PARNASS: Porting Gigabit-LAN components to a workstation cluster

Michael Griebel and Gerhard Zumbusch[a]

[a]Institut für Angewandte Mathematik, Universität Bonn,
Wegelerstr. 6, D-53115 Bonn, Germany

We will report on a cluster of workstations at our department, called Parnass. It is based on different types of MIPS processor workstations and servers, connected by a Myrinet, a Gigabit per second switched LAN, and additionally a Fast Ethernet. We have ported some low level message passing libraries as well as MPI to the Myrinet. A comparison of the performance of various communication libraries on different networks will be presented.

1. INTRODUCTION.

The ever growing demand for high performance computing power led to the advent of medium grained parallel computers based on commodity components. It turned out to be too expensive to keep a dedicated high-end processor development alive, while, in the race for performance, mass-produced microprocessors were approaching dramatically. In fact, todays desktop computers contain many of the architectural features, which were characteristic for supercomputers some years ago. This has two consequences: Todays super-computers are parallel computers and they employ cheap, standard, yet very fast microprocessors.

Hence it seems to be simple to create a super-computer by clustering a large number of off-the-shelf desktop computers. The necessary software is available as variants of standard operating systems and standard communication libraries on top of this networked cluster of processors. However, the main difference between such a pile of computers and an expensive parallel super-computer is the interconnection between the processing nodes. A substantial part of the effort to develop such a super-computer is dedicated to the construction of a low latency, high bandwidth, scalable communication network between the processors.

Thus, in order to build a competitive cluster of workstations, it is essential to provide a competitive, high speed network. Standard LAN technology, such as switched Fast Ethernet and ATM Oc-3, while an improvement compared to the older Ethernet, must be considered as one to two order of magnitude too slow for this purpose. Generally, latencies are very high for standard LANs, while the mass-produced components are cheap. In small networks, the LAN performance scales well. However, standard LANs become either slow or expensive, if one tries to construct large networks of workstations using large LAN

switches.

One way to provide the necessary network performance is to use high-end, expensive equipment like ATM Oc-12, HIPPI, Fibre Channel, or the emerging standard of gigabit Ethernet. These components definitely are expensive and the question for the sustained performance, workstation to workstation, can often not be answered satisfactorily but depend on the specific workstation. This leaves non-standard high performance networks, which do not impose overhead of a protocol standard and are likely to achieve higher performance at lower cost.

There are propriety processor interconnect networks by several parallel computer vendors, which are available solely as complete parallel computers, like the network fabrics of the IBM SP-2, of the Cray T3E or of the SGI Origin 2000. Separate networks components are sold for virtual shared memory systems such as the Scalable Coherent Interface (SCI) [SCI93] and reflective memory/ memory channel by Encore [Enc97, Dig96]. Extensions of the I/O capabilities of the PCI bus in a network fashion are Tandem's ServerNet [Tan95], and the GigaStar system [Gig97]. Traditional message passing networks are the Heterogeneous InterConnect (HIC) [HIC95], the ParaStation system [WTH95], and the Myrinet [BCF+95]. In the following text we will concentrate on the Myrinet, because of the superior network performance and the availability of documentation and software drivers.

2. MYRINET.

The Myrinet system is a low cost, high speed network at local (LAN) or system area (SAN). At present, a set of programmable PCI-bus and S-bus network interface cards (NIC) and 8-port crossbar cut-through switches are available. The components are connected by copper cables at the raw bandwidth of 1280 Mbit/s full duplex. There are two different electric signaling levels and cable types, one for short in-cabinet connections (SAN) and one for longer LAN connections. The network topology (see figure 1) and the communication protocol are completely arbitrary.

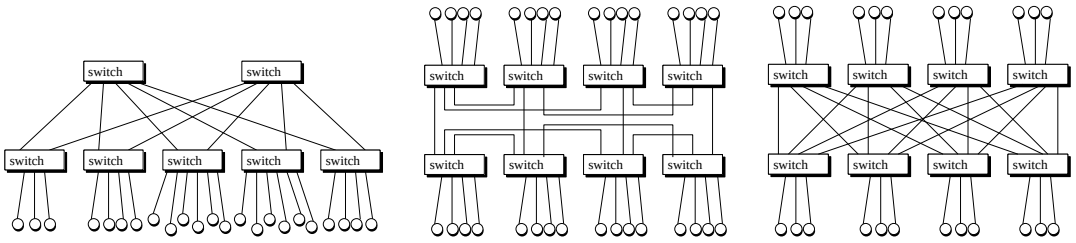

Figure 1. Some possible Myrinet topologies: A fat-tree, a (hyper-) cube, and a two-stage network.

The roots of Myrinet date back to the Caltech Mosaic massive multiprocessor project and the related Atomic LAN [FDCF94]. Today, Myrinet is being developed and sold by Myricom Inc. [BCF+95]. The hardware and software interfaces and protocols are published and open. Hence there are a number of projects to develop drivers and application software for Myrinet. However, there is no second source for the hardware yet.

2.1. Hardware.

A third-generation Myrinet adapter contains a custom RISC processor (LANai 4.x), static 128 kbytes - 1 Mbyte RAM (SRAM) for the program and for the data, a field programmable gate array (FPGA), a network interface with two net DMA engines, and a bus interface to the workstation with a DMA engine, see figure 2. The static RAM contains the control program for the processor and buffers for communication. The DMA engines transfer data from and to this RAM. The host can communicate with the NIC via memory mapped control registers of the NIC and the processor, and via memory access in the RAM. Usually some sort of handshake is performed to the RAM.

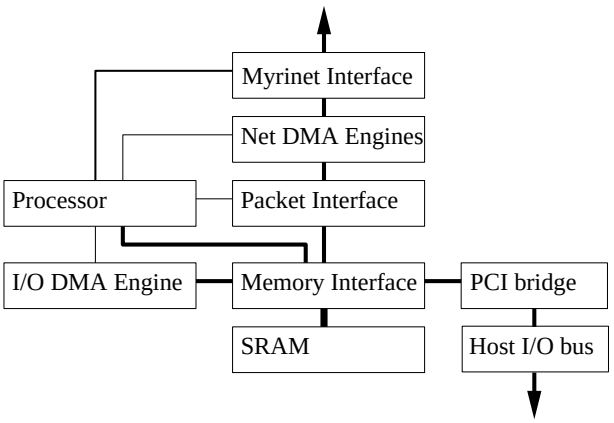

Figure 2. Layout of a Myrinet network interface card (NIC).

The NIC processor can read and write data directly to the LAN and can program the net DMAs to do so. For raw data transfer the net DMA is more efficient. At the same time the I/O DMA can transfer data from or to the host. This gives maximum performance due to pipelining. Of course message headers and administration will be handled by the NIC processor and the host processor. To support this, there is an additional checksum mechanism incorporated into the net DMAs. Usually the NIC processor does not touch most parts of the messages.

2.2. The Message Layer.

The main idea is to keep the hardware fast, cheap and flexible. The message protocol usually is handled by a freely programmable RISC processor on the NIC. Thus, there are many different protocols available and under development. Usually in connection with a specific protocol, one writes a software driver for the NIC processor, a device driver in the host operating system, and an application interface (API) for Myrinet components. TCP/IP drivers and several non-standard message passing protocols for Myrinet are available. There even have been attempts to run an ATM adaptation layer on the LAN, and there exist virtual shared memory implementations.

Messages can be of arbitrary length, defined by the software protocol in use. The topology of the network is completely arbitrary. Routing information is put into the header of a message by the sender, see figure 3. The switches implement hardware routing with header deletion. A switch interprets the first byte of routing information and removes this byte from the message. A subsequent switch will then read the next routing byte. At the tail of a message there is a checksum, which is checked and updated automatically by each switch.

Flow control is implemented in hardware for each link in a leaky bucket manner. In case the message route determined by the message header is in use already, the message is stalled by the flow control mechanism. There is no need for buffers to store whole messages in the switches, as would be for store-and-forward protocols. Messages are also stalled/ slowed down if lower speed (older) links are used.

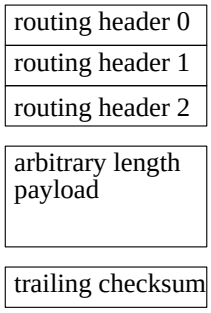

Figure 3. Structure of a Myrinet packet which is going to pass 3 switches.

Up to now there is no standard network equipment, such as routers or bridges, available to connect Myrinet to other types of networks. However, workstations can be used as routers, passing along TCP/IP traffic to a Myrinet. Furthermore, there are projects to specify WAN connections between different Myrinets [CLS+97].

2.3. Two-level Multicomputer.

Myrinet is intended as local area network or system area network for the construction of parallel computers based on message passing. The resulting parallel computer is a two-level system: The NICs can be programmed to implement the message passing protocol and perform the data transfers, while the workstations serve as compute nodes attached to the NIC, see figure 4.

