The ParaStation2 Cluster Environment

Thomas M. Warschko and Joachim M. Blum
University of Karlsruhe, Dept. of Informatics*
Am Fasanengarten 5, 76128 Karlsruhe, Germany

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. But the key to effective and widespread cluster computing in general is to provide a cluster environment which offers high performance and well known programming interfaces as well as an integrated management and administration of the cluster that makes it suitable for a widespread community.

The ParaStation2 cluster environment fulfills these requirements. Its one-way latency for all programming interfaces is about $25\mu s$ *(depending on the hardware platform) and throughput is 100 to 150 MByte/s. It offers standard programming interfaces, including PVM, MPI, Unix sockets, Java sockets, and Java RMI which allow parallel applications to be ported to ParaStation2 with minimal effort. ParaStation's* one system image *is responsible for management and administration of the cluster in terms of fault tolerance (automatic removal and reintegration of faulty nodes), node management (offering logical nodes rather than physical machines), load balancing (automatic mapping of appropriate nodes to applications), and job control and disaster management in case of faulting applications.*

The system is implemented on a variety of platforms (Alpha workstations running Tru64 Unix, as well as Intel and Alpha's running Linux) and is proven to operate large clusters.

1 Introduction and Motivation

Workstation clusters coupled by high-speed interconnection networks offer a promising direction for high performance computing because they are cost-effective and they closely track technology progress. 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. To reduce this overhead, many researchers have proposed *user-space* communication models, which all remove the operating system and regular protocol processing from the communication path. Although being successful in terms of achieved performance (latency as well as throughput), the proposed communication models often use nonstandard programming interfaces and sometimes also nonstandard communication semantics. But application development relies on standardized and well defined programming interfaces such as Unix sockets or programming environments such as PVM and MPI, to ensure portability and maintainability. Thus providing these kind of interfaces is a key issue for a widespread use of high performance cluster systems.

Providing a global cluster environment is a second key issue, especially to handle large clusters. In order to provide clusters which are easy to use and easy to manage, it is necessary to develop a multi-user time-sharing environment with means for global (cluster wide) resource allocation that can respond to resource availability, distribute the workload and utilize the available resources efficiently and transparently. Maintaining a multi-user environment on top of a user-space communication subsystem requires specific mechanisms for process coordination such as co-scheduling of simultaneously communicating processes which is not present in ordinary operating systems. Furthermore, handling clusters in a convenient way needs mechanisms to provide partitioning, global process management, load balancing, node management, fault tolerance and disaster recovery.

This article presents the ParaStation2 architecture and related approaches (section 2), starting with an overview of the ParaStation2 system (section 3). The following sections discuss ParaStation in detail, first its communication subsystem (section 4), then its system environment (section 5) and finally its cluster environment (section 6) which forms

*now: Scarasoft AG, Mühlfelder Straße 10, 82211 Herrsching, Germany, Email: {warschko,blum}@scarasoft.com

ParaStation's *One System Image*.

2 Communication Subsystems and Cluster Environments

There are several systems providing a high performance communication subsystem for Myrinet. First of all GM from Myricom [13], Active Messages-II [8] used in the Berkeley NOW cluster [1], Fast Messages [15] from University of Illinois, the link-level flow control protocol (LFC) [4] used within the distributed ASCI supercomputer, PM [17] as part of the SCore system from Real World Computing Partnership in Japan, virtual memory mapped communication (VMMC-2) [9] from Princeton University, the basic interface for parallelism (BIP)[16] from the University of Lyon, Trapeze [20] from Duke University, and ParaStation2 from the University of Karlsruhe.

Most systems are based on the *user-space* communication principle to achieve high performance, although they support different communication paradigms, different programming interfaces, and a different quality of service. For example, most systems assume Myrinet to be reliable [4] and do not provide any mechanisms to ensure reliability, whereas GM, AM-II, VMMC-2, and ParaStation2 implement TCP-like transmission protocols at firmware level to guarantee reliable communication even in case of network failures (corrupted or lost packets). Besides proprietary programming interfaces which are closely related to the corresponding communication paradigm, nearly all systems provide an optimized MPI package as standardized programming interface. The standard Unix socket interface is supported by the GM system at kernel level (non optimized), by Trapeze at kernel level (optimized), by Berkeley NOW and VMMC-II (optimized, but limited functionality) and by ParaStation2 (optimized with full functionality and compatibility at object code level, see section 5).

