Using Workstations as Building Blocks for Parallel Computing

Thomas M. Warschko, Joachim M. Blum, and Walter F. Tichy
University of Karlsruhe, Dept. of Informatics*

In: Allen Kent and James G Williams, editors, Encyclopedia of Computer Science and Technology, pages 373--391 Marcel Dekker, INC., New York, Basel.

Abstract

The key to efficient parallel computing on workstations clusters is a communication subsystem that removes the operating system from the communication path and eliminates all unnecessary protocol overhead. At the same time, protection and a stable multi-user, multiprogrammed environment cannot be sacrificed.

We have developed a communication subsystem, called ParaStation2, which fulfills these requirements. Its one-way latency is $14.5\mu s$ to $18\mu s$ (depending on the hardware platform) and throughput is 65 to 90 MByte/s, which compares well with other approaches. We were able to achieve an application performance of 5.3 GFLOP running a matrix multiplication on 8 DEC Alpha machines (21164A, 500 MHz).

ParaStation2 offers standard programming interfaces, including PVM, MPI, Unix sockets, Java sockets, and Java RMI. These interfaces allow parallel applications to be ported to ParaStation2 with minimal effort. The system is implemented on a variety of platforms, including DEC Alpha workstations running Digital Unix, and Intel PCs and DEC Alpha's running Linux.

Keywords: Workstation Cluster, High-Performance Cluster Computing, Parallel and Distributed Computing, High-Speed Interconnects, User-Level Communication.

1 Introduction and Motivation

Workstation clusters coupled by high-speed interconnection networks offer a cost-effective and scalable alternative to monolithic supercomputers. In contrast to supercomputers and parallel machines, clustered workstations rely on standardized communication hardware and communication protocols developed for local-area networks and not for parallel computing. As communication hardware is getting faster and faster, the communication performance is now limited by the processing overhead of the operating system and the protocol stack, rather than the network itself. Thus, users who upgrade from ethernet to, e.g., ATM, failed to achive an application speedup comparable to the improvement of the raw communication performance. The peak bandwidth in high-speed networks is often achieved only with extremely large message sizes, and there is no improvement at all when communication is based on small messages. Exchanging small messages, however, is a key issue in parallel computing. Another network-related problem is the scalability of the network. A typical SPMD-like parallel program consists of a sequence of computing and communicating phases. As a consequence, all

*Now: Scarasoft AG, Mühlfelder Straße 10, 82211 Herrsching, Germany. Email: {warschko,blum}@scarasoft.com nodes either compute, not using any network, or exchange data, generating bursty traffic on the network. Burst traffic in LAN topologies sharing a physical medium (bus or ring) results in a worst-case behaviour of the network, causing high latencies and low throughput. In contrast, MPP networks use higher-dimensional topologies such as grids or hypercubes, where bandwidth and bisection-bandwidth increases with the number of connected nodes. Thus, the critical parameters for a network well-suited for parallel computation are

- latency, when transmitting small messages,
- throughput, when transmitting large messages, and
- scalability, when increasing the number of compute nodes.

In addition to these hardware-related parameters, the functional behaviour of the network should be amenable to minimal protocol stacks. In contrast to most local-area networks, MPP networks offer reliable data transmission in combination with hardware-based flow control. As a consequence, traditional protocol functionality such as window-based flow control, acknowledgement of packages, and checksum processing can be reduced to a minimum (or even be left out). If the network further guarantees in-order delivery of packets, the fragmentation and efragmentation task of a protocol is much simpler, because incoming packets do not have to be rearranged into the correct sequence. Implementing as much protocol-related functionality as possible within the network interface results in a minimal and efficient protocol.

The most promising technique to speed up protocol processing is to move it out of the operating system kernel into the address space of a user process. Nevertheless, there are two disadvantages using a *user-level-communication* approach. First, the correct interaction and a certain level of protection between competiting processes has to be maintained. This is necessary, because the operating system as the coordinating instance is no longer on the communication path. This problem is either solved by restricting the network access to a single process, or by implementing mechanisms, which ensure correct intercation between several processes. These mechanisms reside in a software layer between the network interface and the application program. The second critical issue is the design of the user interface to the network. Often vendors support proprietary APIs (e.g., AAL5 for ATM, or Myricom's API for Myrinet) and research groups propose novel but nonstandard communication models (e.g., ActiveMessages, or Memory Mapped Communication), but for reasons of portability a user would prefer a standardized and well-known interface. Thus, the key issues for the design of an efficient communication subsystem are

- sharing the physical network among several processes,
- providing protection between processes using the network simultaneously,
- removing kernel overhead and traditional network protocols from the communication path, and
- still providing well-known programming interfaces.

ParaStation2 is an example of a user-level communication subsystem that is designed with all the given criteria in mind. To reach the goal of efficiency, the kernel is removed from the communication path and all hardware interfacing and protocol processing is done at user level. Correct interaction among concurrent processes is ensured within the system library at user level using semaphores. Besides proprietary user interfaces, we decided to provide an emulation of the standard Unix socket interface on top of our system layer as well as optimized versions of PVM and MPI.

This article describes the ParaStation2 architecture and related approaches (section 2), starting with a description of the ParaStation2 network interface (section 3) and the ParaStation2 software architecture (section 4). Various benchmarks are presented in section 5.

2 Cluster Projects

There are several approaches targeting efficient parallel computing on workstation clusters which can be classified according to the communication model used: memory mapped communication vs. message passing.

Memory mapped communication systems such as Digital's MemoryChannel [1], SCI-based SALMON [2, 3], Sun's S-Connect [4], SHRIMP [5] and virtual memory mapped communication (VMMC and VMMC-II) [6] from Princeton University map remote memory into the applications address space, allowing user processes to communicate without expensive buffer management and without system calls across the protection boundary separating user processes from the operation system kernel.

In contrast to these approaches, message passing systems such as Active Messages [7] and Active Messages-II [8] for the Berkeley NOW cluster [9], Active-Messages for ATM-based U-Net [10], Illinois Fast Messages [11], the link-level flow control protocol (LFC) [12] from the distributed ASCI supercomputer, PM [13] from the Real World Computing Partnership in Japan, the basic interface for parallelism from the University of Lyon (BIP) [14], Hamlyn [15], Trapeze [16], and ParaStation focus on variations of message-passing environments rather than a virtual shared memory. As von Eicken et al. pointed out [7], recent workstation operating systems do not support a uniform address space, so virtual shared memory is difficult to implement.

The Myrinet [17] interconnection hardware can be used to implement both basic communication models (messages passing and virtual memory mapped communication). The Berkeley NOW cluster, the Illinois HPVM cluster, Princeton's VMMC-II cluster, the Dutch ASCI cluster, the french BIP cluster, the japanese PM cluster, Trapeze, Hamlyn, and ParaStation2 all use Myrinet.

