Efficient Parallel Computing on Workstation Clusters

Thomas M. Warschko and Walter F. Tichy and Christian G. Herter
University of Karlsruhe, Dept. of Informatics
Postfach 6980, 76128 Karlsruhe, Germany
email: {warschko|tichy|herter}@ira.uka.de

Technical Report 21/95

Abstract

We present novel hard- and software that efficiently implements communication primitives for parallel execution on workstation clusters. We provide low communication latencies, minimal protocol, zero operating system overhead, and high throughput. With this technology, it is possible to build effective parallel systems using off-the-shelf workstations. Our goal is to develop a standard interfaceboard and the necessary software for interfacing any number of computers, from a workstation to a cabinet full of workstation-boards.

1 Introduction

PCs and Workstations are in widespread use and offer excellent performance per unit cost. Therefore, bundling together a cluster of workstations into a parallel system would seem to be a straightforward solution for computational tasks that are too large for a single machine. However, conventional communication mechanisms and protocols yield communication latencies that make only very large grain parallelism efficient. For example, typical parallel programming environments like PVM[BDG+93], P4[BL92] and MPI[CGH94] have latencies of several milliseconds. As a consequence, the parallel grain size necessary to achieve acceptable efficiency has to be in the range of tens of thousands of arithmetic operations.

We developed new hardware that efficiently implements communication operations and provides low communication latencies, minimal protocol, no operating system overhead, and high throughput. Our current design is capable of performing basic communication operations with a total process to process latency of just a few microseconds (i.e., $5\mu s$ for a 32bit transfer). Even parallel supercomputers have latencies of tens to hundreds of microseconds. Compared to workstation clusters using standard communication hardware (e.g., Message-passing software like PVM using Ethernet/FDDI hardware), our adapter shows performance improvements of more than two orders of magnitude on communication benchmarks. As a result, application benchmarks (i.e., LINPACK equation solver and others) execute with nearly linear speedup on a wide range of different problem sizes.

We are currently working on the second generation of our communication adapter. We expect a process to process latency below $1\mu s$ and a throughput of about 20 MByte/s per link. With this technology, it is possible to build effective parallel systems using off-the-shelf workstations. Our final goal is to offer a standard interfaceboard, capable of building parallel systems ranging from a few workstations or PC's to a cabinet full of workstation-boards.

Related approaches are discussed in section 2. Sections 3 and 4 describe our hardware and software approach in detail. The ParaPC prototype is introduced in section 5 and benchmark results are presented in section 6.

2 Related Work

There are several approaches with related goals targeting low-latency and high throughput parallel computing on workstation clusters. A special issue of IEEE Micro (Feb. 95) presented most advanced ones.

MINI (Memory-Integrated Network Interface) [MBH95] targets a 1-Gbps bandwidth with 1.2 μs latency interconnect using an ATM network. Communication in MINI is based on Channels between participating processes using ATM's virtual channel concept. Presented performance figures – ATM cell round-trip time of 3.9 μs at 10Mbytes/s – are based on VHDL simulations; hardware development is in progress.

SHRIMP (Scalable High-Performance Really Inexpensive Multiprocessor) [BDF+95] supports virtual-memory-mapped communication, allowing user processes to communicate without expensive buffer management and without system calls across the protection boundary separating user processes from the operation system kernel. Using Pentium PCs as platform, the network interface is connected to an EISA-Bus (SHRIMP-I) and the Xpress memory bus (SHRIMP-II). A 16-node (SHRIMP-I) and a 2-node prototype (SHRIMP-II) is expected to be operational in 2Q/95. No performance data about latency and throughput was given.

Myrinet [BCF+95] is a new type of local area network based on technology used for packet communication and switching within massively parallel processors. Measured performance using Myrinet API functions achieve one-way, end-to-end rates of 250 Mbps on 8-Kbyte packets.

Von Eicken et al. adopted their Active Messages [vECGS92, vEBB95] communication architecture developed for the CM-5 to a Sun workstation cluster interconnected by an ATM network. The prototype implementation shows an peak bandwidth of 7.5 MByte/s and a round-trip latency of 52 μs.

