Fast Messages:
Efficient, Portable Communication for Workstation Clusters and MPPs

Scott Pakin, Vijay Karamcheti, and Andrew A. Chien
University of Illinois at Urbana-Champaign

Illinois Fast Messages is a low-level software messaging layer designed to meet the demands of high-performance network hardware. Implementations on a Cray T3D and a Myrinet-based workstation cluster demonstrate that FM can deliver much of the underlying hardware's performance to both higher-level messaging layers and applications.

Prognostications of the National Information Infrastructure's future structure typically include high-performance servers of information, computation, and other specialized services embedded in a high-speed network fabric with hundreds of millions of other hosts. Two system architectures are likely candidates for these servers: massively parallel processors and networks of workstations. Both are attractive because of their ability to scale. However, both critically depend on internal communication performance to be effective servers, and any NII server depends on excellent external networking to support NII service. Interestingly, in recent years, MPP and NOW hardware have become increasingly similar, as both are driven by the significant cost advantages of high-volume products. Hence, many issues involved in delivering communication performance in such systems have converged as well.

The *Illinois Fast Messages* project intends to exploit this convergence to develop communications technology that spans both MPPs and NOWs, supporting both intracluster communication and high-speed external networking. FM is a portable, low-level messaging interface that can deliver high network data rates, even for small messages. It delivers the low-latency communication that is essential for efficient coordination and data movement on large-scale parallel systems. FM implementations not only deliver high performance but also provide the high-level guarantees that enable streamlined implementations of higher-level protocols atop FM. FM delivers performance to the higher-level layers, not just to applications written directly to the messaging layer.

We've implemented FM 1.1 on the Cray T3D MPP and on Myrinet-based workstation clusters. Both implementations perform substantially better than the vendor-supplied messaging layers. We've also implemented two widely accepted standard interfaces—Unix sockets and the Message

1063-6552/97/$10.00 © 1997 IEEE

Comparing Fast Messages and Active Messages

The Illinois Fast Messages 1.1 interface is closely modeled on that of Berkeley Active Messages.[1] AM, in turn, has as its intellectual antecedents hardware architectures that closely integrated communication and computation in message-driven,[2] dataflow,[3] and even systolic[4] architectures.

FM adopts features validated by AM's design and success. First, FM associates small "handlers" with messages, which incorporate incoming data into the ongoing computation. Second, these handlers are user-specified and can be used to implement protocols atop the primitive communication layer. Because these similarities give AM and FM a similar API, simple program examples are often nearly identical.

The crucial differences between FM and AM involve data reception, buffering, deadlock avoidance, and message ordering. AM systems include both explicit and implicit polling operations in each message send. This ensures that data is removed from the network, eliminating the possibility of internal network and store-and-forward deadlock. In contrast, FM provides explicit polling operations but includes no implicit poll in send operations. This subtle difference has two important implications. First, in FM programs, control through extract placement over when a message is processed allows control over data locality in the receiver's memory hierarchy. Second, a receiver cannot process network messages for a long time, so FM requires a modest amount of buffering and flow control to eliminate the possibility of deadlock. In contrast, AM systems can benefit from buffering but do not require it. However, the implicit poll involved in sends does reduce the program's control over the scheduling of message reception. Finally, FM guarantees that messages are delivered in order, while AM makes no such guarantee.

The implications of these differences are only now being investigated in research and prototype systems. For instance, the AM approach normally avoids buffering and might therefore more efficiently integrate direct transfer operations such as **`put/get`** into an implementation or interface. On the other hand, FM's decoupling of the processor and network through a buffered, flow-controlled stream of data might support efficient multitasking or scaling to networks with long latencies. We look forward to the exploration of these issues in future high-performance communication systems.

References

1. T. von Eicken et al., "Active Messages: A Mechanism for Integrated Communication and Computation," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, IEEE Computer Society Press, Los Alamitos, Calif., 1992, pp. 256–266; *http://www.cs.cornell.edu/Info/Projects/CAM/isca92.ps*.

2. W.J. Dally et al., "Architecture of a Message-Driven Processor," *Proc. 14th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, 1987, pp. 189–196.

3. J. Hicks et al., "Performance Studies of Id on the Monsoon Dataflow System," *J. Parallel and Distributed Computing*, Vol. 18, No. 3, July 1993, pp. 273–300; *ftp://csg-ftp.lcs.mit.edu/pub/papers/csgmemo/memo-345-3.ps.gz* or *http://www.csg.lcs.mit.edu:8001/monsoon/monsoon-performance/monsoon-performance.html*.

4. S. Borkar et al., "Supporting Systolic and Memory Communication in iWarp," *Proc. 17th Int'l Symp. Computer Architecture*, IEEE CS Press, 1990, pp. 70–81.

Passing Interface (MPI)[1]—atop FM to build higher-level protocols that deliver much of the underlying hardware's performance.

Illinois Fast Messages

The FM 1.1 interface is modeled after Berkeley's CM-5 Active Messages[2] (see the "Comparing Fast Messages and Active Messages" and the "Related work" sidebars). Primarily, FM borrows the notion of message handlers and uses essentially the same API. However, FM expands on Active Messages by imposing stronger guarantees on the underlying communication.

FM consists of three functions:

- **`FM_send(`*`dest,handler,buf,size`*`)`**,
- **`FM_send_4(`*`dest,handler,i0,i1,i2,i3`*`)`**, and
- **`FM_extract()`**.

`The FM_send()` and **`FM_send_4()`** calls inject messages into the network. **`FM_send()`** sends a long message; **`FM_send_4()`** sends a four-word message. (The latter call sends data from registers, thereby eliminating the need for memory traffic.) Each message has a corresponding handler function, indicated in its header, which is executed to process the message data.

The **`FM_extract()`** call processes received messages. It services the network, dequeueing pending messages and executing their corresponding handlers. Handlers move the message data (if necessary) from temporary FM buffers into user memory. The FM interface provides a simple buffer-management protocol: once an FM send function returns, FM guarantees that the buffer containing the message can be safely reused.

`FM_extract()` need not be called for the network to make progress. FM provides buffering so that senders can make progress while their corresponding receivers are computing and not servicing the network. This buffering allows the decoupling of the processor and network (which we'll discuss in the next section). Also, in contrast to Active Messages, where the send calls implicitly poll the network, FM's send calls do not normally process incoming messages. (**`FM_send()`** and **`FM_send_4()`** call **`FM_extract()`** only when necessary to avoid buffer deadlock.) This lets a program control when communications are processed.

Related work

Hardware architects have long pursued high performance for short messages by integrating the network interface with the CPU.[1,2] Innovations included interfacing the network to the instruction set and register file, message-field-based dispatch, and communication-driven scheduling. However, none of these innovations persisted into mainstream high-performance processors, because they required changes deep in the processor. Also, these innovations did not address the network and output contention that Pull FM (see "Pull FM on the T3D" in the main article) is designed to tolerate.

Less aggressive parallel architectures use commodity microprocessors and add external communication logic (for example, the Thinking Machines CM-5 and Cray T3D). These machines are quite similar to workstation clusters based on, for example, Myricom's Myrinet or DEC's Memory Channel.

Active Messages (see the "Comparing Fast Messages and Active Messages" sidebar) bridged the gap between the aggressive integration of communication into the processor and these hybrid systems, showing how commodity microprocessor-based systems could reap much of the communication performance benefit.[3] Active Messages crystallized a basic low-level communication model that is embodied in many communication layers, including FM.

A number of recent projects have explored novel low-level communication interfaces. U-Net provides buffer management but no flow control, implying that data can be lost because of rate mismatch.[4] The low-level interface supporting the Hamlyn architecture provides flow control but no buffer management, implying that higher-level messaging layers must provide their own buffer-management routines to prevent data from being overwritten.[5] FM provides a more complete solution to ensuring message delivery. By removing the responsibility for flow control and buffer management from higher-level messaging layers, the FM interface permits platform-specific implementations—hence, optimizations—of flow control and buffer management. The Real World Computing Partnership's PM, like FM, runs on Myrinet-connected workstations and performs both flow control and buffer management.[6] However, the PM implementation uses an optimistic flow-control mechanism and variable-sized packets. We are exploring opportunities for a detailed comparison.