3 The ParaStation2 Network Interface

ParaStation was originally developed for the ParaStation hardware [18], a self-routing network with autonomous distributed switching, hardware flow-control at link-level combined with a back-pressure mechanism, and a reliable and deadlock-free transmission of variable sized packets (up to 512 byte). This base system is now being adopted to the Myrinet hardware, which has a fully programmable network interface and a much better base performance than the original ParaStation hardware. The major difference is the absence of reliable data transmission, which has to be implemented at network interface level on the Myrinet hardware (see sections 3.1 and 3.2).

The Myrinet adapter uses a 32bit RISC CPU called *LanAI*, fast SRAM memory (up to 1 MByte) and three programmable DMA engines – two on the network side to send and receive packets and one as interface to the host. The LanAI is fully programmable (in C / C++) and the necessary development kit (especially a modified gcc compiler) is available from Myricom. In fact this capability in addition to the high performance of the Myrinet hardware was the main criterion for choosing Myrinet as the hardware platform for ParaStation2.

3.1 Design considerations

The major questions to answer is how to interface the Myrinet hardware to the rest of the ParaStation software, especially the upper layers with their variety of implemented protocols (Ports, Sockets, Active Messages, MPI, PVM, FastRPC, Java Sockets and RMI). There are three different approaches:

1. Emulating ParaStation on the Myrinet adapter: Simulating ParaStation's transmission FIFO with a small LanAI program running on the Myrinet adapter would not be a problem. But as ParaStation is using programmed I/O to receive incoming packets this approach would lead to an unacceptable performance (see [19]).

2. Emulating ParaStation at software level: As the ParaStation system already has a small hardware dependent software layer called HAL (*hardware abstraction layer*), this approach allows the use of all Myrinet specific communication features as well as a simple interface to the upper protocol layers of the ParaStation system.

3. Designing a new system: This approach would lead to an ideal system and probably the best performance, but we would have to rewrite or redesign most parts of the the ParaStation system.

Because of its simplicity, we choose the second strategy to interface the existing ParaStation software to the Myrinet hardware. The second question to answer is how to guarantee reliable transmission of packets with the Myrinet hardware. As said before, the original ParaStation hardware offers reliable and deadlock free packet transmission as long as the receiver keeps accepting packets. Myrinet instead discards packets (after blocking a certain amount of time) which may happen when the receiver runs out of resources or is unable to receive packets fast enough. Additionally the Myrinet hardware seems to lose packets under certain circumstances, e.g. in heavy bidirectional traffic with small packets. The upper layers of the ParaStation system rely on a reliable data transmission, so a low level flow control mechanism – either within the Myrinet control program running on the LanAI processor or as part of the HAL interface – is required.

3.2 Implementation of the Myrinet Control Program

This section explains the basic ideas of our reliable transmission protocol, how it works and how it is implemented.

3.2.1 Basic Idea

The basic idea of our reliable transmission protocol is that every data packet has to be acknowledged using a technique called *positive acknowledgement with retransmission.* This technique is well known and used as basic principle within the TCP protocol [20, 21]. The sender keeps a record of each packet it sends in one of its transmission buffers and waits for an acknowledgement before it releases the buffer. The sender also starts a timer when it sends a packet and retransmits the packets if the timer expires before an acknowledgement arrives. To achieve better performance our protocol uses multiple buffers and allows multiple outstanding ACK's, a technique known as *sliding windows* [20, 21]. The current implementation of our protocol uses 8-bit sequence numbers and a window size of 8 packets. In case of a corrupted or lost packet (or ACK) our protocol implements a *go back N* behaviour, retransmitting the first unacknowledged and all subsequent packets, rather than using a *selective retransmission* strategy. In contrast to the TCP protocol, the ParaStation2 protocol also uses negative acknowledgements (NACK). In case of insufficient buffer space the receiver sends a NACK back to the sender to prevent further message transmission. As usual, the NACK stops the transmission of further packets and triggers the retransmission of the rejected packet(s).

3.2.2 Basic operation

Figure 1 shows the basic operation during message transmission of the ParaStation2 protocol. The basic protocol has four independent parts: (a) the interaction between the sending application and the sender network interface (NI), (b) the interaction between the sending and the receiving NI, (c) the interaction between the receiving NI and the receiving host, and (d) the interaction between the receiving application and the host.

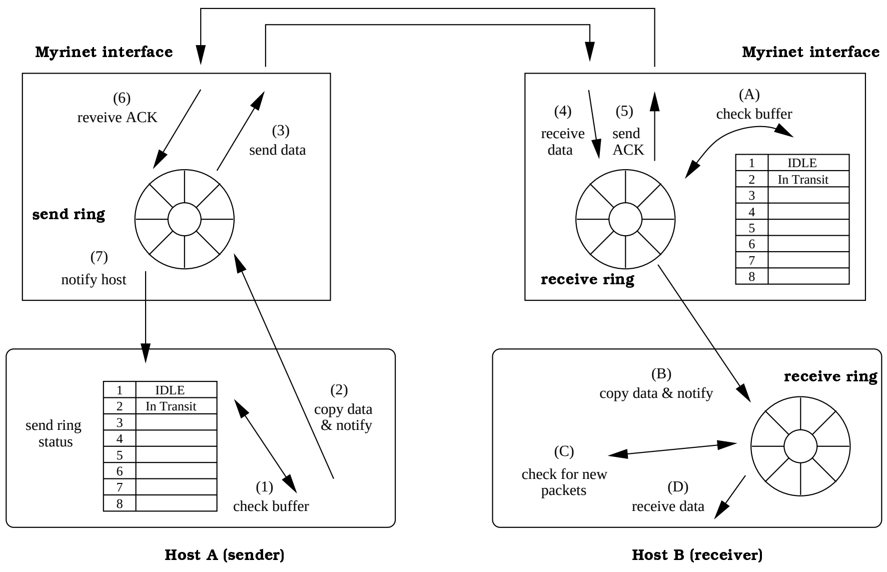

Figure 1: Data transmission in ParaStation2

First, the sender checks if there is a free send buffer (step 1). This is accomplished by a simple table lookup in the host memory, which reflects the status of the buffers of the send ring located in the fast SRAM of the network interface (Myrinet adapter). If there is buffer space available, the sender copies (step 2) the data to a free slot of the circular send buffer located in the network interface (NI) using programmed I/O. Afterwards the NI is notified (a descriptor is written) that the used slot in the send ring is ready for transmission and the buffer in host memory is marked as in transit. A detailed description of the buffer handling is given in section 3.2.3. In step (3), the NI sends the data to the network using its DMA engines.

