An Operating System Support to Low-Overhead Communications in NOW Clusters

P. Marenzoni, G. Rimassa, M. Vignali, M. Bertozzi, G. Conte[1] and P. Rossi[2]

[1] Dipartimento di Ingegneria dell'Informazione,
Università di Parma, Viale delle Scienze
I-43100 Parma, Italy
[2] SMART S.r.l., Via dell'Artigianato 31/2,
I-40057 Granarolo Emilia (BO), Italy

Abstract. This paper describes an Operating System approach to the problem of delivering low latency high bandwidth communications for PC clusters running a public domain OS like Linux and connected by standard, off-the-shelf networks like Fast-Ethernet. The $PARMA^2$ project has the main goal of designing the new light-weight protocol suite PRP, in order to drastically reduce the software overhead introduced by TCP/IP. PRP wants to offer at high level a stream socket oriented interface and at low level compatibility with any device driver. High level compatibility is crucial in facilitating the porting on PRP of existing applications or message passing packages. Moreover, an optimized version of MPI, based on PRP and evolution of the widespread MPICH implementation, is under development, allowing for a very effective reduction of the communication latencies in synchronous communications, compared to the TCP/IP-based MPI.

1 Introduction

Today the use of workstation or PC clusters as platforms for parallel processing is widely spreading [1], often adopting off-the-shelf network solutions. Significant experiences have born recently regarding both the implementation of high performance computing applications on PC or workstation networks [2, 3, 4] and the development of hardware/software components dedicated to speedup communications at the performance level achieved by the parallel machines with special purpose networks [5, 6, 7]. The least effort implementation of a NOW cluster as a parallel computing platform relies on OS services such as the TCP/IP protocol. This has an obvious disadvantage, mainly the high latency associated to the requirements that TCP enforces in order to work in almost all conditions across highly heterogeneous platforms and WANs. The great advantage is that one is automatically endowed of a truly multi user, reliable programming environment.

At the onset of our PARMA PARallel MAchine ($PARMA^2$ in the following) project we decided that the multi user environment had to be preserved by all means, but we did not think necessary to maintain every feature needed by a platform with heterogeneous nodes or by wide area networks with different data link level protocols. In fact, $PARMA^2$ is meant to be a homogeneous PC cluster connected by Fast-Ethernet and running a standard operating system as Linux, whose source code availability allows kernel-level interventions to be made.

Within these goals the first thing to assess is the viability of the idea, that is whether it is possible to design a protocol with a sufficiently low latency to make the project meaningful. This is the primary scope of PaRma Protocol (PRP). The requirement of multi user imposes a heavy burden on PRP, that is to be seamlessly integrated with OS kernel and to cope with overheads associated to interrupt handling, memory management and so on. On the other hand, having limited $PARMA^2$ scope to a homogeneous LAN, we can conceivably design a greatly simplified protocol with respect to TCP/IP.

The solution we propose for integrating PRP within the kernel is the standard one for all network protocols under Unix, that is insertion between socket layer and physical layer (i.e. device driver). This solution offers immediately the added advantage that all of the parallel computing paradigms, developed on socket interfaces, will be effortlessly ported to PRP. In its first implementation PRP has to assess what level of latency is attainable while fulfilling simultaneously all the design constraints. To achieve this goal we realized a stripped down solution, with no mechanism associated to flow control, data recovery etc. (that could be introduced in a light form into the final protocol). The results obtained are extremely encouraging given that we halved TCP/IP latencies and substantially improved bandwidths.

Despite the fact that current PRP release is by no means a complete protocol, in practical applications we observed very few situations where it delivers incorrect results. This validates our optimistic protocol implementation that limits itself to error detection: in fact the tests on real applications described in this paper are performed in this fashion.

Moreover, since the typical discrepancy between latencies visible at higher level (MPI or PVM) are about one order of magnitude greater in NOW or PC cluster environments than in MPP machines with dedicated communication libraries, $PARMA^2$ aims at developing an optimized version of MPI (MPIPR) finely tuned for an homogeneous local NOW cluster running PRP and based on the MPICH implementation by Argonne National Laboratory.

The current $PARMA^2$ implementation is based on a set of PCs connected by a Fast-Ethernet 100 Mbps LAN. Each node comprises an Intel Pentium 100 MHz CPU, equipped with a PCI mother board, 512 KB of secondary cache and 32 MB RAM. The Fast-Ethernet network adapter is a 3COM 3C595-TX. Each PC runs Linux version 2.0 and supports the gcc and g77 GNU compilers.