References

1. S. Borkar et al., "Supporting Systolic and Memory Communication in iWarp," *Proc. 17th Int'l Symp. Computer Architecture*, IEEE Computer Society Press, Los Alamitos, Calif., 1990, pp. 70–81.
2. W.J. Dally et al., "Architecture of a Message-Driven Processor," *Proc. 14th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, 1987, pp. 189–196.
3. T. von Eicken et al., "Active Messages: A Mechanism for Integrated Communication and Computation," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, 1992, pp. 256–266; *http://www.cs.cornell.edu/Info/Projects/CAM/isca92.ps*.
4. T. von Eicken et al., "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," *Proc. 15th ACM Symp. Operating Systems Principles*, ACM Press, New York, 1995, pp. 40–53; *http://www2.cs.cornell.edu/U-Net/papers/sosp.pdf*.
5. G. Buzzard et al., "A High-Performance Network Interface with Sender-Based Memory Management," *Proc. IEEE Hot Interconnects Symp.*, 1995; *http://www.hpl.hp.com/personal/John_Wilkes/papers/HamlynHotIntIII.pdf*.
6. H. Tezuka, A. Hori, and Y. Ishikawa, "PM: A High-Performance Communication Library for Multi-user Parallel Environments," Tech. Report TR-96-015, Real World Computing Partnership, Tsukuba Research Center, Tsukuba, Japan, 1996; *http://www.rwcp.or.jp/papers/1996/mpsoft/tr96015.ps.gz*.

Guarantees

The most important facet of messaging layer design is the service guarantees that the layer provides to higher-level messaging layers and applications. If these guarantees are too weak (that is, they do not provide the functionality that applications expect), other messaging layers built on top will need to supply the missing functionality, thereby incurring additional overhead. On the other hand, if the guarantees are too strong (that is, they provide more functionality than is generally needed), the messaging layer's common-case performance might be needlessly degraded. Analysis of the literature and our ongoing studies to support fine-grained parallel computing[3–5] have led us to conclude that a low-level messaging layer should provide these guarantees:

- Reliable delivery,
- In-order delivery, and
- Control over scheduling of communication work (decoupling of the processor and network).

We designed the FM interface to provide these guarantees. Although FM is certainly not the only approach to delivering high-performance communication,[2,6] these guarantees make it distinctly useful.

Reliability

Previous studies of communication cost in the CM-5 indicate that software overhead for reliability, fault tolerance, and ordering can increase communication cost by over 200%.[3] Yet many low-level communication systems simplify their implementation by discarding packets. When networks were unreliable, this practice made sense. However, modern networks are highly reliable, so such discarding is the major source of data—and therefore, performance—loss.

FM ensures delivery in all cases except uncorrectable hardware communication errors. Such reliable delivery lets protocols eliminate retransmission techniques when dealing with lost packets.

While many high-speed networks have extremely low channel-error rates, reliable delivery also requires buffer management and flow control. FM not only provides these, but its performance demonstrates that these guarantees need not be costly. The small cost these guarantees do incur is offset by the additional performance gains they offer to higher protocol levels, in the form of more simplified, streamlined control.

In-order delivery

In-order delivery lifts the burden of storing and reordering messages off higher protocols. Because FM provides such delivery, we can simplify higher-level protocols by eliminating reordering code.

Decoupling of the processor and the network

Experience with messaging layers in multicomputers, shared-memory systems, and high-speed wide-area networks indicates that cache interference is critical to both communication and local computational performance. Providing control over the scheduling of communication allows programs to control their cache performance, in many cases enabling more efficient computation and communication. However, allowing such scheduling control requires decoupling the processor from the network, so that senders are not blocked because they are waiting for receivers to extract messages from the network.

FM decouples the processor and network through buffering. This allows communication to be truly one-sided. That is, a computation can control when it processes communications, and the sender can still make some progress even if the receiver defers servicing the network.

EXTENSIBILITY

FM also allows easy extension of functionality. For example, some applications might require absolute data integrity. The use of memory buffers at the FM interface allows easy addition of buffer-redundancy schemes. Likewise, applications requiring data security can easily layer encryption atop FM by incorporating a routine that encrypts data in place when they are passed a pointer to a memory buffer. Also, features can be layered above FM with a minimal performance penalty.

PORTABILITY

Often, a tradeoff exists between portability and performance. For FM, however, we identified the key similarities between MPPs and NOWs (present and future) and included in the interface only those features that are generally useful and that can be implemented efficiently on both types of systems. That is, we made the FM interface as portable as possible, up to the point where performance would have been sacrificed. This portability is demonstrated by our multiple implementations on the T3D and workstation cluster.

Cray T3D FM implementation

We first implemented FM on the Cray T3D. Our implementations exploit the T3D's high-speed interconnect and feature-rich network interface to achieve excellent communication performance. The T3D supports up to 4,096 Alpha processors and is a shared-address-space machine with no hardware support for cache coherence. However, because memory access is nonuniform both in mechanism (special addressing mechanisms for remote locations) and performance (four to five times slower than accesses to remote memory), a message-passing programming style can benefit performance as well as portability and relative ease of programming.

Figure 1 diagrams one node of the Cray T3D. The main components of a node are the two processing elements (for clarity, the figure shows only one), the block transfer engine (BLT), and the network interface. Each processing element contains an Alpha microprocessor, local memory, and some communication-specific logic. The parts of the node that are the most relevant to FM are the

- *data-translation buffer (DTB) annex*, which is needed to set up a logical connection with other nodes. The annex supports remote reads and writes as well as atomic fetch & increment and atomic swap operations that can be accessed by remote processors.
- *fetch & increment unit*, which is used for push messaging.
- *atomic swap unit* and *prefetch queue*, which are used for pull messaging.

(We'll describe push and pull messaging in the next two sections.)

Because the network already guarantees reliable, in-order delivery, the most important aspects of implementing the FM interface on the T3D are decoupling communication and computation and achieving high performance. The multiplicity of mechanisms in the

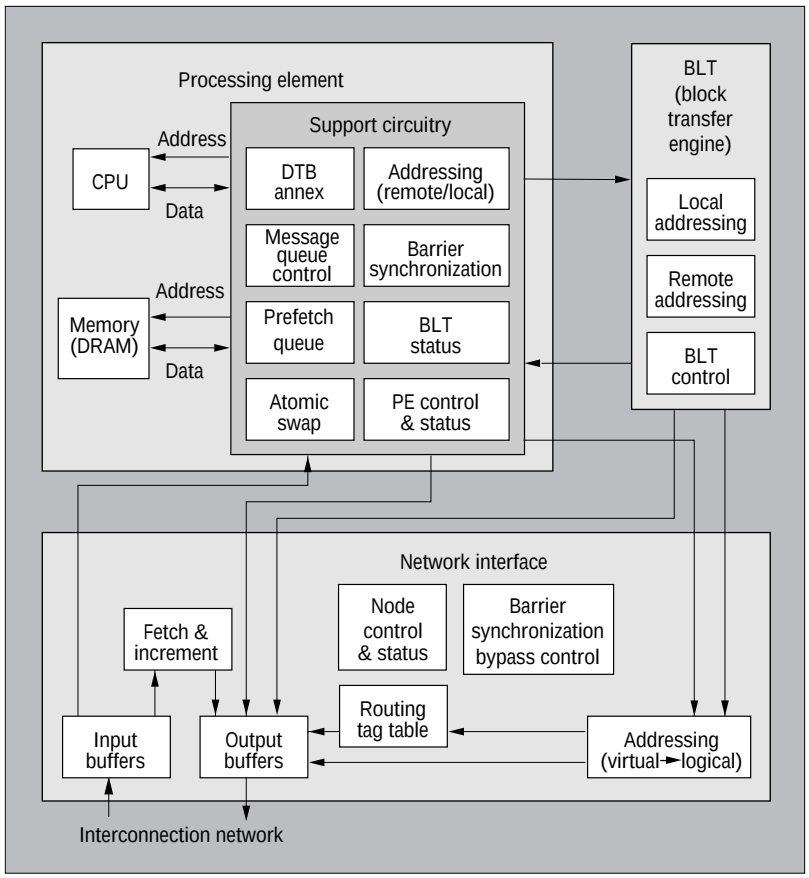

Figure 1. A T3D node.