When the NI receives a packet (step 4) it stores the packet in a free slot of the receive ring using its receive DMA engine. The flow control protocol ensures that there is at least one free slot in the receive ring to store the incoming packet. Once the packet is received completely and if there is another free slot in the receive ring, the flow control protocol acknowledges the received packet (step 5). The flow control mechanism is discussed in section 3.2.4. As soon as the sender receives the ACK (step 6), it releases the slot in the send ring and the host is notified (step 7) to update the status of the send ring.

In the receiving NI the process of reading data from the network is completely decoupled from the transmission of data to the host memory. When a complete packet has been received from the network, the NI checks for a free receive buffer in the host memory (step A). If there is no buffer space available, the packet will stay in the NI until a host buffer becomes available. Otherwise the NI copies the data into host memory using DMA and notifies the host about the reception of a new packet by writing a packet descriptor (step B). Concurrently, the application software checks (polls) for new packets (step C) and eventually, after a packet descriptor has been written in step (B), the data is copied to application memory (step D).

Obviously, the data transmission phases in the basic protocol (step 2, 3, 4, and B) can be pipelined between consecutive packets. The ring buffers in the NI (sender and receiver) are used to decouple the NI from the host processor. At the sender, the host is able to copy packets to the NI as long as there is buffer space available although the NI itself might be waiting for acknowledgements. The NI uses a transmission window to allow a certain amount of outstanding acknowledgements which must not necessarily equal the size of the send ring. At the receiver the NI receive ring is used to temporarily store packets if the host in not able to process the incoming packets fast enough.

3.2.3 Buffer handling

Each buffer or slot in one of the send or receive rings can be in one of the following states:

`IDLE`**:** The buffer is empty and can be used to store a packet (up to 8192 byte).

`INTRANSIT`**:** This buffer is currently involved in a send or receive operation, which has been started but which is still active.

`READY`**:** This buffer is ready for further operation either a send to the receiver NI (if it's a send buffer) or a transfer to host memory (if it's a receive buffer).

`RETRANSMIT`**:** This buffer is marked for retransmission, because of a negative acknowledgement or a timeout (send buffer only).

Figure 2 shows the state transition diagrams for both send and receive buffers in the network interface.

At the sender the NI waits until a send buffer becomes `READY`, which is accomplished by the host after it has copied the data and the packet descriptor to the NI (step 2 in figure 1). After the buffer becomes `READY` the NI starts a send operation (network DMA) and marks the buffer `INTRANSIT`. When an acknowledgement (ACK) for this buffer arrives (step 6 in figure 1), the buffer is released (step 7) and marked `IDLE`. If a negative acknowledgement (NACK) arrives or the ACK does not arrive in time (or gets lost) the buffer is marked for retransmission (`RETRANSMIT`). The next time the NI tries to send a packet it sees the `RETRANSMIT` buffer and

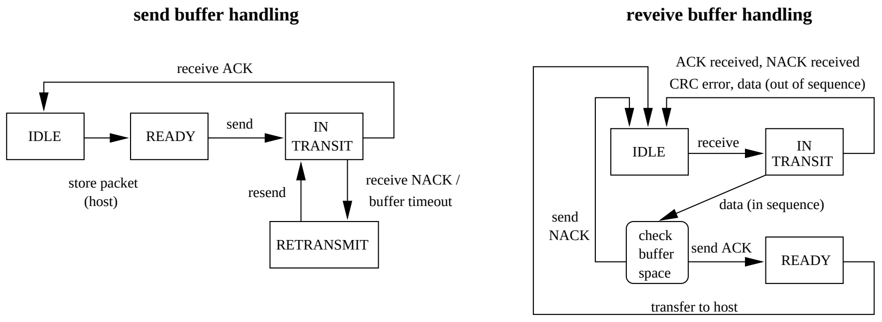

Figure 2: Buffer handling in sender and receiver

resends this buffer, changing the state to `INTRANSIT` again. This `RETRANSMIT` – `INTRANSIT` cycle may happen several times until an ACK arrives and the buffer is marked `IDLE`.
At the receiver the buffer handling is quite similar (see figure 2). As soon as the NI sees an incoming packet it starts a receive DMA operation and the state of the associated buffer changes from `IDLE` to `INTRANSIT` (see step 4 in figure 1). Assuming that the received packet contains user data, is not corrupted, and has a valid sequence number[1] the NI checks for another free buffer in the receive ring. If there is another free buffer it sends an ACK back to the sender and the buffer is marked `READY`. Otherwise a NACK is sent out, the packet discarded and the buffer released immediately (marked `IDLE`). The check for a second free buffer in the receive ring ensures that there is at least one free buffer to receive incoming packets anytime, because any packet eating up the last buffer will be discarded. When the received packet contains protocol data (ACK or NACK), the NI processes the packet and releases the buffer. In case of an error (CRC) the buffer is marked `IDLE` immediately without further processing. If the received data packet does not have a valid sequence number, the packet is discarded and the sender is notified by sending a NACK back. Thus the receiver refuses to accept out-of-sequence data and waits until the sender will resend the missing packet.

3.2.4 Flow control protocol

ParaStation2 uses a flow control protocol with a fixed sized transmission window and 8 bit sequence numbers (related to individual sender/receiver pairs), where each packet has to be acknowledged (either with a positive or a negative acknowledgement) in combination with a timeout and retransmission mechanism in case that an acknowledgement gets lost or does not arrive within a certain amount of time. The protocol assumes the hardware to be **unreliable** and is able to track any number of corrupted or lost packets (containing either user data or protocol information). Table 1 gives an overview of possible cases within the protocol, an explanation of each case as well as the resulting action initiated.

When a data packet is received, the NI compares the sequence number of the packet with the assumed sequence number for the sending node. If the numbers are equal, the received packet is the one that is expected and the NI continues with its regular operation. A received sequence number smaller than expected indicates a duplicated data packet caused by a lost or late ACK. Thus the correct action to take is to resend the ACK, because the

[1]For a discussion of the ACK/NACK protocol see section 3.2.4.

packet type	sequence check	explanation	resulting action
	$<$	lost ACK	resend ACK
DATA	$=$	ok	check buffer space (see fig 2)
	$>$	lost data	ignore & send NACK
	$<$	duplicate ACK	ignore packet
ACK	$=$	ok	release buffer
	$>$	previous ACK lost	ignore packet
NACK	none		mark buffer for retransmission
CRC	none	error detected	ignore packet

Table 1: Packet processing within receiver

sender expects one. Is the received sequence number larger than expected, the packet with the correct sequence number has been corrupted (CRC) or lost. As the protocol does not have a selective retransmission mechanism the packet is simply discarded and the sender is notified with a negative acknowledgement (NACK). Thus, this packet will be retransmitted later either because the sender got the NACK, or because of a timeout. As the missing packet also causes a timeout at the sending side, the packets will eventually arrive in the correct order.

