Universität Karlsruhe
Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation

# PULC –

# ParaStations Kommunikationssystem

Studienarbeit

von

Joachim Blum

Juli 1998

Betreuer: Dr. Thomas Warschko

Ich versichere hiermit, daß ich die vorliegende Studienarbeit ohne fremde Hilfe angefertigt habe.
Die verwendeten Quellen sind im Literaturverzeichnis vollständig aufgeführt.

Karlsruhe, den 3. November 1998

## Zusammenfassung

# PULC–
# ParaStations Kommunikationssystem

von

JOACHIM BLUM

*PULC ist die Kommunikationsschicht des ParaStation Systems. Es erlaubt Zugriff auf die Hardware aus dem Benutzerbereich einer Applikation und bietet somit eine sehr kurze Verzögerungszeit (Latenz) und einen hohen Datendurchsatz. Diese gute Kommunikationsleistung öffnet den Weg einen Verbund aus Arbeitsplatzrechnern als einen Parallelrechner zu benutzen. Als einziges System in der Welt, bietet es im Benutzerbereich eine nahezu vollständig geschützte Multiprozeßumgebung und viele standardisierte Schnittstellen. Hierbei wurden die Kommunikationsprotokolle der einzelnen Schnittstellen soweit optimiert, daß bei TCP Sockets Latenzen von ca. 10 µs erreicht werden. Das Design von PULC ermöglicht, daß auch benutzerdefinierte Protokolle effizient implementiert werden können. Dies wird erreicht, indem in PULC die Protokolle nicht als Stack sondern gleichberechtigt nebeneinander ausgeführt werden.*

*Neben den Kommunikationsprotokollen bietet PULC auch eine* Single System Semantic. *Hierbei wird der gesamte Rechnerverbund nur noch als ein Parallelrechner betrachtet. Innerhalb des Clusters ist ein ortstransparenter Zugriff auf die Prozesse möglich. Das Erzeugen von neuen Prozessen auf entfernten Knoten ist vollständig transparent. Zur Lastverteilung werden unter allen Knoten Lastinformationen ausgetauscht. Der Programmierer kann automatisch oder manuell auf den Knoten neue Prozesse erzeugen, die die geringste Last aufweisen. Größere Cluster können logisch in mehrere Partitionen unterteilt werden, die von einer Applikation alleine benutzt werden kann. Mit dieser* Single System Semantic *bietet PULC eine Abstraktion, die bisher nur von Cluster- und Parallelrechnerbetriebssystemen bekannt war. Durch die Kombination von der* Single System Semantic *und den optimierten Protokollen war es möglich die schnellste PVM Implementierung der Welt zu erstellen.*

*In dieser Arbeit wird das Design und die erste Implementierung des Systems auf der ParaStation Hardware beschrieben. Zur Evaluierung des Gesamtsystems werden verschiedene Kommunikationsbenchmarks und Standardapplikationen herangezogen. Es wird gezeigt, daß das Gesamtsystem mit gängigen dedizierten Parallelrechner in der Leistung konkurrieren kann. Der Preis eines solchen dedizierten Systems überschreitet aber das Mehrfache des Preises einer Clusterlösung. Somit werden Rechnerverbunde mit PULC ein ernstzunehmender Konkurrent zu den kleineren bis mittleren dedizierten Parallelrechnern.*

# Contents

**1 Introduction** **1**

**2 Related Work** **3**

**3 ParaStation Hardware** **4**

**4 Design of PULC** **5**
4.1 Addressing in Communication Subsystems . . . . . . . . . . . . . . 8
4.2 Resources provided by PULC . . . . . . . . . . . . . . . . . . . . . . 8
4.3 PSID: The PULC Coordinator . . . . . . . . . . . . . . . . . . . . . 9
4.4 The process of message transmission . . . . . . . . . . . . . . . . . 10
4.5 The PULC Message Handler . . . . . . . . . . . . . . . . . . . . . . 12
4.5.1 Sending messages . . . . . . . . . . . . . . . . . . . . . . . . 12
4.5.2 Receiving a message . . . . . . . . . . . . . . . . . . . . . . 12
4.5.3 Rawdata Protocol . . . . . . . . . . . . . . . . . . . . . . . . 13
4.5.4 Port-M/S/D Protocol . . . . . . . . . . . . . . . . . . . . . . 14
4.5.5 TCP/UDP Protocols . . . . . . . . . . . . . . . . . . . . . . 14
4.5.6 Other Protocols . . . . . . . . . . . . . . . . . . . . . . . . . 14
4.6 PULC Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
4.6.1 Rawdata Interface . . . . . . . . . . . . . . . . . . . . . . . 15
4.6.2 Port Interface . . . . . . . . . . . . . . . . . . . . . . . . . . 15
4.6.3 Socket Interface . . . . . . . . . . . . . . . . . . . . . . . . . 15
4.6.4 User Defined Interfaces . . . . . . . . . . . . . . . . . . . . . 16
4.6.5 Communication Libraries on Top of PULC . . . . . . . . . . 16
4.7 Scheduling Support in PULC . . . . . . . . . . . . . . . . . . . . . . 17

**5 Implementation** **17**
5.1 Flow Control . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

**6 Performance Evaluation** **19**
6.1 Communication Benchmark . . . . . . . . . . . . . . . . . . . . . . 19
6.2 Application Benchmark . . . . . . . . . . . . . . . . . . . . . . . . . 21
6.2.1 Linear Algebra Package on top of PVM . . . . . . . . . . . 21
6.2.2 NAS Parallel Benchmark on top of MPI . . . . . . . . . . . 22

**7 Conclusion** **24**

**8 Future Work** **24**

**Abstract**

PULC is a user-level communication library for workstation clusters. PULC provides a multi-user, multi-programming communication library for user-level communication on top of high-speed communication hardware. This paper describes the design of the communication subsystem, a first implementation on top of the ParaStation communication card, and benchmark results of this first implementation.

PULC removes the operating system from the communication path and offers a multi-process environment with user-space communication. Additionally, it moves some operating system functionality to the user-level to provide higher efficiency and flexibility. Message demultiplexing, protocol processing, hardware interfacing, and mutual exclusion of critical sections are all implemented in user-level. PULC offers the programmer multiple interfaces including TCP user-level sockets, MPI [CGH94], PVM [BDG+93], and Active Messages [CCHvE96]. Throughput and latency are close to the hardware performance (e.g., the TCP socket protocol has a latency of less than 9 $\mu s$).

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

# 1 Introduction

Common network protocols are designed for general purpose communication in a LAN/ WAN environment. These protocols reside in the kernel of an operating system and are built to interact with divers communication hardware. To handle this diversity, many standardised layers exist. Each layer offers an interface through which the other layers can access its services. This layered architecture is useful for supporting divers hardware but leads to high and inefficient protocol stacks. Protocols which are using standardised interfaces of the operating system are unaware of superior hardware functionality and often reimplement features in software even if the hardware already provides them. Another inefficiency is due to copy operations between kernel- and user-space and within the kernel itself. To transmit a message the kernel has to copy the data from or to user-space. The copying between protected address space boundaries often adds more latency than the physical transmission of a message. In addition, the kernel copies the data several times from one buffer to another while traversing layers of the protocol stack. On the positive side, the traditional communication path with the kernel as single point of access to the hardware ensures correct interaction with the hardware and mutual exclusion of competing processes.

For parallel computation on clusters of workstations, many of the protocols which are designed for wide area networks are too inefficient. Therefore, cluster computing must take new approaches.

