrix rows. In each iteration, processors exchange border rows (12 Kbyte) and perform a single-element reduction to decide on convergence.

Table VII summarizes information about all applications: the PPS that an application runs on, its most important communication pattern, its sequential run time ( $T_1$ ), and parallel efficiency on 64 processors ( $E_{64}$ ), on the PPro and P3 platform. The performance numbers in this table were measured using l-sdma. Although we use small input sets, most applications achieve an efficiency of at least 50%. Radix and Barnes have the lowest efficiency due to their

communication patterns (synchronous RPCs with little opportunity to overlap communication with computation). At the other end of the performance spectrum, ASP shows a significant superlinear speedup due to caching effects. The large arrays used do not fit in the cache in the sequential version of the algorithm, while in the parallel version they do. Furthermore, ASP's broadcasts can be pipelined, allowing for a good overlap of communication with computation. The other applications fall in between these two extremes.

The sequential run times in Table VII on the PPro and P3 platform are for most applications consistent with the factor five difference in processor speed of these platforms. This performance improvement is attained fully when the application's main computation mostly fits in the processor's L1 and/or L2 cache, which on the P3 platform have scaled up in performance with the same factor. However, applications that are more memory bound (ASP, LEQ, QR, and SOR) are hurt by the fact that *memory* latency has only been improved by a factor of about two (66 MHz EDO ram with a measured latency of 236 ns versus 133 MHz SDRAM with a measured latency of 113 ns). For SOR, which sequentially completely thrashes its cache, this influence is most extreme: only a factor of two performance gain is attained.

Previous sections have shown that communication performance has improved less than a factor five on the P3 platform compared to the PPro platform (see, e.g., Tables III and IV). For several applications, the parallel efficiencies are indeed lower on the P3 platform; this can be seen especially for the latency-bound applications Awari, Barnes, and Radix. However, for ASP, LEQ, QR, and SOR, the performance advantage of having more overall cache in parallel runs is significantly higher on the P3 platform. This explains why for these applications the parallel efficiency is almost sustained (QR) or actually improves (ASP is an extreme example of this effect).

Figure 11 gives average per-processor data and packet rates for each application, broken down according to packet type. Data and packet rates refer to inbound traffic, so a broadcast is counted as many times as the number of destinations (63). The figure clearly illustrates that all applications used are dominated by one type of traffic: either unicast or multicast. Especially on the PPro platform, the acknowledgment rate is quite high on average for both the multicast-dominated applications (ASP, LEQ, and QR) and for the unicast-dominated applications that send to arbitrary destinations (Awari, Barnes, Puzzle-4, and Radix). This is an effect of the rather small window size for l-sdma on the PPro platform with 64 nodes (on the P3 platform the send windows are twice as big due to the larger NI memory space). For these applications, there is usually also not enough return traffic to allow acknowledgments to be piggybacked.

### 7.1 Impact of Reliability Implementation

Figure 12 shows the influence of the implementation of reliability protocols on application performance. In this figure, the performance for implementation l-sdma is always set to 1.0, representing the (normalized) parallel run time of the l-sdma implementation on both the PPro and the P3 platform. The results for

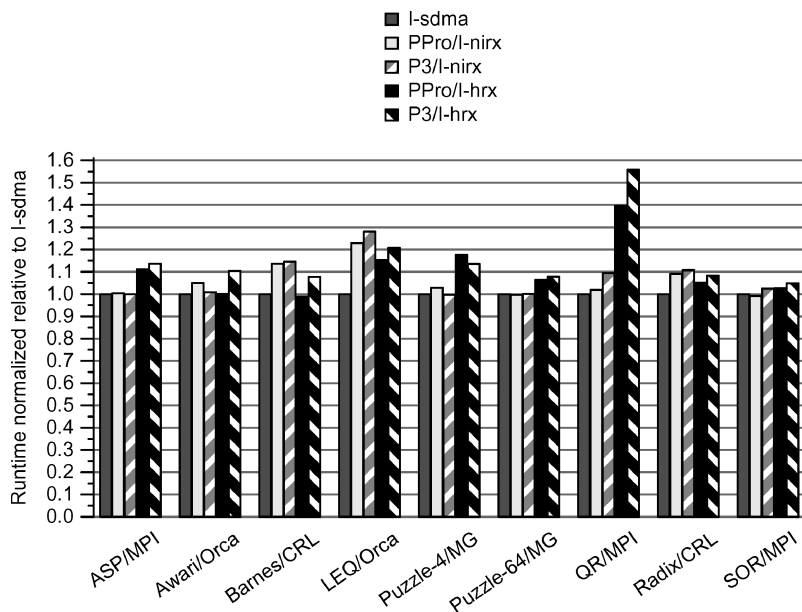


Fig. 12. Application-level impact of different reliability implementations.

the l-nirx and l-hrx implementations are grouped together, thereby highlighting the similarity in performance trends found on both platforms.

Based on these performance figures, three basic (partially overlapping) application classes can be determined:

- Applications that are sensitive to l-nirx’s NI-based reliability support. This is the case for Barnes, Radix, and LEQ.
- Applications that are sensitive to l-hrx’s host-based reliability support. This is the case for the Puzzle applications, LEQ, and to a lesser extent also for Awari, Barnes, Radix, and SOR.
- Multicast applications that are mostly sensitive to host-level forwarding; this is especially true for QR, but also for ASP. It should be noted that the implementation of reliability and multicast forwarding are not completely independent. In particular, l-hrx uses *host*-level multicast forwarding, which affects the performance results. Since QR is very sensitive to multicast latency, its l-hrx results show important slowdowns. The multicast applications will also be discussed separately in Section 7.4.