Providing a high speed communication subsystem is a key issue for parallel computing. In addition to that, Glunix from the NOW project [10], SCore from RWCP in Japan [12], Mosix from the Hebrew University of Jerusalem [2] and ParaStation2 also care about global resource management and administration of clusters by providing a global cluster environment. While Glunix and SCore offer a set of commands to establish the cluster environment, ParaStation uses a combination of kernel extensions and daemon processes on each node of the cluster. Mosix focuses on global resource sharing, load balancing and process migration using kernel extensions to the BSD and Linux kernel, but neglects a high performance communication interface.

3 ParaStation2 Overview

The ParaStation system consists of several modules, located within or outside the unix kernel (see figure 1).

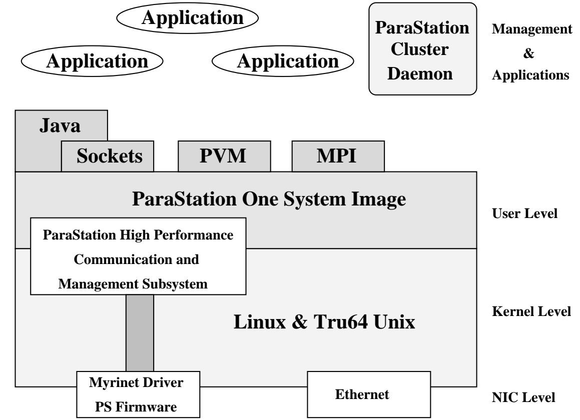

Figure 1. Overview of the ParaStation2 system

A LanAI program acting as firmware on the Myrinet adapter handles all communication between the Myrinet connected nodes. The device driver is responsible for setting up the Myrinet adapter at boot time and for mapping appropriate memory segments at application start time. Together with a thin software layer called HAL[1] these modules form the base of the communication subsystem (see section 4).

All programming interfaces (Unix sockets, PVM, MPI, Java sockets and RMI[2]) are implemented at user level, but rely on services such as process coordination and co-scheduling provided as kernel extensions located within the device driver. We took this approach to support a true multi-user environment on top of a user-space communication subsystem (see section 5).

The ParaStation cluster daemon is responsible to collect and distribute all information from all other nodes in the cluster to set up the global cluster environment. All services to handle participating or faulty nodes, to spawn, signal, and kill processes in a unique but cluster wide fashion, and to handle disaster recovery in case of application crashes are located here. All this features build up what we call a *one system image* (see section 6).

[1]hardware abstraction layer
[2]remote method invocation

4 ParaStation2 Communication Subsystem

ParaStation's communication subsystem is based upon the *user-space* communication principle, which effectively removes all protocol processing, data buffering and operating system overhead from the critical communication path. Unlike most systems which reduce protocol processing to a bare minimum, we've moved protocol processing to the Myrinet control program (MCP) running on the Myrinet adapter. Our protocol implementation called RDP (reliable data protocol) offers a reliable data transmission even in case of network failures such as corrupted or lost packets [19]. Although RDP is message oriented it uses implicit peer-to-peer connections with unique connection identifiers, which are automatically established upon the first transmission request to a destination host. During this process the source and destination node also negotiate unique sequence numbers. The sequence numbers are used to detect lost packets and the connection identifiers to detect when a connection is being reestablished (after a single side breakdown). Thus RDP is able to handle temporary failure or absence of a node gracefully and it will reestablish connection to that node automatically after it is online again. Flow control is implemented on top of a fixed sized transmission window using a combination of a piggybacked acknowledgment (ACK) in case of a bidirectional transmission and explicit but delayed ACKs in case of a unidirectional transmission. Buffer overflow at the receiving node is handled by sending back a negative acknowledgment (capacity NACK) which prevents the sender from transmitting further packets for a certain time. In case of detection of lost packets, a sequence NACK is transmitted back to the sender which results in an immediate retransmission of the missing and all subsequent packets (go back N strategy). In addition to the ACK/NACK scheme RDP associates a timer with each packet sent and upon timeout the packet will be automatically retransmitted. To detect corrupted packets RDP uses Myrinet's built in CRC check and some sanity checks such as minimal packet length and a match between the length parameter in the message header against the amount of data received. If an error is detected the packet is simply discarded, because it will be retransmitted by the sender anyway.