On the reception of an ACK packet, the NI also checks the sequence number and if it got the expected number it continues processing and releases the acknowledged buffer. If the received sequence number is smaller than assumed, we have received a duplicated ACK because the sender ran into a transmission timeout before the correct ACK was received and the receiver has resent an ACK upon the arrival of an already acknowledged data packet[2]. The response in this case is simply to ignore the ACK. A received sequence number larger than what is expected indicates that the correct ACK has been corrupted or lost. Thus the action taken is to ignore the ACK, but the associated buffer is marked for retransmission to force the receiver to resend the ACK. The buffer associated with the assumed (and missing) ACK will timeout and be resent which also forces the receiver to resend the ACK.

A received NACK packet does not need sequence checking; the associated buffer is marked for retransmission as long as it is in the `INTRANSIT` state. Otherwise the NACK is ignored (the buffer is in `RETANSMIT` state anyway). In case of a CRC error the packet is dropped immediately and no further action is initiated, because the protocol is unable to detect errors in the protocol header.

The resulting protocol is able to handle any number of corrupted or lost packets containing either user data or protocol information, as long as the NI and the connection between the incorporated nodes is operational. The protocol was developed to ensure reliability of data transmission at NI level, not to handle hardware failures in terms of fault tolerance. The protocol itself can be optimized in some cases (e.g. better handling of ACK's with a larger sequence number), but this is left to future implementations. In comparison to existing protocols, this protocol can roughly be classified as a variation of the TCP protocol (using NACK's and a fixed size transmission window).

[2]This case may sound strange, but we've seen this behaviour several times.

4 ParaStation User-Level Communication (PULC)

The major goal is to support a standardized, but efficient programming interface such as UNIX sockets on top of the ParaStation network. In contrast to other approaches, the ParaStation network is used by parallel applications exclusively and is not intended as a replacement for a common LAN, so associated protocols (e.g., TCP/IP) can be eliminated. These properties allow using specialized network features, optimized point-to-point protocols, and controlling the network at user level without operating system interaction (see figure 3).

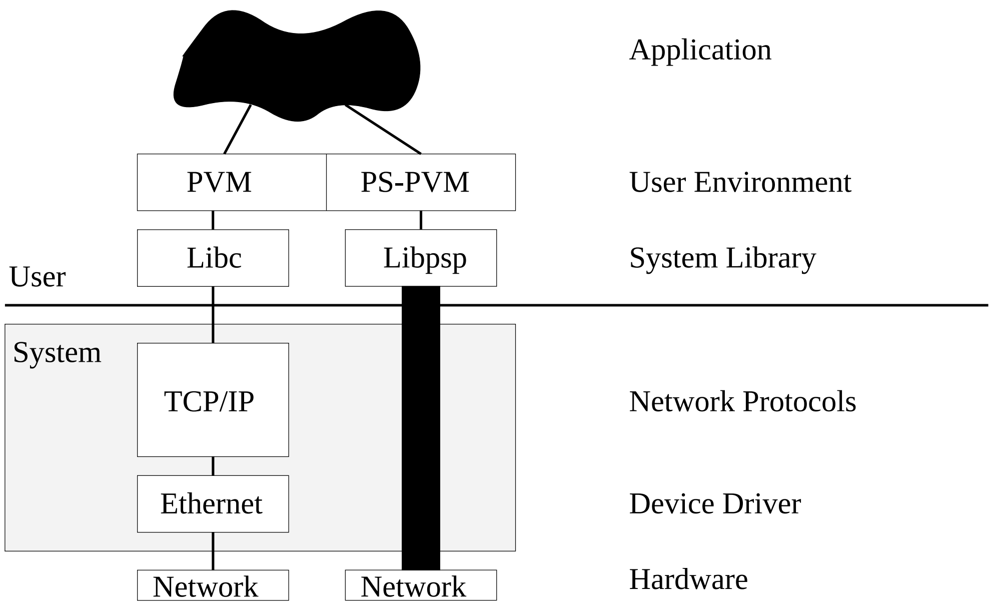

Figure 3: User-level communication highway

The ParaStation protocol software within our system library (libpsp) implements multiple logical communication channels on a physical link. This is essential to set up a multiuser/multiprogramming environment. Protocol optimization is done by minimizing protocol headers and eliminating buffering whenever possible. Sending a message is implemented as zero-copy protocol which transfers the data directly from user-space to the network interface. Zero-copy behaviour during message reception is achieved when the pending message is addressed to the receiving process; otherwise the message is copied once into a buffer in a common message area. Within the ParaStation network protocol, operating system interaction is completely eliminated, removing it from the critical path of data transmission. The missing functionality to support a multiuser environment is realized at user level in the ParaStation system library.

4.1 Architecture of PULC

Figure 4 gives an overview of the three major parts of PULC: The PULC interface, the PULC message handler, and the PULC resource manager.

PULC Programming Interface: This module acts as the programming interface for any application. The design is not restricted to a particular interface definition such as Unix

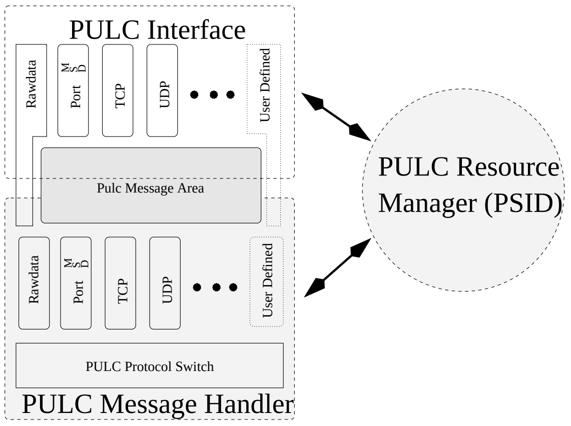

Figure 4: PULC Architecture

sockets. It is possible and reasonable to have several interfaces (or protocols) residing side by side, each accessible through its own API. Thus, different APIs and protocols can be implemented to support a different quality of service, ranging from standardized interfaces (i.e. TCP or UDP sockets), widely used programming environments (i.e. MPI or PVM), to specialized and proprietary APIs (ParaStation ports and a true zero copy protocol called Rawdata). All in all, the PULC interface is the programmer-visible interface to all implemented protocols.