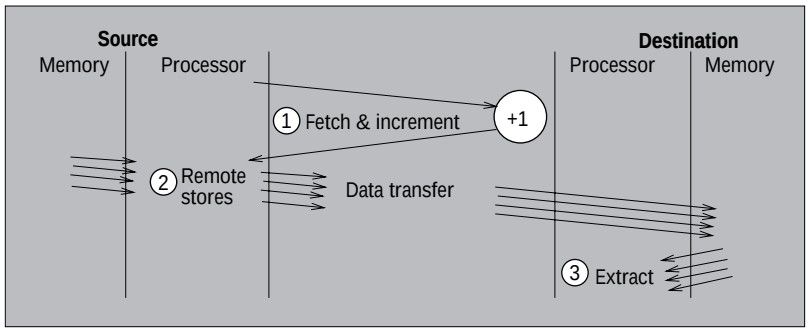

Figure 2. Push messaging.

network interface and support circuitry enables distinct implementations of the FM interface, each optimized for different communication and performance patterns. We have constructed two such implementations: *Push FM* and *Pull FM*.

PUSH FM

Push FM is a traditional style of messaging implementation that pushes data from the source to the destination, and buffers it for the receiving processor. With push messaging, FM "eagerly" propagates messages (that is, right when `FM_send()` is called) from the sender to the receiving node's memory.

As Figure 2 illustrates, the sender uses the fetch & increment operation against the receiver's fetch & increment register to acquire a unique index into a preallocated message buffer at the receiving node (Step 1). With this index, the sender moves the message data into that buffer, using remote write operations (Step 2). Because the fetch & increment operation atomically handles the buffer allocation among competing senders, no other node will try to use the buffer simultaneously. When the receiver wishes to process the message, it simply reads the message from the buffer in its local memory (Step 3).

Although push messaging minimizes latency at low network loads, performance can degrade if output contention exists or if the receiver allows its incoming buffers to fill by not servicing the network often enough.[4] If messages arrive at a receiver faster than the receiver's memory can process them or than the receiver extracts them, the writes back up into the network, adding to network contention and increasing average latency. This effect, which occurs in many irregular parallel computations, provides the impetus for Pull FM.

PULL FM

To avoid output contention, Pull FM moves data "lazily"—only when the receiver is ready to process it. Pull messaging is enabled by the T3D's shared address space, which allows receivers to reach into the sender's memory and pull the message data into local memory. Pull messaging effectively eliminates output contention and is therefore important for irregular computations.

As Figure 3 illustrates, Pull FM first copies the message data into a local memory buffer (Step 1) and then uses atomic swap to link the message into the receiver's receive queue (Steps 2 and 3). This queue is a distributed linked list whose head and tail are stored at the receiver. Data remains in the sender's local memory. When the receiver wants to read a message, it pulls the message into its local memory using remote-memory read oper-

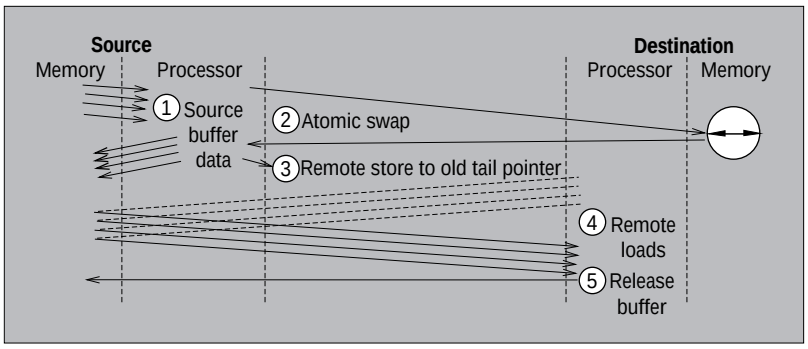

Figure 3. Pull messaging.

ations (Step 4). Push FM uses the T3D's prefetch hardware to mask the latency of these remote reads. Finally, the receiver deallocates the message buffer by storing a "reclaim" flag into the buffer (Step 5).

Because each receiver pulls data across at its own pace, nodes are never swamped with data. This effectively eliminates output contention, because the data arrival rate is matched to the receiver's pulling rate. Furthermore, enqueuing messages is a comparatively low-overhead operation and is therefore beneficial for fine-grained applications. Pull messaging also serendipitously eliminates message-buffer overflow. Responsibility for buffer allocation is transferred back to the senders, where flow control is easily achieved. One drawback of Pull FM is that pulling data across the network requires extra work, increasing the overhead and latency slightly, compared to Push FM.[4]

Performance

We'll now compare T3D FM's performance to two other vendor-supplied communication layers on the T3D: Shmem and PVM. Shmem is a low-level data-movement (not messaging) library that copies data between addresses in the shared address space. It is little more than a set of functions for directly accessing the raw hardware, and therefore is extremely efficient. The Shmem layer provides two main mechanisms: **`put`** and **`get`**. PVM is a widely used, full-featured messaging layer that does both the end-to-end synchronization and the buffer management required for traditional messaging. The PVM implementation was optimized by Cray for the T3D.

Latency

Figure 4a compares FM's one-way latency to that of PVM and Shmem (with **`put`** and **`get`** semantics). Push FM's latency curve is essentially level until around 256 bytes, after which it increases linearly. This is because the fixed overheads—especially the fetch & increment—dominate the latency until that point; afterwards, the per-byte costs dominate. Push FM's latency is 6.1 μs for an 8-byte message and 366.8 μs for a 32-Kbyte message. These times are close to the best achievable on the T3D hardware. Pull FM's latency is comparable to Push FM's for small messages. However, for larger messages, Push FM is quite a bit slower because of the limited performance of the prefetch hardware. The jump in Pull FM's latency from 64 to 128 bytes stems from FM's use of 112-byte packet buffers. Messages larger than 112 bytes require segmentation and reassembly, adding overhead and reducing bandwidth. Still, for the types of messages Pull FM was designed to handle (that is, the small messages used by fine-grained programs), its 8.7-μs,

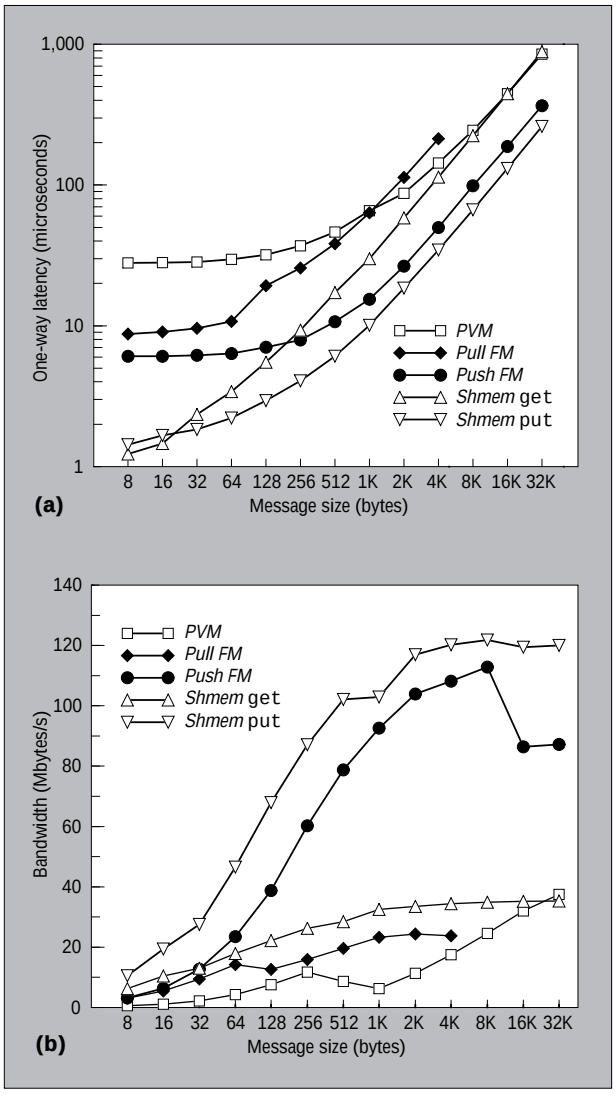

Figure 4. FM performance on the T3D: (a) latency; (b) bandwidth.