2 The *PRP* protocol of $\boldsymbol{PARMA^2}$

UNIX computing environments are characterized by pervasive, smoothly integrated networking capabilities [9]. Many application interfaces (i.e. many sets of *system calls*) have been devised and built, and BSD sockets are nowadays probably the most successful and popular among them, besides being the only one actually implemented in Linux. They support both client/server model (using `connect()/accept()` pair) and peer-to-peer model (using `read()`, `write()`, `recvfrom()` and `sendto()` system calls) and have a plethora of ready-to-run applications based upon them. Programmers can select a socket address family using a special parameter when invoking `socket()` and `bind()` system calls. They use `AF_INET` when setting up network wide, TCP/IP based, internet domain sockets. Subsequent system calls will be automatically redirected to the correct handler, thus providing a uniform and consistent programming interface to interprocess and interprocessor communications. Focusing only on interprocessor communications, which give birth to real networking, one finds network protocol as the software layer immediately under socket address family (by the way, socket domains can be seen as a demultiplexer which routes messages to the proper protocol): Linux, as a UNIX OS, ships along with the complete IP suite support but, as a PC-hosted OS, also has Novell's IPX and Apple's Appletalk capabilities.

The main purpose of PRP project is maximum compatibility with all socket-based Unix applications and message passing interfaces, preserving ordinary Linux functionalities and avoiding the oversized TCP/IP protocol suite [10]. Therefore, a new "light-weight" protocol suite has been inserted in the existing Linux architecture, resulting in two new independent software layers (corresponding to standard network and transport OSI layers). Thus, the OS can continue to perform its functionalities and coordinate the processor activity in a multitasking environment.

2.1 The Linux network architecture

Network protocols are seen by Linux OS as homogeneous subsystems which are supposed to coexist painlessly and can even be added or removed dynamically during normal operations, much in the same way as device drivers and filesystems are (this is accomplished by installable modules, ordinary object files that can be linked in by the kernel without the need to reboot the computer). To ease this task Linux provides well defined hooks and interfaces to the system programmer.

Linux networking structure is composed by a unified application interface, that is socket family, by a set of independent network protocols, by a generic device driver interface and finally by several device drivers for the network adapters present in the system. Since a network operation involves every system layer, each of them maintains a suitable data structure representing its own view of the same data: the structures that accomplish this are, starting from top to bottom, `struct socket`, `struct sock` and `struct sk_buff`. The first two elements describe the connection at socket family and network protocol levels, whereas the third one is an abstraction for a data packet. In particular, `struct sk_buff` must be directly handled by device drivers, so it must reside in a contiguous memory area. Therefore, various functions are provided to insert and remove data at the start or at the end of the buffer, which is allocated all at once.

A kernel level implementation must deal with several facets of the operating system: among them we recall asynchronous events, such as interrupts and context switches, and memory protection with multiple address spaces. When it comes to networking, Linux OS must deal with three address spaces: the network adapter I/O space, the kernel and the user address spaces. Two predefined functions exist to copy data across different address spaces, namely `memcpy_fromfs()` and `memcpy_tofs()`. To obtain maximum performance it would be nice to directly copy messages between user and network card spaces. This approach has been successfully employed in [8]. While this is theoretically possible during send phase, it is not feasible in the callback phase as long as PRP approach not to interfere with process scheduling is maintained. The callback must be regarded as an interrupt handler which cannot assume that the destination process is running and cannot access its user memory. Furthermore, to be completely hardware independent, a network protocol must deliver to device drivers a regular buffer in kernel space. All these considerations have brought to following implementation solutions:

- use of the predefined memory management functions to move data between user and kernel space;
- delegation to network device drivers of the actual copy to and from the I/O space;
- exclusive usage of the `struct socket`, `struct sock` and `struct sk_buff` structures to hold the information about the connection, so that it can be exploited in the callback phase and the protocol code is completely reentrant.

2.2 *PRP* characteristics

The PRP protocol implementation relies on the basic assumption that the *local* homogeneous interconnection network assures correct delivery of the packets. In other words, we assume that the percentage of packets incorrectly delivered is negligible and the software layer dedicated to error recovery can be discarded. While TCP/IP implements both flow control mechanisms and error control coding, PRP only worries about lost packets, because Ethernet cards automatically discard corrupted packets, exploiting Ethernet CRC field. This way a corrupted packet is turned into a missing one and error detection falls into lost packet detection, verifying whether segments arrive at the destination in exact order and returning an error message when appropriate. This choice has been strengthened by the fact that all common user-level MPI-based applications typically running in our cluster never caused protocol failures. Only intensively communicating custom tests have detected problems related to packet loss.

