ELSEVIER

Journal of Systems Architecture 44 (1998) 241–260

JOURNAL OF SYSTEMS ARCHITECTURE

ParaStation: Efficient parallel computing by clustering workstations: Design and evaluation

Thomas M. Warschko [*,1], Joachim M. Blum, Walter F. Tichy

University of Karlsruhe, Dept. of Informatics Am Fasanengarten 5, D-76128 Karlsruhe, Germany

Received 1 November 1996; received in revised form 3 February 1997; accepted 1 April 1997

Abstract

ParaStation is a communications fabric for connecting off-the-shelf workstations into a supercomputer. The fabric employs technology used in massively parallel machines and scales up to 4096 nodes. ParaStation's user-level message passing software preserves the low latency of the fabric by taking the operating system out of the communication path, while still providing full protection in a multiprogramming environment. The programming interface presented by ParaStation consists of a UNIX socket emulation and widely used parallel programming environments such as PVM, P4, and MPI. Implementations of ParaStation using various platforms, such as Digital's AlphaGeneration workstations and Linux PCs, achieve end-to-end (process-to-process) latencies as low as 2 μs and a sustained bandwidth of up to 15 Mbyte/s per channel, even with small packets. Benchmarks using PVM on ParaStation demonstrate real application performance of 1 GFLOP on an 8-node cluster.

Keywords: Workstation cluster; Parallel and distributed computing; User-level communication; High-speed interconnects

1. Introduction

Networks of workstations and PCs offer a cost-effective and scalable alternative to monolithic supercomputers. Thus, bundling together a cluster of workstations – either single-processors or small multiprocessors – into a parallel system would seem to be a straightforward solution for computational tasks that are too large for a single machine. However, conventional communication mechanisms and protocols yield communication latencies that prohibit any but very large grain parallelism.

* Corresponding author. E-mail: warschko@ira.uka.de.

[1] WWW: http://wwwipd.ira.uka.de/parastation.

PII S1383-7621(97)00039-8

For example, typical parallel programming environments such as PVM [1], P4 [2] and MPI [3] have latencies in the magnitude of milliseconds. As a consequence, the parallel grain size necessary to achieve acceptable efficiency has to be in the range of tens of thousands of arithmetic operations.

In contrast, existing massively parallel systems (MPPs) offer an excellent communication/computation ratio. However, engineering lag time leads to a widening gap in the rapidly increasing performance of state-of-the-art microprocessors and low-volume manufacturing of MPPs results in a cost/performance disadvantage. This situation is not unique to MPP systems; it applies to multiprocessor servers as well [4].

ParaStation's approach is to combine the benefits of a high-speed MPP network with the excellent price/performance ratio and the standardized programming interfaces of conventional workstations. Well-known programming interfaces ensure portability over a wide range of different systems. The integration of a high-speed MPP network opens up the opportunity to eliminate most of the communication overhead.

The retargeted MPP network of ParaStation was originally developed for the Triton/1 system [5] and operates in a 256-node system. Key issues of the network design are based around autonomous distributed switching, hardware flow-control at link-level, and optimized protocols for point-to-point message passing. In a ParaStation system, this network is connected to the host system via a PCI-bus interface board. The software design focuses on standardized programming interfaces (UNIX sockets), while preserving the low latency and high throughput of the MPP network. ParaStation implements operating system functionality in user-space to minimize overhead, while providing the protection for a true multiuser/multiprogramming environment.

The current design is capable of performing basic communication operations with a total process-to-process latency of just a few microseconds (e.g., 2 μs for a 32-bit packet). Compared to workstation clusters using standard communication hardware (e.g., message-passing software such as PVM using Ethernet/FDDI hardware), our system shows performance improvements of more than two orders of magnitude on communication benchmarks. As a result, application benchmarks (e.g., ScaLAPACK equation solver and others) execute with nearly linear speedup on a wide range of problem sizes.

2. Related work

There are several approaches targeting to efficient parallel computing on workstation clusters which can be classified as shared-memory and distributed-memory systems. Shared-memory systems such as MINI [6], SHRIMP [7], SCI-based SALMON [8,9], Digital's MemoryChannel [10], and Sun's S-Connect [11] support memory-mapped communication, 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, distributed memory systems such as Active Messages [12] for ATM-based U-Net [13], Illinois Fast Messages [14] for Myrinet [15], and the Berkeley NOW project [4], also based on Myrinet, focus on a pure message-passing environment rather than a virtual shared memory. As von Eicken et al. pointed out [12], recent workstation operating systems do not support a uniform address space, so virtual shared memory is difficult to maintain.

As with Active Messages and Fast Messages, performance improvement within the ParaStation system is based on user-level access to the network, but in contrast to them, we provide multiuser/multiprogramming capabilities. Like Myrinet and

S-Connect, our network was originally designed for a MPP system (Triton/1) and has now been retargeted to a workstation cluster environment. Myrinet, IBM-SP2, and Digital's MemoryChannel use central switching fabrics, while ParaStation provides distributed switches on each interface board.

Recent parallel machines (e.g., Thinking Machines CM-5, Meiko CS-2, IBM SP-2, Cray T3D/T3E) also provide user-level access to the network. These solutions rely on custom hardware and are constrained to the controlled environment of a parallel machine, whereas ParaStation can be (and actually is) implemented on a variety of different platforms from different vendors (i.e., any system that provides a PCI-Bus).

3. Performance hurdles in workstation clusters

Using existing workstation clusters as a virtual supercomputer suffers from several problems due to standard communication hardware, traditional approaches in the operating system, and the design of widely used programming environments.

Standard communication hardware (i.e., Ethernet, FDDI, ATM) was developed for a LAN/WAN environment rather than for MPP communication. Network links are considered unreliable, so higher protocol layers must detect packet loss or corruption and provide retransmission, flow control and error handling. Common network topologies, such as a bus or a ring, do not scale very well. Sharing one physical medium among the connected workstations results in severe bandwidth limitations and high latencies. 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. A fixed or inappropriate packet size wastes bandwidth when transmitting small messages and again leads to large latencies. Thus, critical parameters for a network well-suited for parallel computation are:

- latency, to transmit small messages,
- throughput, to transmit large data streams, and
- scalability, when increasing the number of connected workstations.