In cooperation with the HAL the MCP also handles all data transfer between the Myrinet SRAM and the host memory and vice versa. Here the emphasis is on providing data structures and mechanisms to minimize PCI-bus transmissions and interference of the host and the LanAI processor. Furthermore the HAL code implements all architecture and operating system dependent code as well as special improvements for each architecture. For all upper layers the HAL provides routines to initialize the communication subsystem as well as basic routines to send and receive messages. At this level a reliable in order delivery of packets is already guaranteed.

5 ParaStation2 System Environment

Nowadays most applications are parallelized using the MPI communication interface and therefore all high performance communication systems offer optimized MPI libraries. On the other hand there are several applications and application areas which still use other interfaces such as PVM, any other communication library based on the Unix socket interface or simply the socket interface itself. Therefore supporting only MPI results in a limited usability of the communication subsystem. In contrast to that ParaStation offers a wide range of standardized and well known programming interfaces (MPI, PVM, Unix sockets, and Java RMI) which are all optimized for the ParaStation communication subsystem. All interfaces are placed on top of a flexible protocol switch, which directly transfers incoming packets to protocol specific message handling routines. Choosing this strategy (see [7] for details) all communication interfaces perform nearly at the same speed. MPI, PVM and sockets only add approximately 2-5 μs of latency to the low level ParaStation HAL. This is possible due to the following reasons:

- The HAL in cooperation with the LANai firmware already provides a reliable data transmission. If the HAL accepts a message stream at the sending side it is guaranteed that this message stream arrives at its destination in order. The necessary protocol to ensure reliability is fully implemented in the firmware of the Myrinet adapter (see section 4). This approach allows upper layers to discard any flow control and reordering mechanisms from the critical communication path and thus providing the necessary quality of service at full speed.

- ParaStation's core library offers a flexible message queuing system, which allows upper communication layers to inspect or retrieve messages on request. The interface to the queuing system offers user defined filtering functions to inspect and retrieve specific messages in order to support message tag mechanisms (used both in PVM and MPI). These features enables upper layers (such as PVM or MPI) to discard their own queuing strategies, and to use the ParaStation supplied strategies instead. This approach avoids unnecessary copy operations leaving the data in system supplied buffers as long as possible to reduce overall processing overhead.

5.1 The ParaStation socket interface

All parallel and client-server applications on workstation clusters rely on the Unix socket interface as communication infrastructure. This fact lead us to provide a high performance socket interface to fully support any application to run at full speed on top of the ParaStation communication subsystem.

The first approach was to offer a semantically equivalent interface [18], where all ParaStation socket calls had a simple prefix to distinguish them from the original socket calls. This approach allows a ParaStation socket to switch back to the operating system socket transparently in case that the high speed network is not operational or in case that the requested destination is not reachable via the high speed network. As a consequence all setup calls had to be made by both ParaStation and the operating system since the decision who is the communication partner is not drawn at creation time of a socket but later on during `connect()` or even during a `sendto()`. A small `wrapper` between the application and the socket system calls decided which way to go: high speed through the optimized ParaStation protocol or low speed though the operating system. The disadvantage of this approach is that every application has to be adopted (by exchanging the socket calls) in order to use the ParaStation system.

The next step in optimizing the socket interface was to eliminate the prefix to offer a semantically and syntactically equivalent socket interface. From a technical point of view this can be done by simply overloading the system supplied socket calls (inside `libc.a`) with a second library that uses the same name space. Doing that one gains compatibility at object code level, but looses the transparently fall back mechanism to use the internal system calls when necessary or appropriate. The fall back mechanism is important for at least two reasons: First many applications use the `read()` and `write()` system calls to transfer data through sockets. But overloading `read` and `write` disables any file operations. Second, many applications do not operate in a closed environment in terms that they tend to communicate inside as well as outside a cluster. Not having the transparent fall back mechanism would disable outside communication immediately. To solve this problem we use the same mechanism used within the C library to convert a socket call to a generic system call. All socket calls usually consist only of three lines of code in the `libc.a`, which pack the parameters and switch into to operating system. Exactly this is done inside our new library, when we have to use the operating system sockets instead our own sockets. The resulting ParaStation socket library now offers compatibility at object code level while still providing a fall back mechanism in case a requested destination in not reachable within the cluster. This allows adoption of any application to the ParaStation system by simply linking their code against the ParaStation socket library while combining the advantages (high speed communication), the necessary flexibility (transparent in- and outside communication), and the convenience of not having to adopt any source code.