Therefore, typical flow control mechanisms and acknowledge-retransmission schemes needed by a general purpose, WAN oriented protocol as is TCP/IP are currently not present inside PRP. However, we are studying the impact on performance of some simple packet loss recovery implementations (go-back n vs. selective repeat), which we plan to insert into PRP's next releases. The complex reliable three-way handshake phase present inside TCP/IP has been simplified too, resulting in a straightforward two-way connection setup phase. Besides, PRP can be installed as a module inside all Linux kernels starting from 2.0.0 version.

2.3 The *PRP* Socket Interface

The protocol is implemented as a new family for the high level socket interface and can be called by setting the socket family parameters in the system calls to `AF_PRPF`. The whole set of system calls available in a standard Unix environment for networking and socket management has been implemented from scratch also for PRP. The new structure `sockaddr_prp`, available in the suitable `prp.h` header file, is equivalent (with the family, address, and port fields) to standard `sockaddr_in` structure which supports `AF_INET` sockets. PRP supports up to 1024 port numbers, that can be automatically assigned by the OS. The `sockaddr_prp` structure is defined as:

```
struct sockaddr_prp {
  short int             sprp_family;     /* Address family    */
  unsigned short int    sprp_port;       /* Port number       */
  unsigned short int    sprp_addr;       /* PRP address       */
};
```

In a cluster supporting PRP, machine nodes are numbered with logical addresses from 1 to N_p, in such a way that IP addresses are unknown from inside the new protocol family. In PRP, besides logical node numbers, only Ethernet addresses are needed. The `prphosts` configuration file placed in the `/etc` directory contains the mapping between logical and hardware addresses. The network layer, responsible for converting addresses from logical to Ethernet back and forth, allows us to support any possible routing scheme the user wants to impose to machine configuration. Since PRP layers are completely independent of underlying ones, the protocol must know the device `ethx`, which packets are to be delivered to and collected from. This information also is stored in the configuration file.

The main socket features and parameters seen at application level and the state of the socket in use are stored, as done by other protocol suites in Linux, in the `struct socket`.

2.4 *PRP* Transport Layer

The PRP transport layer is responsible for performing packet segmentation, allocating and deallocating buffers to be sent or that have been received, coding and decoding header flags, copying data to/from user space from/to kernel space. Finally, being the last software level (starting from the application) knowing about both user processes and kernel buffers, PRP transport layer must identify the destination socket (if any) during the callback phase and wake up the receiving process when it's time to do so. Due to the absence of flow control and error recovery this layer does not implement retransmission timeouts nor packet queuing during send phase.

The very light-weight structure of PRP also results in a socket `AF_PRPF` to have only four states named `PRP_CLOSE`, `PRP_CONN_WAIT`, `PRP_LISTEN`, and `PRP_ESTABLISHED`. The two-way handshake phase taking place upon starting a connection is as follows: the client requesting the connection sends a special packet to the server, which in turn responds sending a final acknowledge packet. After a `listen()/accept()` system call pair server's socket goes from `PRP_CLOSE` to `PRP_LISTEN` state. On the other side, the socket requesting the connection goes to `PRP_CONN_WAIT`. After this simple handshake phase, both sides move to `PRP_ESTABLISHED` state.

The `tl_prp_sendmsg()` function applies a segmentation on user buffer using up to the maximum Ethernet size (1484 bytes of data) for each segment; for each packet a suitable socket buffer `sk_buff` is allocated. The header added to each segment contains information on source and destination ports (coded in 16 bit integers), the sequence number of the segment (coded in 16 bits), the packet length (taking other 16 bits), and finally the flags (32 bits). Flags identify a condition of *connection request* when a process makes a `connect()` system call, a *connection acknowledge* by the server side when establishing the connection, an *end of message* when sending the last packet of a sequence, and the *end of connection* request. After adding the header, we are ready to append user data to kernel buffer, through the `memcpy_fromfs()` function, and finally to forward the packet to network layer.