The most promising technique is to move protocol processing to user-level. This technique opens up the opportunity to investigate optimised protocols for parallel processing. With user-level protocols there is no need to use the standardised interfaces between the operating system and the device driver. Thus, the reimplementation of services in software which are already provided by the hardware can be avoided.

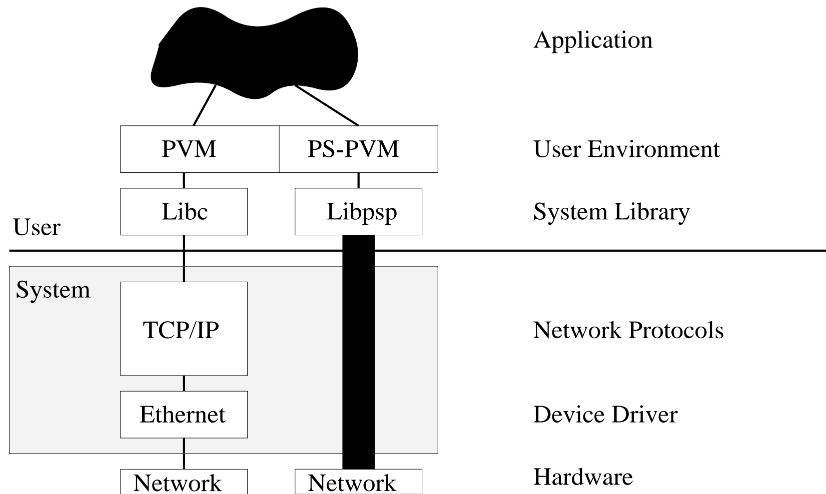


Figure 1: User-level communication highway

User-level communication removes the kernel from the critical path of data transmission. Figure 1 shows how user-level communication shortcuts the access to the communication hardware. High-performance communication protocols are based on superior hardware features to speed up communication. Copying data between kernel- and user-space is avoided and the implementation of true zero-copy protocols is possible. These key issues minimise latency and lead to high throughput.

But user-level communication has also its drawbacks, because now the single point of access to the communication hardware, namely the kernel, is missing. Therefore many user-level communication libraries restrict the number of processes on a node to a single process. Enabling multiple processes on one node in user-level raises difficulties, but also offers a lot of benefits. Once problems, such as demultiplexing of messages and ensuring correct interaction between multiple processes are solved, the high-speed communication network can be used similar to a cluster with regular communication channels such as Unix sockets.

The goal of PULC is to provide a multi-user, multi-programming communication library for user-level communication on top of a high-speed communication hardware. The first implementation of PULC uses the ParaStation communication adapter, which is described in section 3. Section 4 presents design alternatives and the optimisation techniques used. In section 5, this paper describes

the implementation of PULC on top of ParaStation. Performance figures for two different hardware platforms are presented in section 6. The last two sections present the conclusion and the plans for future work.

# 2 Related Work

There are several approaches targeting efficient parallel computing on workstation clusters. Some of them use custom hardware which support memory mapped communication. SHRIMP [DBDF97] builds a client server computing environment on top of a virtual shared memory. Similar to PULC, SHRIMP offers standardised interfaces such as Unix sockets. Digital's Memory Channel [FG97] is proprietary to DEC Alphas and uses address space mapping to transfer data from one process to another. On top of this low level mechanism Memory Channel offers MPI and PVM. Many recent parallel machines, e.g. IBM SP2, are a collection of regular workstations connected with a high speed interconnect.

Others use commodity hardware to implement communication subsystems. OSCAR (e.g. [JR97]) implements MPI on top of SCI cards. Fast Messages [CPL+97] and Active Messages [CCHvE96] are approaches for MPP systems ported to workstation clusters. Both offer low latency protocols which can be used to build other communication libraries on top. As an example the *Berkeley Fast Message* protocol [SR97] is build on top of Active Messages. Similar to PULC, it provides an object code compatible socket interface. It's latency is about 75 $\mu$s and it's throughput reaches 33 MByte/s on Myrinet. But in contrast to PULC it has some restrictions in the use of `fork()` and `exec()` calls. Differently from the current PULC implementation, it provides interoperability between Fast Socket and other applications on the same cluster whereas PULC only provides it for out-of-cluster communication.

BIP [PT97] and Myricom GM [myr] implement low level interfaces to the Myrinet hardware. They are comparable with the PULC hardware abstraction layer but lack on higher protocols. Gamma [CC97] builds Active Messages on top of Fast Ethernet cards and gets nearly full performance by adding a system call and building a special protocol in the Linux kernel. UNet [WBvE97] uses Fast Ethernet and ATM to build an abstraction of the network interface. Dependent on the hardware support, they use kernel or user-level communication. They've even built a memory management system to enable DMA transfer to previously unpinned pages. Such a memory management is not implemented in PULC, but could be done as soon as hardware with DMA transfer and on-board processors are used.

In the Berkeley NOW project [ACP95], GLUnix offers a transparent global view of a cluster. As in PULC the network of workstations can be used similar to a single parallel machine. Their main focus is on Active Messages and therefore no other protocols are implemented.

# 3 ParaStation Hardware

The first implementation of PULC uses the ParaStation high-speed communication card as communication hardware. This section gives a short overview of the ParaStation hardware. ParaStation is the reengineered MPP-network of Triton/1 [HWTP93], an MPP-system built at the University of Karlsruhe. Within a workstation cluster the ParaStation hardware is dedicated to parallel applications while the operating system continues to use standard hardware (e.g., Ethernet).

The network topology is based on a two-dimensional toroidal mesh. Table-based, self-routing packet switching transports data using virtual cut-through routing.

Every node (see figure 2) 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 decouples network operation from local processing. Packets contain the address of the target node, the number 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 are lost. Flow control is provided at link level and the unit of flow control is one packet. These features enable the software to use a simple fragmentation/defragmentation scheme.

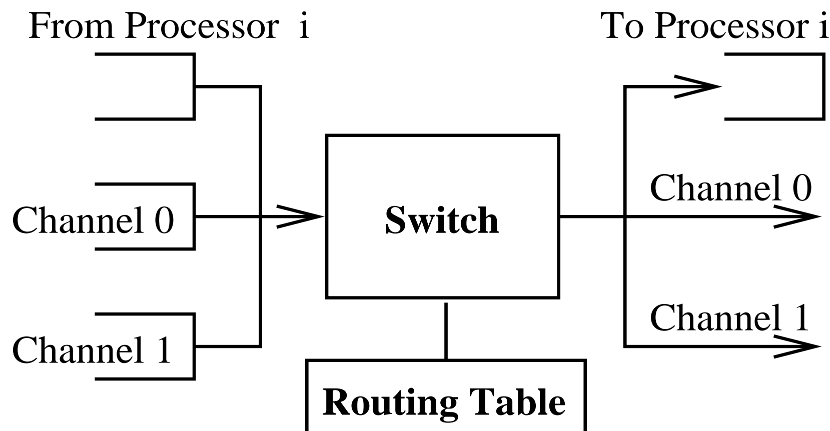


Figure 2: ParaStation Switch

The toroidal mesh provides a good scalability to more than 100 nodes. Actually, Triton/1 is running with 256 nodes with the same network processor and a De-Bruijn network. The communications processor used involves a routing delay of about 250$ns$ per node and offers a maximum throughput of 16 Mbyte/s per link. Additionally, the interface board provides a hardware mechanism for fast barrier synchronisation. To connect several systems, ParaStation uses 60-pin flat

cables, with standardised RS-422 differential signals. Using this technology, the maximum distance between two systems is 10m (about 30 feet).