We will now discuss these three basic classes in more detail. The extra robustness provided by l-nirx’s NI-level retransmission protocol does not always come for free. Barnes and Radix send a large number of small messages to arbitrary destinations in a rather short time; they run up to 15% more slowly on l-nirx. For LEQ, this slowdown is even 23%. Due to their communication patterns, these applications are sensitive to NI overhead: for irregular communication, acknowledgments can be piggybacked less often, so separate acknowledgments have to be scheduled and processed. This overhead increases when

retransmission support is added to the NI firmware. Especially LEQ suffers from this: as shown in Figure 11 the simultaneous multicasts performed by this application cause the highest overall packet rate (both for data and acknowledgments) in our application set. Although the l-nirx's LogP  $g$  is indeed higher than l-sdma's, this does not fully explain LEQ's slowdown on l-nirx. We examined this further using a low-level *all-to-all* throughput benchmark; this showed that the actual NI-level overhead for this traffic pattern is indeed higher than can be exposed by the (point-to-point) LogP  $g$  benchmark or the *one-to-all* multicast throughput benchmark.

l-hrx's host-level retransmission support can have different impacts, depending on the application's traffic patterns. The Puzzle applications are latency tolerant: all communication is one-way and processes generally do not wait for incoming messages. As a result, send and receive overhead are more important than NI-level latency and occupancy. Since host-level retransmission increases send and receive overheads (see Table III), it affects the performance of these applications. Puzzle-4 uses more messages than Puzzle-64 to transfer the same amount of data, so it incurs these higher send and receive overheads more often and therefore suffers more than Puzzle-64.

SOR also performs slightly worse on l-hrx, but for a different reason: SOR suffers from host-level sliding-window stalls. Since all processes send a row to their neighbor at approximately the same time, no receiver transmits its half-window acknowledgments fast enough to prevent window stalls. The acknowledgment is not sent until the receiver itself stalls and needs to poll. In l-sdma and l-nirx, the sliding window is on the NI, so that the host can copy packets to the NI even if the send window has closed. Also, if an NI's send window to one NI closes, that NI can still send packets to other NIs.

LEQ's simultaneous broadcasts (as discussed above) also have an effect on its l-hrx performance, only here the additional overhead is on the host rather than on the NI. For the remaining applications, the effects of l-hrx on performance due to additional host overhead and latency are smaller.

## 7.2 Impact of Data-Transfer Method

Figure 13 shows the impact of using either DMA or PIO at the sender. Since l-sdma and l-pio support the same communication API, all applications can be run with alternative data-transfer methods without modifications (implementing RCPY as an alternative data-transfer method would require significant changes to the internals of all PPSs used). Note that LCI's communication primitives in the l-sdma implementation write the message data to a host-resident packet queue instead of the send queue in firmware memory (alternatively, waiting for the NI to signal completion of a DMA transfer will be more costly in general). This is required to implement the LCI communication primitives correctly: an application may, and often will, modify its data after transmission. When using PIO-based data transfers, as done in l-pio, overhead due to the host send queue is avoided.

The figure shows clearly that on the P3 platform SDMA-based transfers are more efficient than PIO for most applications. This can be explained by the

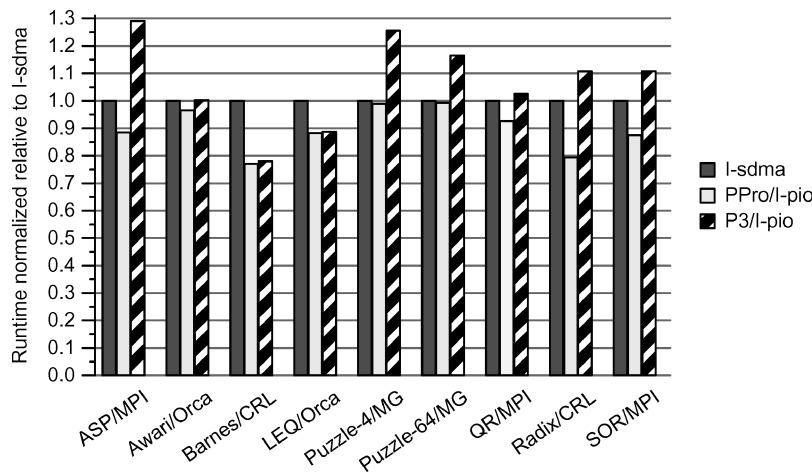


Fig. 13. Impact of SDMA vs. PIO on the PPro and P3 platform.

differences in the overhead and latency curves shown earlier in Figure 7: on the P3 platform SDMA-based transfers become attractive at a smaller message size. Nevertheless, for Barnes and LEQ, which send many small messages, using SDMA results in poor performance, and for QR and Awari we see little or no difference. On the other hand, on the PPro platform the performance advantages attained by DMA'ing the data to the NI send queue are minimized by the overhead due to the transfer via the host send queue. For none of the applications in our suite, the SDMA-based implementation has better performance on the PPro platform.

For best overall performance, a distinction might be made between “small” (PIO) and “large” (SDMA) packets [Prylli et al. 1999]. It should be noted, however, that the boundary value is not a platform-dependent constant in general: for latency-bound applications, the point to switch strategies may be rather different than for applications that are mainly sensitive to host or NI overhead.

To further investigate the impact of the data-transfer method at the application level we conducted several experiments with SOR, since this application exhibits a relatively high data rate, even though its packet rate is modest according to Figure 11. These conditions are in principle most favorable for RCPY transfers, and we see that application performance for the original problem size actually is indeed slightly better with l-rcpy on both platforms.