During callback phase the `tl_prp_rcv()` is the last function executed. It must extract the header information and, on the basis of the port number, identify the destination `sock` which must handle the packet. The header flags are checked and different routes are taken according to socket state and flag values. For example a data packet arriving in a `PRP_ESTABLISHED` condition is enqueued to the list of received buffers. An out-of-sequence data packet, instead, is automatically discarded, in agreement with our error detection policy. The last task the callback must accomplish is the wake-up of the processes waiting for the message. The `tl_prp_recvmsg()` function (called by socket family layer in a `read()` system call) is held responsible for dequeuing received packets from socket list and copy data from `sk_buff` to user supplied buffer, with a `memcpy_tofs()` call.

2.5 *PRP* Network Layer

During send phase the PRP network layer must identify Ethernet addresses and devices to which packets must be delivered. On the other hand, during callback, it must verify if the local node is the destination of the arriving packet, otherwise it must redirect the packet to another device for a new destination (that is packet routing).

The `nl_prp_send()` function adds its own (very simple) header to the buffer coming from upper levels. This header is made by two 16 bit integers storing source and destination logical addresses of the connection endpoints. A suitable kernel routing table structure stores the mapping between logical and Ethernet addresses and the device associated to each destination node. After that, the network layer function is ready to deliver the `sk_buff` to the lower level generic interface of the corresponding `ethx` device, filling Ethernet address fields.

3 The dedicated MPI implementation

Almost every parallel architecture supports a custom MPI version, developed directly on the basis of the hardware/software machine specifics, with the main purpose of achieving maximum efficiency. Therefore, one among the main goals of $PARMA^2$ project is to develop also a dedicated version of MPI, evolution of the MPICH implementation. MPIPR library is mainly devoted to greatly simplify the internal protocol used by the MPI `ch_p4` device [17], in order to dramatically decrease software latencies. Until now only *synchronous* MPIPR primitives have been implemented, as they mainly affect the performance of distributed applications.

The MPICH implementation of MPI adopts a useful layered approach [18]. The *channel interface* is responsible for providing all "high level" data transfer operations and relies by default on the `ch_p4` device layer, which in turn exploits the standard Unix socket facilities. The interface implements its own data exchange protocols and data management mechanisms. Messages are sent in two parts. The control message stores the information about message tag, length, etc. The data part stores user message actual data. Specific routines are designed to send and receive the two message parts, and to check the presence of incoming messages. In case of small user messages sending two packets can determine a substantial latency overhead. Therefore, packets with size below a preset cut-off are sent together with the control part. The default cut-off value is 1024 bytes. The `ch_p4` device derives from the preexisting `p4` parallel programming system [17], developed at Argonne National Laboratory too. Its protocol imposes that, during a send operation, a suitable 40 byte header is built, with all necessary information (source, destination, message-type, length, etc.) and sent apart from data message, in a separate socket `write()` system call. This protocol assures the correct buffer allocation strategy at the receiver side, since message characteristics are available before real data arrives, but on the other hand it doubles the overheads.

Our intervention has concerned the substitution of this layered structure with a single interface, with its own internal protocol, the main purpose being the elimination of duplicate communication latencies, at least for small packets. The new internal protocol assembles the 48 bytes message header needed to store all necessary information and then sends the whole packet (header and data) to the destination. At receiver's side standard stream socket features are exploited in order to correctly scan incoming message headers until the one matching requested criteria is found. Stream socket allow in fact to read only the header packet portion, apart of the rest of the message. This permits to allocate a suitable buffer and then to read remaining message data. The new MPIPR layer relies of course upon the PRP protocol instead of the standard TCP/IP family.

We are also completing the implementation of an optimized version of asynchronous MPI primitives, in order to realize a self-consistent PRP-based MPI version.

4 Performance measurements

A key experimental ingredient, when evaluating quantities as small as latency times (expected to be 10^{-4} seconds or less), is an accurate timing measurement. This is accomplished by means of the TSC Pentium register, which is incremented every clock tick. By reading TSC content before and after the execution of a piece of code we are able to measure the exact execution time of that code with a 10 nanoseconds resolution (on a Pentium 100 CPU).

Latency and bandwidth are parameters commonly used to characterize communication performance [11, 12]. Despite the fact that bandwidth is a much heralded feature of interconnection networks, from a minimal experience in the application field it results, quite obviously, that most of the damage is caused by latency. Almost any reasonable bandwidth turns out to be quite acceptable, while latency can be and often is an unbearable bottleneck. Besides, bandwidth is pretty much bounded by hardware features of the system at hand, while latency is given by a complex interplay of software and hardware to the point that becomes handy to distinguish between network hardware contribution, software overhead introduced by the CPU to perform a network operation, and CPU hardware contribution, in turn made up by memory to memory copies and back and forth memory to device copies. The sum of the various types of overhead is the meaning of latency we like to endorse, in order to relate it to the cost we pay in executing real application code, that is the overhead related to memory to memory delivery of a zero length message.