5.2 Optimizing PVM

The first implementation of PVM on ParaStation was based on the first version of socket layer described above. No modifications, except using the prefixed socket calls, was made to the original code [6]. The resulting PVM was much faster (latency dropped from 220 μs to about 90 μs) as a direct consequence of the improved socket performance (22μs versus 150 μs). Using the optimized socket layer, the latency caused by PVM itself raised from 46% to 318% thus evolving to the major bottleneck of the whole system.

The second step lead us to an optimization of PVM on top of the ParaStation ports [14] called PS-PVM, which fully used the flexibility of the ParaStation communication subsystem. Inside the cluster we used ParaStation's *One System Image* (POSI) (see section 6) to make PVM believe that it is running on a single system with multiple CPUs rather than a cluster of workstations. Furthermore PVM was adopted to use ParaStation's flexible queuing system to leave the messages in place until the user requests them. For outside communication, PS-PVM still uses the socket interface with its own message queuing implementation. The use of ParaStation *One System Image* allowed us to reduce the number of PVM daemons inside on cluster to a single daemon. This daemon is responsible for all tasks connected to him independent of the actual node they are running on. The ports interface reduced the work to be done inside PVM to a minimum, so that the PVM overhead for a message transfer could be reduced from about 70 μs on top of sockets to about 2 μs on top of ParaStation. This internal PVM optimizations and the optimization inside the ParaStation protocols reduced the overall message latency from 220 μs to as low as 25 μs, which is far less than PVM implementations on many dedicated parallel machines even with common shared memory.

5.3 Optimizing MPI

The MPICH distribution from Argonne National Lab used within the ParaStation system offers two separate possibilities to interface to different communication systems: First writing a so called channel interface and second to implement a new ADI (abstract device interface). We took the first approach simply because most of the required functions to set up a new channel interfaces are already present within the ParaStation communication library. Thus the ParaStation channel interface takes about 70 lines of code

mainly just mapping function names and converting parameters. Most code is used in the initialization section allowing a mpi program to use the ParaStation *spawn* mechanism in order to start the necessary processes on the different nodes within the cluster. As a consequence the usual `mpirun` command is not needed any longer and the application can be started by using the `-np` option as usual application parameter. As ParaStation is now responsible to start the tasks on the different nodes of the cluster, ParaStation is also capable to clean up the system in case of application crashes.

5.4 Supporting a multiprocess environment

A drawback when moving the communication path outside of the kernel is that the kernel now lacks of communication dependent information usually used within the process scheduling decision. The kernel now has no more knowledge if a thread is doing real work or if it is busy waiting on an empty receive message queue. But wasting CPU cycles is critical and reduces overall performance especially in a multiprocess environment, because the CPU could be better used by another thread to do real work. Therefore we have developed several coscheduling methods to hand-off the CPU to another thread. With coscheduling activated, executing two independent pairwise exchange benchmarks on the same set of CPUs is about 20 times faster than using the plain system without any coordination between the different tasks. This effect is extremely visible while using Java RMI, where several threads are waiting for a remote method to return and therefore actively consume CPU cycles without doing real work. With ParaStation coscheduling multithreaded applications run efficiently on a cluster of workstation with user-space communication protocols. For more details see [5].

6 ParaStation2 One System Image

Beside an efficient communication subsystem, a very important feature for the success of clusters in general is their ease of use and ease of programing which can be achieved by providing a unified view of the cluster as one single entity. As shown in section 5 ParaStation's *One System Image* (POSI) allowed us to optimize and simplify the adoption of both PVM and MPI.

The ParaStation One System Image is provided by the interaction between the ParaStation system library bound to each process, a cluster daemon running on each node and kernel extension inside the device driver.