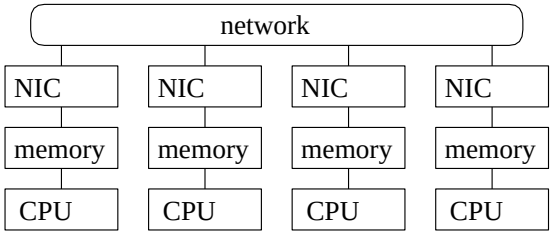

Figure 4. Structure of a two-level multicomputer.

There are multicomputers build of Myrinet components with different kinds of computing nodes. Computing nodes can be signal processors, FPGAs (e.g. for pattern recognition), VME single board computers, PCs and workstations. The strategy is to combine a high performance network by one vendor with high performance processors by other vendors, which suit the applications best. Instead of developing both components at once, one can participate in the advances of processing technology and of networking technology independently. Using mass-produced components gives the best overall price-performance ratio.

2.4. Programming Models.

Software is the key to efficiency combining a high performance network with high speed computing nodes. In former times of Ethernet, it did not really matter, how often messages had to be copied by the processor from source to destination. Hardware requirements for computer I/O bus performance were not that strict. However, with the advance of gigabit per second network speeds, which is to some extent comparable to main memory bandwidth, such considerations are important. Hence some software standards and layers become inefficient due to the demands for high performance. Some Linux PC based implementations of new protocols were indeed able to demonstrate over 1000 Mbit/s sustained bandwidth and latencies below 5μs on Myrinet [PT97].

Furthermore it is essential to provide efficient standard interfaces like MPI in order to re-use code and to port code from one platform to another. Thus, we need efficient implementations of message passing libraries, while some of the software layers to implement them become obsolete.

2.4.1. Message-Passing APIs.

There are many efforts to develop message passing libraries for the Myrinet. The original programming interface designed for the Myrinet is the API by Myricom [Myr96]. It provides a message passing semantic for single packet messages up to a maximum transfer unit (MTU) of 8132 bytes. Routing is done by the NIC processors, which dynamically determine the network topology. The API supports asynchronous communication with a limited amount of buffering. The receiving process has to provide enough and large enough empty receiving buffers in advance, which are returned filled in the case of any message received. If there are no receive buffers left or a buffer is too short, incoming messages are discarded. Hence the protocol is not reliable in case of overflow. Furthermore, the buffer memory has to be contiguous, non-swappable memory allocated by the kernel for implementation reasons. The amount of this memory is very limited. There is support for scatter and gather operations. Furthermore there are no operating system (OS) calls involved during communication, which could slow down the protocol. The API allows complete user access and control of the hardware. There is no support for multitasking, multi-processors or multi-protocol features. It monopolizes the NIC on the computer, resets it and reloads the control code for the NIC processor. Any other process, which accesses the NIC at the same time, may corrupt the data transfer. Such a process could also sniff into all messages processes by the NIC and it could reload the control program to access the memory of the other process. Hence it is not secure in a Unix environment.

An extension of the Myricom API is the message passing system 'GM' by Myricom [Myr97]. Sensitive parts of GM have been put into the kernel's Myrinet device driver. The system becomes secure and multitasking safe. The GM protocol is reliable with an acknowledge mechanism. Different processes can reserve a share of the buffer memory. The OS performs the necessary security checks. The buffer memory does not have to be contiguous memory as for the Myricom API, because page address translation is done by the OS. The routing is done by the host processors instead of the NIC processors, in order to save buffer memory on the NICs.

Another extension of the Myricom API is the message passing protocol 'PM' [THI96]. The send and receive buffers are still located in special contiguous memory. However, the buffers are managed by PM, rather than the user process. Packet routing is done by the host process with static routing tables. Furthermore multi-cast features have been added to the system. PM has been optimized further for performance. On top of PM there is a parallel C++ version and a parallel OS 'SCore' for administration and scheduling of parallel jobs available.

The highest network bandwidth has been achieved so far by the message passing system 'BIP' [PT97]. Several stages of optimization lead to this performance: Short messages are buffered and copied from user space to the NIC and reverse. Basically, an asynchronous semantic is implemented, but the messages can block, if the receiver buffer is full. However, send operations for long messages have a rendezvous semantic instead, where the receiver has to provide the receiving buffer in advance. Long messages are broken down into packets. The packets are transfered by DMA from user buffer to the NIC, from the NIC to the network, from the network to the receiving NIC, and from the NIC to the user

buffer. These four DMA operations are pipelined for maximum performance. The packet sizes are computed accordingly. Since a zero-copy protocol is employed for long messages and the messages are transferred into user memory by DMA, memory mapping is needed. This requires additional expensive OS calls to lock the appropriate memory pages and to translate memory addresses. In order to save time, these OS calls are performed only once and the memory pages remain locked for the lifetime of the process. BIP is not secure and there is no multitasking support for performance reasons. The system is highly optimized.

A reliable message passing system, which very much looks like a subset of MPI is 'BullDog' [MHH+97, HDMS97]. It offers an asynchronous message passing semantics for packets up to an MTU of 8192 bytes. Each packet can be sent as unordered or ordered, reliable (with acknowledge) or non-reliable packet. In the case of buffer overflow, reliable packets are resent by the sending NIC. No host interaction is necessary to implement these features. Routing is done by the host processor via a static routing table. Packets are copied by the host into and from the NIC. No DMA transfer is used yet. Hence any trouble with special memory and memory mapping OS calls is avoided. BullDog does not require OS calls for message passing. The system is not secure and is not multitasking aware, because the user process controls the NIC. The syntax is very similar to some MPI primitives.

The Trapeze message passing system [YCGL97] has been designed mainly for the fast transfer of memory pages. It is used in the implementation of a global memory management service, which extends an OS by a remote paging and a cooperative caching service. Such services may substitute full virtual shared memory support for a parallel application code. Trapeze is currently implemented for Digital Unix on DEC Alpha and for FreeBSD on PCs. Packets of arbitrary size up to 8 kbytes can be sent from one processor to another one. The Trapeze API uses rings of requests and packets. However, packets may be dropped because of overflow, which means that the protocol is not reliable. Additionally it is not secure, the user has full access to the Myrinet hardware. A major effort of the Trapeze development was dedicated to the performance for long packets. The DMA operations host to NIC and NIC to wire are interleaved, resulting in low latencies. Usually, one starts a DMA operation when the previous one has terminated. Initiating the DMA, while the other one is running, eliminates this kind of delay. On top of the Trapeze API, there is a zero-copy TCP/IP implementation available.

2.4.2. Active Message APIs.

Active messages can be considered as a more advanced type of message passing. The message is sent to the receiver in the usual way, including a tag. This tag identifies the receiving thread or process, which is invoked by the message to handle it. In contrast to message passing, there is no corresponding synchronous or asynchronous receive call. This concept is related to remote procedure calls (RPC) and one-sided communication.

'AM-II' is an active message implementation on Myrinet by [LMC97]. There are two different send modes: Short messages up to 32 bytes are buffered. They can be used for passing a few variables to the process. Long messages are allowed up to a length of 8 kbytes packets. Both short and long messages are queued in different queues in the NIC SRAM.

In order to avoid overflow, the queue buffers are paged to the host RAM as secondary memory. Memory transfers do not use DMA. The receiving process is identified by an integer tag, which has to be registered before. The current implementation supports shared memory communication on multiple processor machines connected via multiple NICs by Myrinet. Shared memory and Myrinet communication are transparent to the user. Active messages look the same between different processes on one machine (shared memory) and between processes on different machines (Myrinet). The implementation is not secure, since shared memory blocks and NIC SRAM are accessible to all processes.

The active message protocol 'fast messages' by [LC97] has been ported to several computer architectures. It offers asynchronous send operations. There are send operations for four byte messages and for longer messages. Long messages are broken down to packets. The receiving processor calls a small handler, which decides what to do with an incoming message. The system guarantees reliable and ordered message delivery and offers gather and scatter facilities. It serves as a base for several higher level protocols such as MPI and shared memory primitives.

2.4.3. Parallel Languages.

A high level approach to parallel programming is to use a programming language which includes parallel features rather than calls of a parallel library. Two dialects of standard programming languages for the Myrinet have been developed so far:

'Lyric' is an extension of Objective-C by Myricom [Bro96]. It offers a type of active messages between objects. One object can issue method calls of another remote object. Such a method can be the constructor of the object or some other user defined procedure. The runtime system of Lyric is based on a network of workstations connected by Myrinet. Ports to other environments are planed. A prototype implementation of Lyric is based on Gnu's gcc compiler and on the Myricom API.