The Berkeley NOW (Network of Workstations) project [ACP95] targets 100+ workstation clusters using off-the-shelf components. One initial prototype is a cluster of HP9000/735s using an experimental Medusa FDDI interface. The final demonstration system will use either a second-generation ATM LAN or a retargeted MPP network, such as Myrinet. No performance figures are available at the time of this writing.

Like Myrinet, our network was originally designed for a MPP System (Triton/1)[1] and is now retargeted to a workstation cluster environment. In contrast to all other approaches, we focus on a pure message passing environment rather than a virtual shared memory. As von Eicken at al. pointed out [vEBB95], recent workstation operating systems do not support a uniform address space, so virtual shared memory is difficult to maintain. The major difference, however, is that our prototype adapter is operational and performance figures are based on measurements on real benchmarks running on a real system.

3 Communication Hardware

Using standard communication hardware (i.e.: Ethernet, FDDI, ATM) as communication links for parallel programming on workstation clusters suffers from some or all of the following problems:

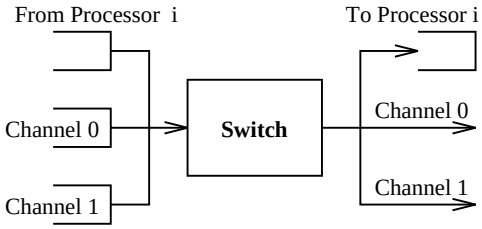
- Unreliable transport mechanisms. Data can be lost during transmission, forcing the protocol to support buffering, handshaking and retransmission. This increases latency and decreases throughput.
- Fixed or inappropriate packet size. Sending small messages within a fixed-size or large packet wastes time and bandwidth.
- Overdesign at the link level. Parts of the related protocols are implemented in hardware. For a parallel application however, only a part of the link-level packet header contains necessary information. All other information to be provided counts as overhead for the application.

[1] Triton/1 is a mixed mode SIMD/MIMD architecture, built with 256 processing elements at the university of Karlsruhe [PWTH93, HWTP93]

- Unscalability of network topologies. Available bandwidth on buses and rings is shared among the connected stations, so connecting more stations results in less bandwidth per station.

To overcome these deficiencies, we took – with slight modifications – the Triton/1 network, which can be characterized as follows:

The network topology is based on an two-dimensional torodial mesh. For small systems a ring topology is sufficient. Data transport is done via a table-based, self-routing packet switching method, which uses virtual cut-through routing. Every node 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 temporally decouples the network 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 in the range of 1 to 508 bytes. Packets are delivered in order and no packets will be lost.



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

The current prototype implementation of our communications processor involves a routing delay of about $1.2\mu s$ per node and offers a maximum throughput of 10 MByte/s per link. The new prototype board will offer a routing delay of about 600ns per node and a maximum throughput of 20 MByte/s per link. Additionally, the new interfaceboard will provide a hardware mechanism for fast barrier synchronization.

4 Communication Software

Message-passing environments on workstations like MPI, PVM, P4 [CGH94, BDG+93, BL92] and others suffer from some or all of the following problems:

- Operating System overhead. Communication hardware is controlled by the operating system. Sending or receiving a message implies at least one system call and copying the message buffer between user- and kernel-space. This takes more time than transmitting the message, especially for small messages and leads to large latencies and lower throughput.

- Several parallel applications per processor. Running several parallel application on a workstation cluster results in scheduling and synchronization delays. Execution time of each application takes much longer than running the applications one after another.

- Several parallel threads per process. Most message-passing environments allow multiple threads of one application running on one processor. Thus the operation system or the message-passing library has to handle the relationship between incoming messages and associated threads.

- Using decoupled send/receive primitives to exchange messages. Sending large messages, which cannot be buffered by the network, implies automatic message acception and buffering at the destination node, unless the receiving process is setup for receipt. Otherwise, the application would run into a deadlock. Automatic message acception and buffering involves interrupt handling and copying the message between kernel- and user-space which counts as overhead.