Figure 14 shows the results for increasing column size, and hence, increasing the amount of communication per iteration. The amount of computation per iteration was kept constant by reducing the row size accordingly. In addition, the number of iterations was also kept fixed (the standard convergence criterion requires more iterations for SOR on non-square grids). The figure shows that a significant increase in communication causes l-rcpy to become relatively more beneficial. We compare the run times relative to l-sdma with the same column size, so changes in parallel efficiency due to a change in column size are not visible in the graph. In fact, SOR's efficiency for larger column sizes deteriorates significantly due to the increasing communication, but for l-sdma more than for l-rcpy.

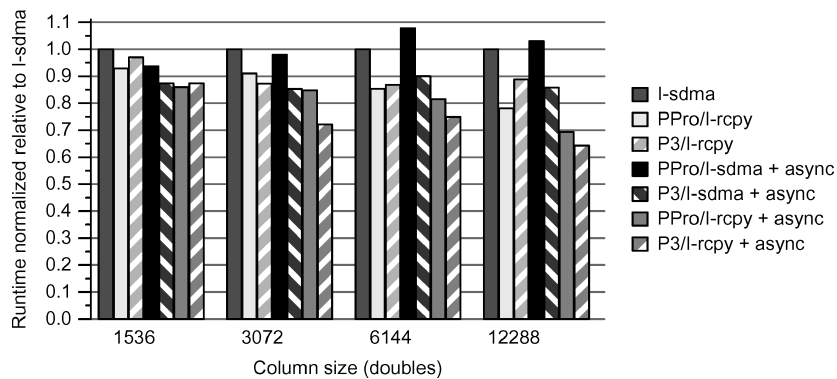


Fig. 14. Impact of zero-copy transfers on SOR for different problem sizes.

The asynchrony of the data-transfer primitives in l-sdma and l-rcpy also allows application-specific optimizations that are not possible with synchronous data transfers. By slightly reordering the column computations in SOR, a node can send the columns that are needed by its neighbors during the next iteration *before* computing its remaining columns. With l-pio, however, the possibility to overlap communication and computation are far more limited, since a large fraction of the data-transfer overhead consists of an inherently synchronous PIO-transfer to the NI at the sender. As shown in Figure 14, using asynchronous column transfers with l-sdma (indicated by “+ async”) indeed improves performance on the P3 platform. On the PPro platform, this only helps noticeably for the original column size (1536). However, asynchronous column transfers with l-rcpy consistently improve SOR’s performance significantly on both platforms.

### 7.3 Impact of MTU

Figure 15 shows the influence of the MTU (packet size) on application execution time. Interestingly, no application benefits significantly from a large, 4-Kbyte MTU. This can largely be explained by the fact that the parallel applications examined in this article are not particularly throughput-sensitive, as is quite common in our experience (this is also reported elsewhere; see, e.g., Martin et al. [1997]).

Although SOR has the largest average message size (about 12 Kbyte) among our applications, these messages are transferred in short exchanges between nodes, so the differences found in the (streaming, one-way) throughput benchmark (see Figure 8) are not directly applicable.

Barnes, LEQ, and QR in fact perform up to 26% *worse* with a 4-Kbyte MTU on the PPro platform. In l-4k, each data packet is acknowledged by a separate, NI-generated acknowledgment packet. These acknowledgment packets increase NI occupancy and this affects these applications. On the P3 platform the maximum slowdown is less (up to 13%) since its NI has more memory, so the sliding windows can be correspondingly larger. Awari and Barnes mainly send round-trip messages and round-trip latency depends on NI occupancy. In LEQ, all processors wait for the termination of an all-to-all communication phase; increased NI occupancy will increase the duration of this phase.

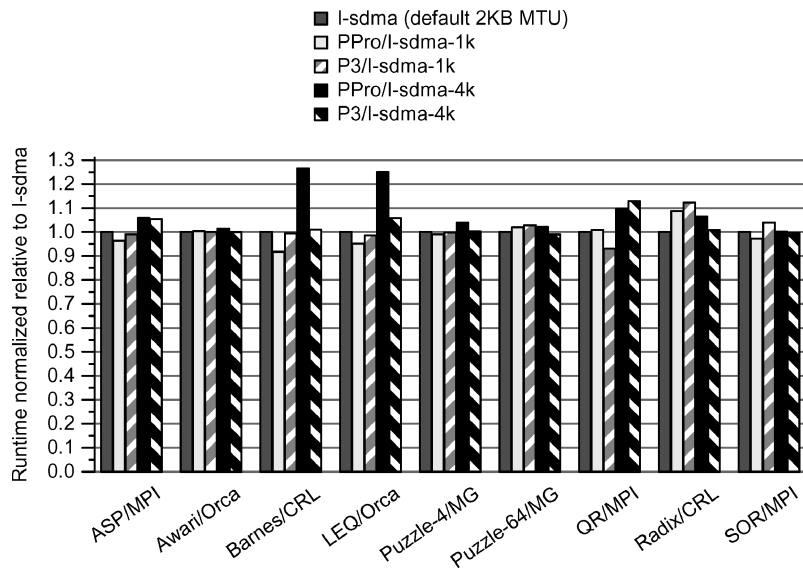


Fig. 15. Application-level impact of different MTU sizes.

Radix suffers slightly with a small, 1-Kbyte MTU, because its dominant message size is just over 1 Kbyte. As a result, all messages require one full and one nearly empty data packet. With a 2-Kbyte or a 4-Kbyte MTU, only one data packet is needed.

Summarizing, if the NI buffer space requirements are to remain constant, a 2-Kbyte MTU appears to find the right trade-off between high throughput and low acknowledgment overhead on both the PPro and P3 platform.

#### 7.4 Impact of Multicast Implementation

Figure 16 shows the l-sdma and l-sdma-hmc performance of the applications that significantly use multicast. All four applications perform better with NI-level multicasting than with host-level multicasting, for various reasons.