The ParaStation hardware resides on an interface card which plugs into the PCI-bus of the host system. Thus, it is possible to use ParaStation on a wide range of machines from different vendors. A more detailed description of the hardware is given in [WBT97].

# 4 Design of PULC

A new communication subsystem has to fulfil several issues to be helpful for parallel computing. First, parallel computing is highly dependent on very low latency and high throughput. The performance available for the user has to be close to the hardware limits. Therefore, deep protocol stacks are deadly for parallel computing.

Second, communication hardware is getting faster and more intelligent. New approaches, such as DMA transfers and communication processors on the interface cards enable high performance and flexible protocol processing. A new communication protocol has to be well-suited for these technologies.

Third, communication libraries offer different interfaces and semantics to the programmer. Not each communication library is well-suited for all users of a cluster of workstations. Therefore, a new communication subsystem has to offer different interfaces (communication libraries). It should also be extensible for new approaches in this field.

Fourth, workstation clusters are often used by several people for parallel computing. Having user-level access to the hardware usually prohibits simultaneous use of one node by several processes. A new approach should support a multi-process environment.

Therefore the main goal was that PULC supports fine grained parallel programming on workstation clusters while still providing the benefits of multi-process environments.

The most challenging problem in a multi-process environment is the demultiplexing of incoming messages. Generally there are three possible places where message demultiplexing can take place:

1. In the operating system: The operating system either checks periodically the hardware for pending messages or it is interrupted by the hardware when a message has arrived. The operating system unpacks the message header and stores the message data in a corresponding queue in kernel space. From the viewpoint of the kernel it doesn't matter if the message is for the currently running process or for any other process.

2. In the communication processor: Each communicating process has a memory area which is accessible by the communication hardware. The communi-

cation processor checks the header and decides where the message fragment should be stored. The number of accessible memory areas is limited, however. To solve this problem the communication system can either limit the number of communicating processes or it buffers the message intermediately, where the processes can access the data (in kernel space, common message area, or a trusted process' address space).

3. In the low level communication software in user-space: A user process periodically checks the hardware (or gets interrupted), and receives the message. If the message is not addressed to the receiving process, the process stores the message in a message pool accessible by the destination process.

In all cases the destination process executes a receive call and gets the data from the intermediate storage and stores it into the final destination. If the final destination is known and accessible at the time of message demultiplexing, the message can be stored directly in this area. This is known as *true zero copy* message reception [BBVvE95].

PULC divides the message demultiplexing and the message reception in two different modules. The *PULC message handler* demultiplexes incoming messages. This message handler can either run on the communication processor or it can be linked to each user process. The *PULC interface* receives the message for the process. It always runs in the address space of the communicating process. Both modules communicate by calling each other or by updating queues in a shared message area.

Another challenging task is resource management. Resources (buffers, sockets, etc.) are usually managed by the operating system. When moving the communication out of the kernel, this task can be accomplished by a regular user process. The resource manager has to control access to the hardware and cleans up after application shutdowns. In PULC, this task is performed by the *PULC resource manager*.

Figure 3 gives an overview of the major parts of PULC.

**PULC Programming Interface:** This module acts as programming interface for any application. The design is not restricted to a particular interface definition such as Unix sockets. It is possible and reasonable to have several interfaces (or protocols) residing side by side, each accessible through its own API. Thus, different APIs and protocols can be implemented to support a different quality of service, ranging from standardised interfaces (i.e. TCP or UDP sockets), widely used programming environments (i.e. MPI or PVM), to specialised and proprietary APIs (ParaStation ports and a true zero copy protocol called Rawdata). All in all, the PULC interface is the programmer- visible interface to all implemented protocols.

**PULC Message Handler:** The message handler is responsible to handle all kind of (low level) data transfer, especially incoming and outgoing messages,

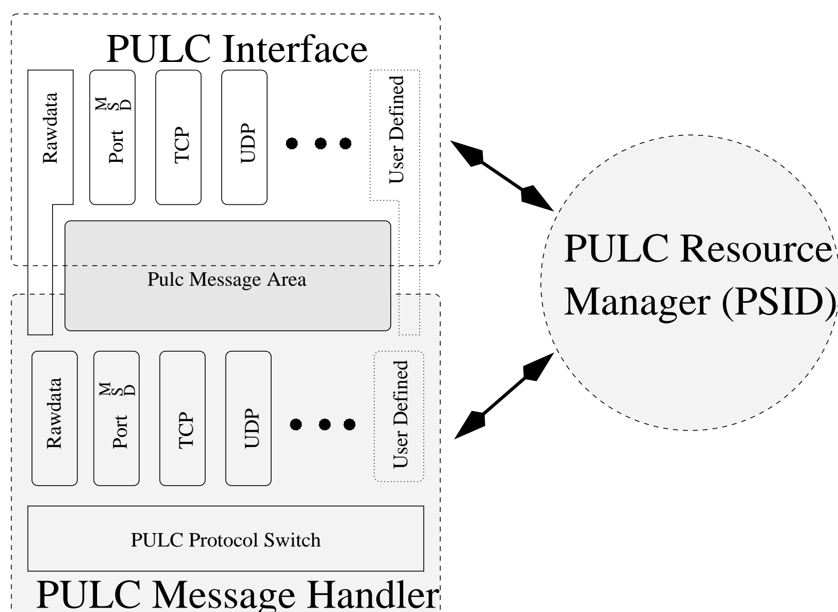


Figure 3: PULC Architecture

and is the only part to interact directly with the hardware. It consists of a protocol-independent part and a specific implementation for each protocol defined within PULC. The protocol-independent part is the *protocol switch* which dispatches incoming messages and demultiplexes them to protocol specific *receive handlers.* To get high-speed communication, the protocols have to be as lean as possible. Thus, PULC protocols are not layered on top of each other; they reside side by side. Sending a message avoids any intermediate buffering. After checking the data buffer, the sender directly transfers the data to the hardware. The specific protocols inside the message handler are responsible for the coding of the protocol header information.

**PULC Resource Manager:** This module is implemented as a Unix daemon process (PSID) and supervises allocated resources, cleans up after application shutdowns, and controls access to common resources. Thus, it takes care of tasks usually managed by the operating system.

To be portable among different hardware platforms and operating systems, PULC implements all hardware and operating system specific parts in a module called hardware abstraction layer (HAL). Choosing an interconnection network with different quality of services would force the adoption of the PULC message handler to these services the communication hardware provides. E.g. if the hardware doesn't provide in-order delivery, the message handler has use the PULC functions which provide a reordering of fragments.

## 4.1 Addressing in Communication Subsystems

There are a few addressing mechanisms for communication protocols. This section describes the key issues to distinguish them:

- Addressing a node: The simplest addressing scheme is node addressing. The sender just specifies the destination node and the receiver knows what to do with the received message. This addressing scheme is limited to one process per node since their is no way to decide a specific destination within the node. Often the sender specifies its own node number. This enables the application to sender specific handling of messages.

- Addressing a process: Many communication libraries, such as PVM, address a specific process. This enables multiple processes running on one node because they can be distinguished by their process id. The process id can be globally unique (often in combination with the node number) or could be specific for a node. In the second case the sender also has to specify the node of the destination process. With globally unique process numbers, it is possible that communicating processes are migrated during execution to other nodes.

- Addressing a communication endpoint: This is the most flexible way of addressing. A process addressing can easily be simulated. This mechanism is more powerful than the process addressing since a process can have multiple communication endpoints. Communication endpoint can also be shared among multiple processes and they can migrate from one process to another.