We intend to minimize overhead as far as possible to speed up message transfer to its maximum. This includes no operation system overhead, no interrupt handling and no buffering of messages. To avoid operating system overhead, all interfacing to the hardware is at user-level. A packet simply contains the address of the target node, the number of data words contained in the packet, and the data itself. Code for sending/receiving small messages (i.e. word transfers) is inserted as inline assembler macros directly into the application code. For large messages, we use a highly optimized library function, which runs at user-level and needs no system call. While sending a message, data is copied directly from user-space memory to the interfaceboard and the receiving function does the same thing vice versa eliminating all intermediate buffering. To avoid interrupt handling and buffering while exchanging large messages, special `exchange` functions are provided rather than using send/receive for the same purpose. These functions send and receive messages simultaneously, so the application cannot deadlock. Multiple parallel applications on the workstations cluster are supported by handling the execution order within the start-up code of each application. In case there is another parallel application running, the second one is put in a queue and execution is delayed until the running application has finished. Having multiple threads per parallel process on one node can simply (and efficiently) be done by simulating them in software using `for`-loops.

5 ParaPC Prototype

Our current prototype testbed consists of two 33 MHz, EISA-Bus based Intel 486 PC's connected through two ParaPC interfaceboards. We use BSD/OS V2.0 (BSD 4.4 from BSDI) as operation system. The ParaPC interfaceboard is a slight modification of a board used to connect the PC to the Triton/1 network. Modifications were simple, but has lead to some deficiencies: The board is only accessible through I/O-commands (i.e.: `in` and `out` instructions of the 486) making fast DMA transfer impossible. Lack of DMA causes the bottleneck in the current prototype design. Second, to interface to the Triton/1 network we needed only one of the two possible network links as shown in section 3. Thus, we can only build a ring as network topology with the prototype – but for a two workstation configuration this is no limitation. Our new PCI-Bus based design will remove these deficiencies and will support fast DMA transfer as well as two-dimensional torodial mesh as network topology.

Although having to live with these problems, it was possible to get reasonable performance. Using carefully designed assembler routines and some EISA-Bus specific features (automatic 32 to 16 bit translation), communication latencies of 5.5 μs and transfer rates up to 4.8MByte/s could be achieved. Bypassing the operation system in BSD/OS is very simple by using the `ioport` command to allow user-level access to all registers of the interfaceboard. Bear in that all performance figures presented in the following sections are performed on the relatively slow EISA-Bus interfaceboard and **not** on the new PCI-Bus design.

6 Benchmark Results

Our benchmark suite consists of several benchmarks and applications collected from literature. For each problem of the communication benchmarks (ping, pingpong [NN94] and pairwise exchange), we implemented the same algorithm using P4/Ethernet and ParaPC. Then we measured the runtime respectively communication latencies using two PC's[2] connected both via ethernet

[2] In lack of more ParaPC interfaceboards we are currently limited to two PC's.

and the ParaPC interface. Detailed results for all communication benchmarks are presented in section 6.1. To evaluate speedup and efficiency, we also implemented two application benchmarks – heat diffusion and LINPACK [Don95]. One version of each application benchmark is running on a uniprocessor system, while the parallelized version is running in parallel on two PC's using the ParaPC interface. We use the UNIX clock function (`getrusage()`) to determine the sum of system and user time. The results of these benchmarks can be found in section 6.2 and 6.3.

6.1 Communication Benchmarks

6.1.1 Ping

The purpose of the *Ping* benchmark is to measure communication latency as well as the effective bandwidth of ParaPC by sending messages from one processor to another. The sender keeps sending data unless it is blocked, and the receiver keeps consuming data. This communication pattern is common to many types of parallel applications and is used to distribute and collect data between the involved processors.

The algorithm for *Ping* is straightforward. Sending out a message is one iteration of the *Ping* sender. We measure the elapsed time of k iterations, and compute the average delay accordingly.

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

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

message size in bytes	latency in μs	throughput in MByte/s	message size in bytes	latency in μs	throughput in MByte/s
word transfer			block transfer		
1	5.50	0.181	4	6.52	0.612
2	5.51	0.362	8	7.31	1.092
4	5.80	0.685	16	8.95	1.786
8	6.99	1.138	32	12.29	2.596
			64	19.10	3.348
			128	32.47	3.937
			256	59.57	4.292
			508	112.81	4.499

For small message sizes, we archive transmission latencies (user- to user-process on UNIX) as low as $5.5\mu s$, which includes the send/receive code and the surrounding loop as overhead – exactly as shown in the code fragment. For larger message sizes, using a block transfers, we get a total throughput of up to 4.5 MBytes/s. Increasing the message size does not increase the latency by the same factor. Overhead decreases for larger messages and the code to send a message is somewhat faster than to receive a message, which causes the sender to run ahead of the receiver. The decoupling of sender and receiver is done by the network interface and its buffers.