In ASP, processors take turns sending a series of rows. The current sender can pipeline the broadcasts of its rows. Due to this pipelining, receivers are often still working on one iteration when broadcast packets for the next iteration arrive. With l-sdma-hmc, these packets are not processed until the receivers invoke a receive primitive. Consequently, the sender is stalled, because acknowledgments do not flow back in time. To improve performance, we augmented ASP with application-level polling statements (using the MPI primitive *MPI\_Iprobe*) and Figure 16 shows these improved numbers. With l-sdma, this problem does not occur, because acknowledgments are sent by the NI, not by the host processor. The remaining performance difference is due to the processor overhead caused by failed polls and host-level forwarding: the time spent on these activities is not available to the application.

Since ASP requires high multicast throughput for good performance, using a binary forwarding tree is essential. The standard MPICH host-level forwarding implementation always uses binomial trees, so we modified MPICH to support

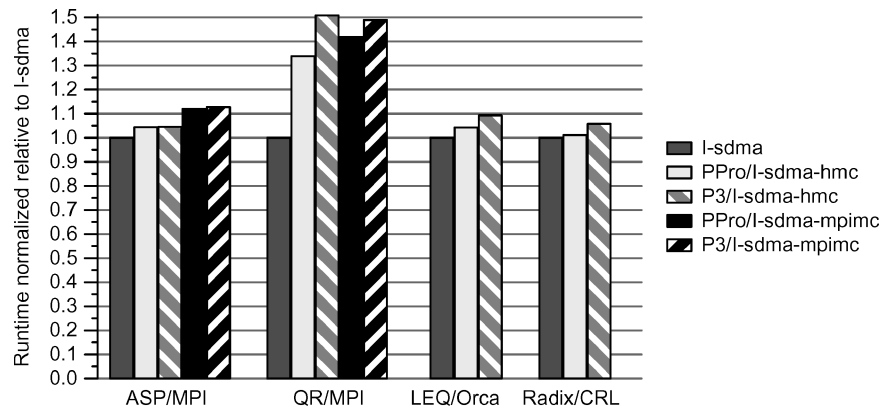


Fig. 16. Application-level impact of different multicast implementations.

binary trees as well (in our modified version, the multicast tree topology can be specified by means of a command-line option). The results are shown in Figure 16 as *l-sdma-mpimc*. The results for MPICH would be 58% worse when using its default binomial forwarding strategy.

QR is very sensitive to broadcast latency, because at the start of each iteration, each receiving processor must wait for the arrival of a new Householder vector  $H$ , which is broadcast. There is no opportunity to pipeline broadcasts and hide this latency, because all processors synchronize in each iteration through a reduce-to-all operation. Also, due to pivoting, the sending processor changes in almost every iteration.

QR runs up to 34% more slowly on *l-sdma-hmc* on the PPro platform; on the P3 platform this slowdown is even 51%. Host-level forwarding increases broadcast latency and reduces the time available for executing application code. Figure 16 shows that the results for the default multicast implementation in MPICH (*l-sdma-mpimc*) is similar to LCI's host-level forwarding in *l-sdma-hmc*. In both cases, binomial trees were used.

Although LEQ is dominated by totally ordered broadcast traffic, the performance difference between host-level forwarding and NI-level forwarding is relatively small. Since all processes simultaneously broadcast small messages, NI and host occupancy influence LEQ's performance more than broadcast latency (see the discussion of LEQ in Section 7.1).

Radix predominantly employs unicast communication, but the barriers it uses during every iteration exploit multicast. Multicast only accounts for about 10% of the packets sent by Radix, but on the P3 platform the latency reduction due the NI-level multicast is quite noticeable in the overall performance on *l-sdma*.

## 7.5 Discussion

There are several conclusions we can draw from the higher-level evaluation that are not a simple extrapolation of our findings at the microbenchmark level.

*Data-Transfer Method.* The application-level results presented in the previous sections showed that the choice of a data-transfer method generally has the largest impact on performance. The optimal choice was shown to be both platform and application dependent. Using DMA-based message passing or a primitive for remote-memory copies instead of PIO-based message passing generally gives better performance for throughput-bound applications. Most of the applications in our suite, however, are not bandwidth-hungry and send small or medium-size messages, which is typical for many fine-grained (non-trivial) parallel applications. Also, to fully benefit from this type of data transfer, it is important to restructure applications or runtime systems to use asynchronous transfers, so that data transfers and computations can be overlapped. For remote-memory transfers, runtime systems must also be changed so that data is copied directly to its final destination (in the unmodified systems the sender usually does not know to which remote address the data should be sent, so this may result in additional bookkeeping and synchronization). Making the required changes is possible with a reasonable amount of effort on systems like CRL and MPI, which employ a data-shipping paradigm, but it is harder for function-shipping systems such as Orca.

*Reliability.* The additional overhead due to a reliability protocol at the NI or the host also had some performance impact, but it was shown to be more modest: usually less than 15%. It should be noted, however, that LCI's base implementation makes strong assumptions about the underlying hardware: it assumes a programmable network interface and reliable network links. Furthermore, the fact that this overhead is relatively limited can be interpreted as a positive outcome for production clusters that need additional reliability guarantees. NI-level retransmission allows host code and NI code to be executed in parallel, but increases NI occupancy. As a result, NI-level retransmission reduces performance for applications and PPSs that generate messages at a high rate. This is especially true when many round-trips are required, that is, when increased message latency cannot be hidden. Host-level retransmission does not exploit parallelism between the host and the NI and therefore yields the largest round-trip latency in microbenchmarks. In several cases, however, increased host occupancy is less visible at the application level than the increased NI occupancy that comes with NI-level retransmission. A notable exception is the Multigame programming system, which is latency tolerant: outbound messages are one-way and can be aggregated. In such a system, send and receive overhead are more important.