In addition to these hardware-related parameters, an efficient interface between a high-speed network and parallel applications is an essential requirement. By now, the communication subsystem lies within the operating system, which provides two standardized interfaces: one to the hardware at device driver level, and the other to the user in the form of system calls. System calls guarantee the protection needed in a multiuser environment, and the device driver level provides a transparent interface for different hardware. As the communication subsystem of the operation system has to deal with different hardware abilities, the overall service at the device driver level is limited to the least common denominator of the hardware. This structure prohibits the use of specialized features of the communication hardware. Implementing the user interface to the communication subsystem as system calls implies at least a context switch overhead and the copying of message buffers between user and kernel space. Often, this takes more time than transmitting the message over the wire, especially for small messages. Furthermore, built-in communication protocols (e.g., TCP/IP) are designed to support communication in local and wide area networks and therefore are not very well suited for the needs of parallel computing. Their complex protocol stacks tend to limit the throughput of the whole communication subsystem. To avoid processing overhead in a protocol stack, the underlying network has to maintain as much functionality as possible.

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 task and especially the reassembly task of a protocol become much easier, because incoming packets do not have to be rearranged into the correct sequence. Implementing as much protocol-related functionality as possible directly in hardware results in minimal and thus efficient protocols.

The most promising technique to improve the performance of the network interface as seen by a user is to move protocol processing into the user's address space. The ParaStation system in fact does all protocol processing at user-level while still providing protection. Another critical issue is the design of the user interface to the network. Often, vendors support proprietary APIs (e.g., AAL5 for ATM), 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 for the ParaStation architecture are:

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

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. Even protection is done within the system library at user-level using processor-supported atomic operations to implement semaphores. Besides proprietary user interfaces, we decided to offer an emulation of the standard Unix socket interface on top of our system layer.

Popular message-passing environments such as MPI [3], PVM [1], P4 [2], and others are usually built on top of the operating system communication interface which typically provides either reliable stream based communication (TCP) or unreliable message-based communication (UDP). But the type of service needed for message-passing is reliable message-based communication. Thus, all these message-passing environments have to implement a reliable message-based communication protocol on top of TCP sockets. This causes additional overhead due to different addressing and naming conventions, buffering considerations, and the mapping of data streams to messages. A second inefficiency arises from the multiuser and multiprogramming environment in workstation clusters. Running several parallel applications simultaneously on a workstation cluster results in scheduling and synchronization delays. The operating system of each individual node is only able to use local information to schedule processes, where global information would be appropriate (e.g., gang scheduling within the cluster). Thus, execution time of each application takes much longer than running the applications one after another. Running multiple processes of one application on one processor causes an additional process switching overhead, especially if these processes communicate with each other. Furthermore, all these environments allow processes to consume messages in arbitrary order and at arbitrary times. This implies message buffering at the destination node unless the receiving process is set up for receipt.

A promising technique to overcome the performance bottlenecks of current message-passing environments is to support a specialized API within the user-level communication subsystem. This API could provide all necessary functionality such as reliable message-based communication channels, message multiplexing and demultiplexing, dynamic coscheduling of concurrent threads, or even

gang scheduling of applications within the cluster to better meet the requirements of standardized message-passing libraries.

4. The ParaStation architecture

The ParaStation architecture is centered around the reengineered MPP network of Triton/1 [16,5]. The goal is to support a standard but efficient programming interface such as UNIX sockets. The ParaStation network provides a high data rate, low latency, scalability, flow control at link-level, minimized protocol, and reliable data transmission. Furthermore, because the ParaStation network is dedicated to parallel applications and is not intended as a replacement for a common LAN, it can therefore eliminate the associated protocols. These properties allow the use of specialized network features, optimized point-to-point protocols, and controlling the network at user-level without operating system interaction. The ParaStation protocol implements multiple logical communication channels on a physical link. This is essential to set up a multiuser/multiprogramming environment. Protocol optimization is achieved by minimizing protocol headers and eliminating buffering whenever possible. Sending a message is implemented as a zero-copy protocol which transfers the data directly from user-space to the network interface. Zero-copy behavior 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 commonly accessible message area. Within the ParaStation network protocol, operating system interaction is completely eliminated, removing it from the critical path of data transmission. The functionality missing to support a multiuser environment is implemented at user-level in the ParaStation system library.

4.1. Hardware Architecture

Experience with network interfaces in parallel machines made it clear that providing a self-routing network, flow control, reliable data transmission, and in-order delivery of packets at the network interface level opens up the opportunity to design fast and efficient protocols. A self-routing network frees the protocol layers from any routing decision, because packets will not show up at any software layer on intermediate nodes. Hardware-based flow control at link-level eliminates all related mechanisms usually found in communication protocols, such as transmission windows, packet sequence numbers, acknowledgements, timeouts, and retransmissions. In conjunction with transmission lines with a very low error rate, hardware-based flow control acts as base technology to offer a reliable end-to-end data transmission. In-order delivery of packets is a more practical issue, because it simplifies the fragmentation and especially the reassembly task of a protocol implementation when transferring large data sets.

4.1.1. Hardware overview

We use the reengineered MPP-network of Triton/1 as communication hardware. The network topology is based on a two-dimensional toroidal mesh. For small systems a ring topology is sufficient. Data transport is done via a table-based, self-routing packet switching method which uses virtual cut-through routing. Every node (see Fig. 1) is equipped with its own routing table and with three input buffers: two for intermediate storage of data packets coming from other nodes and one for receiving packets from its associated processing element (workstation). An output buffer delivers data packets to the associated workstation. The buffering (1 KB fifos) decouples the network operation from local processing. Packets contain the address of the target node, the number

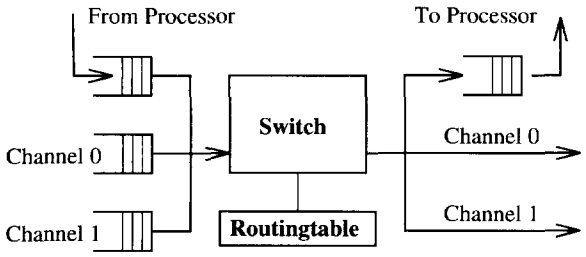

Fig. 1. ParaStation network.

of data words contained in the packet, and the data itself. The size of the packet can vary from 4 to 508 bytes. Packets are delivered in order and no packets will be lost. Flow control is done at link-level and the unit of flow control is one packet.

For both topologies – ring and toroidal mesh – we provide a deadlock-free routing scheme. Deadlock-free routing on a ring is simple, as long as the network is prevented from overloading. This problem is solved by inserting new packets into the network only when both channel fifos are empty. Deadlock-free routing on a toroidal mesh is done by using X–Y dimension routing. First, a packet is routed along the x-axis of the grid until it reaches its destination column. Then it is routed along the y-axis to its final destination node. Providing similar insertion rules as in the ring routing for both dimensions and giving the y-axis priority over the x-axis prevents deadlock.