Another project led to an extension of C++: 'MPC++' is a parallel C++ dialect [IHS+95], where dynamic syntax and semantics extensions are implemented in a preprocessor type fashion. Some parallel extensions, which have been implemented already, are mutual exclusion for critical regions in multi-threaded programming, active objects and global addresses. MPC++ is based on the message passing system PM.

2.4.4. TCP/IP.

In addition to provide programming interfaces for a networked cluster of computers, it is also necessary to support ordinary network services. There are several TCP/IP implementations for the Myrinet available. Usually TCP/IP is implemented in a kernel device driver, based on an existing message passing interface.

TCP/IP is available based on the Myricom API, GM, Trapeze, BIP, 'fast messages' and MPC++. It is of course secure and can be used by several processes at the same time, due to its implementation as a Unix kernel driver. Such a TCP/IP implementation can be used as a foundation for several other standard message passing interfaces, socket libraries etc. However, this introduces a lot of computational overhead, both by the OS and by the protocol stack of TCP/IP. Furthermore, some of the TCP/IP implementations cannot coexist with a message passing interface directly based on the hardware.

Myrinet is a local area network. Running IP on such a network poses the question of connecting the network to some wide area network. Besides switches and NICs there is no equipment to do this. However, a workstation with a Myrinet NIC and some other network interfaces may serve as a gateway between a Myrinet and other lower speed networks. Such a WAN connection could also be useful for other protocols than IP on the Myrinet. In order create long distance Myrinet conections through other networks, an encapsulation protocol 'PacketWay' [CLS+97] is being specified. PacketWay establishes a secure worm-hole connection between two Myrinets through some insecure IP network.

3. PARNASS.

The parallel computing system Parnass[1] consists of single processor workstations and shared memory multiple processor servers. They are connected by a Myrinet and a Fast Ethernet. There are three different ways of communication between processors: Shared memory between processes on one multiple-processor server, message passing libraries on TCP/IP on the Fast Ethernet between different computers, and a message passing library directly based on the Myrinet.

3.1. Hardware.

All Parnass computers are Silicon Graphics (SGI) O2 workstations and Origin 200 servers. They are based on MIPS4 binary compatible R5000 and R10000 processors. A binary executable, which has been compiled and optimized for one machine will run on all machines, although the computers run different OS versions and have different processors. The operating system allows for a homogeneous programming environment with NIS and NFS and identical software tools.

MIPS4 processors are 64 bit RISC with 64 bit data registers and 32 bit addressing and 64 bit addressing modes. The R10000 processor performs out of order execution of statements, while the R5000 is not capable of this runtime optimization. The processors run at clock rates between 150 MHz and 180 MHz, delivering a floating point peak performance of 300-360 MFlop/s each. They have two levels of cache, a 32 kbyte 1st level data and a 32 kbyte 1st level instruction cache and 0.5 MB - 1 MB unified 2nd level cache organized in 128 byte cache lines. The memory layout of the computers is non-uniform memory access (NUMA), both on multiprocessor as on single processor machines. Each node is equipped with 64 Mbytes - 256 Mbytes of main memory per processor board in custom SDRAMs and a 2 - 4 Gbyte disk drive.

Each system library is available in three versions of application binary interfaces (ABI). Binary objects, which have been compiled with different ABIs, that is different e.g. procedure call sequences, cannot be linked together. There is an ABI with 32 bit addressing and 32 bit data registers for backwards compatibility, one with with 32 bit addressing and 64 bit data registers, which usually is the fastest version, and one with 64 bit addressing and

[1] All workstations have been given the names of poets and philosophers by our system administrator. The resulting meta-computer is named after the mythological poets' place of assembly, the mountain Parnass. The similarity to the name of some parallel computing projects and companies is accidental.
See also URL `http://wwwwissrech.iam.uni-bonn.de/research/projects/parnass/`

64 bit data registers for huge applications. The operating systems themselves run in 32 bit mode for single processor workstations and in 64 bit mode for multi-processor servers. Codes compiled on one machine run on all of them. The compilers are in fact identical. There are additional compiler options to detect or to include implicit multi-threading for loops, which are useful for multiprocessor machines only.

All machines support standard Unix services and OpenGL graphics. The single processor SGI O2 workstations provide hardware support for double buffering, Z-buffer, alpha accumulation, texture mapping and JPEG decoding. All buffers including the frame buffer are part of the main memory in a kind of unified memory architecture. Hence, theoretically each buffer can be accessed and managed like ordinary memory and is limited just by the amount of main memory. Of course it does not make sense to swap an active frame buffer to disk, or let the user access it directly. However, some memory copy operations can be avoided this way, resources like texture buffers are no longer limited by special purpose memory, and CPU accesses to buffers can be optimized and cached easily. A special purpose memory controller is required to control the memory accesses by the CPU, the graphics engine and the I/O bus, which is bridged to a PCI bus interface and SCSI ultra interfaces. All data streams travel trough the central memory controller, even the regular frame buffer refreshes. This can result in a major bottleneck, if the CPU or the PCI bus happen to demand a large memory bandwidth. The effect is visible in the degradation of the I/O performance. Standard PCs, where the PCI bus is directly bridged to the memory bus, perform much better in this respect, see chapter 4.1.5.

The double processor SGI Origin 200 servers are designed as one shared memory double processor board attached to a memory controller. The controller provides links to an I/O system. There is no frame buffer, just some PCI and SCSI slots. The machines are used as parallel compute servers. The quadruple processor machines are constructed of two complete double processor servers connected by a shared memory CrayLink. While cache coherence on one double processor board is implemented via bus snooping, it is implemented between processor boards on the base of cache directories. One 64 bit multiprocessor version of the OS kernel is running on such a machine with two or four processors. Processes are assigned to processors by the OS scheduler, which can migrate processes. A process tries to allocate memory located on its double processor board for performance reasons. However there is no guarantee for this. If memory on the board is exhausted, memory on the other machine via the link is used with higher latencies. The architecture is transparent to user. The same architecture is used in the Silicon Graphics/ Cray Research Origin 2000 computer series, where additional routers manage the links to neighbor double processor boards connected in a hyper-cube topology.

3.2. Network.

The basic interconnection network of Parnass is a switched Fast Ethernet LAN. Each single processor workstation and each double processor server has a full duplex auto-negotiating Fast Ethernet adapter at the speed 100 Mbit/s. Quadruple processor servers have two adapters, since they are composed of two complete double processor machines. On top of the Fast Ethernet, the IP protocol is running, with UDP and TCP, and higher

level services such as NFS, NIS and the BSD 'r'-tools. There are several public domain implementations of message passing libraries available, such as MPI and PVM.

Quadruple processor machines possess two Ethernet adapters, two IP addresses and two names. The average network bandwidth is doubled this way, compared to a single adapter, although it cannot be guaranteed that communication with a processes is routed through the nearest Fast Ethernet adapter. Hence some messages will flow through the shared memory link in addition to the Fast Ethernet adapter. However, the Parnass Ethernet network is balanced with one to two processors per Fast Ethernet port.

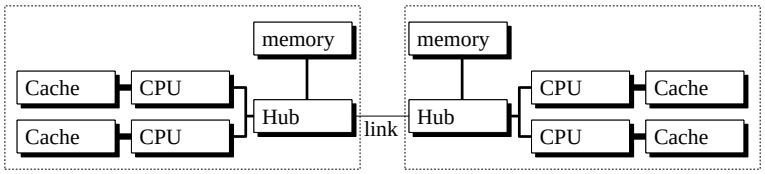

Figure 5. Shared memory quadruple processor architecture.

Shared memory connections can be considered as a system area communication network. Double processor boards in a multiprocessor server share the same memory bus. This gives a shared memory communication with cache coherence via bus snooping. The CrayLink is a shielded custom copper cable to connect two double processor machines via shared memory that operates at a bandwidth of 5.76 Gbit/s, see figure 5. There are several programming models for parallel computing available: The simplest one is multi-threading. Lots of lightweight threads access a single image of shared memory. The code for multiple threads can be created by the auto-parallelizing compiler for simple code structures. In more complicated cases, additional 'pragma' compiler directives initiate multi-threading. Of course, threads can also be used explicitly by library calls. Heavy processes may communicate via vendor implementations of standard message passing libraries such as MPI and PVM and via specific shared memory library calls.

Parnass' high performance interconnection network is a Myrinet. Each single processor workstation and each double processor server is equipped with one Myrinet NIC, which means that quadruple processor servers have two adapters. The NICs are linked in a two stage fat tree topology, see figure 6. Connections between two computers go either trough one switch or through three switches (hops), which results in a latency of 0.1μs to 0.7μs caused by the switches. The numbers depend on the port types, because SAN ports are faster than LAN ports. Latencies due to the cable lengths are of the same order. Electrical signals travel at the light speed of the transmission medium, which is lower than vacuum light speed. This takes some time for the maximum 50m distance.