*MTU.* While the use of larger MTU sizes results in better throughput in microbenchmarks, the application-level benefit of a large MTU is small and sometimes negative. There are several reasons for this phenomenon:

- Many applications are not throughput-bound.
- Copying at the receiving side destroys the advantage of a large MTU for applications that *are* throughput-bound. This effect, however, is specific to the PPro platform.

—In l-sdma and l-pio, a larger MTU implies fewer send and receive buffers and more acknowledgments. This is specific to credit-based protocols that employ a static receive buffer reservation scheme.

*Multicast.* In our experiments, NI-level multicast forwarding almost always yields better performance than host-level multicast forwarding, even though the NI itself is significantly slower than the host. But only in one case it was found that the impact of an optimized multicast implementation had more influence on the application run time than the choice of data-transfer method. The smaller impact of a host-level multicast implementation is partly due to the smart use of NI memory, however. Most existing host-level multicast implementations transfer a packet to the NI multiple times, once per forwarding destination.

*Influence of the Parallel-Programming System.* Our results show clearly that different programming systems and different applications are sensitive to different parameters of the underlying communication system. Applications that synchronize frequently tend to be sensitive to increases in latency and occupancy parameters. Synchronization patterns are frequently built into the programming system that an application runs on. Orca, for example, uses remote procedure calls (RPCs) and totally ordered broadcasts for all communication. The sender of an RPC request must wait for an RPC reply and the sender of a totally ordered broadcast must fetch a sequence number which involves a round-trip to a sequencer node. In MPI, the implementation of collective-communication operations may synchronize participating processors. In CRL, read and write misses are resolved through round-trip messages to a home node. Unicast latency and occupancy are increased by moving retransmission support to the NI; multicast latency is decreased by forwarding multicast packets on the NI. Multigame applications are latency tolerant, and again this is a result of the way the Multigame programming system is structured: since it only sends asynchronous one-way messages and also needs only little other synchronization, it is more sensitive to host-level send and receive overhead.

## 8. RELATED WORK

Several papers discuss possible uses of the NI in the implementation of high-speed communication systems.

Karamcheti and Chien [1994] studied the division of protocol tasks between network hardware and host software for CM-5 Active Messages, a communication system similar to LCI. They argue for higher-level services (ordering, reliability, flow control) in the network hardware to reduce costs in the software communication system. Our work also considers multicast *and* it considers the impact of different communication system implementations on PPSs and applications.

Krishnamurthy et al. [1996] studied the role of programmable NIs in different implementations of a single PPS, Split-C. Their work focuses on NI support for remote memory access and discusses neither reliability nor multicast.

Prylli et al. [1999] describe an MPI implementation which uses a combination of PIO and DMA transfers. In their implementation, the choice of the transfer

method is made statically. However, as discussed in Sections 5.2.4 and 7.2, an optimal choice of the point to switch strategies depends both on the platform and the communication pattern.

NI-supported synchronization can reduce the number of asynchronous requests that have to be processed by the host processor. For this reason, Bilas et al. [1999b] use NI-supported locks to implement distributed locking in a shared virtual-memory system. For the same reason, LCI provides an NI-level fetch-and-add primitive (see Section 4.3); it is used by the Orca system to reduce the costs of totally ordered multicasts.

LCI's reliability and multicast implementations make similar assumptions to existing and proposed protocols. Fast Messages [Pakin et al. 1995] and PM [Tezuka et al. 1997] assume that the hardware never drops or corrupts packets. Active Messages II combines an NI-level reliability protocol with a host-level sliding-window protocol for reliability and flow control [Chun et al. 1997]. Several papers describe NI-supported multicast protocols [Bhoedjang et al. 1998a; Huang and McKinley 1996; Kesavan and Panda 1997; Verstoep et al. 1996]. We are the first to compare efficient NI-level and host-level multicasts *and* their impact on application performance.

Araki et al. [1998] used LogP measurements [Culler et al. 1996] on several user-level communication systems with different reliability strategies. They compare systems with different programming interfaces (e.g., memory-mapped communication and message passing) and do not consider multicast. We compare implementations of *one* interface and also consider the application-level impact of different reliability and multicast designs.

We have considered only the recovery from transient network errors. Recovery from permanent link failures involves discovering new routes. Work by Tang and Bilas [2002] indicates that such recovery does not have a strong impact on critical-path software, which is the focus of this article.

Martin et al. [1997] studied the sensitivity of Split-C applications to LogP parameter values. They varied individual LogP parameters using delay loops in a single communication system (Active Messages). We look at a much smaller set of parameter values, but we know that each set corresponds naturally to a particular communication-system implementation. We can therefore correlate parameter values and communication systems. We also consider a larger range of PPSs and show that some are more sensitive to particular parameters than others.

NI-supported zero-copy data-transfer mechanisms are described among others by Welsh et al. [1997] and Chen et al. [1998]. Welsh et al. resolve NI-level TLB misses through interrupts. Such interrupts are processed by the kernel, which updates the NI-level TLB. The user-managed TLB approach by Chen et al. relies on caching entries from a host-level master TLB in NI memory. When an NI-level TLB miss occurs, the NI fetches an up-to-date TLB entry from host-level kernel memory. Our scheme (described in Section 5.2.3) resolves misses asynchronously and relies on programmed I/O at the sending side and transfer redirection [Dubnicki et al. 1997a] at the receiving side as backup mechanisms when a miss occurs. NI-level TLB misses are signaled to the host without using interrupts, by transferring a miss notification to the

receive queue. In contrast with the alternative TLB schemes just discussed, our current implementation of this scheme is not multiprogramming-safe.

