Dipartimento di Informatica e
Scienze dell'Informazione

A Communication System for Efficient Parallel Processing on Clusters of Personal Computers

Giuseppe Ciaccio

Theses Series DISI-TH-1999-02

DISI, Università di Genova
v. Dodecaneso 35, 16146 Genova, Italy http://www.disi.unige.it/

Università degli Studi di Genova

**Dipartimento di Informatica e
Scienze dell'Informazione**

Dottorato di Ricerca in Informatica

Ph.D. Thesis in Computer Science

A Communication System for Efficient Parallel Processing on Clusters of Personal Computers

Giuseppe Ciaccio

June, 1999

Dottorato di Ricerca in Informatica
Dipartimento di Informatica e Scienze dell'Informazione
Università degli Studi di Genova

DISI, Università di Genova
via Dodecaneso 35
I-16146 Genova, Italy
http://www.disi.unige.it/

Ph.D. Thesis in Computer Science

Submitted by Giuseppe Ciaccio
DISI, Università di Genova
ciaccio@disi.unige.it

Date of submission: December 1998

Title:
A Communication System for Efficient Parallel Processing
on Clusters of Personal Computers

Advisor: Giovanni Chiola
DISI, Università di Genova, Italy
chiola@disi.unige.it

Ext. Reviewers:

Robert D. Russel
Computer Science Department, University of New Hampshire
Durham, NH 03824 USA
rdr@cs.unh.edu

Djamshid Tavangarjan
Institut für Technische Informatik, Universität Rostock, Germany
tav@informatik.uni-rostock.de

Abstract

Current trends indicate that multiprocessor platforms will eventually replace uniprocessors in every application field, as the performance improvement exhibited by uniprocessor architectures during the last decade will soon become unable to satisfy the ever growing demand for higher and higher application performance.

Modern high-end Personal Computers (PCs) provide computation speed as well storage capacity at the best price/performance ever. Therefore an obvious way to obtain higher performance, parallel systems is to build a distributed-memory platform out of a pool of PCs interconnected by a fast Local Area Network (LAN) hardware, to obtain what is commonly called a *cluster of PCs*. Clusters are potentially able to deliver high performance at the unbeatable price/performance typical of their building blocks, namely commodity PCs and LAN interconnects, thus providing a low-cost alternative to both shared-memory multiprocessors and distributed-memory Massively Parallel Processors (MPPs). However standard communication protocols like TCP/IP run at a very low efficiency rate on modern LANs. This has the immediate consequence of poor performance achieved by parallel programs on clusters.

The Genoa Active Message MAchine (GAMMA) is an attempt to obtain good performance out of a cluster of PCs by adopting an optimized communication protocol and a suitable abstraction of the interconnect. Both the protocol and the abstraction are implemented mainly as a device driver for 100base-T LAN adapters in the Linux Operating System kernel. The virtualization of the network interface provided by GAMMA is called *Active Ports*, a mechanism derived from Active Messages. Such a virtualization is close enough to the raw LAN hardware to guarantee high communication performance while still providing a fairly flexible programming interface.

In this document the state of the art in the field of efficient parallel computing on clusters is briefly accounted. Then the basic principles, the software architecture, the main optimizations and the application programming interface of GAMMA are described in detail. The communication performance of GAMMA is reported and compared with other communication systems running on clusters; performance measurements from some typical parallel benchmarks are provided as well. All the performance figures confirm that GAMMA provides a viable solution to cost-effective parallel computing on clusters of PCs.

Acknowledgements

The hardware for the experimental research activity described in this book has so far been kindly provided by the Dipartimento di Informatica e Scienze dell'Informazione (DISI), Università di Genova, in order to allow this project to start without any specific financial support from external institutions.

Paolo Marenzoni, Giovanni Rimassa and Michele Vignali (former PhD and undergraduate students of University of Parma, Dept. of Information Engineering) provided practical suggestions and a number of "tricks of the trade" in the early stages of my experimental research action.

Luca Landi (undergraduate student at DISI), Angelo Margarino (system manager at DISI) and Alessandro Polverini (former undergraduate student at DISI) contributed to the installation of the systems and provided useful information and suggestions on some practical aspects.

Most of the basic ideas and optimizations described in this book, as well as a substantial contribution to the implementation phase, are due to Giovanni Chiola (full professor at DISI).