The current implementation of our communications processor involves a routing delay of about 250 ns per node and offers a maximum throughput of 20 Mbyte/s per link. In addition to the communication network, the interface board provides a hardware mechanism for fast barrier synchronization. To connect several systems, we use 60-wire flat ribbon cables, with standardized RS-422 differential signals. Thus, the maximum distance between any two systems is 10 m (about 30 feet). The balanced interface circuits of RS-422 in combination with a correct line termination acts as base to ensure reliable data transmission. Nevertheless, each 16 bit wide data word is protected with two parity bits to detect transmission errors. [2]

4.1.2. Hardware details

The following functional diagram shows details of the ParaStation network and will be used to explain how packet transmission, routing, and flow control is currently implemented. For clarity, the second communication link is omitted from Fig. 2.

The network interface presents itself as a set of registers, namely a status register (SREG), a control register (CREG), and a fifo interface for transmitting data (NREG) to the (low level) programmer. The SREG provides information about the current state of the hardware, such as full, half-full, and empty signals from both transmission fifos (input and output buffer, see figure). The CREG is used to control various operations related to interrupt handling, programming of the synchronization lines, and loading the Xilinx FPGA (network processor). The data interface (NREG) to transfer packets into or out of the network is implemented as a pair of fifos to decouple the host processor from the operation of the network processor.

The message transfer between host processor, the network interface, and within the network itself is accomplished by a packet-based flow control mechanism. We choose a packet as the unit of flow control for several reasons: first, easier buffer management and transmission control within the network, and second, faster interface routines between the host processor and the network interface. If the network processor has knowledge about buffer space in a destination *before* transmitting a message, it can either refuse to transmit (insufficient buffer space) or it can transmit the

[2] The only transmission errors we have ever detected were caused by faulty hardware.

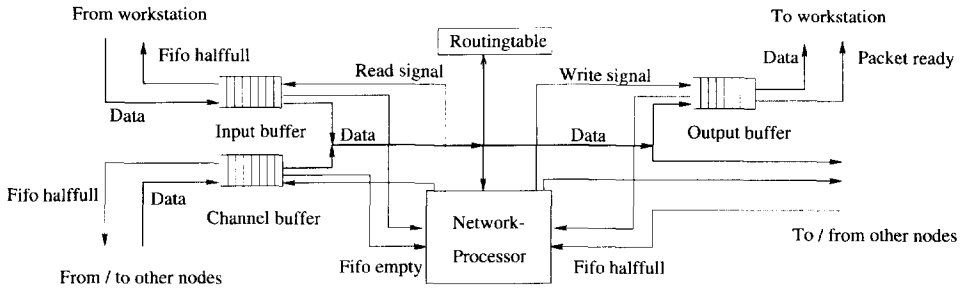

Fig. 2. Flow control details.

complete message at once (sufficient buffer space). Thus, the network processor of the sending node cannot be blocked during a message transfer and there is no need at the destination node to drop incoming packets due to buffer overflow. Blocking a network node is critical because of deadlock situations, and dropping packets causes a network to be unreliable, unless an appropriate (expensive) flow control mechanism is provided (transmission windows, sequence numbers, acknowledgements, timeouts, and retransmission).

From the host processor's point of view, knowledge about buffer space within the network interface can be used to speed up transmission routines. Depending on the buffer space available, the host processor can refuse to send the message or it can send the whole message at once. Otherwise, the host is forced to poll the buffer state each time a single word is transferred, which takes about twice as long. Receiving a complete message is even more difficult, because without special hardware support it is hard to figure out if a complete message is available at the output buffer. The output fifo provides empty, half-full, and full signals; but that is insufficient information to keep track of the number of received messages. The ParaStation network solves this problem with additional hardware to provide a 'packet ready' signal if a complete message resides pending in the output fifo ('packet ready' signals of successive messages are buffered in an additional fifo). This information is then used by the host processor to receive a complete message without additional polling.

Packet-based flow control and autonomous routing are closely related within the ParaStation network. As stated above, the network processor transmits a packet only if there is enough buffer space at the destination node. To determine the path of an incoming packet, the network processor uses the empty and half-full signals of all incoming and outgoing links as follows. (In combination with X–Y dimension routing in a toroidal mesh, this information is sufficient to determine if there is sufficient buffer space at the destination node.) According to the X–Y dimension routing, a packet is first routed along the x-axis and then along the y-axis to its final destination node. Thus, a packet pending at the y-channel has to stay on the y-channel. If the half-full signal of the destination node along the y-axis is inactive, this packet is transferred. In case of pending packets at the x-channel, the network processor has to check the half-full signals of both outgoing links, because without further inspection of the packet it is impossible to determine the correct destination link. Sufficient buffer space at both neighbor nodes and an empty incoming y-channel causes the network controller to transfer a pending packet at the x-channel. If there is a packet pending at the

input buffer of the associated host processor, the network controller checks the empty signals of both incoming as well as the half-full signals of both outgoing channels. To protect the network from overloading and blocking, all incoming channels have to be empty and there has to be sufficient buffer space at both outgoing channels. Under these conditions, the network processor decides to insert a new packet into the network. Once the network controller has drawn its decision on which packet to transfer next, it reads the first message flit (the destination address of that packet) out of the fifo and puts it on the internal data bus. The routing table listening to this bus provides information about the final destination link of that packet. Finally, the network controller transfers the packet.

4.2. Software architecture

The basic principle behind the ParaStation software architecture is user-level communication. Accessing the communication hardware directly from user space removes the operating system and traditional protocol stacks from the communication path. This concept opens up the opportunity to improve both the performance and the flexibility of communication protocols as well as to investigate highly optimized and carefully designed high-performance protocols to better meet the requirements for parallel computing in a workstation cluster.

To avoid operating system overhead, all interfacing to the ParaStation hardware is at user-level (see Fig. 3). The device driver is used at system startup to configure the communication boards and within the startup-code of an application program to get information about the hardware. During normal operation (i.e., message transfers), the ParaStation system library interfaces directly to the hardware without using the operating system. The gap between hardware capabilities and user requirements is bridged within the ParaStation system library.