PULC Message Handler: The message handler is responsible to handle all kinds of (low level) data transfer, especially incoming and outgoing messages, and is the only part to interact directly with the hardware. It consists of a protocol-independent part and a specific implementation for each protocol defined within PULC. The protocol-independent part is the *protocol switch* which dispatches incoming messages and demultiplexes them to protocol specific *receive handlers*. To get high-speed communication, the protocols have to be as lean as possible. Thus, PULC protocols are not layered on top of each other; they reside side by side. Sending a message avoids any intermediate buffering. After checking the data buffer, the sender directly transfers the data to the hardware. When receiving a message, the data is first written to a message buffer and then delivered to the application. Only the *Rawdata* protocol behaves differently and copies the received data directly to its final destination in the application space.

PULC Resource Manager: This module is implemented as a Unix daemon process (PSID) and supervises allocated resources, cleans up after application shutdowns, and controls access to common resources. Thus, it takes care of tasks usually managed by the operating system. All PSIDs communicate with each other. They exchange local information and transmit demands of local processes to the PSID of a remote node. With this cooperation, PULC offers a distributed resource management and supports a single system view.

To be portable amongst different hardware platforms and operating systems, PULC implements all hardware and operating system specific parts in a module called hardware abstraction layer (HAL). Thus, choosing a different interconnection network as discussed in section 3.1

only forces the adoption of the HAL to the quality of service the new communication hardware provides.

One possible drawback in the design of user level protocols is using a polling strategy to wait for the arrival of new messages, because polling consumes CPU cycles while waiting. If the sender is on the same node, the sender is slowed down and it takes even longer to execute the send call for which the receiver is waiting for. Thus, PULC uses different co-scheduling strategies to hand off the CPU to the sender. Therefore even node-local message transfers have acceptable latencies.

The following sections presents the three main parts of PULC – the resource manager, the message handler, and the programming interface – in detail.

4.2 PSID: The PULC Coordinator

Since PULC is entirely implemented in user-space, the operating system does not manage the resources. This task is done by a resource manager (PSID: ParaStation Daemon). It cleans up resources of dead processes and organizes access to the message area. Before a process can communicate with PULC, the process has to register with the PSID. The PSID can grant or deny access to the message area and the hardware.

The PSID also checks if the version used by the PULC interface and the PULC message handler are compatible, which makes corruption of data impossible. The PSID can restrict the access to the communication subsystem to a specific user or a maximum number of processes. Thus it is possible to run the cluster in an optimized way, since multiple processes slow down application execution due to scheduling overhead.

All PSIDs communicate with each other. They exchange local information and transmit demands of local processes to the PSID of a remote node. With this cooperation, PULC offers a distributed resource management and provides the semantic of a single system. PULC allows spawning (remote execution) and killing (remote termination) processes on any node of the cluster. To do so, the PSID first transmits a request to the PSID of the remote node. The remote PSID then uses regular operation system calls to spawn (`fork()` and `exec()`) or kill (`signal()`) processes. Afterwards, the spawned process runs with the same user id as the spawning process. Furthermore, PULC redirects the output of a spawned process back to the terminal of the parent process. Therefore it offers a transparent view of the cluster.

The PSIDs periodically exchange load information. Using this information, load balancing is possible when spawning new tasks. Several strategies are feasible:

- Spawn a new task on a specified node: No selection is done by PULC. The spawn request is transfered to the remote PSID, which creates the new task. As result a new task identifier is returned.
- Spawn a task on the next node: PULC keeps track of the node which was used to spawn the last task on. This strategy selects the next node by incrementing the node number.
- Spawn a task on an unloaded node: Before spawning, PULC sorts the available nodes by their load. After that, PULC spawns on the node with the least load.

These strategies allow a PULC cluster to run in a balanced fashion, while still allowing the programmer to specify an explicit node layout, when an application requires a specific communication pattern.

4.3 The PULC Message Handler

The PULC message handler is responsible for sending and receiving messages.

4.3.1 Sending messages

Sending a message avoids any intermediate buffering. After checking the buffer, the sender directly transfers the data to the hardware. The specific protocols inside the message handler are responsible for the coding of the protocol header information. PULC doesn't restrict the length or form of the header. PULC just specifies the form of the hardware header with its protocol id. The rest of the message header must be interpretable by the protocol specific receive handler. If the receiver is on the local node, the receive handler optimizes message transfer by directly calling the appropriate receive handler of the protocol.

4.3.2 Receiving a message

The major task while receiving a message is demultiplexing of incoming messages. Demultiplexing means to distinguish between different protocols and in a multiprogramming environment to distinguish between different receiving processes.

Demultiplexing of protocols is accomplished within the *PULC protocol switch*, which reads the hardware dependent header of the message with its unique protocol identifier. After decoding the identifier, the protocol switch directly transfers control to the receive handler of the protocol, which reads the rest of the message. This *header forwarding* is extremely fast and avoids any unnecessary copy stage. After that its up to the protocol specific part where to store the incoming data – either directly in user data structures, as it is done in the rawdata protocol, or queuing the data in protocol specific message queues (TCP, UDP, PORT-M/S/D).

The second demultiplexing task is to distinguish between different receiving processes, which is accomplished within the protocol specific part of the message handler. In general, the message handler stores the incoming messages in a common accessible message pool (a memory segment shared between all PULC processes) by ordering them into protocol specific message queues. These message queues form the interface between the PULC message handler and the PULC interface (which is described in the next section). Thus handling multiple concurrent processes does not present a problem, because all these processes use different protocols and different port identifiers for their connections. Each port has it's own queue and the message handler delivers incoming messages to the associated queue.

PULC uses several optimizations to speed up message reception. Some of these are *preallocated fragments*, *destination prediction*, and *end queuing*.

- *Preallocated fragments* guarantees that there is always an allocated fragment which is used by the message handlers to enqueue a message in a port. Thus receive handlers do not have to spend time to allocate a new fragments.

- With *destination prediction* PULC tries to predict the destination of the next fragment. It keeps track of recently addressed ports and therefore minimizes table lookups for destination ports. This is reasonable, because a large message is fragmented into different parts and there is a high probability that these parts are received one after another.

- When queuing a fragment *end queuing* does the same technique inside a port. It knows which fragment was queued last and checks if the new fragment is related to the old one. If they are related, this technique speeds up queuing.

The following subsections describe the protocol specific actions of the predefined message handlers.

Rawdata Protocol The rawdata protocol is a true zero copy protocol. If the receive handler knows the final destination of a message at the time of receiving, it directly transfers the data to this location. This kind of message transaction minimizes latencies and maximizes throughput. If the receive handler does not know or can not access the final destination, the fragments are placed in a queue of the *rawdata port*. Since there is only one rawdata port on a node, only one application can use this protocol at a time. This is a restriction similar to many other user-level protocols.