8-byte latency is good, even if its 213.2-μs, 4-Kbyte latency (messages in Pull FM are limited to 4 Kbytes) is a bit high.

PVM's latency parallels Push FM's. However, PVM's latency is much higher than Push FM's for large messages and than both FMs for small messages. Its latency ranges from 28.0 μs for 8-byte messages to 850.2 μs for 32-Kbyte messages. So, even though FM also provides a messaging interface that supplies buffer management and flow control, its no-frills operation delivers performance superior to PVM's.

Because Shmem **put** performs neither flow control, buffer management, nor notification of message arrival, its latency is fairly good: 1.4 μs for an 8-byte message, increasing almost linearly to only 260.9 μs for a 32-Kbyte message. However, Push FM's latency approaches Shmem **put**'s as the message size increases. Because Push FM and Shmem **put** both rely on remote stores for data movement, this behavior is expected.

Shmem **get**, however, uses a different mechanism—remote loads. Because remote stores are nonblocking but remote loads block, Push FM's latency exceeds Shmem **get**'s for messages larger than 128 bytes, after which Push FM's fixed overhead is sufficiently amortized. Still, Shmem **get** can retrieve an 8-byte message in only 1.2 μs, although its 32-Kbyte latency, 885.9 μs, exceeds even PVM's. Figure 4a shows, therefore, that even though FM performs flow control and buffer management as PVM does, its latency is comparable to that of the lower-level Shmem layer.

Bandwidth

Figure 4b demonstrates FM's bandwidth over a range of message sizes. In general, the performance follows the same ordering as in Figure 4a. However, there are fewer crossovers than in the latency graph. Push FM's bandwidth increases smoothly until the message size exceeds 8 Kbytes, where it drops slightly. Cache effects cause that drop—the Alpha processor has only 8 Kbytes of data cache, and the T3D has no additional off-chip cache. Even so, Push FM's 112.9 Mbytes/s for 8-Kbyte messages is quite good and reflects the limitation of the Alpha's write buffer, which peaks at approximately 130 Mbytes/s.

Pull FM's bandwidth is comparable to Push FM's bandwidth for small messages, but is lower for large messages. The lower large-message bandwidth is due primarily to the prefetch queue's peak bandwidth, approximately 32 Mbytes/s. As in the latency curve, Pull FM's bandwidth decreases from 64 to 128 bytes because of the segmentation and reassembly used for messages larger than 112 bytes. Pull FM's bandwidth ranges from 3.1 Mbytes/s for 8-byte messages to 23.8 Mbytes/s for 4-Kbyte messages.

Both versions of FM have much better bandwidth than PVM for small messages. For 8-byte messages, PVM's bandwidth—0.6 Mbytes/s—is one-fifth of FM's. For larger messages, Push FM's bandwidth is still over twice PVM's. However, because PVM's bandwidth is not limited by the prefetch queue, its bandwidth eventually surpasses Pull FM's. PVM's bandwidth is 37.5 Mbytes/s for 32-Kbyte messages.

As expected, Shmem **put**'s bandwidth exceeds FM's. However, for messages between 1 and 8 Kbytes, Push FM's bandwidth is almost identical to Shmem **put**'s. Unlike Push FM, however, Shmem **put** does not suffer from cache effects for messages larger than 8 Kbytes. Therefore, its bandwidth does not drop after reaching its peak of 120.0 Mbytes/s for 32-Kbyte messages. Because Shmem **put** does no buffer management, its fixed overhead is reduced enough to give it an 8-byte message bandwidth of 10.5 Mbytes/s. Shmem **get**'s bandwidth is less than Shmem **put**'s, and even less than Push FM's for messages larger than 32 bytes. However, it performs better than Pull FM. While Shmem **get**'s 8-byte bandwidth is a reasonably high 6.2 Mbytes/s, remote loads limit its bandwidth to 35.3 Mbytes/s for 32-Kbyte messages.

Two useful implementations

Figure 4 demonstrates that while FM supplies PVM-like delivery guarantees, it does so at a cost comparable to the raw data-movement cost (Shmem **put**). Although Pull FM might appear unnecessary—Push FM exhibits superior latency and bandwidth—it performs much better than Push FM for heavy network traffic, especially when communication patterns are irregular.[4] The Concert runtime,[5] for instance, uses Pull FM exclusively, because the communication irregularity inherent to Concert programs makes communication robustness dominate overall performance more than baseline latency or bandwidth does.

These two FM implementations demonstrate the advantages of a well-defined communication interface. The FM interface's design enabled two distinct implementations which, although suitable for different application and network behaviors, can be used interchangeably. Because the implementations have significantly different performance characteristics, application programs benefit from selective use of the two libraries—

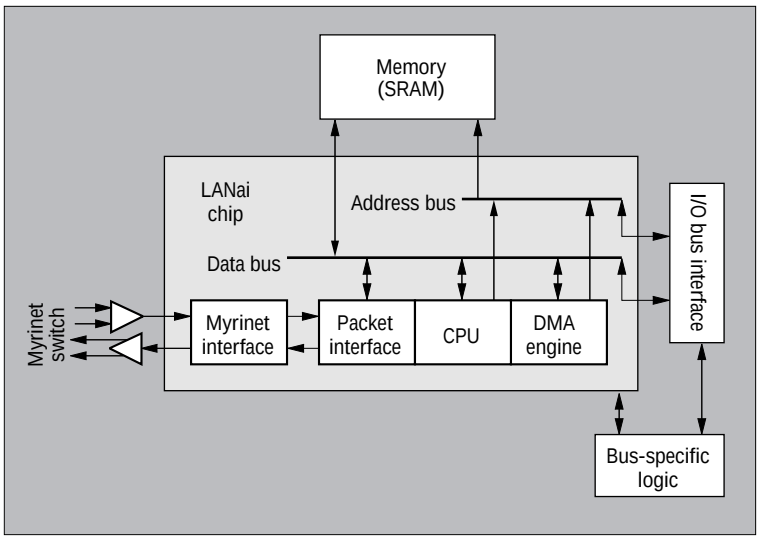

Figure 5. The Myrinet interface.

even in the same program. FM delivers communication performance close to the maximum achievable by the underlying hardware, demonstrating that its interface need not incur significant overhead.

Myrinet FM implementation

Our workstation cluster implementation of FM uses Myricom's 640-million (10^6, as opposed to megabits—2^{20}) bits-per-second switched network[7] and a collection of Sun Sparcstation workstations. The network exhibits per-hop latencies of approximately 0.5 μs. Packets are wormhole-routed, so if an output port is busy, the packet is blocked in place. Packets blocked for greater than 50 ms are dropped.

In place of the DTB annex on the T3D nodes' memory bus, the Myrinet card has the *LANai*, a programmable CPU that has 128 Kbytes of SRAM and attaches to the Sparcstation's I/O bus (SBus) (see Figure 5). (Our Myrinet cards contain LANai version 3.2, a prototype of the LANai 4.0 processor, but with less memory bandwidth and more limited clock speeds—less than 25 MHz.) The LANai contains three DMA (*d*irect *m*emory *a*ccess) engines—one that transfers between the LANai and host memory (shown in the figure) and two that transfer between the network and the LANai memory (not shown, but included in the Myrinet inteface). The host and the LANai communicate through the sharing of LANai memory (mapped into the host's address space) or host memory (via DMA to or from the LANai).