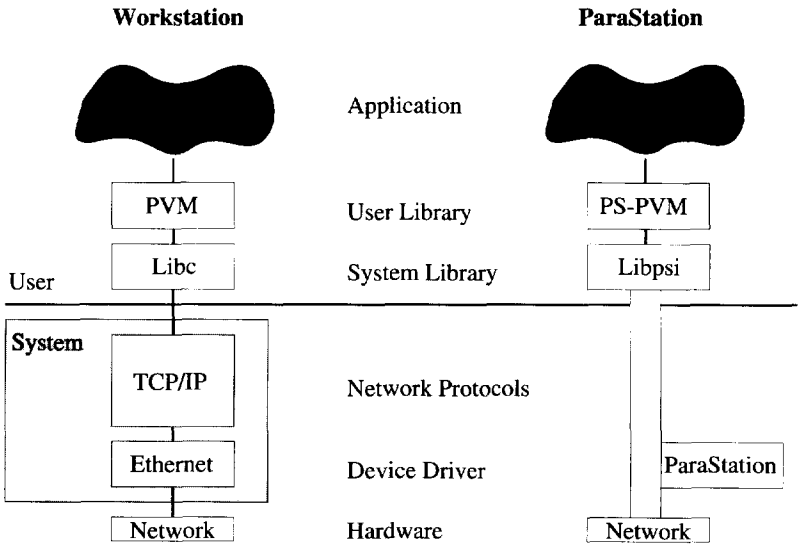

Fig. 3. Network interfacing techniques.

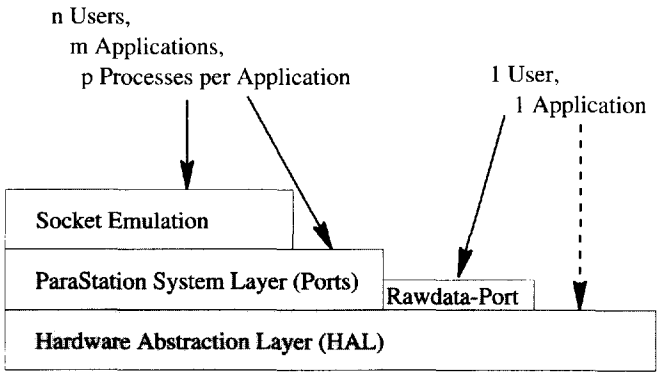

Fig. 4. ParaStation system library.

The ParaStation system library (see Fig. 4) acts as a second communication subsystem besides that of the operation system, but offers at top level the same communication services as the operating system, namely the Unix socket interface. The core layer (ports) implements all necessary abstractions (independent communication channels) to provide a multiuser/multiprogramming environment. The hardware abstraction layer was introduced to hide architecture and hardware specific operations from upper layers. The following sections give a brief description of all layers within the ParaStation communication library.

4.2.1. Hardware abstraction layer

This layer intends to hide architecture and hardware specific operations from upper layers. It is intended to be used only by the ParaStation system layer and is not considered as an application programming interface. The hardware abstraction layer provides highly optimized send/receive operations, a status information call, and an initialization call. The initialization call is used to map the hardware registers (status register, command register, and communication fifos) into user space. The information call looks for pending messages and checks if the network is ready to accept new messages. The send call transmits a packet into the network and the receive call delivers the next pending packet to the caller.

Since messages at this level are addressed to nodes rather than individual communication channels, message headers simply contain the address of the target node, the number of data words contained in the packet, and the data itself. While sending a message, data is copied directly from user space memory to the interface board. Receiving a message is split into two separate calls. The first call (`receive_header`) delivers only the message header to the caller whereas the second call (`receive_body`) copies the message data itself to the appropriate location in memory. This approach allows eliminating all intermediate buffering and leads to true zero-copy behavior.

4.2.2. System layer (ports)

The system layer provides the necessary abstraction (multiple communication channels) between the basic hardware capabilities (the hardware just handles packages) and a multiuser/multiprogramming environment. The major tasks within this layer are message multiplexing and

demultiplexing, mutual exclusion and correct interaction between competing processes, and fragmentation and reassembly of arbitrary-sized messages.

To support individual communication channels – called *ports* in ParaStation –, the system layer maintains a minimal software protocol, which adds information about the sending and receiving port in each packet. This concept is sufficient to support multiple processes by using different port identifiers for different processes. To handle multiple communication channels, multiplexing of outgoing and especially demultiplexing of incoming messages is accomplished within this protocol layer. Message multiplexing is done by ensuring the correct interaction between competing processes. The demultiplexing task instead has to keep track of the relationship between incoming messages, port identifiers, and associated processes. In a first step, the complete message is copied to a common accessible message pool (shared memory segment). Afterwards it is easy to decide whether this message is going to be delivered to the calling process. Although it introduces additional overhead in some cases, we choose this buffering technique for the following reasons. First, if the incoming messages belongs to a different process, it has to be buffered anyway. Second, if the incoming packet is part of a fragmented message, it first has to be reassembled to a complete message before it can be delivered to the calling process. Third, message reception may be invoked while sending large messages to prevent the network from overloading and blocking. In this case, incoming messages have to be buffered because no application is going to receive a message.

The general problem of demultiplexing and buffering messages is closely related to the way applications are used to receive messages. Receiving a message is an active process where an application issues a receive operation and the communication subsystem is expected to deliver the requested message. Furthermore, an application assumes the reception of complete messages and is usually not prepared to receive message fragments.

Maintaining a correct interaction between processes while sending or receiving messages necessitates the locking of critical code regions by semaphores. For reasons of efficiency, we also implemented these semaphores at user-level, using processor supported atomic operations. A fine granularity while locking critical code regions provides fairness among competing processes. Busy waiting while trying to enter a critical code region already locked is prevented by hands-off scheduling. Using this technique, the locking process is able to continue operation much earlier. This improves overall performance.

To obtain as much performance (and as little overhead) as possible, the system layer provides a so called *rawdata port*, which differs from regular ports as follows. Rawdata connections are implemented as a separate protocol which uses less protocol information than regular ports. Second, receiving small messages is done by copying them directly to user-space without intermediate buffering (true zero-copy). Third, the rawdata protocol uses another locking schema of critical code regions which gives a higher priority to the rawdata connection. Nevertheless, the *rawdata port* and regular ports can be used simultaneously, but applications using the *rawdata port* are scheduled one after another.

Our implementation of these concepts does not need a single system call. Furthermore, we provide a zero-copy behavior (no buffering) whenever possible. This leads to high bandwidth and low latencies.

4.2.3. Socket layer

The socket layer provides an emulation of the standard UNIX socket interface (TCP and UDP connections), so applications using socket communication can be ported to the ParaStation system

with little effort. To provide a greater flexibility, this layer supports a fall-back mechanism which transparently uses regular operating system calls if a communication request cannot be satisfied within the ParaStation cluster. Calls that can be satisfied within the ParaStation cluster do not need any interaction with the operating system.