It must be pointed out that Ethernet allows frame sizes ranging from 60 bytes to 1514 bytes, so that every packet smaller than 60 bytes (including headers) is actually padded by the network card, resulting in a constant network hardware overhead for a data size from 0 to 34 bytes. Let then N_{min} be the data size resulting in a 60 byte frame and be N_{max} the data size giving a frame of 1514 bytes; let also a be the overall constant communication overhead (i.e. a is the sum of the three terms independent of data size, each referring to a different latency contribution). Finally, let $1/B_N$ be the coefficient of the linear part (i.e. the one proportional to data size) arising from network hardware and let $1/B_{CPU}$ be the same for CPU hardware and software contributions. If we want to parameterize communication time T versus message size N we cannot use a single formula for the whole range, since network contribution has different expressions for data size greater or lesser than N_{min}:

$$T = a + \frac{N_{min}}{B_N} + \frac{N}{B_{CPU}} = L + \frac{N}{B_0} \quad 0 \le N \le N_{min}, \tag{1}$$

$$T = a + \frac{N}{B_N} + \frac{N}{B_{CPU}} = L' + \frac{N}{B_1} \quad N_{min} \le N \le N_{max}, \tag{2}$$

where $L = a + N_{min}/B_N$, $B_0 = B_{CPU}$, $L^{'} = a$ and $1/B_1 = 1/B_N + 1/B_{CPU}$ are the resulting parameters for the model.

	Latency (μs)	Bandwidth (MB/s)
PRP	74	6.6
TCP/IP	146	5.5

Table 1. Latency and bandwidth for the ping-pong operation.

The usual "ping-pong" application is employed to carry out our measurements. A standard socket-based implementation is adopted for ping-pong experimental setup with both TCP/IP and PRP, the only difference being **`AF_PRPF`** sockets used instead of **`AF_INET`** sockets. Experiments performed with very large messages give a measure of effective bandwidth, whereas relevant latency, as previously defined, is clearly L, measured through experiments conducted with message length going to zero. The ping-pong results taken on 100 MHz Pentium processors with Fast-Ethernet are reported in Table 1. PRP shows half overall latency overheads, if compared to TCP/IP, while the improvement of effective bandwidth is about 20%.

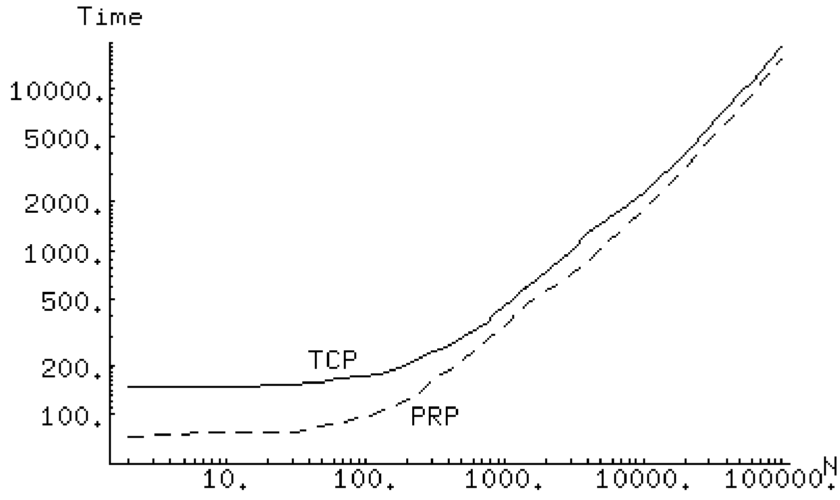

Fig. 1. Delay as a function of the message length N.

The behavior of transmission delay T as a function of message length N on a wide range of values (from 0 to 200 KB) is drawn in Fig. 1 for TCP/IP and PRP. Despite the significant difference between the two curves near the origin, measuring the difference in latency of the two protocols, we notice that they tend to narrow the gap asymptotically, offering further support to the fact that bandwidth is not much affected by TCP/IP. In order to address $PARMA^2$ scaling problem measurements have been carried out with increasing numbers of nodes (up to 16): as expected from a bus-based topology, a remarkable decrease in available bandwidth has been observed for ping-pong applications with eight or more PCs. In this heavy load conditions both PRP and TCP/IP performed equally, being the physical channel the main system bottleneck.