Port-M/S/D Protocol The Port-M/S/D protocol is a frontend to the PULC functionality. It mainly implements additional functionality which is not available within the standard UNIX socket specification. The Port protocols offers dynamic process creation, group scheduling, and a testbed for new functionalities. The differences between Port-M (multiple stream), Port-S (single stream), and Port-D (datagrams) is just which queuing function the receive handler calls.

TCP/UDP Protocols TCP and UDP protocols in PULC use ports as their communication channel. UDP and TCP have differences in connection establishment and they use different queuing strategies: TCP uses the single stream strategy whereas UDP uses the datagram strategy. In ParaStation, the data transfer is reliable and so UDP derives this property from the underlying hardware.

Other Protocols PULC is open for user defined protocols. New protocols plug into the protocol switch and the switch will redirect fragments to the new protocols. New protocols have to ensure correct locking to exclude deadlocks, as they have to co-exist with the predefined protocols. This co-existence allows implementing new protocols with the same efficiency as the predefined PULC protocols.

4.4 The PULC Interface

Each protocol in the message handler can have its own interface. The interface is the counterpart of the message handler. The message handler receives a message and puts it in the message area whereas the interface functions get these messages as soon as they are received completely. The cooperation between the interface functions and the receive handler of the protocol includes correct locking of the port and its message queues. Correct interaction is necessary since PULC does not have control of the scheduling decisions of the operating system. Thus the receive handler could be in a critical section while the operating system switches to a process which conflicts with this critical section. This would leave PULC in an inconsistent state.

A process can use several interfaces at the same time. E.g., it can use the sockets for regular communication and PULC's ability to spawn processes through the Port-M interface.

Rawdata Interface The Rawdata interface offers a true zero copy operation when receiving messages, if the final destination in memory space is accessible by the PULC interface at the time of reception. In case that the message handler runs in the address space of the rawdata process this is always true. If the message handler runs on the communication processor the final destination must reside in a mapped and pinned-down message area, because the communication processor needs a physical address of destination memory. During receive, the rawdata interface first checks the rawdata port if any appropriate message is available. If there is not any message available, it registers the receive buffer at the rawdata receive handler, which places the incoming data into this buffer. VMMC-II [22] uses a similar approach, which they call *transfer redirection.*

Port Interface The interface is similar to the standard Unix socket interface, but has additionally functionality for dynamic process creation on remote nodes (spawning) and logging of the child processes. The ports are addressed by an index (descriptor) into the own private port table. A destination port is addressed by a port identifier which is a combination of the node number and the peer address of the port. The message queues of the ports are either single streams, multiple streams, or datagram oriented as described in section 4.3.2.

Socket Interface The socket interface to PULC is the same as for BSD Unix sockets – it even uses the same name space. This allows easy porting of applications and libraries to PULC, by simply linking the application code with an additional library. Nevertheless, PULC provides a fall-back mechanism, which redirects calls to destination nodes not reachable inside the PULC cluster transparently to the operating system.

PULC sockets use specially tuned methods with caching of recently used structures. This allows an extremely fast communication with minimal protocol overhead. Each socket has a port as its communication channel. The socket receive handler only knows about the ports and uses different enqueuing strategies for UDP (datagram ports) and TCP sockets (single stream ports). The socket interface provides the interaction between the communication ports and the socket descriptor. Sockets can be shared among different processes due to a `fork()` call and can be inherited by a `exec()` call. During `fork()`, the socket is duplicated but both sockets share the same communication port (the count attribute of the port is incremented). Thus, both processes have access to the message queue of the socket. After an `exec()` and a reconnection to PULC the sockets of the message area are inserted into the private socket descriptor table. Therefore the process has access to these abstractions again.

User Defined Interfaces PULC is open for extensions. Users are able to define their own interfaces to existing message handlers or to define a new message handler and an interface to it. The interfaces just have to ensure that the interaction with the used message handler is correct. We plan to support other interfaces and message handlers in the near future. Candidates for this are Active Messages, MPI, and Java RMI.

4.5 Communication Libraries on Top of PULC

There are several communication libraries built on top of PULC. Most of them are just the standard Unix distributions on top of sockets, which are relinked with the PULC socket library. Using this technique, P4 [23] and tcgmsg [24] have been adopted to PULC. Other programming environments, such as PVM [25], have been modified [26] in a way that they can co-exist simultaneously to the standard socket implementation. This allows a direct comparison of operating system communication and PULC. The comparison shows that PVM adds a significant overhead to the regular socket communication. This lead to a new approach [27], which optimized PVM on top of the port-D interface. PULC already provides efficient and flexible buffer management and therefore this functionality could be eliminated in the PVM source. The resulting PSPVM2 is still interoperable with other PVMs running on any other cluster or supercomputer.

The PULC MPI implementation is based on MPICH [28]. MPICH provides a *channel* interface which hardware manufacturers can use to port MPICH to their own communication subsystem. This channel interface is implemented on top of PULC's port-D protocol. MPICH on PULC uses PULC's dynamic process creation at startup. The implementation is well-suited for MPI-2, which is supporting dynamic process creation at run-time. It is possible to support MPI directly as an interface of PULC. Most of the necessary functionality is already provided in the Port protocol.

5 Performance

The benchmarks described in this section cover two different scenarios. The communication benchmark provides information about the application-to-application performance that can be achieved at ParaStation's different software layers. The second scenario deals with application performance, namely run-time efficiency and application speedup.

5.1 Basic performance of the protocol hierarchy

Table 2 compares the performance of the ParaStation2 protocol to VMMC-2 and AM-II, which both use a reliable transmission protocol for Myrinet.

System	Latency [μs]	Throughput [MByte/s]
ParaStation2	14.5 – 18	65 – 90
VMMC-2	13.4	90
AM-II	21	31

Table 2: Performance comparison between reliable systems

The communication latency of ParaStation2 is between $14.5\mu s$ and $18\mu s$ (platform dependent, see table 3) and compares well to the $13.4\mu s$ of VMMC-2 (Intel/PCI/Linux platform) and the $21\mu s$ of AM-II (Sun Sparc/SBUS/Solaris platform). ParaStation2's 65 MByte/s to 90 MByte/s throughput is as high as the 90 MByte/s of VMMC-2 (using the same platform, see table 3), and two to three times as high as AM-II (31 MByte/s). The low performance for AM-II is caused by the Sparc/SBUS interface and not due to the AM-II transmission protocol.