The bisection bandwidth as a measure of the network throughput is optimal for the

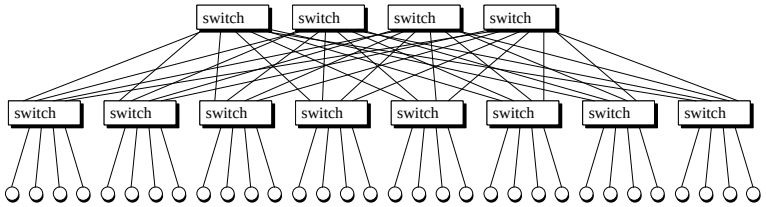

Figure 6. Topology of the Parnass Myrinet: A fat tree with full bisection bandwidth of 40 Gbit/s.

fat tree in figure 6. Every line, which cuts the network into two pieces of 16 computers each, cuts at least 16 links. This means that the network has the same performance as a 32 port crossbar switch. Any set of point-to-point communications between different computers can be routed without collisions or blocking. However, a smart routing scheme is necessary to avoid collisions in the fat tree.

Roughly half of the computers are linked with short distance unshielded SAN copper cables (up to 3m distance) and the other half is linked with long distance shielded LAN copper cables. The diameter of the network is about 50m to connect all office workstations. The longest high speed cable offered by Myricom is only 10m long. Hence, some of the LAN cables had to be extended by a LAN repeater. Some of the switches with long distance LAN ports are mounted in offices, while the bulk of the switches is located in a cabinet along with the multiple processor servers. Eight short distance SAN port switches (top row and four switches from bottom row) are packaged altogether with the connections in a single board 19" rack-mountable box.

4. MYRINET ON PARNASS.

We are interested in using the Myrinet on Parnass. In order to do message passing over a Myrinet on MIPS based machines, we have to port an operating system kernel driver, a control program for the NIC, and a message passing application interface for the user program. Such components are available in source code for Linux PCs and for Suns, some even for Power PCs and Windows NT on Intel and DEC Alpha based machines. However, no such drivers had been written for Silicon Graphics workstations so far. We will describe some considerations of the port we did.

4.1. Myricom API.

First we describe the port of the Myricom API (see chapter 2.4.1 and [Myr96]) to Parnass. We implemented kernel device drivers for SGI IRIX and we ported the Myricom API to the SGI platform. We encountered several problems such as little/ big endian adressing, many memory copy operations, cache coherence problems of DMA transfers and a lack of API protocol reliability. Furthermore the SGI hardware PCI bus implementation

showed some high latencies. This resulted in performance problems. To overcome these problems, we chose a different API and furthermore applied some modifications, which we describe in the next chapter 4.2.

4.1.1. Device Driver.

We wrote one kernel device driver for IRIX 6.3 (O2 workstations) and one IRIX 6.4 (multiprocessor machines) using some of the structure of other Unix device drivers as a template. However, the detailed OS calls and data structure differ from one Unix dialect to another. The IRIX device driver interface has many similarities with other BSD Unix variants, see [Cor96]. Furthermore it offers some System V compatibility. Because the PCI bus support in IRIX has been added recently by SGI, all PCI bus related parts differ from previous I/O bus support and from other Unix version's PCI functions. Hence we had to start from scratch.

The new kernel device driver registers for the Myrinet NIC vendor and adapter numbers, see [ET88, Cor96]. The driver is loaded in case the kernel detects the NIC on the PCI bus. The main purpose of the driver is to provide the mapping of the PCI address space into the user address space. The control registers and the SRAM of the NIC are mapped for programmed I/O (PIO) accesses. Additionally the driver allocates a chunk of contiguous, non-swappable memory, the 'copy-block'. This memory section can also be mapped into user space. Its physical addresses can be programmed into the DMA engine located on the NIC. It is used as a buffer for DMA transfers.

4.1.2. Byte Swapping.

The Myricom API itself is portable C and C++ code, which can be compiled with minor modifications. The main change we did is due to a problem with byte swapping. The MIPS processors in use are build as big endian machines. The PCI standard originates from Intel based PCs, which are little endian. Hence most of the PCI adapters are little endian, as is the PCI interface of the Myricom NICs. The difference between big endian and little endian for 32 bit accesses means that 32 bit addresses are wired correctly and data is wired in the wrong order. The 4 bytes have to be swapped from one format to the other format, see figure 7.

Usually, byte-swapping is implemented in hardware to make the access to little endian devices transparent. Such facilities are implemented in the various PCI bridges. The single processor workstations provide mappings of the PCI devices to two different addresses, one for big endian and one for little endian. The multi-processor servers use PCI bridges, which are programmable to big endian and little endian for certain address ranges. Unfortunately these hardware byte swapping features are not available in the current OS releases IRIX 6.3 for O2 and IRIX 6.4 for Origin 200 computers.

The byte swapping can also be accomplished by appropriate calls to the library functions 'ntohl' and 'htonl' on little endian machines. On big endian machines, these function calls often are 'no-operation', because network services like NFS are considered as big endian. Of course it is easy to write a substitution for the above function. Furthermore, byte and 16 bit accesses additionally lead to an address problem. Since all accesses to a 32 bit PCI bus are done in full 32 bits, there is no way to filter the right byte or two bytes out of the

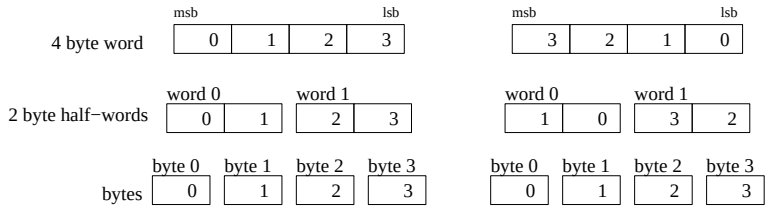

Figure 7. Big endian (left) and little endian (right) storage of 32 bits as words, half-words and byte.

32 bits. The hardware assumes a big endian scheme and accesses the wrong bytes. Hence, the address for byte and word accesses has to be permuted in addition to byte swapping for 16 bits. The byte-address has to be xor'ed by 3 for bytes and by 2 for 16 bit words. This means an xor operation by 1 for 16 bit array index accesses. Things become even more tricky for un-aligned data, which could be avoided in our case.

If we are interested to transfer raw data of a multiple length of 4 bytes, we do not have to bother with byte-swapping at all. Whether the bytes are exchanged by the sender and reversed by the receiver does not matter, because we have a homogeneous computing environment. This would change for the communication with other computer platforms. However, for the communication with the NIC hardware, as well as some data on the NIC SRAM, we have to care about the byte order. That is why we had to modify several parts of the codes to implement the right byte order.

4.1.3. Data path of the Myricom API.

Consider some contiguous user data to be transferred to another processor. If we want to do this with the Myricom API, first, we have to copy the data into a DMA-able region, which can be any location in the copy-block, see figure 8. The copy-block is a limited resource allocated by the kernel. We call an API function to transfer the data. The API passes the addresses to the processor on the NIC, which initiates a DMA transfer from host memory into the SRAM. The NIC processor looks up the route to the destination processor and puts the route onto the wire. Afterwards it initiates a DMA transfer for the data from the SRAM onto the wire. The NIC hardware adds a trailing checksum.

Given a receiving process that has put a receiving buffer of the copy-block into the NIC receive queue, see figure 9. The incoming message is stored in the NIC SRAM and transferred by the NIC DMA into one of the receiving buffers. This message can now be fetched by the receiving process, which has to copy the data to the destination location.

This procedure holds for packets shorter than the message transfer unit (MTU) of 8132 bytes. Longer messages have to be split and sent in several packets. There are two problems associated with this message passing semantics: If the receiver does not provide a receive buffer, the message is dropped and gets lost. No acknowledge or flow

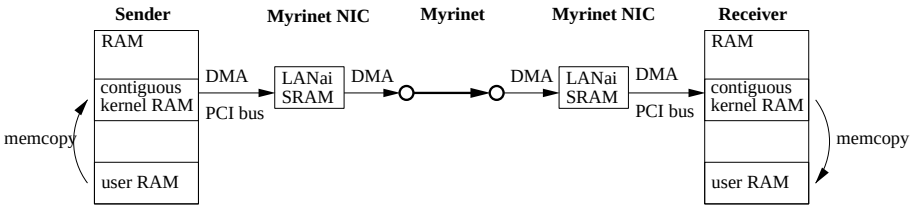

Figure 8. Data path for the Myricom API.