An usual countermeasure is breaking out network traffic using switches; since they work directly with Ethernet frames, they support every higher level protocol without distinctions. Alternatively, network topologies exclusively composed by point to point links (an hypercube is a typical example) are a feasible way to overcome scaling problems and are indeed supported by means of PRP static routing scheme.

5 $PARMA^2$ applications

Preliminary results allowed by MPIPR have been measured, in order to compare them with other assessed MPI implementations (the standard MPI, based on TCP/IP, and the MPI version simply adapted to PRP). The test is a ping-pong experiment, implemented using `MPI_Isend()` and `MPI_Recv()` primitives. Comparative results on two Pentium 100 processors are reported in Table 2, that report also commercial MPP platform results [13, 14, 15] with some available message passing libraries. On Cray T3D ping-pong results obtained using the *shared memory* configuration are also reported, giving much better results. Latencies with MPIPR are 2.2 times smaller than with MPI plus TCP/IP, allowing our environment to be competitive with parallel machines, at least within a factor of four or five, using only standard low-cost hardware and suitably modified public domain software. In particular, we are only two times slower than the CM-5, at the same application level.

	Latency (μs)	Bandwidth (MB/s)
CM-5 CMMD	93.7	8.3
T3D MPI	43.3	29.6
T3D SHMEM	1.5	58.5
SP2 MPI	44.6	33.9
MPI + TCP/IP	401.5	4.51
MPI + PRP	256	5.37
MPIPR	181.5	5.37

Table 2. Latency and bandwidth for the ping-pong operation using the three MPI versions available on the $PARMA^2$ platform.

The availability of the MPI interfaces, with TCP/IP and PRP, and the MPIPR dedicated version, allows us to measure how user applications can benefit from improvements obtained in communication latencies. We have chosen a common and widely used regular lattice problem [16], based on the Cellular Neural Network (CNN) computational paradigm, in order to implement a test where communication latencies can play a very important role.

CNNs [19] [20] are defined on discrete regular N-dimensional spaces. The basic element of the paradigm is the *unit* (or *cell*), corresponding to a point in the grid. Cells are characterized by an *internal state variable.* The main characteristic of CNN is the *locality* of the connections among the units: the most important difference between CNN and other Neural Network paradigms is the fact that information is directly exchanged only between neighboring units. From another point of view, it is possible to consider the CNN paradigm as an evolution of the Cellular Automata paradigm. A formal description of the discrete time CNN model is:

$$x_j(t_{n+1}) = I_j + \sum_{k \in N_r(j)} A_j[y_k(t_n), P_j^A] + \sum_{k \in N_s(j)} B_j[u_k(t_n), P_j^B] \tag{3}$$

$$y_j(t_n) = \phi[x_j(t_n)], \tag{4}$$

where x_j is the internal state of a cell, y_j is its output, u_j is its external (*control*) input and I_j is a local value (e.g. a threshold) called *bias.* A_j and B_j are two generic parametric functionals, P_j^A and P_j^B are the corresponding parameter arrays (typically the inter-cell connection weights). The y and u values are collected from the cells of the neighborhood N_r (for the functional A) and N_s (for the functional B). A and B are also called *templates.* The *activation function* ϕ, which generates the output from the internal state, can be typically a linear with saturation, a sigmoidal, a step, a quantizer or a Gaussian [19]. All data are represented as 32-bit floating-point numbers.

At each iteration, and for each lattice point, the CNN must perform the collection of the u and y neighbor values of the previous iteration, compute the A and B templates, then x is determined adding to I the A and B results. Finally, the activation function is applied to x to obtain the new value of the output y. For our purposes only the *linear* template subset of 2-D CNN is considered. Moreover, only periodic boundary conditions are implemented, as usually happens in many physics and engineering problems. In all tests a linear with saturation activation function ϕ has been used. In spite of its simplicity, the CNN formalism may express many complex problems or applications, for example in Image Processing, Field-Dynamics, and Theoretical Physics. An exhaustive list of many CNN applications can be found in [19].