6.1 Node management

The cluster daemons of all node stay in contact with all other cluster daemons in the system. Each daemon distributes local information such as the current load situation and currently connected applications to all other daemons. Additionally all daemons can request status information from remote nodes through a build in daemon-daemon protocol.

The first implementation of this daemon-daemon protocol was based on TCP connections between each node resulting in a separate connection to each other node. This approach caused more and more problems while increasing the number of nodes, but TCP connections were necessary to ensure a reliable communication and to detect remote system shutdowns. Now the new daemon-daemon protocol is based on our reliable datagramm protocol (RDP) which is also used inside the Myrinet adapter to ensure reliable communication. RDP allows us to switch a single UDP socket per node while still having a reliable data channel. Additionally, the traffic on the network was reduced by broadcasting the load information (also used as *alive* message) of the nodes with UDP multicast messages. This reduces the number of messages per node to send from N to 1 as well as the total number of messages from N^2 to N.

If a node fails, the other daemons detect this because the *alive* message of this daemon is missing for a specific period of time. If this happens the daemon declares all tasks running on this nodes as dead and takes appropriate action (see notification mechanism below).

The communication infrastructure provides by the daemon-daemon protocol enables us to expand the local view of a single node to a global, cluster-wide coordination of the connected client processes. Parts of the global coordination are global process management, partitioning, load balancing and output redirection.

6.2 Process management

The global process management in POSI is formed by providing a unique global task identifier for each (parallel) process. The global task identifier consists of the unix process identifier (PID) and the logical node number the task is running on. Any operation inside the POSI on tasks use this task identifier to address the task. Sending signals in Unix is limited to the local node. POSI expands the delivery of signals to any task in the cluster. Inside ParaStation signals are sent to the global task identifier. This task identifier is used to address the node where the task resides and a request is sent to the daemon on the node. When the request arrives the daemon sends the signal locally to the final task. Any error codes, such as operation not permitted, are sent back to the calling task and it can handle the return code in the same manner as a local `kill()` command.

Furthermore POSI introduces a new and very powerful mechanism, which notifies a task if another task changes its status. Any task in the system can register at the POSI to be

notified as soon as the status of another tasks changes. This helps the registering process to know when any task, e.g. the parent task, dies and may react in a proper manner. PS-PVM uses this feature to expand the responsibility of the single PVM-Daemon to the whole cluster. The clients register to be notified as soon as the daemon dies. This mechanism allows to clean up the virtual machine even if the coordinating PVM daemon is not present any longer. The same mechanism is used inside MPI, where all processes register to be notified as soon as a cooperating task dies. It helps to clean up all processes in a parallel MPI application even if a process gets a segmentation fault.

POSI allows a task to spawn new task dynamically on any node transparently. As a result of spawning the calling task gets back the global task identifier with which it can use to send signals to or retrieve information about the new task.

6.3 Partitioning

It is often useful to partition a large cluster into smaller subsets. E.g. one part of the system is used for programming and another part is used for productive runs. POSI can be instructed to limit total number of available nodes to any specific subset. New tasks are only spawned inside this specific subset of allowed nodes. This allows a close cooperation with batch systems such as DQS [3] or PBS [11]. When launching new applications the batch system sets the subset of possible nodes and executes the master process, which then spawns its clients only inside this subset.

While spawning POSI allows the user to tell the system to use a specific node or to choose an appropriate node to spawn a new task. When no specific node is given, POSI sorts the available nodes in the active partition by load and spawns the requested number of processes on the least loaded nodes. This balances the load among all nodes in a partition.

6.4 I/O redirection

All spawned client processes send their output to the terminal of the master process. This allows even to use the `printf()` debugger in the parallel application. Due to the fact that the real parent process of the client is the daemon on the remote side, it inherits the IO channels of the daemon. To prevent this, the daemons redirect the IO channels to a logger process which is running on the node of the master process after forking and before changing to the new executable. The logger process is forked by the master process before the master sends its spawn request to the daemon and transmits the peer addresses of this logger with its spawn message. This technique then allows redirecting all client output to the master process.

7 Conclusion

In this paper we concentrated on the importance of an integrated communication and management subsystem for clusters. We presented available systems for Myrinet and focused on details of the ParaStation2 system, which provides all necessary features and offers a promising direction for high speed cluster computing.