In table 3, performance figures of all software layers in the ParaStation2 system are presented. The various levels presented are the hardware abstraction layer (HAL), which is the lowest layer of the hierarchy, the so called *ports* and TCP layers, which are build on top of the HAL, and standardized communication libraries such as MPI and PVM, which are optimized for ParaStation2 and build on top of the ports layer. Latency is calculated as round-trip/2 for a 4 byte ping-pong and throughput is measured using a pairwise exchange for large messages (up to 32K). N/2 denotes the packet size in bytes when half of the maximum throughput is reached. The performance data is given for three different host systems, namely a 350MHz Pentium II running Linux (2.0.35), a 500MHz and a 600MHz Alpha 21164 system running Digital Unix (4.0D).

		Programming interface				
System	Measurement	HAL	Ports	TCP	MPI	PVM
Pentium II 350 MHz	Latency [μs]	14.5	18.7	20.2	25	30
	Throughput [MByte/s]	90	78	76	73	58
	N/2 [Byte]	512	1000	1000	2000	2000
Alpha 21164 500 MHz	Latency [μs]	17.5	24	24	30	29
	Throughput [MByte/s]	65	55	57	50	49
	N/2 [Byte]	512	500	500	1000	1000
Alpha 21164 600 MHz	Latency [μs]	18.0	24	25	27	32
	Throughput [MByte/s]	75	65	71	62	57
	N/2 [Byte]	1024	1000	1000	2000	2000

Table 3: Basic performance parameters of ParaStation2

The latency at HAL level of $14.5\mu s$ to $18\mu s$ is somewhat higher than for systems which do not ensure reliable data transmission such as LFC ($11.9\mu s$) or FM ($13.2\mu s$) [12]. This is because neither LFC nor FM copies the data it receives to the application and second, both LFC and FM incorrectly assume Myrinet to be reliable. The 90 Mbyte/s throughput of ParaStation2 for the Intel platform is between FM (up to 60 MByte/s), LFC (up to 70 MByte/s), PM (90 MByte/s), and BIP (up to 125 MByte/s) [29].

Switching from a single-programming environment (HAL) to multi-programming environments (upper layers) results in a slight performance degradation regarding latency as well as throughput. The reason for increasing latencies is due to locking overhead to ensure correct interaction between competitive applications. The decreased throughput is caused by additional buffering, a complex buffer management, and locking overhead.

5.2 Performance at application level

Focusing only on latency and throughput is too narrow for a complete evaluation. It is necessary to show that a low-latency, high-throughput communication subsystem also achieves a reasonable application efficiency. For this reason we installed the widely used and publicly available ScaLAPACK[3] library [30], which uses BLACS[4] [31] on top of MPI as communication subsystem on ParaStation2. As benchmark we use the parallel matrix multiplication for general dense matrixes from the PBLAS library, which is part of ScaLAPACK. Table 4 shows the performance in MFLOP's running on our 8 processor DEC-Alpha cluster (500 MHz, 21164A). First, we have measured the uniprocessor performance of a highly optimized matrix multiplication (cache aware assembler code), which acts as reference to calculate the efficiency of the parallel versions. A uniprocessor performance of 772 to 790 MFLOP on a 500 MHz processor proves that the program is highly optimized (IPC of more than 1.5). Obviously the parallel version executed on an uniprocessor has to be somewhat slower, but the measured efficiency

[3]Scalable Linear Algebra Package.

[4]Basic Linear Algebra Communication Subroutines

Problem size (n)	Uniprocessor MFlop (Eff.)	1 Node	2 Nodes	4 Nodes	6 Nodes	8 Nodes
		Performance in MFlop (Efficiency)				
1000	782 (100%)	731 (93.5%)	1276 (81.6%)	2304 (73.6%)	3243 (69.1%)	3871 (61.9%)
2000	785 (100%)	743 (94.6%)	1359 (86.6%)	2546 (81.1%)	3582 (76.1%)	4683 (74.6%)
3000	790 (100%)	755 (95.6%)	1396 (88.4%)	2700 (85.4%)	3908 (82.4%)	4887 (77.3%)
4000	772 (100%)		1398 (90.5%)	2694 (87.2%)	4044 (87.3%)	**5337** (86.4%)

Table 4: Parallel matrix multiplication on ParaStation2

of 93.5% to 95.6% is very high. Using more nodes, the absolute performance in MFLOP increases steadily while the efficiency decreases smoothly. The maximum performance achieved was 5.3 GFLOP using 8 nodes which is quite good compared to the 10.1 GFLOP of the 100 nodes Berkeley NOW cluster[5].

Table 5 shows the performance of the same experiment using FastEthernet instead of ParaStation2 as communication subsystem; the application code was just linked with another MPI-library.

Problem size (n)	Uniprocessor MFlop (Eff.)	1 Node	2 Nodes	4 Nodes	6 Nodes	8 Nodes
		Performance in MFlop (Efficiency)				
1000	782 (100%)	731 (93.5%)	1085 (69.4%)	1358 (43.4%)	1270 (27.1%)	1135 (18.1%)
2000	785 (100%)	743 (94.6%)	1274 (81.2%)	1861 (59.2%)	2007 (42.6%)	1722 (27.4%)
3000	790 (100%)	755 (95.6%)	1291 (81.7%)	2011 (63.6%)	2121 (44.7%)	1925 (30.5%)
4000	772 (100%)		1337 (86.6%)	2047 (66.3%)	**2342** (50.6%)	2312 (37.4%)

Table 5: Parallel matrix multiplication using FastEthernet

The maximum performance achieved using FastEhternet was 2.3 GFLOP, which is far less than the 5.3 GFLOP of ParaStation2 (only 44%). Although a small increase in performance can be achieved while adding up to 6 processors the efficiency drops dramatically. Using more than 6 processors even results in a performance decrease. The reason for this is due to the different raw performance of FastEtherent and Myrinet (100 Mbit/s vs. 1280 Mbit/s) and due to the topology of the Myrinet installation (8x8 crossbar vs. 8-port hub).

6 Conclusion

In this paper we presented the design and evaluation of ParaStation2 as example for a high-performance cluster based on off-the-shelf workstations and PCs. The design of ParaStation2

[5]see `http://now.cs.berkeley.edu`

at hardware level focuses on how to implement a reliable transmission protocol on top of the Myrinet hardware. This reliable data transmission is a prerequisite for the design of PULC with it's variety of different protocols. Again, the focus within the design of PULC is the implementation of efficient but standardized protocol interfaces.

The evaluation of ParaStation2 shows that ParaStation2 compares well with other approaches in the cluster community using Myrinet. ParaStation2 is not the fastest systems in terms of pure latency and throughput, but in contrast to most other approaches it offers a reliable interface which is – in our experience – more important to the user than an ultra high-speed, but unreliable interface. Similar arguments hold for the performance of PULC. Providing a multiuser and multiprogramming environment results in a decrease in performance, but offering standardized and well known programming interfaces justifies the approach taken.