Most PULC predefined protocols use the *addressing to communication endpoints* and therefore they offer the most flexibility to the programmer. Only the rawdata protocol (see section 4.5.3) uses *node addressing*. This is acceptable since the rawdata protocol only allows on processes at the same time.

## 4.2 Resources provided by PULC

PULC supports the implementation of different protocols by offering a variety of resources together with associated interfaces to access them. The protocol independent resources are *message fragments*, *communication ports*, *semaphores*, and *process control blocks*. A *Message fragment* consists of a fragment control block and the message data and several fragments are concatenated to form a messages. Fragmentation is essential, because the underlying hardware has limited packet size. Therefore, PULC fragments have fixed sizes in memory and fragments are allocated as fixed sized memory blocks. This may waste memory, but allocating and managing variable sized junks of memory is time consuming. Several messages together form a message queue of a *port*. The port is the basic

addressable element in PULC communication. Different protocols use the ports as the channels to their communication partners. The resource manager frees a port and all fragments inside its message queue when no process is using it anymore. For the TCP/UDP protocol, another resource called *socket* is provided. A socket uses a port as its communication channel and stores additional socket specific information. To know about all the resources which are allocated by a specific process, PULC keeps information about a process in a *process control block*. These information are use to clean up the allocated resources when the process exits.

If the PULC message handler runs on the host processor, several processes can access common resources. To ensure mutual exclusion of processes to protect critical sections (manipulating queues or other resources), PULC provides user-level semaphores. Processor specific atomic operations, such as *test and set* or *load/store locked*, are used to implement them.

For an easy implementation of the protocols, PULC offers support functions to access the resources. E. g., PULC provides routines to store fragments into the message queue of a port. There are only three different strategies to store fragments in a message queue. PULC classifies the ports and the protocol calls its appropriate routine. In general, message queues of a port can be classified in the following way:

- Single stream: All fragments are stored in a single queue disregarding any message boundaries or message sources.
- Multiple Stream: All fragments of a the same source are stored in a queue. Fragments of different sources are stored in different queues.
- Datagrams: Fragments of different messages and different sources are stored in different queues. Each message has its own queue.

In addition to this classification these routines have to know if the hardware delivers the fragments of a message in order or if a reordering of the fragments is necessary. Fortunately the ParaStation hardware provides in-order delivery. The same holds for our HAL implementation for the Myrinet card.

## 4.3 PSID: The PULC Coordinator

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

The PSID also checks if the version used by the PULC interface and the PULC message handler are compatible. The version check makes corruption of data

impossible. The PSID can restrict the access to the communication subsystem to a specific user or a maximum number of processes. This enables the cluster to run in an optimised way, since multiple processes slow down application execution due to scheduling overhead.

All PSIDs are connected to each other. They exchange local information and transmit demands of local processes to the PSID of the destination node. With this cooperation, PULC offers a distributed resource management. The single system semantic of PULC is ensured by the PSIDs. They spawn and kill client processes on demand of other processes. PULC transfers remote spawning or killing requests to the PSID of the destination node. PULC uses operating system functionality to spawn and kill the processes on the local node. The spawned process runs with same user id as the spawning process. PULC redirect the output of spawned process on the terminal of the mother process. Therefore it offers a transparent view of the cluster.

The PSIDs periodically exchange load information. With this information PULC provides load balancing when spawning new tasks. There are several spawning strategies possible:

- Spawn a new task on the specified node: No selection is done by PULC. The spawn request is transfered to the remote PSID, which creates the new task. A new task identifier is returned in the result.

- Spawn a task on the next node: PULC keeps track of the node which was used to spawn the last task on. This strategy selects the next node by incrementing the node number.

- Spawn a task on a unloaded node: Before spawning, PULC orders the available nodes by their load. After that, PULC spawns on the nodes with the least heavy load.

These strategies allow a PULC cluster to run in a balanced fashion, while still allowing the programmer to specify the exact node, when problem solved requires a specific communication pattern.

The following subsection presents an example how message transfer works and how the PULC modules interact. After this, the paper proceeds with a more detailed view of the PULC modules, its functions, and its resources.

## 4.4 The process of message transmission

The following example (see figure 4) explains how PULC processes and transmits data. For explanation the well-known TCP socket protocol is used. Two major components, the *message handler* and the *programming interface*, are clearly separated. The programming interface as communication library is part of each calling process. The message handler could either reside within the communication

library (current implementation) or it could run on a communication processor within the communication adapter. Depending on the location of the message handler, the *message area* is either shared among all processes or each process has its own message area if a communication processor is capable of storing data to multiple areas in the host memory.

The sending process invokes a send call to transmit the data to a destination which is already connected to the associated socket. The TCP interface checks the parameters and invokes the TCP message handler. The message handler determines the destination address and sends the data – possibly in multiple fragments – by invoking the send routine of the HAL. The message data is always transferred to the underlying layer by pointers without copying data.

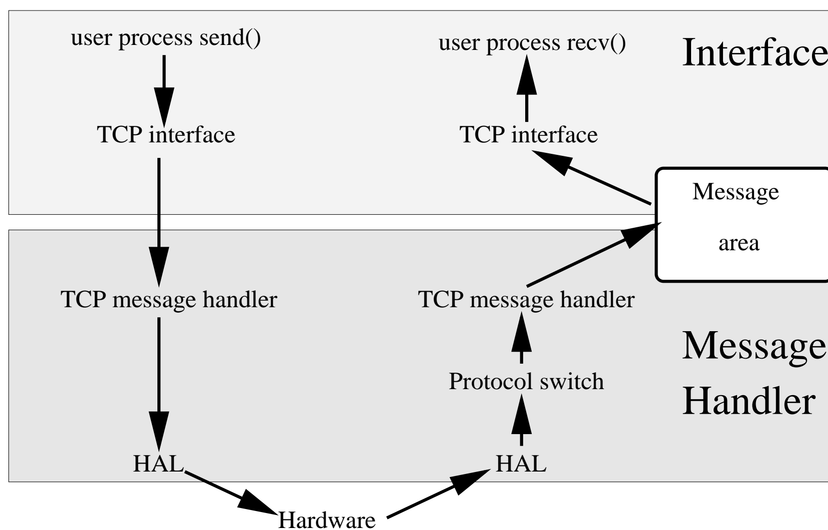


Figure 4: TCP message transmission in PULC

At the receiving node, the receive handler checks the HAL if any message is pending. When a message arrives, the receive handler determines inside the protocol switch which protocol handles the incoming message and transfers control to the specific receive handler. The TCP receive handler receives the body of the possibly fragmented message, determines the receiving port[1], and stores the fragment in the message queue associated with this port. The message queue itself is located in the message area. When the receiving process issues a receive call, the TCP interface checks the communication port of the associated socket for pending messages and delivers them to the application.

The port protocol acts in a similar way. Differences are due to new functionality which is not included in the socket standard. There are additional calls to view the whole cluster as one single virtual machine and to allow more flexible re-

[1]PULC's TCP implementation uses *single streams ports* 4.2 to store incoming messages.

ceives, sends, and selects. Due to these extensions, other communication systems, such as MPI, PVM and AM can be build very easily on top of this protocol.

The rawdata protocol has some differences to allow a true zero copy protocol implementation. The sending is equivalent to the TCP protocol. On the receiver side, the rawdata receive handler directly receives in a user supplied buffer, if the location of the buffer is accessible at the time of executing the receive handler.