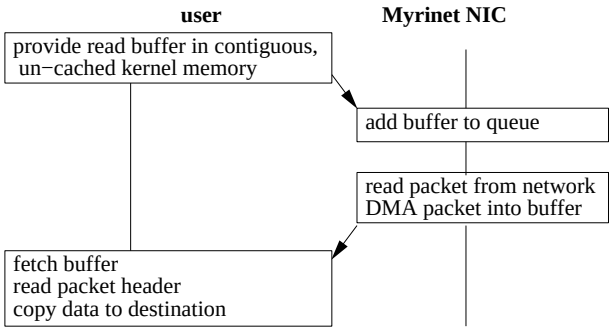

Figure 9. Control flow for the Myricom API, receiving one packet, without any operating system calls.

control protocol is used on this software level. Such a situation may occur, if the sending process is too fast. The receiver does not expect a message yet, or the receive buffers are exhausted. Unfortunately there is a lack of a flow control protocol in the API layer, which is present on the hardware link level. Hence the network reliability is lost due to the API layer.

A more subtle problem is about the two copy operations necessary. The semantics of MPI, for example, gives the user the freedom to transfer any data in the user address space. Since the Myricom API limits this address space to the limited amount of the copy-block, such copy operations cannot be avoided in general. However, they may have a severe performance impact for high speed networks.

4.1.4. Myricom API Results.

We have measured the performance of our port of the Myricom API with a simple ping-pong test between two computers. We took the timing for different packet sizes to compute the latency and the bandwidth, which we computed from the latency and

the packet length. There were 10^4 packets sent and received. The numbers reflect the average value. The three lines in figure 10 are timings for two single-processor machines, a single-processor machine and a multiprocessor machine, and a single-processor machine and a Linux PC. The performance of the pure API is measured, without any additional data copy, which may be necessary.

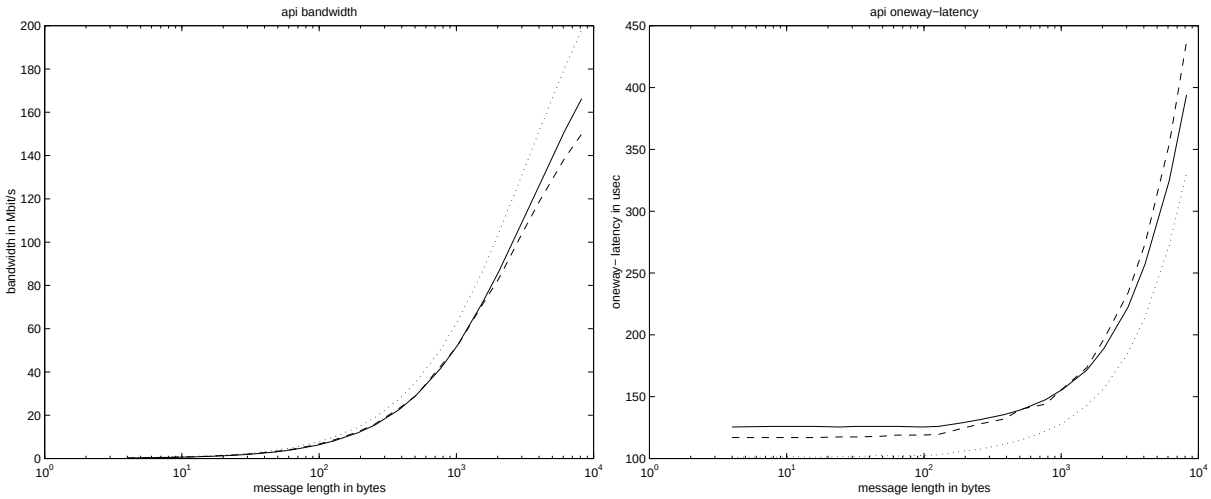

Figure 10. Bandwidth in Mbit/s (left) and latency in μs (right) for the Myricom API and for different packet lengths. Connection between single processor machines solid line, single and double processor machine dashed line, and single processor machine to a Linux PC dotted line.

The Myricom API has a low communication latency of 125 μs and a maximum bandwidth of 150 Mbit/s. However, there is no flow control and messages tend to get lost due to a lack of receive buffers. The results are not much better than the nominal performance of Fast Ethernet. However, the Linux PC performance demonstrates the potential of much better Myrinet values, which could also be seen in other tests e.g. [PT97]. Bandwidths up to 1000 Mbit/s and latencies as low as 5 μs can be obtained on some Linux PCs, with the appropriate software.

Even worse, in order to implement a standard message passing protocol such as MPI on top of the Myricom API, one has to implement flow control. Adding reliability to the protocol requires at least some hand-shaking to avoid buffer overflow. The communication lines themselves can be considered as reliable. Thus, we can avoid a more elaborate acknowledge protocol. Furthermore one has to copy data from and to the copy-block. This degraded the overall performance in some tests we did to a bandwidth of 37 Mbit/s with 250 μs latency, which we can roughly obtain with the cheaper Fast Ethernet, see

section 4.3.

4.1.5. PCI Bus Implementation.

If we really want to exploit the performance of the Myrinet components, we have to identify the problems and bottlenecks of the previous Myricom API port. Almost identical software shows completely different speed on PCs on the one hand and on our MIPS based machines on the other hand. One obvious reason can be found in the characteristic of the different PCI bridge implementations, which are compared in table 1.

	SGI O2	SGI Origin 200	good PC boards
byte swapping	disabled	disabled	not necessary
PIO prefetch	disabled	disabled	available
sequence of PIO accesses	48 cycles = 1.44 μs = 22 Mbit/s	48 cycles = 1.44 μs = 22 Mbit/s	3 cycles = 0.090 μs = 355 Mbit/s
cache coherence of DMA	not available	implemented	implemented
DMA performance read	681 Mbit/s	338 Mbit/s	1015 Mbit/s
write	727 Mbit/s	742 Mbit/s	1015 Mbit/s

Table 1. Comparison of 32 bit 33 MHz PCI bus implementations.

We have discussed the lack of byte swapping support of the OS already. However, only few words actually have to be swapped. The majority is just raw data, which is copied.

Another problem is DMA performance, see also figure 11. The 32 bit wide PCI bus at 33.33 MHz is able to achieve a theoretical DMA performance of 1066.67 Mbit/s. We see that some PC implementations are pretty close. The MIPS numbers are not that good, but they cannot explain a performance gap of a factor of 8 between PC and our machines. The DMA read performance on multiple processor machines is especially bad. This is in part due to a special PCI bus arbitration mechanism, which allows for a maximum transfer block of 128 bytes (the cache line size) to ensure fairness.

The main difference between PC PCI implementations and our machines however is the programmed I/O (PIO) access. This is also due to the PCI bus arbitration, which even slows down if no other PCI device, like the SCSI adapters, requires the PCI bus. Hardware support for PIO prefetch was provided by SGI, which unfortunately had to be turned of by the OS to prevent some errors. This means that any CPU access to the NIC is slow on any recent SGI machine. This includes communication with the NIC processor via SRAM accesses and the access of NIC registers. If some API implementation uses too many PIO accesses, it will be slow on Parnass.

Furthermore there is a problem with DMA on single processor workstations. There is

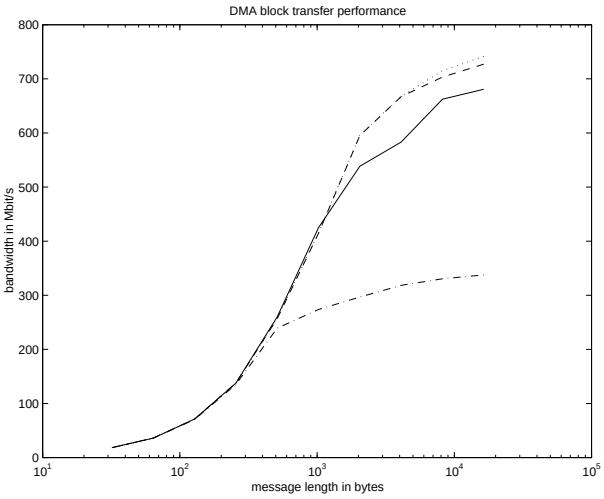

Figure 11. DMA performance of single and double processor PCI-bus implementations for 32 bit cards in Mbit/s. PCI reads from single proc. solid line, PCI writes to single proc. dashed line, PCI reads from double proc. dash-dotted line, PCI writes to double proc. dotted line.

no cache coherency. Any DMA operation by the NIC has to be done in conjunction with appropriate cache write-back or cache invalidate OS calls, which can be quite expensive. However, such calls are not included in the Myricom API, which does not require OS calls at all. An alternative would be to use un-cached memory. Unfortunately the access to un-cached memory is also very slow. The lack of cache coherence may result in additional latencies in the order of μsecs.

4.2. BullDog and BullFrog API.

If we want to achieve a higher performance with a message passing implementation over the Myrinet, we have to avoid the drawbacks of the standard Myricom API mentioned in the last section, which were the many memory copy operations and a lack of API protocol reliability. We cannot avoid the high latencies of the SGI hardware PCI bus implementation. However, we can try to use as little PIO as possible by changing the host NIC software interface. First we describe our port of the BullDog API [HDMS97]. Tests showed poor performance of the API on SGIs for long packets. However, BullDog proved to be a good base for further development. We add support for DMA operations for long packets and we create an API, which switches between DMA for long packets and PIO for short packets. We call this API BullFrog[2].