While the FM interface for Myrinet is identical to that on the T3D, the differences in hardware structure dictate a substantially different implementation.

FULFILLING THE GUARANTEES

Of course, the three guarantees (reliable delivery, in-order delivery, and decoupling of the network and processor) also apply to FM implementations for workstation clusters. Decoupling is particularly important in workstation clusters, which often operate without coordinated process scheduling. Tens or even hundreds of milliseconds might pass before a receiver process is scheduled. These challenges are exacerbated by several additional hardware constraints:

- The LANai processor is approximately 20 times slower than the host processor;
- The LANai has insufficient memory to effectively buffer messages—only 1.5 milliseconds' worth at network speed; and
- DMA to or from host memory requires that the pages be pinned (that is, marked unswappable).

Myrinet FM's design ensures reliable, in-order delivery with end-to-end window-based flow control and FIFO queueing at all levels. FM uses three queues: a send queue in the LANai, a receive queue in the LANai (used as a staging area), and a larger receive queue in the host's memory (in a pinned-down region).

Message transmission consists of four steps (see Figure 6). In Step 1, the host processor uses double word stores to move the message data directly into the send queue. Processor I/O eliminates the cost of copying the data into a pinned DMA-able buffer accessible by the network interface. (In our SBus-based Sparcstation 20's, processor I/O cannot use burst transaction mode. Hence, this solution improves latency at the expense of bandwidth. We expect this penalty will be eliminated in future

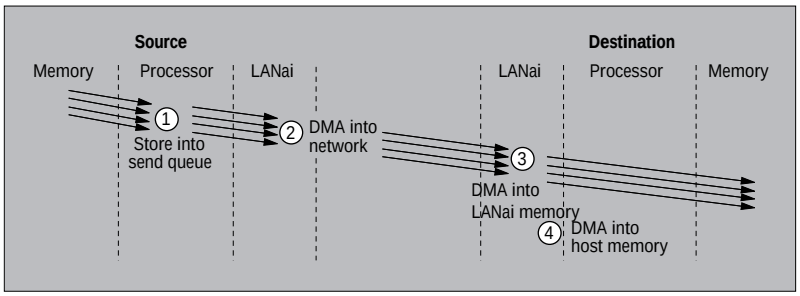

Figure 6. Messaging on the Myrinet.

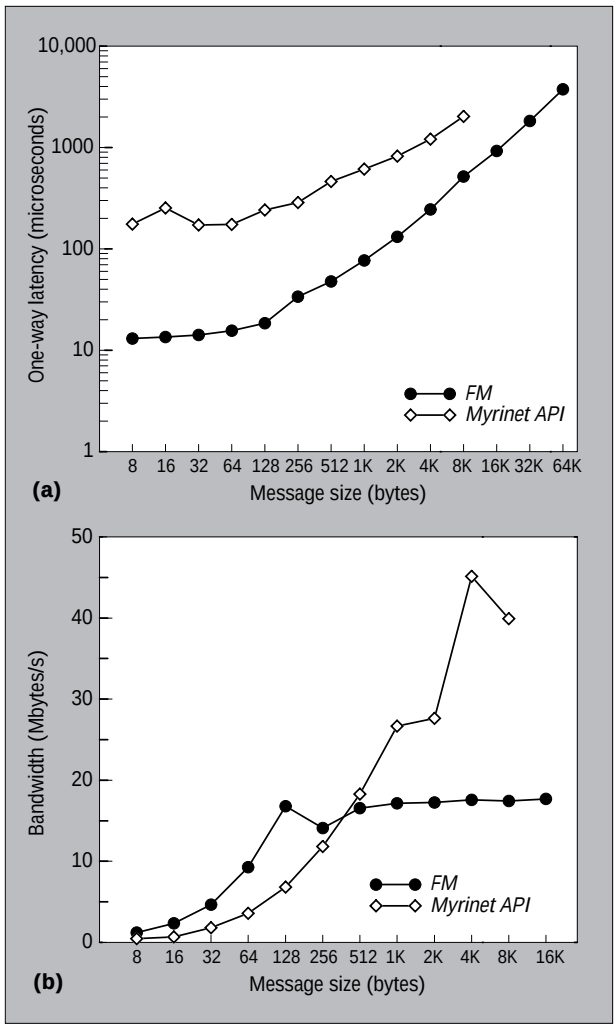

Figure 7. Myrinet FM versus the Myrinet API: (a) latency; (b) bandwidth.

machine designs—particularly those based on PCI—thereby doubling or tripling FM's peak bandwidth.)

The host LANai detects the outgoing packet and sends it via DMA into the network (Step 2). The remote LANai then detects the incoming packet and receives it into its memory (Step 3). Finally, the remote LANai sends the message data buffered in its memory to the host, where the receiving process can directly access it with high efficiency (Step 4). This last step ensures good bandwidth, quick delivery of the packet, and prompt draining of the network, even when the host process is busy or not available (that is, if the FM host program is descheduled).

Myrinet FM's end-to-end window flow-control scheme ensures that no packets are lost because of buffer overflow. Each sender is allocated a fraction of a receiving node's host memory queue size (credits), and the sum of the allocated fractions cannot exceed the total storage in the queue. So, even if a process neglects to remove data from the network, eventually all of the senders will stop, having used up their credits (that is, filled their allocated buffers), and no message data will be lost. Furthermore, because FM decouples the host and the network, host unresponsiveness will not prevent messages from being drained from the network. Thus, the Myrinet 50-ms time-out never occurs, and packets are never dropped. Whenever a receiver does remove a message from the queue, however, it sends a credit to the sender to recycle the buffers. (We aggregate credit messages and—when possible—piggyback credit on ordinary messages for greater efficiency.) Because the network and all the queues in the sender and receiver are FIFO, message ordering is preserved.

Performance

We'll now compare FM to Myricom's Myrinet API, a similar low-level messaging layer. Like FM, the Myrinet API is a combination of LANai and host code. The API's structure is similar to that of traditional networking interfaces for Ethernet.

The Myrinet API has three significant differences from the FM API. First, in the Myrinet API, the host and the LANai communicate through command FIFOs (for example, the host tells the LANai it has a packet to send by writing a send command into a queue). Using command queues requires additional synchronization between the host and the LANai, increasing the overhead for communication. Second, the Myrinet API uses DMA on the send side. This requires an extra memory copy (to move the data from the user's data structures into the pinned memory region) and some host-LANai synchronization, which increases overhead. FM uses programmed I/O on the send side to eliminate this overhead. Third, the Myrinet API does not guarantee reliable, in-order delivery. The user must allocate receive buffers in a timely manner (coupling the processor and network), or message data might be lost. These differences produce significantly worse latency for the Myrinet API, as we'll show next.

Although FM provides much stronger guarantees than the Myrinet API, it still delivers superior performance. Figure 7 illustrates the one-way latency and bandwidth of Myrinet FM running on a Myrinet network containing a pair of Sparcstation 20s with 75-MHz SuperSparc-II processors and 1 Mbyte of level 2 cache.

Myrinet FM 1.1 uses 128-byte fixed-size packet frames. Messages greater than 128 bytes require segmentation and reassembly, which produces some slight performance anomalies in the transitional region around that message size. Hence, the latency curve is flat up to 128 bytes and then increases linearly with message size. FM's bandwidth

peaks at 17.5 Mbytes/s, limited by the I/O bus bridge implementations. This limit disappears with I/O bus bridges that support write aggregation (that is, exploit bus burst mode for programmed I/O), such as Pentium Pro systems with PCI buses. As in the T3D version, Myrinet FM's performance approaches the maximum possible by the hardware (in the Sparcstation's case, approximately 23 Mbytes/s for programmed I/O bandwidth over the bus bridge).