## 4.5 The PULC Message Handler

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

### 4.5.1 Sending messages

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

### 4.5.2 Receiving a message

If the hardware supports a demultiplexing of messages, the PULC message handler runs on the communication processor of the hardware. It has some memory common with each receiving process. The data can directly be transferred to this memory area.

The first generation of the ParaStation card does not support any message demultiplexing at hardware level and so the PULC message handler has to be part of a process and runs in the address space of its host process. During reception of a message the PULC message handler can detect that it is not addressed to its own host process. It has to store the message in a commonly accessible message area (SHM) where the destination process can read the message.

Whether a message is received with true zero copy, or it is stored intermediately, depends on the used protocol.

The *PULC protocol switch* reads only the hardware header of the message and the protocol identifier. After decoding the id, the protocol switch directly transfers control to the receive handler of the protocol, which reads the rest of the message. This *header forwarding* is extremely fast and does not do any unnecessary copy of the data. The protocols can store the data directly in user data structures, as it is done in the rawdata protocol, or queue the data in the

a message queue (TCP,UDP,PORT-M/S/D). Other protocols can do it in their specific way.

PULC allows multiple processes to communicate concurrently since different processes can use different communication ports. The protocol interface and the protocol receive handler have to ensure the correct cooperation while receiving a message.

In a hardware-supported PULC message handler a shared port must reside in an area where both processes can access it. If both processes trust each other, the port can reside in a message area which is mapped in both processes. If they do not trust each other, the message handler has to protect the port in its own memory area. Both processes would have to access the message in the port through the message handler API. This is much slower than the solution with a direct access.

PULC includes many optimisations to speed up message reception. Some of these are *preallocated fragments*, *destination prediction*, and *end queueing.*

- *Preallocated fragments* guarantees that there is always an allocated fragment ready which the message handlers use to enqueue a message in a port. The receive handlers don't spend time to allocate a new fragments. Especially when resources get low, this mechanism guarantees that the receiving process does not have to be interrupted until a new fragment gets available.

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

- When queueing a fragment *end queueing* does the same technique inside a port. It knows which fragment was queued last and checks if the new fragment is related with the old one. If they are related, it shortcuts the queueing.

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

### 4.5.3 Rawdata Protocol

The rawdata protocol is a true zero copy protocol. If the receive handler knows the final destination of a message at the time of receiving, it directly transfers the data to this location. This kind of message transaction minimises latencies and maximises throughput. If the receive handler does not know or can not access the final destination, the fragments are placed in a queue of the *rawdata port.*

Since there is only one rawdata port on a node, only one application can use this protocol at a time. This is a restriction similar to many other user-level protocols (see section 2).

### 4.5.4 Port-M/S/D Protocol

The Port-M/S/D protocol is an example of using predefined PULC functionality. Other protocols (e. g., TCP) use this functionality but have to provide additional standardised functionality. PULC offers a framework for implementing them.

The Port protocol is just a frontend to the PULC functionality and it is implemented to create additional functionality which is not standardised in sockets. PULC offers dynamic process creation, group scheduling, and a testbed for new functionalities. The differences between Port-M (multiple stream), Port-S (single stream), and Port-D (datagrams) is just which queueing function the receive handler calls.

### 4.5.5 TCP/UDP Protocols

TCP and UDP protocols in PULC use ports as their communication channel. UDP and TCP have differences in connection establishment and they use different queueing strategies: TCP uses the single stream strategy whereas UDP uses the datagram strategy. In ParaStation, the data transfer is reliable and so UDP derives this property from the underlying hardware. If PULC were implemented on top of unreliable hardware, the TCP protocol implementation would have to guarantee reliability. Methods to guarantee reliability and in-order delivery are already supported by PULC and do not have to be implemented by the protocol.

### 4.5.6 Other Protocols

PULC is open for user defined protocols. New protocols plug into the protocol switch and the switch will redirect fragments to the new protocols. New protocols have to ensure correct locking to eliminate deadlocks.

They are on the same level as other PULC protocols. This means that they can be realized with the same efficiency as the predefined PULC protocols.

## 4.6 PULC Interface

Each protocol in the message handler can have its own interface. The interface is the counterpart of the message handler. The message handler receives a message and puts it in the message area whereas the interface functions get these messages as soon as they are received completely. The cooperation between the interface functions and the receive handler of the protocol includes correct locking of the port and its message queues. Correct interaction is necessary since PULC doesn't have control of the scheduling decisions of the Operating System. Thus the receive

handler could be in a critical section while the Operating System switches to a process which conflicts with this critical section. This could destroy consistency.

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

### 4.6.1 Rawdata Interface

The Rawdata interface enables the user to have true zero copy operations when receiving messages. The final place of the message must be accessible at the time of receiving. If the message handler is running in the address space of the rawdata process this is always true. If the message handler is running on the communication processor the final place must reside in the mapped message area. During receive, the rawdata interface first checks the rawdata port if any appropriate message is available. If no message is available, it registers the receive buffer at the rawdata receive handler, which places the incoming data into this buffer.

### 4.6.2 Port Interface

The interface is similar to the standard Unix socket interface, but has additionally functionality for dynamic process creation on remote nodes (spawning) and logging of the child processes. The ports are addressed by an index (descriptor) into the own private port table. A destination port is addressed by a port identifier which is a combination of the node number and the peer address of the port.

The message queues of the ports are either single streams, multiple streams, or datagram oriented as described in section 4.5.4.

### 4.6.3 Socket Interface

The socket interface to PULC is the same as for BSD Unix sockets. This interface allows easy porting of applications and libraries to the fast communication protocols. Destinations which are not reachable inside the PULC cluster are redirected to regular operating system calls.

This interface minimises costs when applications are ported to the PULC library. All communication in Unix is based on the socket interface. By providing a compatible interface, porting applications to PULC is just a relinking.

PULC sockets use specially tuned methods with caching of recently used structures. This allows an extremely fast communication with minimal protocol overhead. Each socket has a port as its communication channel. The socket receive handler only knows about the ports and uses different enqueuing strategies for UDP (datagram ports) and TCP sockets (single stream ports). The socket interface provides the interaction between the communication ports and the socket

descriptor. Sockets can be shared among different processes due to a `fork()` call and can be inherited by a `exec()` call. During `fork()`, the socket is duplicated but both sockets share the same communication port (the count attribute of the port is incremented). Thus, both processes have access to the message queue of the socket. After an `exec()` and a reconnection to PULC the sockets and the ports of the message area are inserted into the private socket and port descriptor tables. Therefore the process has access to these abstractions again.

#### 4.6.4 User Defined Interfaces

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

#### 4.6.5 Communication Libraries on Top of PULC

There are several communication libraries built on top of PULC. Most of them are just the standard Unix distributions on top of sockets. The applications which use these libraries just have to be linked with the PULC sockets. These libraries include P4 [BL92] and tcgmsg [Har91]. Others such as PVM [BDG+93] have been changed [BWT96] in a way that they can be used simultaneously to the standard sockets. This enables a direct comparison of the operating system communication and PULC. The implementation shows that PVM adds a significant overhead to the regular socket communication. This isn't obvious when PVM is used with regular sockets (see section 6). This lead to a new approach[OBWT97], which optimised PVM on top of the port-D interface. PULC already provides efficient and flexible buffer management and therefore this functionality could be eliminated in the PVM source. This PSPVM2 is still interoperable with other PVMs running on any other cluster or supercomputer. PSPVM2 views the whole PULC cluster as one single parallel system.