[2]The original 'BullDog' API was named in honor of the Mississippi State Bulldog Basketball Team. We chose the name 'BullFrog' for our extended version, because the API jumps between the original BullDog

4.2.1. Structure of BullDog.

The BullDog API [HDMS97] uses a smarter, multi-threaded control program for the NIC than the original Myricom API. The control program implements a reliable and ordered message passing protocol with acknowledgment and delivery receipt without interference of the host CPU. This speeds up any higher level reliable message passing protocol. Furthermore, the data transfer between the NIC processor and the host processor is minimized. Both processors communicate via several single-producer single-consumer queues, where packet headers are exchanged. This reduces CPU overhead further, but even more valuable, it reduces the amount of PIO accesses. Another feature of BullDog is the semantic and syntactic similarity to some MPI calls, which streamlines an MPI implementation on top of BullDog.

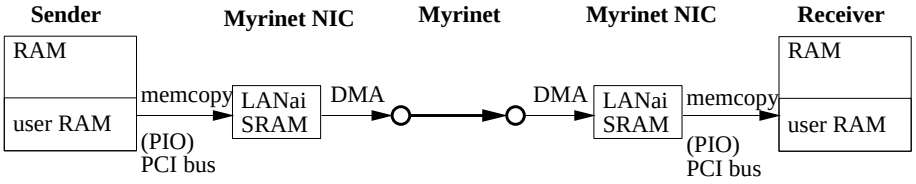

Figure 12. Data path for the original BullDog API.

The data path of BullDog (see figure 12) looks different than the path used in the Myricom API. There is no DMA in BullDog, as there are no other privileged operations. All data is copied directly from and to the SRAM with PIO accesses. This circumvents the copy-block and looks like an improvement compared to the Myricom API. However, PIO accesses are very slow on our workstations. We cannot expect bandwidths with BullDog above the peak PIO performance of 22 Mbit/s, which is well below the performance of Fast Ethernet.

4.2.2. BullFrog with Additional DMA.

The main features of BullDog sound very attractive for Parnass. The drawback however is the memory copy with slow PIO accesses. The only alternative to fix the problem is DMA data transfer. The DMA performance is a factor of 20 to 40 times higher than the PIO performance, as we saw in table 1. Hence we modified the existing BullDog API by DMA capabilities. We call this new API BullFrog. The data flow with DMA is depicted in figure 13. The memory copy operations by the host CPU are substituted by the faster DMA operations by the NIC.

---

and our DMA version back and forth, depending on the block sizes, like a frog. Alternatively we could have named it after our successful local basketball team 'Telekom' sponsored by the national telephone company.

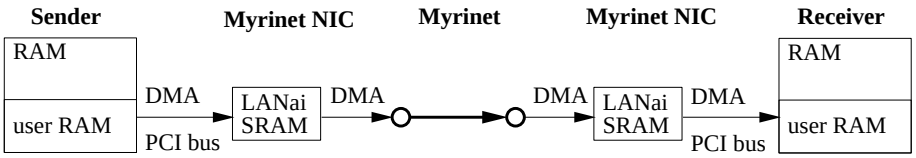

Figure 13. Data path for the BullFrog API with DMA block transfers instead of memory copy.

The main problem with DMA is that it is initiated by the NIC. During DMA transfers, the NIC acts as a PCI bus master. When the host node receives a packet for example, which is stored in NIC SRAM, a process on the host can retrieve the packet, see figure 14. In the semantics of MPI for example, the receiving user process determines the destination address. The receiver process decodes the tag and sender ID and scans the pending receive calls. Sometimes the matching receive call is issued at a later time. Only at that time it is clear where to transfer the packet to. The NIC does not know in advance the final destination address in user space. However, we want to implement a zero-copy protocol and we want avoid intermediate storage.

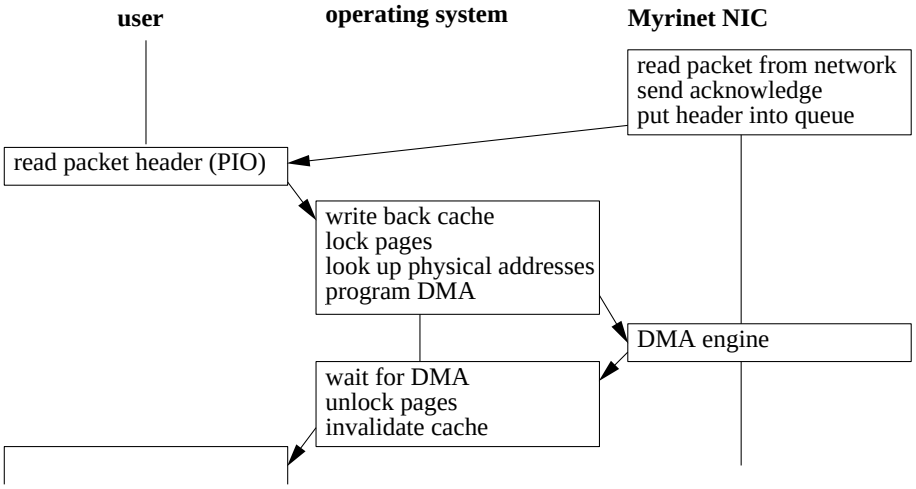

Figure 14. Control flow for the BullFrog API with DMA block transfers, receiving one packet.

At the time the destination address is known, the DMA transfer can be initiated. On single processor O2 machines this requires a cache invalidate call for the destination region (on the sender a cache write-back). The destination address usually does not point into physically contiguous memory. The memory is contiguous from the users view, which is done by the processors memory management unit (MMU). However, the memory block looks fragmented from the point of view of an I/O device. Hence the block has to be split into memory pages, whose addresses have to be translated. This gives a list of addresses and lengths, which can be programmed into the DMA unit of the NIC. Fortunately, we do not need an NIC processor interaction. Register access is sufficient. Before we are allowed to initiate the DMA, we have to lock the pages in memory to avoid paging. This also means that we have to wait for the DMA to terminate to unlock the pages and to return to the user context. The DMA operation becomes blocking.

To modify BullFrog for the DMA transfers, we have to substitute the memory copy routine calls (memcopy) by calls of a new device driver function. The DMA transfers, along with all the other OS tasks described, are located in a single new function, which is added to the Myrinet kernel device driver. The function has the standard 'memcopy' semantics. The resulting message passing API can be considered as zero-copy protocol, since the host CPUs do not touch the data at all.

4.2.3. BullFrog Switching between PIO and DMA: Results.

We have measured the bandwidth and latency with a simple ping-pong test for the our port of the BullDog and the BullFrog APIs. The performance of the original BullDog ordered protocol with memory copy and the BullFrog API with DMA transfers is shown in figure 15. The final BullFrog API uses a mixed mode: Short packet are transferred by memcopy and long packets are transferred by DMA. The resulting latency is the minimum of both latencies and the bandwidth is the maximum bandwidth of the values shown in figure 15.

BullDog has very low communication latencies of about 19 μs. The DMA extension of BullFrog boosts bandwidth from original 20 Mbit/s to over 250 Mbit/s, but increases the latencies by 100μs. The original maximum packet sizes of 8 kbytes have been increased to 32 kbytes produce the timings for the larger packets.

We cannot improve the performance of the original BullDog, since the PIO bandwidth is 22 Mbit/s and we are close to that. However, it pays off for data packets larger than 320 bytes to switch to the BullFrog DMA version. Comparing the data with the Myricom API data, we have to mention that data transfer now take place from and to ordinary buffers supplied by the user, like in MPI. The copy-block is not used any longer.