4.2.4. Application layer

The installation of standard programming environments such as PVM [1], MPI [3], P4 [2], TCGMSG [17], and others on ParaStation is simply done by replacing the standard socket interface with the ParaStation sockets for high-speed communication. This approach allows us to easily port, maintain, and update these packages. We use the out-of-the-box software distribution.

4.3. Interaction of components

The ParaStation system consists of five major components (see Fig. 5): the communication hardware (a PCI-bus board), a device driver, the communication library, a common accessible message buffer, and the ParaStation daemon process.

The ParaStation device driver is responsible for initializing the ParaStation hardware at system startup, mapping the hardware registers to user space at application startup, and – together with the ParaStation daemon process – for ensuring protection. The ParaStation daemon process acts as a rather passive component, as it is not capable of handling any communication request. It is involved in application startup, maintaining information about the state of the ParaStation cluster, and administration, and is also responsible for keeping the shared message buffer consistent in case an application crashes. The shared message buffer is used to keep control information, such as used ports and sockets, and process control blocks of active processes, as well as to buffer incoming messages not yet deliverable to the associated application.

The ParaStation library itself acts as the trusted base within a ParaStation system. The library is statically linked to each application using ParaStation and ensures correct interaction between all parts of the system. At application startup, it contacts the ParaStation daemon to register the application as a ParaStation process. During this

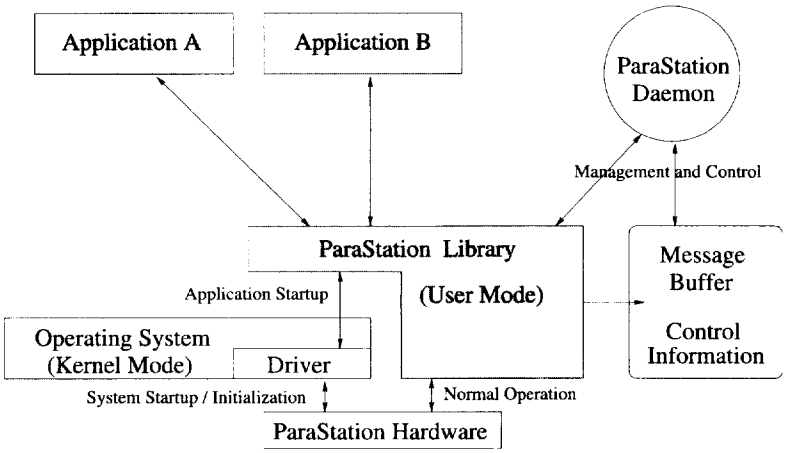

Fig. 5. Interaction of ParaStation components.

procedure, the daemon checks permissions as well as the current library version to prevent outdated applications from disturbing the system. After permission is granted, it maps all necessary interfaces to the address space of that application. During normal operation (i.e., sending and receiving messages), the library interacts only with the application, the hardware, and the shared message buffer. Applications are not allowed to directly access either the hardware or the message buffer; all operations have to be provided by the library. If an application finishes, the library informs the daemon process to release all resources used by this application. Application crashes are automatically detected by the ParaStation daemon and used resources can be freed.

4.4. Parastation platforms

Currently, ParaStation supports AlphaGeneration workstations from Digital Equipment running Digital UNIX (OSF/1) and Intel PCs (486, Pentium, PentiumPro) running Linux. Ports to the DEC-Alpha platform running Linux as well as ports to the Intel-PC and DEC-Alpha platform running Windows NT are in progress. Ports to other platforms (e.g., Sun/Solaris, IBM-PowerPC/AIX, SGI/IRIX) are possible, but not yet scheduled.

Our current testbed consists of three different ParaStation clusters. One cluster is based on 21064A Alpha workstations (275 MHz, 64 MB memory) with 8 nodes. The additional cost to equip this Alpha cluster with the ParaStation communication boards was about 16.000 USD, which is less than what we paid for a single Alpha workstation. The second cluster consists of four 21066 Alpha workstations (233 MHz, 64 MB memory). All Alpha machines are running Digital Unix 3.2c. The third cluster consists of two Intel PCs (120 MHz, 40 MB memory) running Linux 2.0.

5. Performance evaluation

The evaluation described in this section covers three different scenarios. The communication and synchronization benchmarks provide information about the raw performance of ParaStation. Although we call this 'raw performance', these benchmarks reflect application-to-application performance measured at the hardware abstraction layer. Second, we present the level of performance that can be achieved at ParaStation's different software layers (see Section 4.2). The third scenario directly deals with application performance, namely run time efficiency.

```
Process1:
  measure start-time;
  DO i = 1,k
    send(message)
    receive(message)
  ENDDO
  measure stop-time;
  calculate latency and throughput;

Process2:
  measure start-time;
  DO i = 1,k
    send(message)
    receive(message)
ENDDO
measure stop-time;
Calculate latency and throughput;
```

Algorithm 1. Pairwise Exchange codefragment

5.1. Communication benchmark

To measure the end-to-end delay, we implemented a *Pairwise Exchange* benchmark (see algorithm 1) where two processes send a message to each other simultaneously, and then receive simultaneously. Unlike a *Ping-Pong* benchmark, the second process does not wait for receipt of a mes-

Table 1
ParaStation performance at the hardware abstraction layer

Message size (bytes)	Alpha 21064A, 275 MHz		Alpha 21066, 233 MHz		Intel pentium, 120 MHz	
	Time per iteration (μs)	Through-put (Mbyte/s)	Time per iteration (μs)	Through-put (Mbyte/s)	Time per iteration (μs)	Through-put (Mbyte/s)
Word transfer						
1	2.52	0.794	2.39	0.978	1.75	1.143
2	2.51	1.592	2.40	1.957	1.75	2.272
4	2.48	3.228	1.94	4.108	1.76	4.545
8	3.24	4.939	2.58	6.188	2.26	7.080
Block transfer						
4	3.54	2.260	3.62	2.205	2.93	2.730
8	4.27	3.739	4.13	3.860	3.22	4.969
16	5.71	5.596	5.39	5.921	4.09	7.824
32	8.69	7.358	8.25	7.956	5.72	11.189
64	14.56	8.772	12.90	9.894	9.29	13.778
128	26.40	9.693	22.74	11.238	17.25	14.772
256	50.31	10.227	42.45	12.019	33.25	15.366
508	95.90	10.506	81.43	12.450	65.05	15.592

sage before transmitting. This is a more practical scenario for two processes exchanging messages.