Bilas et al. [1999a] identify bottlenecks in DSM systems. Their simulation study revolves around the same layers as used in this article: low-level communication software and hardware, PPS, and applications. Bilas et al. [1999a] use memory-mapped communication and analyze the performance of page-based and fine-grained DSM systems. Our work uses packet-based communication and PPSs that implement message passing or object sharing.

Although our work has focused on Myrinet, there are several other networks for which our analysis regarding the work division between the host and NI is relevant. Cluster interconnection technologies that are also implemented with a programmable NI to execute a variety of protocol tasks include QsNet of Quadrics Supercomputer World [Petrini et al. 2001], the interconnection network in the IBM SP series [Chang et al. 1996], and DEC's Memory Channel [Fillo and Gillett 1997]. Other technologies (e.g., Virtual Interface (VI) Architecture implementations ServerNet and GigaNet [Speight et al. 1999]) place most or all protocol processing in hardware. This approach can in principle result in lower NI or host overheads, but the question remains if this potential performance increase outweighs the significant loss of flexibility. Software-based implementations of the VI architecture also exist, however [Begel et al. 2002].

Finally, InfiniBand [InfiniBand Trade Association 2002] is an ambitious standardization initiative for System Area Networks (connecting processor nodes, I/O platforms as well as individual I/O devices), introducing a new multi-layer, high-speed network architecture. Both acknowledged (reliable) and unacknowledged (unreliable) unicast communication are defined. Atomic operations (like LCI's fetch-and-add) are also included, but for multicast communication only an unreliable datagram version is available. We expect that, if InfiniBand actually gains ground, for the foreseeable future many implementations will make use of NIs with modifiable firmware, as on Myrinet, due to the complexity and evolving nature of the standard.

## 9. CONCLUSIONS AND FUTURE WORK

We studied four important implementation issues for low-level communication systems: NI- versus host-based reliability protocols, SDMA- versus PIO-based message passing and remote-memory copy, maximum transfer unit, and multicast protocols. We compared different implementations, mostly of the same API, using microbenchmarks, PPS-specific benchmarks, and application measurements. The experiments were performed on two generations of cluster hardware, thus giving insight into how computer architecture influences the performance aspects investigated.

Regarding the relative importance of the different implementation choices we draw several conclusions. The choice of a data-transfer method which fits both the architecture, parallel-programming system *and* application generally has the highest impact on performance. Implementing multicast at the lowest layer can have important performance advantages as well, but mainly for applications that are specifically dependent on multicast latency, either explicitly

or implicitly via a barrier implementation. The introduction of a reliability protocol can degrade performance somewhat, but whether a host- or NI-based retransmission protocol performs best, has been shown to depend highly on the application. Only in one case we found that using a higher MTU than the default resulted in a (slight) performance improvement.

In this article, we used well-accepted microbenchmarks to investigate low-level performance differences between the various implementations. The LogP model proved to be valuable in providing additional insights due to its explicit modeling of host overhead and NI bottleneck. In some cases, though, it was seen that contention at the network or the NI due to nontrivial traffic patterns may cause application slowdown as well. For future work, it would be interesting to look at appropriate models for this contention (e.g., as suggested by Frank et al. [1997]) that are simple and yet general enough, and to correlate this with some of the implementation choices discussed.

Most of our parallel applications achieved an acceptable efficiency on both the PPro and the P3 platform, even though some important communication performance indicators were shown not to scale with the increased host and network interface performance. The efficiency of applications that are sensitive to round-trip latency deteriorated significantly on the P3 platform, however. It is therefore likely that application restructuring (e.g., to reduce the number of synchronizations) will become increasingly more important in future work on clusters.

A recurring theme in this study is the division of work between host and NI. The NI processor can be used to reduce the load on the host processor, to reduce the number of data transfers between host and NI, to reduce the number of network interrupts, to direct inbound data to its final destination in host memory, etc. Such optimizations must be balanced against raised NI occupancy. There is, however, more to the host-NI division than processing tasks. Both the reliability protocols and the multicast protocols benefit from explicit NI buffer-space management. If a host has sufficient control over NI buffer space, it can avoid unnecessary buffering in host memory and avoid the repeated injection of almost identical multicast packets. Also, we have focused on static work divisions of general-purpose implementation aspects. An interesting topic for future research would be to consider mechanisms that allow a more dynamic shifting of functionality from the host to the NI.

Summarizing, there is no single best system. What is best depends on application requirements and the communication patterns induced by programming systems and applications. Some directions can be given, though. NI-level or host-level retransmission is somewhat dubious on very reliable network hardware, but a production environment will in general require stronger reliability guarantees than predominantly experimental environments. NI-level multicast forwarding improved performance in almost all cases in our study, but we showed under what conditions an efficiently implemented host-level forwarding scheme can also achieve acceptable application-level results.

Finally, microbenchmarks and application-based measurements on the PPro and P3 platform suggest that for future low-level communication interface designs and PPS implementations it will be advantageous to allow a flexible choice

of the data-transfer method between the host and the NI. Making an appropriate choice of the data transfers at run time can have significant performance impacts, depending both on application and PPS characteristics and the hardware platform used. An interesting option to achieve maximum performance given a system and an application is to use profiling information from previous runs to tune low-level network parameters towards the needs of the application. One possibility is to automatically adapt a bound for message sizes, to choose for PIO send transfers for small messages and DMA for larger messages. For applications that lend themselves to remote memory copy, the same technique can be used to choose between PIO, DMA and remote memory copy. From profiling runs, multicast application performance can also be inspected intelligently. If the application can be analyzed to be throughput-bound, a slim spanning tree is best. If it is latency-bound, a tree of higher fan-out is preferable. It may be advantageous to support different multicast tree shapes within one application run, and to switch on-demand, possibly even per broadcast, between trees. The choice of fan-out offers a rich range of possibilities. However, determining whether an application is throughput-bound or latency-bound *automatically* seems a challenging research topic in itself.