The final version of BullFrog jumps between PIO and DMA mode, depending on the packet size. The special memcopy procedure calls either and ordinary memcopy, if the packet size is below some threshold, or it translate addresses and calls the device driver function, which initiates a DMA transfer. This switching is done transparently to the API layer. The switching threshold can be chosen for each library. This means that in a heterogeneous environment, different thresholds on different machines, such as the SGI O2 and Origin O200, are possible. Furthermore, the BullFrog API with PIO/DMA is

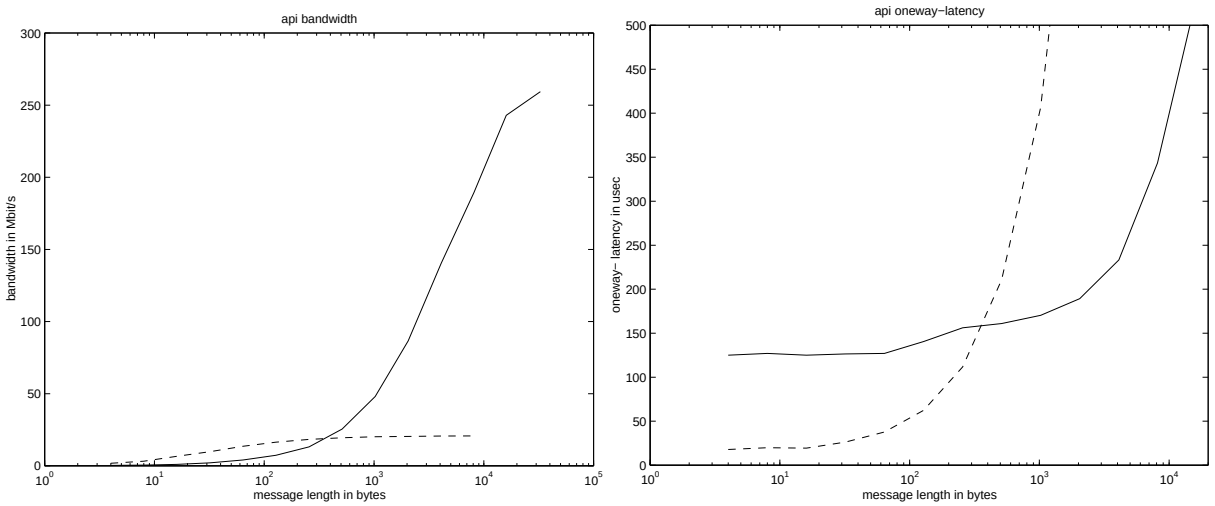

Figure 15. Bandwidth in Mbit/s (left) and latency in μs (right) for the BullDog and BullFrog APIs and for different packet lengths. BullDog API with memcopy through PIO dashed line, BullFrog API with DMA solid line. The final BullFrog version switches between memcopy and DMA.

inter-operable with the BullDog API with PIO in a heterogeneous network.

4.3. MPI on Parnass.

4.3.1. Available MPI Implementations.

We compare the MPI implementations available on Parnass. There is a vendor optimized shared memory version of MPI which was used on the multiple-processor Origin 200 machines. The processes communicate via (virtual) shared memory segments. The data passes the CrayLink, if two distant processors communicate. Local processors can exchange data over the common memory bus. However, there is no guarantee, where a specific process resides. It may even happen, that two processes are located on the same processor and can communicate through the shared memory segement located in the processor's cache.

Furthermore we use a version of the MPI implementation MPICH [GLDS96] on top of TCP/IP on the Fast Ethernet. It uses the p4 message passing library. This version is not limited to the Fast Ethernet and can also be used for other computers connected by Ethernet or some other media. Additionally we can use it for communication between processes on a shared memory machine. Of course shared memory communication is more efficient in this case.

4.3.2. Porting MPICH to Myrinet.

We have ported MPICH to the BullFrog message passing system on SGI workstations. The MPICH implementation for the BullDog system [MHH+97] compiles on the SGI machines. The OS dependent parts of the MPICH device are hidden in the BullDog library. Some smaller adjustments for the compilers were needed.

The main difference between BullDog and BullFrog is the use of DMA. The DMA transfers have to be initiated by the receiving process. We try to avoid buffering and we want to achieve a zero-copy protocol. This means that the DMA is started in the MPICH device. The memcopy calls are to be substituted by the special memcopy of the BullFrog library. In this sense, we had to patch the existing MPICH device. This special memcopy routine either calls the standard memcopy and does PIO for short packets, or it translates addresses and calls the kernel device driver to initiate the DMA transfers. The switching point for BullFrog on SGI O2 machines actually is set to 512 bytes.

4.3.3. Comparing MPI Implementations.

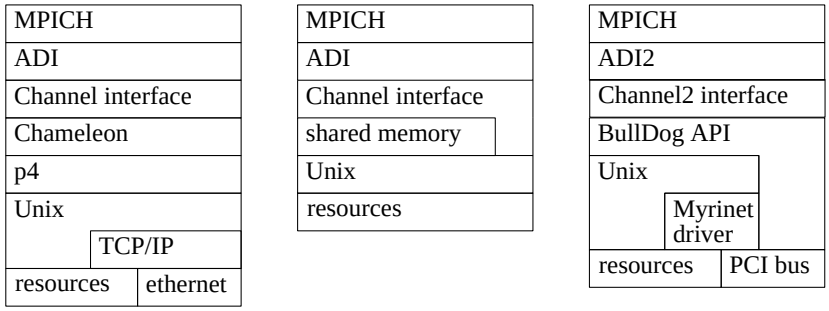

Figure 16. Layers of MPI implementations on Ethernet (left), shared memory (middle), and BullFrog (right).

The different MPI implementations are shown schematically in figure 16. They are based on an ADI [GL96], which is based itself on some channel interface implementations. The TCP/IP version of MPICH is based on the p4 message passing system, the shared memory version uses windows of shared memory to transfer the data, and the Myrinet version is based on BullFrog.

We compare the performance of all MPI implementations available on Parnass. We use a ping-pong test to measure latencies and performance between single processor workstations. The shared memory tests were run on a quadruple processor Origin 200. The results are depicted in figure 17.

The ping-pong one-way latencies vary from 16 μs for shared memory, 70 μs for BullFrog on Myrinet to 0.63 ms for Fast Ethernet and 1.15 ms for Ethernet. The peak bandwidth

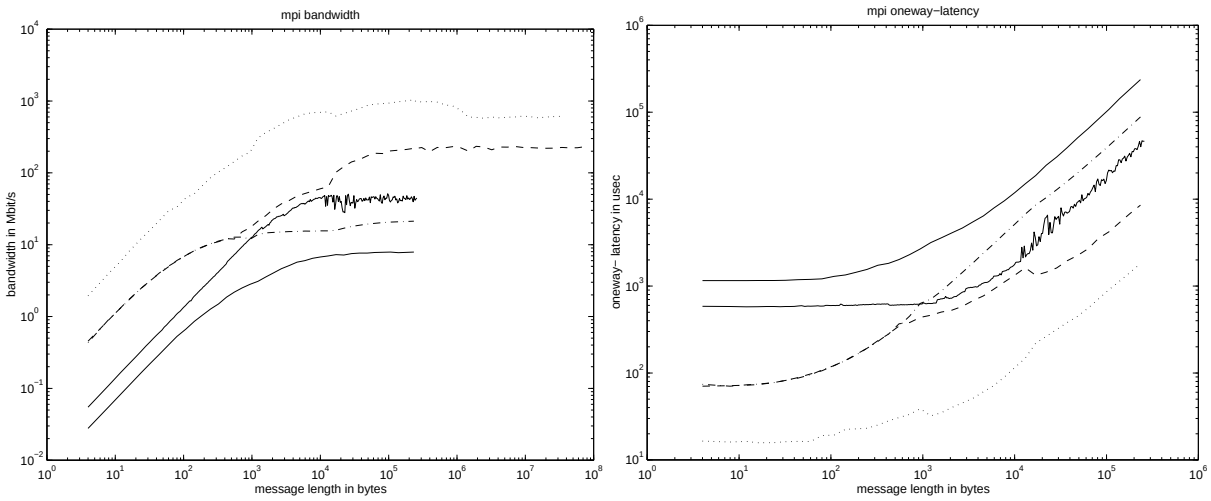

Figure 17. Bandwidth in Mbit/s (left) and latency in μs (right) for different MPI implementations and for different packet lengths. MPICH on Ethernet (10 Mbit/s) and Fast Ethernet (100 Mbit/s) solid lines, MPICH on BullDog with memory copy (Myrinet) dash dotted line, MPICH on BullFrog with swicthing between memory copy and DMA (Myrinet) dashed line, MPI on shared memory dotted line.

performance is 7 Mbit/s for Ethernet, 40-47 Mbit/s for Fast Ethernet, up to 220 Mbit/s for Myrinet and 1 Gbit/s for shared memory. The high latencies of Ethernet are in part due to the TCP/IP overhead involved. Fast Ethernet compared to Ethernet starts paying off for packets longer than 1.5 kbytes. While it is a factor of 2 faster for small messages, the peak bandwidth is 6-7 times higher than Ethernet.

We compare two versions of BullFrog for Myrinet: The DMA version is used for packets longer than 512 bytes, while the 'memcopy' version achieves only 10% of the DMA version's bandwidth. Both numbers have to be compared to Myrinet's theoretical peak bandwidth of 1.28 Gbit/s.

The latencies for the shared memory device mainly demonstrate OS and MPI administration overhead and time for system calls. The peak bandwidth is limited by some 'memcopy' operations through shared memory. Bandwidth even decreases for packets larger than the size of the 2nd level cache down to 610 Mbit/s. The theoretical peak bandwidth of the shared memory connection between two double processor boards (CrayLink) is 5.76 Gbit/s and the on board connection is even faster.