The following table contains the results from the *Pairwise Exchange* benchmark, while varying message size from 1 to 508 bytes. [3] Transmitting larger messages can be done by fragmentating them into several smaller packets. To get accurate timing information, we measured run time of one million iterations ($k = 10^6$ in the above code fragment) for each packet size. For very short message sizes (word transfer), we use specialized routines with less overhead than the general block transfer routine.

For small message sizes (see Table 1), ParaStation achieves transmission latencies (sending and receiving a message in user-space) as low as 2.5 μs on systems with the 21064A processor, 1.9 μs on systems with the 21066 processor, and 1.8 μs on Pentium machines. For larger message sizes, with decreasing overhead per byte, we get a total throughput of up to 10.5 Mbytes/s (21064A), 12.5 Mbytes/s (21066), and 15.5 Mbytes/s (Pentium) respectively. The performance differences are due to the location of the PCI interface. The Alpha 21064A is using a board-level chipset (21072), where as the Alpha 21066 benefits from its on-chip PCI interface. Furthermore, the Alpha processor has a write buffer which is capable of combining writes to the same memory addresses. As the ParaStation communication interface is implemented as a fifo buffer, we had to insert memory barrier (MB) instructions after each write to the fifo to overcome the write combining problem. The MB instruction itself waits for all outstanding read and write operations and thus limits the performance on these two architectures. Although the

[3] 508 bytes user data is the maximum packet length of the ParaStation interface.

Pentium system uses a board-level chipset (Intel Triton), this system shows the best performance because there is no write combining problem. The Alpha 21164 processor has a write memory barrier instruction (WMB), which prohibits write combining but does not interfere with outstanding read operations. Early measurements on an Alpha 21164 system (300 and 500 MHz) show quite the same performance as a Pentium system.

5.2. Synchronization benchmark

As mentioned above, SPMD-style parallel programs often need barrier synchronizations to keep their processes in synchrony. The following code fragment (see algorithm 2) was used to measure the performance of our hardware-supported synchronization mechanism on ParaStation.

To get accurate timing information, we measured run time of one million iterations ($k = 10^6$) of the given code fragment. To compare our results to conventional methods, we also implemented a logarithmic barrier synchronization using standard operating system calls.

The performance improvement of our hardware mechanism shown in Table 2 is so overwhelming that no further explanation is needed. The results were measured on the 21064 cluster; the 21066 and Pentium clusters are about 17% faster.

```
Process1:
  measure start-time;
  DO i = 1,k
    sync()
  ENDDO
  measure stop-time;
  calculate timing;

Process2:
  measure start-time;
  DO i = 1,k
    sync()
  ENDDO
  measure stop-time;
  calculate timing;
```

Algorithm 2. Synchonization code fragment

5.3. Performance of the protocol hierarchy

Switching from single- to multiprogramming environments often suffers from a drastic performance decrease. In Table 3, performance figures of all software layers in the ParaStation system are presented.

To support a true multiprogramming environment, our system layer (ports) only adds about 10 μs (8 μs on the PC) additional latency to communication calls, and the loss of throughput compared to the hardware abstraction layer on the

Table 2
Synchronization performance

Number of stations	ParaStation		Ethernet	
	Runtime per iteration (μs)	Synchronizations per second	Runtime per iteration (μs)	Synchronizations per second
2	1.6	625.000	576	1739
4	1.7	588.000	1223	818
8	2.3	435.000	1856	539

Table 3
Performance of the protocol hierarchy

Protocol layer	Alpha 21064A, 275 MHz				Pentium, 120 MHz			
	ParaStation		OS/Ethernet		ParaStation		OS/Ethernet	
	Latency (μs)	Band-width (MB/s)	Latency (μs)	Band-width (MB/s)	Latency (μs)	Band-width (MB/s)	Latency (μs)	Band-width (MB/s)
Hardware	1.24	10.5			0.87	15.6		
Rawdata	4.15	9.6			3.05	13.6		
Port	10.7	8.9			8.9	10.8		
Socket	11.4	8.8	283	0.99	9.2	10.7	159	1.08
P4	108	7.5	344	0.95				
PVM	129	6.7	539	0.84	102	7.7	388	0.86
Socket (self)	6.4	85	195	33	4.82	88	288	30

Alpha system is within 15% (30% on the PC system). The results justify our decision to maintain the rawdata port which is more than twice as fast in latency than regular ports and the loss of throughput drops to 8.5% (13% on the PC system) compared to the performance of the hardware abstraction layer. 4.15 μs (3.05 μs) latency of the rawdata port is even less than 4.5 μs (3.9 μs) for a null system call on the Alpha (PC).

The performance difference between the hardware abstraction layer (HAL) and the rawdata interface is due to the guarantee of mutual exclusion and correct interaction between competitive processes. Applications using the HAL as a communication interface assume exclusive access to the ParaStation network. Only one application per node is allowed to interface to the hardware at this level, so there is no need to regulate any interaction between processes. Typically the HAL is only used by the ParaStation library and not considered as a user programming interface. The rawdata interface too, is limited to one application per node, but ensures correct interaction between applications using upper layers such as ports or sockets. Thus, several critical code regions (interaction with the hardware while sending or receiving a message, updating global information, etc.) are locked by semaphores. Furthermore and in contrast to the HAL interface, the rawdata layer supports automatic fragmentation and reassembly of large messages (>500 bytes), receiving messages only from a particular node (and not just the next packet as at the HAL level), and it uses a larger protocol header than the HAL (8 vs. 4 bytes). All these operations are responsible for the additional latency at the rawdata level compared to the HAL.

The performance difference between the rawdata interface and upper layers (ports, sockets) is mostly due to the enhanced functionality of the port and socket layer. First of all, the upper layers provide a multiuser and multiprogramming environment. As a consequence, the protocol has to maintain the relationship between incoming messages and associated processes. This is done in a step of demultiplexing incoming messages within the library. To ensure mutual exclusion and correct interaction between competing processes, the library is forced to lock critical code regions at a finer granularity than at the rawdata level. This maintains fairness and good interaction possibilities

between multiple processes, but causes higher latencies. [4] Furthermore, the intermediate buffering of incoming messages in the common message pool is responsible for the performance degradation (see Section 4.2.2).