The Myrinet FM implementation demonstrates that FM ports well to rather different types of hardware. Because the underlying hardware does not provide all the needed guarantees, our implementation includes some LANai firmware and a host-processor library that together provide the requisite service guarantees. The cost for supporting these guarantees is not excessive, as the high absolute performance of Myrinet FM demonstrates. Also, the resulting performance is significantly better than that of the traditional organization presented by the Myrinet API, especially for small to medium messages.

Delivering performance to higher layers

Many high-speed messaging layers achieve good performance by relegating costly operations to higher-level layers. Although this makes the low-level layer appear fast, it generally degrades actual, application-level performance. This is because higher-level layers must add the missing functionality, often at a higher price than the lower-level layer would have paid.

A successful low-level messaging layer must enable the construction of efficient higher-level messaging software. It must also be able to deliver a significant amount of its performance to that software and, ultimately, to applications.

To evaluate FM from this perspective, we built two higher-level messaging interfaces atop it: MPI[1] and Unix sockets. MPI is a standard of increasing importance for both parallel and distributed message-passing programs. Sockets are a long-standing standard for building multiprocess and, particularly, client-server applications. Our implementation of MPI is a port of the mpich[8] version. However, we significantly restructured the code to reduce both buffer copies and control-path overhead.[9] We ran this version of FM on the same system we used to test Myrinet FM.

Figure 8 indicates that FM makes the high performance of the underlying network hardware accessible to higher-level messaging layers. MPI imposes additional processing overhead (approximately 5 μs, which is modest compared to many other implementations), which reduces performance for short messages. However, we have worked hard to minimize this effect, and messages of modest size—as small as 256 bytes—achieve near-peak network performance.[9]

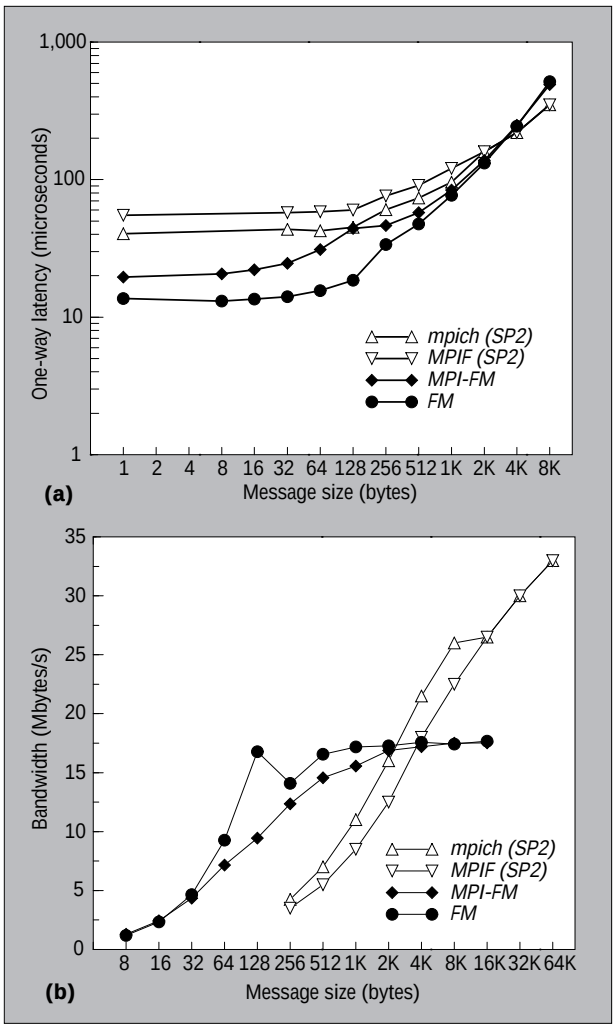

Figure 8. MPI-FM performance: (a) latency; (b) bandwidth.

For comparison, Figure 8 includes performance data for the IBM SP2's two versions of MPI. Mpich (SP2) is a port of mpich to the SP2, and MPIF (SP2) is an MPI implementation written and optimized specifically for the SP2. MPI-FM exhibits latency and bandwidth superior to both SP2 MPIs for messages smaller than 4 Kbytes. At that point, MPI-FM's performance levels off because of the limited programmed I/O bandwidth over the SBus bridge.

To implement the Unix sockets interface, we built a user-level library that is linked to an application program and that makes calls to the underlying FM implementation. FM Sockets supports a socket interface to both byte-stream and datagram transport services (such as TCP and UDP). Preliminary performance numbers

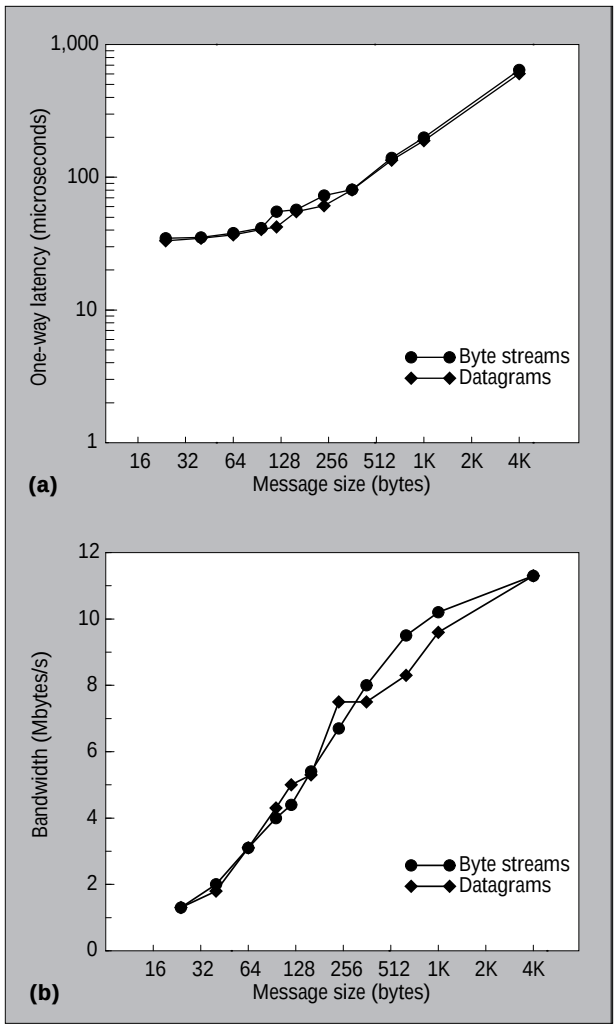

Figure 9. FM Sockets performance: (a) latency; (b) bandwidth.

for FM Sockets (see Figure 9) indicate a much larger control overhead than the MPI-FM implementation. The major reason is the rather complex means by which sockets implementations exchange data with applications and the network. Because sockets implement byte streams without clear message boundaries, both applications and the socket layer itself cannot easily optimize buffer allocation and data movement. (Some sockets implement datagrams, but byte streams are the general case.) The latency numbers reflect the additional control overhead from dealing with irregular and unpredictable amounts of data.

Also, FM Sockets does not achieve raw FM's bandwidth of 17.5 Mbytes/s, because FM's receive queue needs to be marked uncacheable to ensure consistency when using hyperSparc CPUs (as were used in Figure 9). Hence, the cost of segmentation and reassembly greatly increases.

Given those limitations, FM Sockets achieves good performance, partly because FM already provides reliable, in-order delivery, thereby obviating the need to implement it in the socket layer. Because FM provides the important guarantees, higher-level messaging layers can achieve performance approaching FM's.

We compared FM Sockets to Myricom's TCP and UDP implementations. (The Myricom results are based on third-party measurements.[10]) As Table 1 shows, FM Sockets outperforms the Myricom implementation. In particular, FM Socket's latency is an order of magnitude better than that of Myricom's TCP and UDP. However, this is not really a fair comparison. Myricom implemented TCP and UDP as standard, kernel-level drivers, while FM Sockets runs entirely at the user level. Still, it illustrates the benefits of implementing sockets on top of FM.