To summarize, one has to admit that the message passing system on a shared memory system achieves only a fraction of the high theoretical memory performance. It is still faster than shared memory implementations by other computer vendors. Generally, the faster a network is, the higher are the losses due to software overhead and management.

This rule of thumb is in part true also for other high speed networks, like Myrinet on SGI and the Fast Ethernet. However, the difference between two fastest MPI versions, shared memory and BullFrog melts down for long packets to a factor of 2.5.

4.3.4. Conclusion.

We have written and presented a set of drivers and libraries for Myrinet on SGI computers. We have developed kernel device drivers, ported the Myricom API and the BullDog API, extended the last one by DMA transfers to the BullFrog API and ported MPICH. We have compared several measurements concerning the performance of message passing libraries and the PCI bus hardware, which again gave hints for improvements of the libraries. Finally we arrived at a high performance MPI implementation for SGI machines, which we compared to the shared memory links.

A gap between the performance of shared memory communication and performance of the Myrinet communication could be observed, the first one available only between a few processors and the latter one available between all processors. However, one has to keep in mind that the Myrinet network performance scales better than shared memory links. This is also true for higher numbers of processor if we compare the Origin 2000 network fabric with Myrinet. Furthermore, workstations equiped with Myrinet posses a much more attractive price/ performance ratio than large shared memory machines.

4.4. Future work.

The MPI library for Myrinet on Parnass is operational. However, we still have to address security issues in a multi-user Unix environment. Sensitive parts of the API should be placed under kernel protection. Direct hardware access should be denied to the user. This has been achieved already for a different message passing system GM [Myr97].

A related question is about a safe multi-user and multi-process computing environment. The BullFrog API, as well as many other APIs, monopolizes the Myrinet hardware. It re-loads the NIC control program, which communicates exclusively with one application. Extensions to share the NIC between several applications also have to be placed into the OS kernel. If there are several parallel applications running concurrently, this does not deliver the maximum performance of Parnass. However, during production runs, such a situation may occur. All TCP/IP implementations on Myrinet are able to overcome this problem, since the OS monopolizes the Myrinet. Furthermore the GM message passing system provides a mechanism to share the network resources on a lower level. This is done under the administration of the kernel, which handles the privileged DMA-able memory.

We intend to run the BullFrog API on additional computer architectures. Porting the API to Linux based computers should is straightforward. We expect the performance numbers to be much better than for O2 and Origin 200 computers. We will report on such experiments in a forthcoming paper.

Furthermore, we intend to improve the routing scheme in our fat tree network topology. It is easy to substitute the present static routing by some other schemes. We expect a random routing scheme to improve the overall bandwidth for a wide class of parallel applications. We will also report on this topic in a forthcoming paper.

We would like to see some improvements of the workstations themselves: Maybe some

patches of the IRIX OS by Silicon Graphics for the PCI bus become available to turn on hardware byte-swapping and PIO prefetch. A new layout of the memory controller may also reduce PIO latency and improve DMA performance to meet the speed of (second generation) PCI bus implementations by many other vendors.

4.5. Outlook.

The main goal of the Parnass parallel computer is to serve as a platform for our activities in high performance scientific computing. Some leading edge applications for the solution of partial differential equations are running already or are under development. We have developed a parallel adaptive finite element/ finite difference code with a multigrid equation solver. A space-filling curve gives optimal dynamic load-balancing. At the same time data is stored highly efficient in a parallel hash table, see [GZ97].

For Schrödinger's equation we have developed a parallel code. It is based on the sparse-grid combination technique [GSZ92]. The electron density of a Helium atom under a strong magnetic field leads to an eigenvalue problem in six spatial dimensions, which we is solved efficiently.

We are developing a molecular dynamics code based on the adaptive Barnes-Hut approximation. It is implemented with space filling curves and hash table like [WS95]. The parallel version is currently under construction.

Furthermore, we also run conventional parallel codes, such as a three dimensional incompressible Navier-Stokes solver. It is parallelized by standard domain decomposition techniques.

References

[BCF+95] N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W.-K. Su. Myrinet – a gigabit-per-second local-area network. *IEEE-Micro*, 15(1):29–36, 1995.

[Bro96] G. Brown. *Draft Lyric Programmer's Guide.* Myricom, Inc., 1996.

[CLS+97] D. Cohen, C. Lund, T. Skjellum, T. McMahon, and R. George. *Proposed Specification for the End-to-End (EEP) PacketWay Protocol.* The Internet Engineering Task Force, draft edition, 1997.

[Cor96] D. Cortesi. *Device Driver Programming in IRIX 6.3 For O2.* Silicon Graphics, Inc., 1996.

[Dig96] Digital Equipment Co. Memory channel. URL: http: //www.digital.com:80/info/hpc/systems/symc.html, 1995, 1996.

[Enc97] Encore Computer Co. Reflective memory. URL: http: //www.encore.com/realtime/products/systems/rms.html, 1997.

[ET88] J. I. Eagan and T. J. Teixeira. *Writing a UNIX Device Driver.* Wiley, 1988.

[FDCF94] R. Felderman, A. DeSchon, D. Cohen, and G. Finn. Atomic: A high-speed local communication architecture. *Journal of High Speed Networks*, 3(1):1–30, 1994.

[Gig97] GigaLabs Inc. Gigastar. URL: http: //www.gigalabs.com/ prodinfo/pci.htm, 1997.

[GL96] W. Gropp and E. Lusk. *The Second-Generation ADI for the MPICH Implementation of MPI.* MCS Division, Argonne National Laboratory, MPICH working note edition, 1996.

[GLDS96] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. Technical Report MCS-P567-0296, MCS Division, Argonne National Laboratory, 1996.

[GSZ92] M. Griebel, M. Schneider, and C. Zenger. A combination technique for the solution of sparse grid problems. In P. de Groen and R. Beauwens, editors, *Iterative Methods in Linear Algebra*, pages 263–281. IMACS, Elsevier, North Holland, 1992.

[GZ97] M. Griebel and G. Zumbusch. Parallel multigrid in an adaptive pde solver based on hashing. In *Proceedings of ParCo '97*. North-Holland, 1997. submitted.

[HDMS97] G. Henley, N. Doss, T. McMahon, and A. Skjellum. BDM: A multiprotocol program and host application programmer interface. Technical Report MSSU-EIRS-ERC-97-3, Mississppi State University, Dept. Computer Science, 1997.

[HIC95] IEEE Service Center, Piscataway, NJ. *IEEE Standard for Heterogeneous InterConnect (HIC) 1355-1995*, 1995.

[IHS+95] Y. Ishikawa, A. Hori, M. Sato, M. Matsuda, J. Nolte, T. Tezuka, H. Konaka, M. Maeda, and T. Tomokiyo. RWC massively parallel software environment and an overview of MPC++. In *Workshop on Parallel Symbolic Languages and Systems*, 1995.

[LC97] M. Lauria and A. Chien. MPI-FM: High performance MPI on workstation clusters. *Journal of Parallel and Distributed Computing*, 1997.

[LMC97] S. Lumetta, A. Mainwaring, and D. Culler. Multi-protocol active messages on a cluster of SMP's. In *Proceedings of Super Computing '97*, San Jose, California, 1997.

[MHH+97] T. McMahon, G. Henley, S. Hebert, B. Protopopov, R. Dimitrov, and A. Skjellum. MCP and MPI for Myrinet on Sun and Windows NT. Slides from Super Computing '96, 1997.

[Myr96] Myricom, Inc. *Myrinet User's Guide*, 1996.

[Myr97] Myricom, Inc. *The GM API*, 1997.

[PT97] Loïc Prylli and Bernard Tourancheau. New protocol design for high performance networking. Technical Report 97-22, LIP-ENS Lyon, 1997.

[SCI93] IEEE Service Center, Piscataway, NJ. *IEEE Standard for Scalable Coherent Interface (SCI), 1596-1992*, 1993.

[Tan95] Tandem Computers Inc. ServerNet. URL: http: //www.tandem.com/ prod_des/srvnetpd/srvnetpd.htm, 1995.

[THI96] H. Tezuka, A. Hori, and Y. Ishikawa. PM: A high-performance communication library for multi-user parallel environments. Technical Report TR-96015, RWC, Parallel Distributed System Software Tsukuba Laboratory, 1996.

[WS95] M. Warren and J. Salmon. A portable parallel particle program. *Comput. Phys. Comm.*, 87:266–290, 1995.

[WTH95] T. M. Warschko, W. F. Tichy, and C. G. Herter. Efficient parallel computing on workstation clusters. Technical Report 21/95, University of Karlsruhe, Dept. of Informatics, 1995.

[YCGL97] K. G. Yocum, J. S. Chase, A. J. Gallatin, and A. R. Lebeck. Cut-through delivery in Trapeze: An exercise in low-latency messaging. Technical report, Dept. of Computer Science, Duke Univ., Durham, NC, 1997.