The real advantage of ParaStation becomes obvious when comparing its performance to that of regular operating system calls. ParaStation socket calls on the DEC Alpha are about 25 times faster in latency than the regular OS calls while offering the same services. Similar results are measured on the PC system where ParaStation is about 17 times faster in latency than equivalent operating system calls. Throughput, however, is not comparable because the ParaStation network is much faster than Ethernet. Even the relative loss in throughput is not comparable because it is much harder to interface to a fast network than to a slower one. We did not try to fill the empty areas of Table 3 – which would in fact be especially interesting – because our approach heavily relies on the superior functionality of the ParaStation hardware, which is not present within common network adapters such as (Fast) Ethernet, FDDI, and ATM.

Another interesting insight is the additional overhead caused by the programming environments, P4 and PVM. Within ParaStation on the Alpha system, these environments add an overhead of factor 9.5 (P4) and 11.3 (PVM) to the latency of our system layer. Even in the standard operating system environment, P4 adds about 21% and PVM 116% overhead. Similar results are measured on the PCs where PVM adds an overhead of factor 11 to the latency and an overhead of 144% to the regular operating system, respectively. PVM even decreases throughput when built on top of the ParaStation sockets by 24% on the Alpha system and 28% on the PC system. This shows that both packages are not well designed for high-speed networks.

Finally, we measured the performance of a socket-to-socket communication within a single process, where network hardware is not needed at all. This test aims to measure the protocol performance for local communication in the absence of process switching. Local communication on ParaStation is optimized and enqueues the sent message directly into the receive queue of the receiving socket. Thus, the presented 85 Mbyte/s (88 Mbyte/s on the PCs) reflects mainly the `memcopy` performance of the system. The TCP/IP implementation within both Digital Unix and Linux seem to optimize local communication because a throughput of 33 MBytes/s (30 MBytes/s on the PC) is achieved with this benchmark test.

5.4. Application performance

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. Our approach is twofold. First, we took a *heat diffusion* benchmark to test application performance on our proprietary interface. Second, we installed the widely used and publicly available ScaLAPACK [5] library [18], which first uses BLACS [6] [19] and then PVM as communication subsystem on ParaStation.

All ParaStation application benchmarks were executed on the Alpha 21064A (275 MHz) cluster.

The heat diffusion benchmark starts with an even temperature distribution on a square metal plate. On all four sides different heat sources and

[4] Semaphores are expensive, especially on the Alpha processor.

[5] *Scal*able *L*inear *A*lgebra *Pack*age.

[6] *B*asic *L*inear *A*lgebra *C*ommunication *S*ubroutines.

Table 4
Heat diffusion on ParaStation

Problem size (n)	1 workstation Runtime (ms/iter)	2 workstations		4 workstations		8 workstations	
		Runtime (ms/iter)	Speedup	Runtime (ms/iter)	Speedup	Runtime (ms/iter)	Speedup
64	1.5	0.99	1.51	0.9	1.66	2.0	0.75
128	6.0	3.5	1.71	2.3	2.61	3.4	1.77
256	22.3	12.0	1.86	7.5	2.97	7.0	3.19
512	89.2	46.7	1.91	26.4	3.38	17.2	5.19
1024	424	217	1.95	113	3.75	57.3	7.40

heat sinks are asserted. The goal is to compute the final heat distribution of the metal plate. This can easily be done with a Jacobi or Gauss–Seidel iteration by calculating the new temperature of each grid point as the average of its four neighbours.

Parallelizing this algorithm is simple: we use a block distribution of rows of the $n \times n$ matrix, so during each iteration, each process has to exchange two rows with its neighbouring processes. To visualize the progress, all data is periodically collected by one process. Table 4 shows the effective speedup for different problem sizes. Each experiment was measured with at least 5000 iterations, visualizing the result every 20 iterations.

As expected (see Table 4), the execution time on uniprocessor and multiprocessor configurations quadruples as problem size is doubled. This is obvious, because the asymptotic work of a Jacobi-iteration on a $n \times n$ matrix is $O(n^2)$. As shown, we achieve a reasonable speedup for relevant problem sizes on all configurations. Taking the last line as an example, the efficiency of two workstations is close to its maximum. In the four and eight processor configurations, we achieve an efficiency of 93.75% and 92.5%, respectively. The reason for the decreasing efficiency when using more workstations is due to visualizing the progress every 20 iterations, which is inherently sequential. In general, there are only two points where performance decreases when switching to the next larger configuration. But this only happens for problem sizes where parallelizing is doubtful.

The second application benchmark for ParaStation – *xslu* taken from ScaLAPACK – is an equation solver for dense systems. Numerical applications are usually built on top of standardized libraries, so using this library as benchmark is straightforward. Major goals within the development of ScaLAPACK [18] were efficiency (to run as fast as possible), scalability (as the problem size and number of processors grow), reliability (including error bounds), portability (across all important parallel machines), flexibility (so that users can construct new routines from well-designed parts), and ease of use. ScaLAPACK is available for several platforms, so presented results are directly comparable to other systems.

Table 5 confirms the scalability of performance while problem size as well as the number of processors increase. The efficiency of the two, four, and eight processor clusters are 94%, 87%, and 77%, respectively. It is remarkable that we get more than a GFLOP for the 8-processor cluster. These are real measured performance figures and not theoretically calculated numbers. The last line shows the performance one can get using ScaLAPACK configured with standard PVM (Ethernet). The best performance in this scenario

Table 5
ScaLAPACK on ParaStation

Problem size (n)	1 workstation		2 workstations		4 workstations		8 workstations	
	Time (s)	MFlop	Time (s)	MFlop	Time (s)	MFlop	Time (s)	MFlop
1000	5.0	134	3.36	199	2.95	226	2.74	244
2000	34.4	155	20.8	257	13.6	394	9.80	545
3000	109	165	62.3	289	39.2	459	27.9	647
4000			138	309	84.0	508	54.6	782
5000					152	547	96.4	865
6000					251	573	157	920
7000							234	978
8000							334	1022
Ethernet	$n=3000$	165	$n=4000$	232	$n=6000$	320	$n=8000$	261

is reached at a problem size of $n=6000$ on a 4-processor cluster. Using more processors results in a drastic performance loss due to bandwidth limitation on the Ethernet. For ParaStation, we see no limitation when scaling to larger configurations. It is even possible to further improve the ParaStation performance by optimizing the library hierarchy below ScaLAPACK (ScaLAPACK – BLACS – PVM – ParaStation sockets – Hardware).

In general, using various application codes such as digital image processing and finite element packages, we achieved relative speedups of 3–5 on ParaStation over regular PVM or P4 on our 4-node and 8-node ParaStation clusters. In all of these studies, we used the same object codes, just linking them with different libraries.