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

## 4.7 Scheduling Support in PULC

In a workstation cluster each node is controlled by its own operating system. All scheduling decisions are made locally. Usually no remote information is used to choose the process to be activated.

Since each node does scheduling decisions locally, communicating partners may not be scheduled at the same time. This decreases performance, because one process has to wait for the partner during synchronisation. Synchronisation can be implicit or explicit. A synchronisation is called implicit when the network interface is no longer accepting messages for sending or when there is no message ready to be received. A synchronisation is called explicit when both processes try to synchronise their execution by a synchronisation call.

When using a zero-copy protocol, implicit synchronisation is common. Message transmission is only decoupled by using the internal hardware buffers of the network. The sender blocks when the hardware buffers get full. The hardware buffer capacity is very small and so senders usually have to wait until a receiver on the other side is accepting the message.

In user-space communication, the operating system does not know whether the running process is actively polling for a message or if it is doing useful work. PULC solves these deficiencies by supporting two different scheduling strategies. *Passive Scheduling* (spin-block synchronisation) deactivates processes which are waiting for a communication or a lock. Other processes become active and can do their computation. Busy wait on locks and empty queues would decrease the performance of the whole system. When a process gets blocked on a busy lock, the process which holds the lock gets activated. *Active Scheduling* (gang scheduling) activates only the processes of one parallel application at a time. This approach puts more responsibility to the programmer of the application. The programmer is responsible that the time needed for the computational phase between two communication phases is the same for all communicating processes. The system places the processes in groups which work together on one problem. This is easy for distinct users, but can get very complicated when using sockets for communication with processes of other users. to solve this problem, hints from the programmers are needed.

# 5 Implementation

There exists two implementations of PULC, one for Intel-PCs running Linux and the other for DEC-Alpha workstations running Digital Unix. Both of them use the ParaStation high-speed communication card as communication hardware. As described in section 3, ParaStation offers many useful services to the software protocols, but unfortunately, it has no communication processor on board. Thus, the implementation uses a commonly accessible shared memory area (see figure 5)

to store messages and control information. The PULC library itself, in particular the PULC message handler, acts as the trusted base within the whole system. The library is statically linked to each application and ensures correct interaction between all parts of the system. The operating system is only invoked at system and application startup.

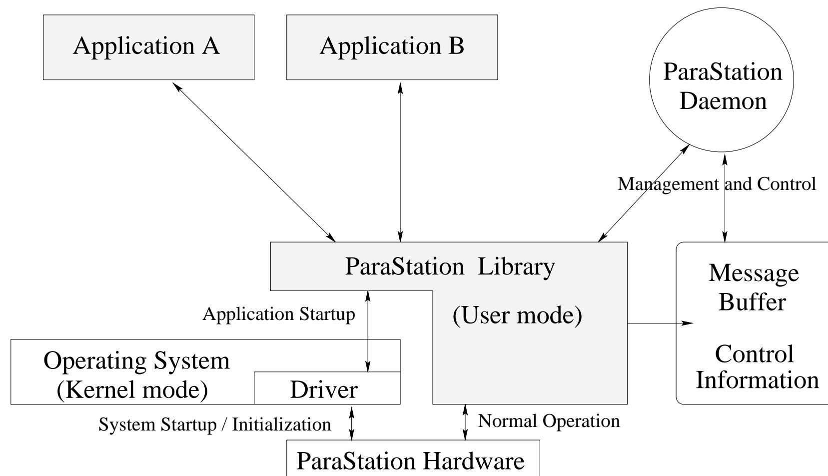


Figure 5: ParaStation User-Level Communication

Operating system and hardware specific parts of the library are placed in a separate module (the HAL). Therefore only this module has to be change when porting PULC to another platform. This is currently done for the Myrinet communication card.

Since the message handler is part of each process, the message area is mapped into each communicating process. This enables the message handler to receive messages for different processes and to demultiplex them to the correct receiving port. The multi-process ability of this solution is quite expensive due to the locking of ports, as well as locking data transmission to and from the hardware.

Using a commonly accessible message area suffers from a (minimal) lack of protection. The implemented message demultiplexing implies that all communicating processes trust each other. A malfunctioning process accessing the common message area directly is able to corrupt data owned by another process and can possibly crash the system. But the risk is minimal since the address space of Alpha processors (64 bit addresses) is approximately $2^{52}$ times larger than the size of the message area (configuration dependent). If a wrong address is produced once a second, a corruption of data in the message area could happen approximately every $2^{27}$ years. On the other hand, the trusted system is open for malicious hackers with access to the cluster, but this is a tolerable disadvantage when compared to the performance benefits gained from this policy. If this lack of protection is considered harmful, PULC can be configured to allow only

a specific number of process or only a specific user access to the communication system concurrently.

## 5.1 Flow Control

There are two levels of flow control implemented in the first version of PULC. First, there is the flow control adopted from the hardware. No packets are lost. The hardware checks for errors in the packets and guarantees the delivery, but the software has to protect against buffer overflow. To avoid bombarding a node with many messages while no process at this node is reading them, the receive handler checks if there is enough space in the message area. If a high water mark is reached (about 90% of the message area), the receive handler doesn't accept additional messages and already received fragments are swapped out to user-space. For each message one fragment is kept in the shared memory. This doesn't prevent the system from getting filled up again. If there are no more possibilities to swap out more fragments, the fragments stay in the hardware buffers. The the system blocks back to the sender, which stops its sending until new buffers are available again. The ParaStation hardware ensures that no packets are lost during back blocking.

# 6 Performance Evaluation

This section shows the efficiency of the PULC implementation. The performance of the different protocols is presented and the results are explained. Performance is measured on a cluster where each node in the cluster is a fully configured workstation.

## 6.1 Communication Benchmark

Communication subsystems can be compared by evaluating the latency and throughput of the systems. PULC offers several interfaces and runs on several hardware/operating system environments. Our test clusters consist of two Pentium PCs (166MHz) running Linux 2.0 and two Alphas 21164 (500MHz) running Digital Unix 4.0.

PULC's results are compared with the operating system performance whenever possible. The test consists of a pairwise exchange program to measure the throughput and a ping-pong test to measure the latency, which is calculated by the round trip time divided by two.

In the exchange program(see figure 6) both processes send a message to the other and wait for the receive of the other. Therefore both processes execute always the same command.

```
/* Sender code */                    /* Receiver Code */
StartTimer()                         StartTimer()
for(i=0;i<LOOPS;i++)                 for(i=0;i<LOOPS;i++)
   SendMessage(buffer,size)             SendMessage(buffer,size)
   ReceiveMessage(buffer,size)          ReceiveMessage(buffer,size)
end                                  end
EndTimer()                           EndTimer()
```

Figure 6: Pairwise exchange test program

```
/* Sender code */                    /* Receiver Code */
StartTimer()                         StartTimer()
for(i=0;i<LOOPS;i++)                 for(i=0;i<LOOPS;i++)
   SendMessage(buffer,size)             ReceiveMessage(buffer,size)
   ReceiveMessage(buffer,size)          SendMessage(buffer,size)
end                                  end
EndTimer()                           EndTimer()
```

Figure 7: Pingpong test program

In the ping-pong test(see figure 7), one process sends and the other receives, after receiving the message, the receiver sends the message back to the sender.