#### ACKNOWLEDGMENTS

We thank Matthieu Roy for his work on the remote-memory copy implementation described in the paper and John Romein for his help with the Multigame system. We thank Frans Kaashoek, Thilo Kielmann and the anonymous referees for their useful comments on this article.

#### REFERENCES

- ARAKI, S., BILAS, A., DUBNICKI, C., EDLER, J., KONISHI, K., AND PHILBIN, J. 1998. User-space communication: A quantitative study. In *Proceedings of Supercomputing'98* (Orlando, Fla.).
- AUMAGE, O., BOUGÉ, L., DENIS, A., MÉHAUT, J.-F., MERCIER, G., NAMYST, R., AND PRYLLI, L. 2000. A portable and efficient communication library for high-performance cluster computing. In *Proceedings of the IEEE International Conference on Cluster Computing*. IEEE Computer Society Press, Los Alamitos, Calif., 78–87.
- BAL, H., BHOEDJANG, R., HOFMAN, R., JACOBS, C., LANGENDOEN, K., RÜHL, T., AND KAASHOEK, M. 1998. Performance evaluation of the Orca shared object system. *ACM Trans. Comput. Syst.* 16, 1 (Feb.), 1–40.
- BEGEL, A., BUONADONNA, P., CULLER, D., AND GAY, D. 2002. An analysis of VI architecture primitives in support of parallel and distributed communication. *Concurr. Comput.: Pract. Exper.* 14, 1 (Jan.), 55–76.
- BHOEDJANG, R., RÜHL, T., AND BAL, H. 1998a. Efficient multicast on Myrinet using link-level flow control. In *Proceedings of the 27th International Conference on Parallel Processing (ICPP)*. (Minneapolis, Minn.). 381–390.
- BHOEDJANG, R., RÜHL, T., AND BAL, H. 1998b. User-level network interface protocols. *IEEE Comput.* 31, 11 (Nov.), 53–60.
- BHOEDJANG, R., VERSTOEP, K., BAL, H., AND RÜHL, T. 2000a. Reducing data and control transfer overhead through network-interface support. In *Proceedings of the 1st Myrinet User Group Conference* (Lyon, France).
- BHOEDJANG, R., VERSTOEP, K., RÜHL, T., BAL, H., AND HOFMAN, R. 2000b. Evaluating design alternatives for reliable communication on high-speed networks. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Cambridge, Mass.). 71–81.

- BILAS, A., JIANG, D., ZHOU, Y., AND SINGH, J. 1999a. Limits to the performance of software shared memory: A layered approach. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture (HPCA)*. (Orlando, Fla.). 193–202.
- BILAS, A., LIAO, C., AND SINGH, J. 1999b. Using network interface support to avoid asynchronous protocol processing in shared virtual memory systems. In *Proceedings of the 26th International Symposium on Computer Architecture (ISCA)* (Atlanta, Ga.). 282–293.
- BODEN, N., COHEN, D., FELDERMAN, R., KULAWIK, A., SEITZ, C., SEIZOVIC, J., AND SU, W. 1995. Myrinet: A gigabit-per-second local area network. *IEEE Micro* 15, 1 (Feb.), 29–36.
- BUZZARD, G., JACOBSON, D., MACKEY, M., MAROVICH, S., AND WILKES, J. 1996. An implementation of the Hamlyn sender-managed interface architecture. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Seattle, Wa.). 245–259.
- CANONICO, R., CRISTALDI, R., AND IANNELLO, G. 1999. A scalable flow control algorithm for the Fast Messages communication library. In *Proceedings of the Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing (CANPC)* (Orlando, Fla.). 77–90.
- CHANG, C.-C., CZAJKOWSKI, G., HAWBLITZEL, C., AND VON EICKEN, T. 1996. Low-latency communication on the IBM RISC System/6000 SP. In *Proceedings of Supercomputing '96* (Pittsburgh, Pa.).
- CHEN, Y., BILAS, A., DAMIANAKIS, S., AND DUBNICKI, C. 1998. UTLB: A mechanism for address translation on network interfaces. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, Calif.). 193–204.
- CHUN, B., MAINWARING, A., AND CULLER, D. 1997. Virtual network transport protocols for Myrinet. In *Proceedings of Hot Interconnects V* (Stanford, Calif.).
- CULLER, D., LIU, L., MARTIN, R., AND YOSHIKAWA, C. 1996. Assessing fast network interfaces. *IEEE Micro* 16, 1 (Feb.), 35–43.
- DRUSCHEL, P., PETERSON, L., AND DAVIE, B. 1994. Experiences with a high-speed network adaptor: A software perspective. In *Proceedings of the 1994 Conference on Communications Architectures, Protocols, and Applications (SIGCOMM)* (London, U.K.). ACM New York, 2–12.
- DUBNICKI, C., BILAS, A., CHEN, Y., DAMIANAKIS, S., AND LI, K. 1997a. VMMC-2: Efficient support for reliable, connection-oriented communication. In *Proceedings of Hot Interconnects V* (Stanford, Calif.).
- DUBNICKI, C., BILAS, A., LI, K., AND PHILBIN, J. 1997b. Design and implementation of virtual memory-mapped communication on Myrinet. In *Proceedings of the 11th Int. Parallel Processing Symp. (IPPS)* (Geneva, Switzerland). 388–396.
- FILLO, M. AND GILLET, R. 1997. Architecture and implementation of Memory Channel 2. *Dig. Tech. J.* 9, 1, 27–41.
- FRANK, M., AGARWAL, A., AND VERNON, M. 1997. LoPC: Modeling contention in parallel algorithms. In *Proceedings of the 6th Symposium on Principles and Practice of Parallel Programming (PPOPP)* (Las Vegas Nev.). 276–287.
- GROPP, W., LUSK, E., DOSS, N., AND SKJELLUM, A. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Paral. Comput.* 22, 6 (Sept.), 789–828.
- HUANG, Y. AND MCKINLEY, P. 1996. Efficient collective operations with ATM network interface support. In *Proceedings of the 25th International Conference on Parallel Processing (ICPP)* (Bloomington, Ill.). 34–43.
- INFINIBAND TRADE ASSOCIATION. 2002. *InfiniBand Architecture Specification Release 1.1*. Available from <http://www.infinibandta.org>.
- JOHNSON, K., KAASHOEK, M., AND WALLACH, D. 1995. CRL: High-performance all-software distributed shared memory. In *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP)* (Copper Mountain, Col.). 213–226.
- KARAMCHETI, V. AND CHIEN, A. 1994. Software overhead in messaging layers: Where does the time go? In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, Calif.). 51–60.
- KESAVAN, R. AND PANDA, D. 1997. Optimal multicast with packetization and network interface support. In *Proceedings of the 26th International Conference on Parallel Processing (ICPP)* (Bloomington, Ill.). 370–377.