The overall computational times required to execute 100 complete CNN iterations have been measured on $L \times L$ square lattices. Fig. 2 reports the overall performance (in M-Flop/s) measured with four $PARMA^2$ nodes (Pentium 100), as a function of L, using the general MPI version with TCP/IP and PRP and the dedicated MPIPR version. The graph emphasizes the significant performance

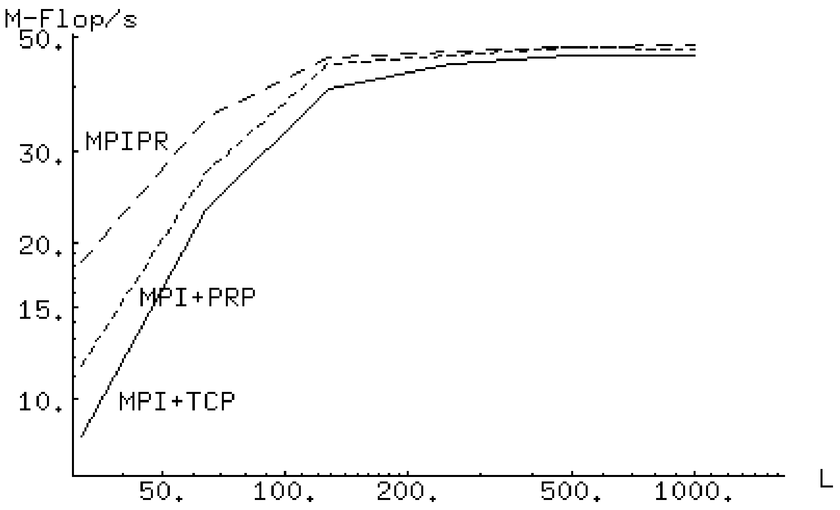

Fig. 2. Performance (in M-Flop/s) on four PCs for the CNN application, with the standard MPI version with TCP/IP and PRP, and the dedicated MPIPR version, as a function of the 2-D $L \times L$ lattices.

improvement (more than a factor of two) allowed by MPIPR on a real synchronous application when communications become a bottleneck, that is when frequently exchanging small packets. The difference with respect to the standard TCP/IP-based MPI version becomes asymptotically negligible, the application being computation bounded at this regime.

6 Lessons learned and future work

In this work a development suite supporting fast communications in PC clusters running Linux was presented. The PRP network protocol realized with the purpose of preserving complete OS functionality and maintaining the high level socket interface, although very simple, showed remarkable latency improvements with respect to TCP/IP. Moreover, the MPI custom implementation (MPIPR) furtherly encourages the $PARMA^2$ approach of comprehensive interventions in both OS kernel and user-level libraries.

Several annotations arose during software development. First of all, when operating with modern and fast networks the major latency contributions come from system and software overheads. In particular, every system component (i.e. device drivers, virtual memory management, and to a lesser extent - 33% - PRP code itself) yields a non negligible percentage of total communication time when sending very small packets, as is explained in Fig. 3. Furthermore, the effective bandwidth allowed by system hardware (bus architecture, cache memory hierarchy, CPU speed) is still far (53%) from Fast-Ethernet theoretical peak

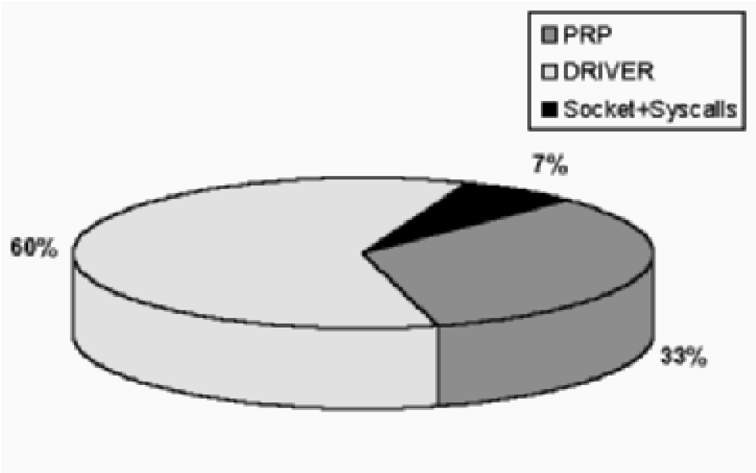

Fig. 3. Contributions to PRP latency.

performance. Having implemented a minimal protocol, it results quite evident that for both performance aspects a crucial role is played by the complex interaction between device driver and system bus. Therefore, future investigations and enhancements will concern on one hand testing improved device drivers, and, on the other hand, adding error control features to PRP protocol.