Furthermore, the comparison between ParaStation2 and FastEthernet shows a tremendous gain in performance (factor 2.3) when using a carefully design communication subsystem instead the common TCP/IP stack implementation within recent operating systems. The achieved application performance of 5.3 GFLOPs has not been achieved before with this small number of nodes and an efficiency of 86.4% compares well to commercial parallel machines. All in all this proves that high-performance cluster computing using workstations and PC as building blocks is feasible.

References

[1] Peter Ross. Unix ^{TM} clusters for technical computing. Technical report, Digital Equipment Coropration, December 1995.

[2] IEEE. *IEEE – P1596 Draft Document. Scalable Coherence Interface Draft 2.0*, March 1992.

[3] Knut Omang. Performance results from SALMON, a Cluster of Workstations Connected by SCI. Technical Report 208, University of Oslo, Department of Informatics, November 1995.

[4] Andreas G. Nowatzyk, Michael C. Browne, Edmund J. Kelly, and Michael Parkin. S-connect: from networks of workstations to supercomputer performance. In *Proceedings of the 22nd International Symposium on Computer Architecture (ISCA), Santa Margherita Ligure, Italy*, pages 71–82, June 22-24 1995.

[5] Matthias A. Blumrich, Cezary Dubnicki, Edward W. Felten, Kai Li, and Malena R. Mesarina. Virtual-Memory-Mapped Network Interfaces. *IEEE Micro*, 15(1):21–28, February 1995.

[6] C. Dubnicki, A. Bilas, K. Li, , and J. Philbin. Design and Implementation of Virtual Memory-Mapped Communication on Myrinet. In *11th Int. Parallel Processing Symposium*, pages 388–396, Geneva, Switzerland, April 1997.

[7] Thorsten von Eicken, Anindya Basu, and Vineet Buch. Low-Latency Communication Over ATM Networks Using Active Messages. *IEEE Micro*, 15(1):46–53, February 1995.

[8] B. Chung, A. Mainwaring, and D. Culler. Virtual Network Transport Protocols for Myrinet. In *Hot Interconnects'97*, Stanford, CA, April 1997.

[9] Thomas E. Anderson, David E. Culler, and David A. Patterson. A Case for NOW (Network of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[10] Anindya Basu, Vineet Buch, Werner Vogels, and Thorsten von Eicken. U-net: A user-level network interface for parallel and distributed computing. In *Proc. of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, Colorado*, December 3-6, 1995.

[11] Scott Pakin, Mario Lauria, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, San Diego, California, December 3-8 1995.

[12] Raoul A. F. Bhoedjang, Tim Rühl, and Henri E. Bal. LFC: A Communication Substrate for Myrinet. In *Fourth Annual Conference of the Advanced School for Computing and Imaging*, Lommel, Belgium, June 1998.

[13] H. Tezuka, F. O'Carrol, A. Hori, , and Y. Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In *12th International Parallel Processing Symposium*, pages 308–314, Orlando, Florida, Mar 30 - Apr 3, 1998.

[14] L. Prylli and B. Tourancheau. Protocol Design for High Performance Networking: A Myrinet Experience. Technical Report 97-22, LIP-ENS Lyon, July 1997.

[15] G. Buzzard, D. Jacobson, M. MacKey, S. Marovich, and J. Wilkes. An Implementation of the Hamlyn Sender-Managed Interface Architecture. In *The 2nd USENIX Symp. on Operating Systems Design and Implementation*, pages 245–259, Seattle, WA, October 1996.

[16] K. Yocum, J. Chase, A. Gallatin, and A. Lebeck. Cut-Through Delivery in Trapeze: An Exercize in Low-Latency Messaging. In *The 6th Int. Symp. on High Performance Distributed Computing*, Portland, OR, August 1997.

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

[18] Thomas M. Warschko. *Effiziente Kommunikation in Parallelrechnerarchitekturen.* PhD thesis, Universität Karlsruhe, Fakultät für Informatik, March 1998. In: VDI Fortschritt-Berichte, Reihe 10, Nr. 525, VDI-Verlag, ISBN: 3-18-352510-0.

[19] Raoul A. F. Bhoedjang, Tim Rühl, and Henri E. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):52–60, November 1998.

[20] Douglas E. Comer. *Internetworking with TCP/IP. Volume I: Principles, Protocols, and Architecture.* Prentice-Hall International Editions, 1991.

[21] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall International Editions, second edition, 1989.

[22] Cezary Dubnicki, Angelos Bilas, Yuqun Chen, Stefanos N. Damianakis, and Kai Li. VMMC-2: Efficient support for reliable, connection-oriented communication. Technical Report TR-573-98, Princeton University, Computer Science Department, February 1998.

[23] Ralph Buttler and Ewing Lusk. *User's Guide to the p4 Parallel Programmimg System.* ANL-92/17, Argonne National Laboratory, October 1992.

[24] R. J. Harrison. Portable tools and applications for parallel computers. *International Journal on Quantum Chem.*, 40:847–863, 1991.

[25] A. Beguelin, J. Dongarra, Al Geist, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User's Guide and Reference Manual.* ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[26] Joachim M. Blum, Thomas M. Warschko, and Walter F. Tichy. PSPVM: Implementing PVM on a high-speed Interconnect for Workstation Clusters. In *Proceedings of the Third Euro PVM Users' Group Meeting*, pages 235–242, München, Germany, October 1996.

[27] Patrick Ohly, Joachim M. Blum, Thomas M. Warschko, and Walter F. Tichy. PSPVM2: PVM for ParaStation. In *Proceedings of 1st Workshop on Cluster-Computing*, pages 147–160, Chemnitz, 6. - 7. November 1997.

[28] William Gropp, Ewing Lusk, and Nathan Doss adn Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. Technical report, Mathematics and Computer Science Division, Argonne National Laboratory and Missisipi State University, 1996.

[29] Soichiro Araki, Angelos Bilas, Cezary Dubnicki, Jan Edler, Koichi Konishi, and James Philbin. User-space communication: A quantitative study. In ACM, editor, *SC'98: High Performance Networking and Computing: Proceedings of the 1998 ACM/IEEE SC98 Conference: Orange County Convention Center, Orlando, Florida, USA, November 7–13, 1998.* ACM Press and IEEE Computer Society Press, November 1998.

[30] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPCK: A Portable Linear Algrbra Library for Distributed Memory Computers – Design Issues and Performance. Technical Report UT CS-95-283, LAPACK Working Note #95, University of Tennesee, 1995.

[31] J. Dongarra and R. C. Whaley. A user's guide to the blacs v1.0. Technical Report UT CS-95-281, LAPACK Working Note #94, University of Tennesee, 1995.