Surprisingly, the slower Pentium system performs better than the Alpha system in both latency and throughput at lower layers (see Table 8). This is due to the architectural differences between the two systems. In particular Alpha's capability to combine writes to the same memory location requires additional synchronisation. As the ParaStation communication interface is implemented as a FIFO buffer, memory barrier instructions (MB) are inserted after each write to the FIFO. The MB instruction itself waits for all outstanding read and write operations and thus limits the performance. In addition to the write combining bottleneck, the semaphore mechanism which is used in the Alphas is not as fast as the semaphores on the Pentium. A lock operation on the Alphas takes about 1 $\mu$s whereas a Pentium provides mutual exclusion within 200 ns. The semaphore bottleneck is also visible in multi-process protocols .

The line titled *hardware* in the table above shows the performance of the hardware abstraction layer described in section 5 and reflects the maximum performance one can get using ParaStation on the stated workstation.

The additional latency of 0.9 $\mu$s on the Alpha (3 $\mu$s on the Pentium) introduced by the rawdata protocol is due to guarantee mutual exclusion and correct interaction between concurrent processes. Multiple ports are addressed by the port protocol. This multi-programming environment adds additional 3.8 $\mu$s (8.2

| protocol-layer | Alpha 21164, 500 MHz ParaStation latency [$\mu s$] | bandwidth [MB/s] | Alpha 21164, 500 MHz OS/Ethernet latency [$\mu s$] | bandwidth [MB/s] | Pentium, 166 MHz ParaStation latency [$\mu s$] | bandwidth [MB/s] | Pentium, 166 MHz OS/Ethernet latency [$\mu s$] | bandwidth [MB/s] |
|---|---|---|---|---|---|---|---|---|
| hardware | 4.2 | 12.4 | | | 3.4 | 15.6 | | |
| rawdata | 5.1 | 11.9 | | | 6.4 | 14.8 | | |
| port-M | 8.9 | 9.5 | | | 14.6 | 11.8 | | |
| socket | 9.0 | 9.6 | 115 | 1.1 | 13.9 | 11.9 | 308 | 1.0 |
| PVM | 78.0 | 8.7 | 289 | 1.0 | 158 | 7.8 | 776 | 0.8 |
| PVM (port-M) | 11.5 | 9.4 | | | 27.2 | 11.5 | | |
| socket (self) | 3.2 | 318.8 | 390 | 33.0 | 19.0 | 107.0 | 578 | 30.0 |

Figure 8: Communication Performance of the PULC system

$\mu$s on Pentium) to the rawdata protocol. Providing full TCP socket functionality within 9 $\mu$s opens up a wide range of fine grained parallel programs on top of sockets. As reported in [BWT96] standard programming environments, such as PVM, add a huge amount of latency to the sockets. This is not noticeable when slow operating system sockets are used. When running PVM on top of PULC sockets 89 % (91% on a Pentium) of the latency is caused by these packages. These numbers show that these standard environments do not adopt well to high speed protocols. This lead to an optimisation of PVM on top of ports. As reported, the port-M protocol already provides most of the functionality that PVM has to implement on top of sockets, e.g. a very inefficiently implemented buffer management. Using the whole functionality of PULC, PVM only adds 2.5 $\mu$s ( 13.4 $\mu$s on the Pentium) to the port-M protocol latency. This shows that even with standardised interfaces, PULC offers great performance.

## 6.2 Application Benchmark

A user doesn't focus on the pure message passing numbers. The more important fact is how the system behaves with real applications. This section presents performance measurements of the system in two different areas with two different communication libraries. First, the PVM implementation is measured with a widely used linear algebra package and second, the NAS parallel benchmark is used to compare the system to the Cray T3E, a dedicated parallel system.

### 6.2.1 Linear Algebra Package on top of PVM

This test uses a linear equation solver for dense systems, called *xslu*, which is part of ScaLAPack [CDD+95], a popular linear algebra package. ScaLAPack uses

BLACS as a communication interface to different communication libraries such as MPI or PVM. In this test PVM acts as the underlying subsystem. The test is run on up to 8 Alphas (500 MHz, 256 MB Ram, Digital Unix 4.0b) connected with the ParaStation hardware.

| **ScaLAPACK on 160 MBit ParaStation with PSPVM2** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Problem | 1 workstation | | 2 workstations | | 4 workstations | | 8 workstations | |
| size (n) | Speed up | MFlop | Speed up | MFlop | Speed up | MFlop | Speed up | MFlop |
| 3000 | 1 | 443 | 1.75 | 759 | 2.18 | 966 | 2.62 | 1161 |
| 4000 | | | 1.70 | 753 | 2.47 | 1093 | 3.23 | 1431 |
| 5000 | | | 1.85 | 821 | 2.68 | 1187 | 3.74 | 1656 |
| 6000 | | | | | 2.90 | 1285 | 4.04 | 1789 |
| 7000 | | | | | 3.11 | 1379 | 4.37 | 1939 |
| 8000 | | | | | | | 4.61 | 2044 |
| 9000 | | | | | | | 5.07 | 2247 |
| 10000 | | | | | | | 5.22 | 2312 |

The table shows that on top of ParaStation the application scales good in terms of problem size and number of processors. A maximum performance of 2.3 GFlops is achieved which compare quite well to dedicated parallel machines. Unfortunately, *xslu* depends on high bandwidth and thus ParaStation with about 10MByte/s throughput is the real bottleneck.

### 6.2.2 NAS Parallel Benchmark on top of MPI

The second test measures the performance of the system with the NAS Parallel Benchmark suite. This suite is widely used to compare different parallel platforms. It is based on top of MPI and it runs without any source code modifications.

Some tests require a power of two number and others a square number of processors. Therefore not all columns are filled in each test.

The FT benchmark is a 3-D FFT application. MG is a multi grid benchmark. The LU benchmark does a matrix decomposition. It is the only benchmark in the NPB 2.0 suite that sends large numbers of very small (40 byte) messages. Therefore it shows the performance of the communication subsystem for fine-grained applications. EP (embarrassing parallel) usually shows the performance of a single node. The communication subsystem is not used frequently. IS (integer sort) sorts a number of integers in parallel. CG (conjugate gradient) and IS exchange a lot of data in huge data junks. All of these codes require a power-of-two number of processors. The SP(pentadiagonal solver) and BT (block diagonal solver) algorithms are more coarse grained implementations. They solve three sets of uncoupled systems of equation using multi partition schemes. Both the SP and BT codes require a square number of processors[2].

[2] For a detailed description of the tests please refer to

As a comparison the numbers achieved by a Cray T3E-900 are presented, which has similar processors per node. The communication subsystem is a highly optimised three dimensional torus. The third level cache is eliminated and therefore tests which are memory intensive run good on the T3E and test which can mostly run in the cache perform worse than on regular workstations.

The Cray T3E provides a bandwidth of about 300 MB/s and a latency of about 2 $\mu$s at hardware level . Therefore one could expect a comparable performance for tests which do not depend on bandwidth.

**NAS Parallel Benchmark on ParaStation and T3E**

| Test on no. of nodes Class A | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| BT ParaStation | n/a | n/a | 144.4 | n/a |
| BT Cray T3E-900 | n/a | n/a | 226.7 | n/a |
| CG ParaStation | 19.7 | 44.5 | 55.15 | 75.72 |
| CG Cray T3E-900 | | 46.5 | 86.0 | 241.4 |
| EP ParaStation | 4 | | | 31.96 |
| EP Cray T3E-900 | | 5.2 | 10.4 | 20.8 |
| IS ParaStation | 1.46 | 2.23 | 2.15 | 3.72 |
| IS Cray T3E-900 | | 6.6 | 12.9 | 22.1 |
| LU ParaStation | | | | 579.48 |
| LU Cray T3E-900 | | 134.4 | 270.4 | 531.1 |
| MG ParaStation | | | | 299.77 |
| MG Cray T3E-900 | | 171.5 | 313.9 | 720.8 |
| FT ParaStation | | | | 86.02 |
| FT Cray T3E-900 | | 85.3 | 169.5 | 330.4 |
| SP ParaStation | | n/a | 106.55 | |
| SP Cray T3E-900 | n/a | n/a | 172.4 | n/a |

The table shows the results measured on ParaStation and the results taken from the NAS homepage for the T3E. Higher numbers mean better performance.