Finally, all performance advantages of PRP over TCP/IP measured with custom tests are gradually reduced by successive software layers (common socket interface and MPI library); in order to avoid masking out kernel level improvements it is necessary to extend upward software optimizations. This is another main direction in our future developments: completion of the dedicated MPIPR implementation and integration of enhanced higher level programming paradigms on top of MPI, like HPF as compiled by ADAPTOR.

Acknowledgments

The authors warmly thank Prof. G. Chiola and Dr. G. Ciaccio of Università di Genova for the helpful discussions had during early phases of their research.

References

1. G. Bell: 1995 Observations on Supercomputing Alternatives: Did the MPP Bandwagon Lead to a Cul-de-Sac ? Communications of the ACM **Vol 30** No. 3 (1996)
2. L. Colombet and L. Desbat: Speedup and efficiency of large size applications on heterogeneous networks. Proc. EURO-PAR96 (1996)
3. C. C. Lim and J. P. Ang: Experience on Optimization and Parallelization of Existing Scientific Applications on Network of Workstations. Proc. PDPTA96 (1996)

4. R. van Drunen, C. van Teylingen and M. Kroontje: The Amfisbaena: A Parallel Supercomputer System Based on i860 as a Generic Platform for Molecular Dynamics Simulations. Proc. PDPTA96 (1996)
5. M.A.Blumrich, K.Li, R.Alpert, C.Dubnicki, E.W.Felten, and J.Sandberg: Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. Proc. "International Symposium on Computer Architecture" ISCA94 (1994) 142–153.
6. T. Sterling, D. Savarese, B. Fryxell, K. Olson, and D. J. Becker: Communication Overhead for Space Science Applications on the Beowulf Parallel Workstation. Proc. "High Performance Distributed Computing" HPDC95 Pentagon City Virginia USA (1995)
7. H. Lu, S. Dwarkadas, A. L. Cox and W. Zwaenepoel: Message Passing Versus Distributed Shared Memory on Networks of Workstations. Proc. Supercomputing95 (1995)
8. G. Chiola and G. Ciaccio: GAMMA: a Low-cost Network of Workstations Based on Active Messages. Proc. "5th EUROMICRO workshop on Parallel and Distributed Processing PDP'97" London UK (1997)
9. W. R. Stevens: Unix Network Programming. Prentice Hall New Jersey (1990)
10. D. E. Comer and D. L. Stevens: Internetworking with TCP/IP. Prentice Hall New Jersey (1991)
11. Z. Xu and K. Hwang: Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2. IEEE Parallel & Distributed Technology **Vol. 4** No. 1 (1996) 25–42
12. R. W. Hockney: The Communication Challenge for MPP: Intel Paragon and Meiko CS-2. Parallel Computing **Vol. 6** No. 3 (1994) 389–398
13. J. J. Dongarra and T. Dunigan: Message-Passing Performance of Various Computers. Tec. Report ORNL/TM-13006 Oak Ridge National Laboratory (1996)
14. P. Marenzoni: Performance Analysis of Cray T3D and Connection Machine CM-5: a Comparison. Proc. Int. Conf. "High-Performance Computing and Networking HPCN95" Milan Italy Springer-Verlag LNCS **919** (1995) 110–117
15. P. Marenzoni and P. Rossi, Benchmark Kernels as a Tool for Performance Evaluation of MPP's,
Concurrency Practice and Experience, 1997, in press, John Wiley & Sons.
16. G. Destri and P. Marenzoni: Cellular Neural Networks as a General Massively Parallel Computational Paradigm. Special Issue on Cellular Neural Networks of "International Journal of Circuits Theory and Application" **Vol. 24** No. 3 (1996) 397–408
17. R. M. Butler and E. L. Lusk: Monitors, Messages, and Clusters: The p4 Parallel Programming System. Parallel Computing **Vol. 20** (1994) 547–564
18. W. Gropp and E. L. Lusk: MPICH Working Note: Creating a New MPICH Device Using the Channel Interface. Tec. Report Argonne National Laboratory
19. L.O. Chua and T. Roska: The CNN Paradigm. IEEE Trans. on Circuit and Systems - I **Vol. 40** (1993) 147–155
20. L.O. Chua and L. Yang: Cellular Neural Network: Theory. IEEE Trans. on Circuit and Systems **Vol. 35** (1988) 1257–1272

This article was processed using the LaTeX macro package with LLNCS style