6. Conclusion and future work

The integrated and performance-oriented approach of designing fast interconnection hardware and a system library with a well-defined and well-known programming interface has lead to a workstation cluster environment that is well-suited for parallel processing. With low communication latencies, minimal protocol, and no operating system overhead, it is possible to build effective parallel systems using off-the-shelf workstations. While ParaStation is still a workstation cluster rather than a parallel system, presented performance results compare well to parallel systems. ParaStation's flexibility, scalability (from 2 to 100+ nodes), portability of applications (providing standard environments such as PVM, MPI, P4 and Unix sockets), and the performance level achieved have led us to market ParaStation. [7]

In future, we will work on next-generation hardware, ports to other platforms and support for various programming environments. Current issues for a new network design are fiber optic links, optimized packet switching, and flexible DMA engines to reach an application-to-application bandwidth of about 100 Mbyte/s. Second, due to the PCI-bus interface, the ParaStation system is not limited to Alpha or PC platforms. Currently, we are working on a port to Alpha

[7] For further information, see http://wwwipd.ira.uka.de/parastation or http://www.hitex.com/parastation.

machines running Linux, and Pentium PCs as well as Alpha workstations running Windows NT. Ports to other platforms (e.g., Sun/Solaris, IBM-PowerPC/AIX, SGI/IRIX) are possible, but not yet scheduled. Finally, we plan to support MPI as a future standard as well as PVM directly within the ParaStation system layer. This will give PVM and MPI applications a performance boost over a socket-based implementation. Besides MPI and PVM, Active Messages and Fast Messages, respectively are considered as additional interfaces to the system layer.

References

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

[2] R. Buttler, E. Lusk, User's Guide to the p4 Parallel Programmimg System (ANL-92/17), Argonne National Laboratory, October 1992.

[3] L. Clarke, I. Glendinning, R. Hempel, The MPI message passing interface standard, Technical report, March 94.

[4] T.E. Anderson, D.E. Culler, D.A. Patterson, A Case for NOW (Network of Workstations), IEEE Micro 15 (1) (1995) 54–64.

[5] C.G. Herter, T.M. Warschko, W.F. Tichy, M. Philippsen, Triton/1: A massively-parallel mixed-mode computer designed to support high level languages, Seventh International Parallel Processing Symposium, Proceedings of the Second Workshop on Heterogeneous Processing, Newport Beach, CA, April 13–16, 1993, pp. 65–70.

[6] R. Minnich, D. Burns, F. Hady, The memory-integrated network interface, IEEE Micro 15 (1) (1995) 11–20.

[7] M.A. Blumrich, C. Dubnicki, E.W. Felten, K. Li, M.R. Mesarina, Virtual-memory-mapped network interfaces, IEEE Micro 15 (1) (1995) 21–28.

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

[9] K. Omang, Performance results from SALMON, a cluster of workstations connected by SCI, Technical report 208, University of Oslo, Department of Informatics, November 1995.

[10] P. Ross, UNIX ™ clusters for technical computing, Technical report, Digital Equipment Coropration, December 1995.

[11] A.G. Nowatzyk, M.C. Browne, E.J. Kelly, M. Parkin, S-Connect: From networks of workstations to supercomputer performance, Proceedings of the 22nd International Symposium on Computer Architecture (ISCA), Santa Margherita Ligure, Italy, June 22–24, 1995, pp. 71–82.

[12] T. von Eicken, A. Basu, V. Buch, Low-latency communication over ATM networks using active messages, IEEE Micro 15 (1) (1995) 46–53.

[13] A. Basu, V. Buch, W. Vogels, T. von Eicken, U-Net: A user-level network interface for parallel and distributed computing, Proceedings of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, Colorado, December 3–6, 1995.

[14] S. Pakin, M. Lauria, A. Chien, High performance messaging on workstations: Illinois fast messages (FM) for Myrinet, Proceedings of the 1995 ACM/IEEE Supercomputing Conference, San Diego, California, December 3–8, 1995.

[15] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, W.-K. Su, Myrinet: A gigabit-per-second local area network, IEEE Micro 15 (1) (1995) 29–36.

[16] M. Philippsen, T.M. Warschko, W.F. Tichy, C.G. Herter, Project Triton: Towards improved programmability of parallel machines, 26th Hawaii International Conference on System Sciences, vol. I, Wailea, Maui, Hawaii, January 4–8, 1993, pp. 192–201.

[17] R.J. Harrison, Portable tools and applications for parallel computers, International Journal on Quantum Chemistry 40 (1991) 847–863.

[18] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, 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.

[19] J. Dongarra, 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.

Thomas M. Warschko is a Ph.D. student at the Department of Informatics at the University of Karlsruhe. He has been working in the Parallel Systems Group under Professor Walter F. Tichy since 1991. Currently he is heading the ParaStation project at the University of Karlsruhe. Prior to that he was a leading member of the Triton project and was involved in the design and development of the Triton/1 parallel computer. His main research interests are high-performance communication networks and high-speed communication protocols for cluster-computers, parallel and distributed computer architectures, parallel and distributed processing, performance evalutation, and latency hiding, latency tolerating and latency reducing techniques. Thomas received his diploma in informatics from the University of Karlsruhe, in 1990, and is going to receive his Ph.D. in 1997. He is a member of the GI and the ACM.

Walter F. Tichy is professor of Computer Science at the University of Karlsruhe in Germany. Previously, he was senior scientist at Carnegie Group, Inc., in Pittsburgh, PA, and associate professor of Computer Science at Purdue University in West Lafayette, IN. He has consulted widely for industry. His primary interests are Software Engineering and parallelism. He has obtained international recognition for his work in software configuration management and compiler technology. He is currently heading a research group involved in a number of projects. These projects include the study of software architectures, software configuration management, networks of workstations for high performance computing, optimizing compilers for parallel machines, and optical interconnects. Dr. Tichy received his B.S. in Mathematics from the Technical University Munich, Germany, in 1974, and his M.S. and Ph.D. in Computer Science from the Carnegie-Mellon University (PA), in 1976 and 1980, respectively. He is a member of the GI, the IEEE, the ACM, and Sigma Xi.

Joachim M. Blum is a Ph.D. student at the Department of Informatics at the University of Karlsruhe. He has been working in the Parallel Systems Group under Professor Walter F. Tichy since 1995. Currently he is a member of the ParaStation project at the University of Karlsruhe. His main research areas are high-speed communication protocols for cluster-computers, performance evaluation, distributed processing, and distributed operating systems. Joachim received his M.S. in Computer Science from the University of Massachusetts, Dartmouth, in 1995. He is a member of the GI.