[3] 508 bytes user data is the maximum packetlenght of the ParaPC interface.

6.1.2 PingPong

The *PingPong* benchmark is aimed at measuring the end-to-end delay by sending a message back and forth between two processes. Unlike *Ping*, the processor takes turns to become sender and there is only one sender at a time. This implies that sender and receiver are synchronized on each message transfer and results in a *worst case* scenario for exchanging messages.

Process1:	**Process2:**
measure start-time;	measure start-time;
DO i = 1,k	DO i = 1,k
send(message)	receive(message)
receive(message)	send(message)
ENDDO	ENDDO
measure stop-time;	measure stop-time;
calculate latency and throughput;	calculate latency and throughput;

We perform our *PingPong* benchmark with the same scenario as the *Ping* benchmark. We use message sizes from 1 to 508 and all runtimes were measured with $k = 10^6$ iterations.

message size in bytes	latency in μs	throughput in MByte/s	message size in bytes	latency in μs	throughput in MByte/s
word transfer			block transfer		
1	18.04	0.110	4	25.55	0.312
2	18.05	0.221	8	28.34	0.564
4	18.79	0.425	16	34.68	0.927
8	26.18	0.610	32	45.85	1.393
			64	70.63	1.811
			128	119.36	2.141
			256	215.77	2.370
			508	406.40	2.496

Message transfer times are – as expected – about 3.5 times slower compared to the *Ping* benchmark. A factor of 2 is obvious, because twice the amount of data has to be transferred and the remaining factor of 1.5 is due to implicit synchronization between sender and receiver on each message transfer. So both sender and receiver face the complete transmission overhead.

To compare our interface (hardware and software) to standard communication mechanisms, we have implemented the *PingPong* benchmark using P4 message passing software. The P4/Ethernet implementation shows latencies of $4.3ms$ for small messages and is therefore about 240 (!) times slower compared to the ParaPC interface. Comparing throughput performance difference is somewhat better – about 850kbyte/s vs. 2.5Mbyte/s for large messages – but this is caused by the maximum ethernet throughput of 1Mbyte/s.

6.1.3 Pairwise Exchange

The *Pairwise Exchange* benchmark is aimed to measure the end-to-end delay. Two processes send a message to each other simultaneously, and then receive simultaneously. Unlike *PingPong*, process two does not wait for receipt of a message before transmitting. This is a more practical scenario for two processes exchanging messages.

Process1:	**Process2:**
measure start-time;	measure start-time;
DO i = 1,k	DO i = 1,k
send(message)	send(message)
receive(message)	receive(message)
ENDDO	ENDDO
measure stop-time;	measure stop-time;
calculate latency and throughput;	calculate latency and throughput;

As above, we increased the message size from 1 to 508 bytes and used $k = 10^6$ iterations for each size.

message size in bytes	latency in μs	throughput in MByte/s	message size in bytes	latency in μs	throughput in MByte/s
word transfer			block transfer		
1	10.17	0.196	4	15.26	0.523
2	10.17	0.393	8	16.10	0.992
4	10.57	0.756	16	19.06	1.676
8	15.26	1.047	32	24.78	2.579
			64	36.06	3.546
			128	60.78	4.207
			256	111.26	4.597
			508	210.94	4.811

With simultaneous sending and receiving messages, the resulting runtime is about twice as long as to the *Ping* benchmark, because twice the amount of data has to be transferred. *Pairwise Exchange* actually shows even better maximum throughput than *Ping* (4.8 MByte/s vs. 4.5 MByte/s). Both processes start with sending a message, so there is no delay for receiving the first message.

6.2 Heat diffusion

The *Heat diffusion* benchmark starts with an even temperature distributions on a metal plate. On all four sides different heat sources and heat sinks are asserted. The goal is to compute the final heat distribution of the metal plate. This can easily be done with a Jacobi- or Gauss-Seidel iteration, by calculating the new temperature of each gridpoint as average of it's four neighbours.