- KIELMANN, T., BAL, H., GORLATCH, S., VERSTOEP, K., AND HOFMAN, R. 2001. Network performance-aware collective communication for clustered wide area systems. *Paral. Comput.* 27, 11, 1431–1456.
- KRISHNAMURTHY, A., SCHAUSER, K., SCHEIMAN, C., WANG, R., CULLER, D., AND YELICK, K. 1996. Evaluation of architectural support for global address-based communication in large-scale parallel machines. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Cambridge, Mass.). 37–48.
- MAQUELIN, O., GAO, G., HUM, H., THEOBALD, K., AND TIAN, X. 1996. Polling watchdog: Combining polling and interrupts for efficient message handling. In *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)* (Philadelphia, Pa.). 179–188.
- MARTIN, R., VAHDAT, A., CULLER, D., AND ANDERSON, T. 1997. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24th International Symposium on Computer Architecture (ISCA)* (Denver Colo.). 85–97.
- MPI FORUM. 1994. A message passing interface standard. *Int. J. Supercomput. Appl.* 8, 3/4.
- PAKIN, S., LAURIA, M., AND CHIEN, A. 1995. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95* (San Diego, Calif.).
- PETRINI, F., FENG, W., HOISIE, A., COLL, S., AND FRACHTENBERG, E. 2001. Quadrics network (QsNet): High-performance clustering technology. In *Proceedings of Hot Interconnects IX* (Stanford, Calif.).
- PRYLLI, L., TOURANCHEAU, B., AND WESTRELIN, R. 1999. The Design for a high-performance MPI implementation on the Myrinet network. In *Proceedings 6th European PVM/MPI Users' Group* (Barcelona, Spain). Lecture Notes in Computer Science, vol. 1697 Springer Verlag, New York, 223–230.
- ROMEIN, J. AND BAL, H. 2003. Solving the game of Awari using parallel retrograde analysis. *IEEE Comput.* 36, 10 (Oct.), 26–33.
- ROMEIN, J., BAL, H., SCHAEFFER, J., AND PLAAT, A. 2002. A performance analysis of transposition-table-driven scheduling in distributed search. *IEEE Trans. Paralle. Distrib. Syst.* 13, 5 (May), 447–459.
- SPEIGHT, E., ABDEL-SHAFI, H., AND BENNETT, J. 1999. Realizing the performance potential of the Virtual Interface architecture. In *Proceedings of the 13th International Conference on Supercomputing (ICS)*. Rhodes, Greece, 184–192.
- TANG, J. AND BILAS, A. 2002. Tolerating network failures in system area networks. In *Proceedings of the 2002 International Conference on Parallel Processing (ICPP)* (Vancouver, B. C., Canada), 121–130.
- TEZUKA, H., HORI, A., ISHIKAWA, Y., AND SATO, M. 1997. PM: An operating system coordinated high-performance communication library. In *High-Performance Computing and Networking* (Vienna, Austria) Lecture Notes in Computer Science, vol. 1225.
- TEZUKA, H., O'CARROLL, F., HORI, A., AND ISHIKAWA, Y. 1998. Pin-down cache: A virtual memory management technique for zero-copy communication. In *Proceedings of the 12th International Parallel Processing Symposium (IPPS)* (Orlando, Fla.). 308–314.
- VERSTOEP, K., LANGENDOEN, K., AND BAL, H. 1996. Efficient reliable multicast on Myrinet. In *Proceedings of the 25th International Conference on Parallel Processing (ICPP)* (Bloomington, Ill.). 156–165.
- VON EICKEN, T., BASU, A., BUCH, V., AND VOGELS, W. 1995. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP)* (Copper Mountain, Colo.). 303–316.
- VON EICKEN, T., CULLER, D., GOLDSTEIN, S., AND SCHAUSER, K. 1992. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th International Symposium on Computer Architecture (ISCA)* (Gold Coast, Australia). 256–266.
- WANG, R., KRISHNAMURTHY, A., MARTIN, R., ANDERSON, T., AND CULLER, D. 1998. Modeling and optimizing communication pipelines. In *Proceedings of the 1998 Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (Madison, W. Sc.). 22–32.
- WELSH, M., BASU, A., AND VON EICKEN, T. 1997. Incorporating memory management into user-level network interfaces. In *Proceedings of Hot Interconnects V* (Stanford, Calif.).

Received May 2002; revised May 2003; accepted October 2003