ParaStation2 features a high performance communication system with a wide range of programming interfaces operating closely at hardware speed. E.g. latency of all interfaces is as low as 25 μs and throughput rise up to about 150 MB/s. Especially the highspeed sockets enable a wide range of applications to run more efficiently on a highspeed network such as Myrinet. The ParaStation One System Image enhances the handling of a cluster, because the whole system is viewed as one single entity. Programming environments such as MPI and PVM use this functionality to operate in an optimized fashion. Additionally the user does not have to be aware of temporarily inactive nodes, since node failures are automatically detected by the ParaStation system and new processes won't get spawned on them.

References

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

[2] A. Barak and O. La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Future Generation Computer Systems*, 13(4-5):361–372, Mar. 1998.

[3] E. Bellcastro, A. Dutkowski, W. Kaminski, M. Kowalewski, C. L. Mallamaci, S. Mezyk, T. Mostardi, F. P. Scrocco, W. Staniszkis, and G. Turco. An overview of the distributed query system DQS. In M. M. J.W. Schmidt, S. Ceri, editor, *Proceedings of the International Conference on Extending Database Technology (EDBT '88)*, volume 303 of *LNCS*, pages 170–189, Venice, Italy, Mar. 1988. Springer.

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

[5] J. Blum, T. Warschko, and W. Tichy. Coscheduling: Proze"swechselentschung in user-level kommunikationsbibliotheken. In *Proceedings of Cluster Computing Workshop, Karlsruhe*, March 1999.

[6] J. M. Blum, T. M. Warschko, and W. F. Tichy. PSPVM:Implementing PVM on a high-speed Interconnect for Workstation Clusters. In *Proc. of 3rd Euro PVM Users' Group Meeting*, Munich, Germany, Oct.7-9, 1996.

[7] J. M. Blum, T. M. Warschko, and W. F. Tichy. PULC: ParaStation User-Level Communication. Design and Overview. In J. Rolim, editor, *Parallel and Distributed Processing*, number 1388 in Lecture Notes in Computer Science, pages 498–509. Springer Verlag, March 1998.

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

[9] C. Dubnicki, A. Bilas, Y. Chen, S. N. Damianakis, and K. Li. VMMC-2: Efficient support for reliable, connection-oriented communication. Technical Report TR-573-98, Princeton University, Computer Science Department, Feb. 1998.

[10] D. P. Ghormley, D. Petrou, S. H. Rodrigues, A. M. Vahdat, and T. E. Anderson. GLUnix: A Global Layer Unix for a network of workstations. *Software Practice and Experience*, 28(9):929–961, July 1998.

[11] R. L. Henderson et al. Job Scheduling Under the Portable Batch System. In D. G. Feitelson and L. Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – IPPS'95 Workshop Proceedings*, volume 949 of *Lecture Notes in Computer Science*, pages 279–294, Santa Barbara, CA, Apr. 1995. Springer.

[12] A. Hori, H. Tezuka, and Y. Ishikawa. User-level Parallel Operating System for Clustered Commodity Computers. In *Proceedings of Cluster Computing Conference*, March 1997.

[13] Myricom Inc., Arcadia, California. *The GM Message Passing System*, 1999. http://www.myri.com/GM/doc/gm.pdf.

[14] P. Ohly, J. M. Blum, T. M. Warschko, and W. F. Tichy. PSPVM2 – PVM for ParaStation. In *Proceedings of First Workshop on Cluster Computing, Chemnitz*, Nov.6-7 1997.

[15] S. Pakin, M. Lauria, and A. 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.

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

[17] 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.

[18] T. M. Warschko, J. M. Blum, and W. F. Tichy. The ParaStation Project: Using Workstations as Building Blocks for Parallel Computing. In *Proceedings of the International Conference on Parallel and Distributed Processing, Techniques and Applications (PDPTA'96), New Horizons*, Sunnyvale, California, USA, August 9–11, 1996.

[19] T. M. Warschko, J. M. Blum, and W. F. Tichy. A Reliable Transmission Protocol for Myrinet. In *Proceedings of the 2nd Workshop on Cluster-Computing*, pages 135 – 144, Karlsruhe, Germany, March 25 - 27, 1999.

[20] 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, Aug. 1997.