Parallelizing this algorithm is simple: We use a block-distribution of rows of the $n \times n$ matrix. So during each iteration each process has to exchange two rows with its neighboring processes. To visualize the progress, all data is periodically collected by one process. The following table shows the effective speedup for different problem sizes. Each experiment was measured with at least 1000 iterations, visualizing the result every 20 iterations.

size (n)	time for 1 workstation [ms/iter]	time for 2 workstations [ms/iter]	speedup
32	5.16	3.40	1.52
64	19.91	11.07	1.80
128	79.96	41.77	1.91
256	330.96	164.47	2.00
512	1371.30	699.15	1.96
1024	5526.37	2841.46	1.94

As expected, execution time on uniprocessor and multiprocessor configuration quadruples as problem size is doubled. This is obvious, because the asymptotic work of a Jacobi-iteration on a $n \times n$ matrix is $O(n^2)$. For a wide range of problem sizes (starting with n=128 up to n=1024), an ideal speedup and therefore an efficiency of more than 95% is achieved. Only when benchmarking very small problem sizes, we see a decreasing speedup because the overhead for collecting and visualizing the data is quite large compared to computational task within the Jacobi-iteration.

6.3 Linpack

The *Linpack* benchmark is an equation solver using LU decomposition with partial pivoting and backsubstitution. We have parallelized the core routine (SGEFA). We use a cyclic distribution of lines to achieve an optimal load balance. Thus, pivot searching, scaling and row elimination can be done in parallel.

The following table shows runtime, achieved Mflops and effective speedup for different problem sizes.

size (n)	1 workstation time[s]	1 workstation Mflop	2 workstations time[s]	2 workstations Mflop	speedup
100	0.39	1.76	0.22	3.14	1.78
200	2.96	1.80	1.57	3.45	1.92
500	46.1	1.82	23.1	3.62	1.99
1000	368	1.82	183	3.64	2.00
1500	1255	1.80	627	3.60	2.00

The measured performance in Mflops of the uniprocessor configuration is quite stable over the whole range of different problem sizes and compares well to results presented by J. Dongarra [Don95] for a 33MHz 80486. Using two workstations we obtain a perfect speedup (greater than 1.98) and therefore an efficiency close to the maximum for all relevant problem sizes.

7 Conclusion

The integrated and performance oriented approach of designing fast interconnection hardware and a message-passing library has lead to a workstation cluster environment that is well suited for parallel processing. With low communication latencies, minimal protocol, and no operating system overhead it is possible to build effective parallel systems using off the shelf workstations. Our final goal is to offer a standard interfaceboard, capable of building parallel systems ranging from a few workstations or PC's up to a cabinet full of workstation-boards.

Status and schedule
Prototype boards based on EISA-Bus are operational. Currently we are designing new PCI-Bus boards, which will be manufactured and tested in summer 1995. Our next testbed will consist of an eight DEC-Alpha workstation cluster which should be running by December 1995.

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.

[BCF+95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jarov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[BDF+95] Matthias A. Blumrich, Cezary Dubnicki, Edward W. Felten, Kai Li, and Malena R. Mesarina. Virtual-Memory-Mapped Network Interfaces. *IEEE Micro*, 15(1):21–28, February 1995.

[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.

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

[Don95] Jack J. Dongarra. The Complete Linpack Report. Technical report, University of Tennesse, January 95.

[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.

[MBH95] Ron Minnich, Dan Burns, and Frank Hady. The Memory-Integrated Network Interface. *IEEE Micro*, 15(1):11–20, February 1995.

[NN94] Natawut Nupairoj and Lionel M. Ni. Performance Evaluation of Some MPI Implementations on Workstation Clusters. Technical report, Department of Computer Science, Michigan State Univerity, October 94.

[PWTH93] Michael Philippsen, Thomas M. Warschko, Walter F. Tichy, and Christian G. Herter. Project Triton: Towards improved programmability of parallel machines. In *26th Hawaii International Conference on System Sciences*, volume I, pages 192–201, Wailea, Maui, Hawaii, January 4–8, 1993.

[vEBB95] Thorsten von Eicken, Anindya Basu, and Vineet Buch. Low-Latency Communication Over ATM Networks Using Active Messages. *IEEE Micro*, 15(1):46–53, February 1995.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19st Annual International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.