In some test ParaStation behaves very good compared to the expensive dedicated system. Unsurprisingly, these are the test with minimal communication (EP) and the test with many small messages (LU), because the MPI latency is about the same on both systems.

During the other tests, which depend on high throughput, the ParaStation system began to swap received messages to user-space due to an overflow of the message storage. This effect limited the performance and a new version PULC will optimise this swapping. But even with this swapping effect, the resulting numbers are even better than other much more expensive dedicated machines, such as the IBM SP/2, SGI Origin, and Cray T3D[3].

---

`http://science.nas.nasa.gov/Software/NPB/`

[3]See the performance numbers at `http://science.nas.nasa.gov/Software/NPB/NPB2Results/index.html`

# 7 Conclusion

PULC shows extremely good performance on all protocols. Many programs benefit from the high speed of the PULC library. PULC's design offers nearly the raw performance of high-speed communication cards to the user while still providing standardised interfaces. The design goal of a multi-user/multi-programming environment at full speed was reached. PULC is also easily adapted to new hardware and brings efficient parallel processing to workstations clusters. Presented performance results compare well with parallel systems. PULC is included in the ParaStation system, which was introduced into market in 1996[4] and is currently ported to the Myrinet communication adapter. First results show that TCP throughput will raise up to 60 MB/s while latency will increase to about 20 $\mu$s. These are first numbers, where the message handler still runs in low level software.

The pure user-level approach in the ParaStation system showed many drawbacks which could only be resolved by introducing some security holes and performance limitations. Especially the performance of multiple processes on one node is dependent on the coscheduling strategies used. Unfortunately, I couldn't find a coscheduling strategy which is good for multi-threaded and interprocess communication at the same time. More research has to be done in this area.

The user-level access to the hardware FIFOs of ParaStation raised difficulties when a process gets killed during sending or receiving a message. Partly sent or received messages had to be completed or fully received. Before introducing this crash protection enhancements, the system crashed very often after I had introduced cluster wide signalling.

For future hardware developments, hardware FIFOs which cause system crashes when they are accessed incorrectly should be avoided. New hardware should support the software by giving hints how many bytes are missing to complete the message or how many bytes still have to be received.

# 8 Future Work

In future, the ParaStation team will work on next-generation ParaStation hardware. Current issues for a new network design are fiber optic links, optimised packet switching, and flexible DMA engines to reach an application-to-application bandwidth of about 100 MByte/s. Similar to Myrinet the new hardware will be able to run the message handler on board. Therefore any security whole will be eliminated.

PULC and the full ParaStation environment is going to be ported to other systems with PCI bus (e.g., Sun/Solaris, IBM-PowerPC/AIX, SGI/IRIX). PULC itself will be ported to other communication hardware. Additional interfaces

[4]For further information, see `http://wwwipd.ira.uka.de/ParaStation`.

and protocols, such as Active Messages, are considered to be implemented as protocols inside of PULC. This would give them a performance boost over the current implementation which are implemented on top of sockets or ports.

Furthermore, the analysis of the message demultiplexing showed that this task can be done in the OS, the communication hardware, and the low level software. All three cases will be implemented and evaluated.

# References


[ACP95] Thomas E. Anderson, David E. Culler, and David A. Patterson. A Case for NOW (Network of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[BBVvE95] Anindya Basu, Vineet Buch, Werner Vogels, and Thorsten von Eicken. U-net: A user-level network interface for parallel and distributed computing. In *Proc. of the 15th ACM Symposium on Operating Systems Principles, Copper Mountain, Colorado*, December 3-6, 1995.

[BDG+93] A. Beguelin, J. Dongarra, Al Geist, W. Jiang, R. Manchek, and V. Sunderam. *PVM 3 User's Guide and Reference Manual.* ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

[BL92] Ralph Buttler and Ewing Lusk. *User's Guide to the p4 Parallel Programmimg System.* ANL-92/17, Argonne National Laboratory, October 1992.

[BWT96] Joachim M. Blum, Thomas M. Warschko, and Walter 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.

[CC97] G. Chiola and G. Ciaccio. Gamma: a low-cost network of workstations based on active messages. In *5th EUROMICRO workshop on Parallel and Distributed Processing*, 1997.

[CCHvE96] Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, and Thorsten von Eicken. Low-Latency Communication on the IBM RISC System/6000 SP. In *ACM/IEEE Supercomputing '96, Pittsburgh, PA*, November 1996.

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

[CGH94] Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI Message Passing Interface Standard. Technical report, March 94.

[CPL+97] Chien, Pakin, Lauria, Buchanan, Hane, Giannini, and Prusakova. High Performance Virtual Machines (HPVM): Clusters with Supercomputing APIs and Performance. In *Eighth SIAM Conference on Parallel Processing for Scientific Computing (PP97)*, 1997.

[DBDF97] Stefanos N. Damianakis, Angelos Bilas, Cezarz Dubnicki, and Edward W. Felten. Client Server Computing on Shrimp. *IEEE Micro*, pages 8–17, January/February 1997.

[FG97] Marco Fillo and Richard B. Gillett. Architecture and implementation of memory channel 2. Technical report, Digital Equipment Coropration, 9 1997.

[Har91] R. J. Harrison. Portable tools and applications for parallel computers. *International Journal on Quantum Chem.*, 40:847–863, 1991.

[HWTP93] Christian G. Herter, Thomas M. Warschko, Walter F. Tichy, and Michael Philippsen. Triton/1: A massively-parallel mixed-mode computer designed to support high level languages. In *7th International Parallel Processing Symposium, Proc. of 2nd Workshop on Heterogeneous Processing*, pages 65–70, Newport Beach, CA, April 13–16, 1993.

[JR97] H. Jin and W. Rehm. Performance of message passing and shared memory on sci-based smp cluster. In *Proceedings of Fifth High Performance Computing Symposium, Atlanta, Georgia*, April 6-10 1997.

[myr] *The GM API.*

[OBWT97] Patrick Ohly, Joachim M. Blum, Thomas M. Warschko, and Walter F. Tichy. PSPVM2:PVM for ParaStation. In *Proc. of 1st Workshop on Cluster Computing*, Chemnitz, Germany, Nov.6-7, 1997.

[PT97] Loic Prylli and Bernard Tourancheau. New protocol design for high performance networking. Technical report, LIP-ENS Lyon, 69364 Lyon, France, 1997.

[SR97] David Culler Steve Rodrigues, Tom Anderson. High-performance local-area communication using fast sockets. In *USENIX '97*, 1997.

[WBT97] Thomas M. Warschko, Joachim M. Blum, and Walter F. Tichy. ParaStation: Efficient Parallel Computing by Clustering Workstations: Design and Evaluation. *Journal of Systems Architecture*, 1997. Elsevier Science Inc., New York, NY 10010. *To appear.*

[WBvE97] Matt Welsh, Anindya Basu, and Thorsten von Eicken. ATM and Fast Ethernet Network Interfaces for user-level communication. In *roceedings of the Third International Symposium on High Performance Computer Architecture (HPCA), San Antonio*, 1997.