FM sacrifices a tiny amount of its raw performance for a large increase to the system's overall performance.[11] By providing generally useful service guarantees such as reliable, in-order delivery, FM saves messaging layers and applications built on top of it from the burden and performance penalty of adding those features. Hence, not only is FM capable of low-latency, high-bandwidth communication, but so are the messaging layers built on top of it.

However, while we have achieved good performance, the design of the MPI and Unix sockets interfaces has given us insight into ways to improve the FM interface, which we've incorporated into FM 2.0 (see the "FM 2.0" sidebar).

Although FM's original goal was to support more efficient parallel computation, such high-speed communication systems can have a much broader impact on distributed computing. Such a system's additional requirements include

- support for multiple users and processes per node,
- coordinated scheduling,
- integrating separately managed computing domains, and
- interoperability with traditional networks and networking protocol structures.

Distributed systems typically run multitasking operating systems on each node. This implies that if multiple, independent processes on the same node want to use the network concurrently, the communication sub-

system must prevent inadvertent interactions. Messaging layers such as FM should therefore provide protection between processes as well as a way to demultiplex messages to the appropriate process on a destination node. The challenge in doing so efficiently involves a complex tradeoff between dedicating resources and inline interpretation overhead.

Because each node runs its operating system independently from other nodes, communicating processes are often not scheduled simultaneously. This lack of coscheduling can drastically increase response time—often into the tens-of-milliseconds range—and thereby severely reduce performance. Gang scheduling is inappropriate in the context of multiuser, multitasking operating systems because it requires scheduling decisions to be globally agreed upon. This is costly and scales poorly. A better approach is to integrate communication and scheduling. That is, FM and the operating system should work together to coschedule communicating processes by exploiting local process information and information about message arrival and launch. We have already implemented a version of FM that does exactly that, but our implementation is limited to a single communicating thread per node. Ongoing work involves generalizing the algorithm and implementation to support an arbitrary number of threads and processes per node.

Among the challenges in converting from a strictly parallel to an integrated parallel and distributed model is support for large-scale computing—computing on systems with more nodes than fit in a single administrative domain. For such systems, communication layers support extremely large numbers of nodes. Again, this entails challenges in resource management. For example, we might need to replace FM's static window-based flow-control scheme with something more dynamic.

Finally, much networking code exists and low-speed networks still predominate. To incrementally integrate high-speed communication systems into existing networks running legacy software, these high-speed and low-speed systems must interoperate. That is, messaging layers such as FM must be able to interact with established networking protocol structures without sacrificing an inordinate amount of performance on high-speed networks.

Table 1. FM Sockets performance compared to Myricom TCP and UDP. (Myricom results are based on third-party measurements.[10])

Messaging layer	8-byte, one-way latency (μs)	Peak bandwidth (Mbytes/s)
FM Sockets—byte streams	34.8	11.3
Myricom TCP	751.0	9.4
FM Sockets—datagrams	33.3	11.3
Myricom UDP	721.5	9.8

Acknowledgments

The research described in this article was supported in part by DARPA Order #E313 through US Air Force Rome Laboratory Contract F30602-96-1-0286, NSF Grant MIP-92-23732, and NASA Grant NAG 1-163. We also gratefully acknowledge support from the Intel Corp., Tandem Computers, Hewlett-Packard, Microsoft, and Motorola. Andrew Chien is supported in part by NSF Young Investigator Award CCR-94-57809.

References

1. Message Passing Interface Forum, *MPI: A Message Passing Interface Standard*, Tech. Report Version 1.1, Univ. of Tennessee, Knoxville, Tenn., 1995; *ftp://ftp.mcs.anl.gov/pub/mpi/mpi-1.jun95/mpi-report.ps.Z.*

2. T. von Eicken et al., "Active Messages: A Mechanism for Integrated Communication and Computation," *Proc. 19th Ann. Int'l Symp. Computer Architecture*, IEEE Computer Society Press, Los Alamitos, Calif., 1992, pp. 256–266; *http://www.cs.cornell.edu/Info/Projects/CAM/isca92.ps*.

3. V. Karamcheti and A.A. Chien, "Software Overhead in Messaging Layers: Where Does the Time Go?" *Proc. Sixth Symp. Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, ACM Press, New York, 1994, pp. 51–60; *http://www-csag.cs.uiuc.edu/papers/asplos94.ps*.

4. V. Karamcheti and A.A. Chien, "A Comparison of Architectural Support for Messaging on the TMC CM-5 and the Cray T3D," *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, IEEE CS Press, 1995, pp. 298–307; *http://www-csag.cs.uiuc.edu/papers/cm5-t3d-messaging.ps.*

5. V. Karamcheti, J. Plevyak, and A.A. Chien. "Runtime Mechanisms for Efficient Dynamic Multithreading," *J. Parallel and Distributed Computing*, Vol. 37, No. 1, 1996, pp. 21–40; *http://www-csag.cs.uiuc.edu/papers/rtperf.ps*.

6. H. Tezuka, A. Hori, and Y. Ishikawa, "PM: A High-Performance Communication Library for Multi-user Parallel Environments," Tech. Report TR-96-015, Real World Computing Partnership, Tsukuba Research Center, Tsukuba, Japan, 1996; *http://www.rwcp.or.jp/papers/1996/mpsoft/tr96015.ps.gz*.

7. N.J. Boden et al., "Myrinet: A Gigabit-per-Second Local-Area Network," *IEEE Micro*, Vol. 15, No. 1, Feb. 1995, pp. 29–36; *http://www.myri.com/research/publications/Hot.ps*.

8. W. Gropp and E. Lusk, *User's Guide for mpich, a Portable Implementation of MPI*, Tech. Report ANL/MCS-TM-ANL-96/6, Mathematics and Computer Science Div., Argonne Nat'l Laboratory, Argonne, Ill., 1996; *http://www.mcs.anl.gov/mpi/mpiuserguide/paper.html* and *ftp://info.mcs.anl.gov/pub/mpi/userguide.ps.Z*.

9. M. Lauria and A.A. Chien, "MPI-FM: High Performance MPI on Workstation Clusters," to appear in *J. Parallel and Distributed Computing*, Feb. 1997; *http://www-csag.cs.uiuc.edu/papers/jpdc97-normal.ps*.

For further reference

FM software distributions and the latest information about the FM project are available from *http://www-csag.cs.uiuc.edu/projects/comm/fm.html*.

10. K.K. Keeton, T.E. Anderson, and D.A. Patterson, "LogP Quantified: The Case for Low-Overhead Local Area Networks," *Hot Interconnects III: A Symp. High Performance Interconnects*, 1995; *http://now.cs.berkeley.edu/Papers/Papers/hotinter95-tcp.ps*.

11. S. Pakin, M. Lauria, and A.A. Chien, "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet," *Proc. Supercomputing '95* (on CD-ROM), IEEE CS Press, 1995; *http://www-csag.cs.uiuc.edu/papers/myrinet-fm-sc95.ps*.

Scott Pakin is in the doctoral program in computer science at the University of Illinois at Urbana-Champaign. He is also a member of Andrew Chien's Concurrent Systems Architecture Group. His research interests include high-speed communication for workstation clusters and coordinated scheduling. He received his MS in computer science from the University of Illinois at Urbana-Champaign in 1995 and his BS in mathematics/computer science from Carnegie Mellon University in 1992. He is a student member of the IEEE. He can be contacted at 1304 W. Springfield Ave., Urbana, IL 61801; pakin@cs.uiuc.edu.

Vijay Karamcheti is completing his PhD in the Electrical and Computer Engineering Department at the University of Illinois at Urbana-Champaign. He is also a member of Andrew Chien's Concurrent Systems Architecture Group. His research interests include high-performance communication for parallel machines and runtime system design for supporting irregular parallelism. He received his MS in electrical and computer engineering from the University of Texas, Austin, in 1990 and his B.Tech. in electrical engineering from the Indian Institute of Technology, Kanpur, in 1988. He can be contacted at 1304 W. Springfield Ave., Urbana, IL 61801; vijayk@cs.uiuc.edu.

Andrew A. Chien is an associate professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign, where he holds a joint appointment as an associate professor in the Department of Electrical and Computer Engineering and as a senior research scientist with the National Center for Supercomputing Applications. He also leads the Concurrent Systems Architecture Group. The primary goals of his research involve the interaction of programming languages, compilers, system software, and machine architecture in high-performance parallel systems. He received his BS in electrical engineering in 1984 and his MS and PhD in computer science in 1987 and 1990, all from the Massachusetts Institute of Technology. He received an NSF Young Investigator Award in 1994, the C.W. Gear Outstanding Junior Faculty Award in 1995, and the Xerox Senior Faculty Award for Outstanding Research in 1996. He can be contacted at 1304 W. Springfield Ave., Urbana, IL 61801; achien@cs.uiuc.edu.

FM 2.0

The FM 2.0 interface consists of these functions:

- **`FM_begin_message(`*`dest,size,handler`*`)`** opens a streamed message.
- **`FM_send_piece(`*`stream,buf,size`*`)`** adds data to a streamed message.
- **`FM_end_message(`*`stream`*`)`** closes a streamed message.
- **`FM_extract(`*`maxbytes`*`)`** processes up to *maxbytes* of received messages.
- **`FM_receive(`*`buf,stream,size`*`)`** receives data from a streamed message into a buffer.

It incorporates improvements based on our experience with FM Sockets and MPI-FM (see "Delivering performance to higher layers" in the main article). Although FM 1.1 supports ordered, reliable delivery and decoupling of communication and computation, it sometimes still requires higher-level messaging layers to perform memory copies, because it lacks network pacing and gather/scatter functionality.

Network pacing

FM Sockets exposed the need for network pacing. **`FM_extract()`** is called only when an FM Sockets program posts a receive (that is, calls **`recv()`**, **`recvfrom()`**, or **`recvmsg()`**). But because **`FM_extract()`** processes the entire receive queue, FM Sockets is forced to buffer (that is, copy) a potentially large amount of data. When subsequent receives are posted, FM Sockets must again copy data, this time from the unposted receive buffer into a program-specified location.

FM 2.0 reduces such additional memory copying by allowing higher-level messaging layers to pace message processing by specifying the maximum number of bytes that **`FM_extract()`** processes. Thus, rather than processing all available data on a socket receive operation, FM Sockets needs to process data only until the overlying program's receive request is satisfied. This control also lets FM 2.0 programs more effectively schedule communication processing with computation (for example, to amortize communication costs behind computation time).

Gather/scatter

MPI-FM validated the importance of gather/scatter—initially deferred to higher-level messaging layers for implementation if needed. MPI-FM adds protocol-specific headers to the beginning of each message. Without gather/scatter built into FM, higher-level messaging layers must copy

program data twice: once on the send side to append protocol-specific headers and once on the receive side to strip those headers. These extra copies hurt performance significantly.[1] Traditional gather vectors (lists of <length,offset> pairs sent in sequence) remove copies on the send side, but traditional scatter vectors are insufficient for removing copies on the receive side, because a message's target location is known only when the message header is parsed.

Instead of gather/scatter vectors, FM 2.0 implements gather and scatter using *streaming messages*. Message data is written and read piecewise via a stream abstraction. On the send side, a higher-level messaging layer opens a message with `FM_begin_message()`, appends zero or more pieces of data to the message with `FM_send_piece()` (for example, a header followed by program data), and finally closes the message with `FM_end_message()`. On the receive side, a message handler is passed an opaque "stream" object instead of a pointer to a piece of memory. The handler receives data piecewise from the stream object, specifying a target memory location for each `FM_receive()`. Because each message is a stream of bytes (unlike TCP, which does not possess the concept of a message), the size of each piece received need not equal the size of each piece sent, as long as the total message sizes match. Thus, higher-level receivers can examine a message header and, based on its contents, scatter the message data to appropriate locations. The streaming interface is possible only because FM provides end-to-end flow control.

The streaming-message interface also allows the pipelining of individual messages; message processing can begin at the receiver even before the sender has finished. This increases the throughput of higher-level messaging layers built atop FM.

Performance

Although FM 2.0 is more powerful than FM 1.1, this strength does not necessarily penalize performance severely. Figure A shows the performance of Myrinet FM 2.0 on a pair of Dell Optiplex GXPros (200-MHz Pentium Pro-based PCs) and a pair of Sun Ultra-1 computers.

This figure shows the performance of FM 2.0 on two different types of workstations. As a rough point of comparison, the figure also repeats the FM 1.1 results from the main text of this article. Even with the streaming-message interface, FM 2.0 exhibits low latency and high bandwidth. FM 2.0's minimum latency is similar on the two types of systems—11.7 ms on the Dells and 12.81 ms on the Ultra-1s. However, FM 2.0's maximum bandwidth is far superior on the Dells than on the Ultra-1s (56.3 ms versus 32.0 ms). This is because FM 2.0 takes advantage of the Dell's PCI bus, which is much faster than the Ultra-1's SBus. Specifically, FM 2.0 employs PCI write combining on the send side, which enables consecutive writes to occur as rapid burst transactions.

FM 2.02, a heavily optimized but fully compatible implementation of the FM 2.0 interface, achieves a minimum latency of 8.0 ms and a maximum bandwidth of 88.0 Mbytes/s on the same Dells as we used above. FM 2.02 will be available soon from *http://www-csag.cs.uiuc.edu/projects/comm/sw-releases.html*.

Reference

1. M. Lauria and A.A. Chien, "MPI-FM: High Performance MPI on Workstation Clusters," to appear in *J. Parallel and Distributed Computing*, Feb. 1997; *http://www-csag.cs.uiuc.edu/papers/jpdc97-normal.ps*.

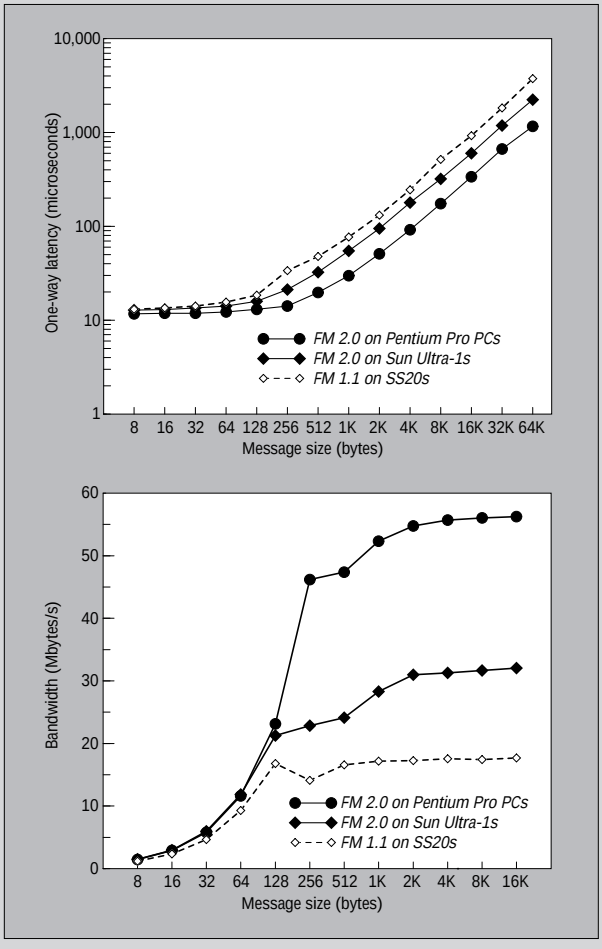

Figure A. FM 2.0 performance: (a) latency; (b) bandwidth.