Also Gianni Conte (full professor at Dipartimento di Ingegneria Informatica, Università di Parma) and Luigi Mancini (associate professor at Dipartimento di Scienze dell'Informazione, Università di Roma "La Sapienza") supported our research project by sharing important ideas and supporting my work.

In the late phases of my PhD program I had the nice opportunity to exchange ideas with Loic Prylli and Bernard Tourancheau at the University "Claude Bernard" of Lyon, France.

The final draft of this thesis has been carefully reviewed by Djamshid Tavangarjan (professor at the University of Rostock, Germany) and Robert Russel (professor at the University of New Hampshire, USA). In particular, the excellent and very detailed review by Robert Russel pointed out a number of important "bugs" in the preliminary version of my thesis, thus substantially contributing to improve the quality of my work.

I'm grateful to all of the above persons, who contributed to make my PhD program an exciting and useful experience.

Table of Contents

Chapter 1

Introduction

1.1 Clusters of Personal Computers: convenient and necessary

PCs and workstations are becoming more and more powerful and Local Area Networks (LANs) are becoming more and more reliable and performing. At the same time, both PC and networking hardware components are becoming low-cost, off-the-shelf commodities. As a consequence, the idea of considering a Network of Workstations (NOW) or even a cluster of Personal Computers (PCs) as an efficient, low-cost parallel computer to be used in place of insufficiently performing uniprocessors or too expensive Massively Parallel Processors (MPPs) gains more and more consensus in the community of computer scientists/engineers, as pointed out by some well-known influential papers [ACPtNt95, Ste96]. Even more importantly, this idea is now largely accepted in the wide community of potential users.

There is no conceptual reason why parallel processing could not be successfully exploited in every kind of application domain. This has not yet occurred so far due to a number of reasons, including the following:

- The cost of what appeared to be the only reasonable support to parallel processing, namely an MPP platform, was and is still too high.
- Performance of microprocessors was and is still increasing at a fast rate. This rate has been evaluated to 55% per year for integer computation speed and about 75% per year for floating-point speed [CSG98]. Moreover, such an improvement occurred at lower and lower cost.

Given the above trends, it is no wonder that uniprocessors and sequential programming enjoy such a long-lasting and healthy life, while traditional MPP-based parallel processing did not expand significantly in real applications.

However the above reasons seem to vanish as time goes by. Indeed, due to the very trend of microprocessor price/performance, modern NOWs and clusters based on commodity components provide a viable alternative to MPPs at a much more reasonable cost. Moreover, a study based on simulation of aggressive uniprocessor architectures [CSG98] shows that such a trend is expected to reach a plateau, at least with respect to the ever increasing demand for computation speed. This is not due to a saturation of the underlying VLSI technology, which is expected to keep a fast improvement rate. Rather, further improvements of uniprocessor architectures are expected to be more and more hampered by the very structure of sequential programs. Actually microprocessors could increase their performance thanks to a more and more efficient exploitation of parallelism at the level of data representation (bit-level parallelism) and instruction execution (instruction-level parallelism), and memory could provide greater bandwidth mainly thanks to a hierarchical organization exploiting code/data locality. All this is now going to be hampered by the intrinsic structure of sequential programs: the logical dependencies among instructions in a typical sequential code limit the degree of instruction-level parallelism to an average value of five. Further improvements in clock speed and memory bandwidth will not be able to sustain the demand for higher performance for a long time.

To summarize, parallel processing no longer requires expensive hardware and is going to become the only way to keep taking advantage of the ever improving underlying VLSI technology. Therefore a change in the attitude of programmers with respect to parallel programming is expected: in much the same way as multi-threaded programming is going to be widely accepted for the currently ubiquitous shared-memory Symmetric Multi Processor (SMP) computers, message-passing is expected to be more and more exploited for programming cluster computers, which are expected to become ubiquitous as well. In the near future, virtually all significant applications will be written as parallel programs mixing multi-thread and message-passing programming models, and clusters of SMPs will be the most common high-performance computing architecture.

1.2 The "technology bottleneck" of communication software

Currently, to build an *efficient* cluster is not simply a matter of PCs, LAN and cables; to build an efficient cluster is mainly a matter of communication software. Every modern Operating System (OS) provides industry-standard software for computer communication, but this software is so inefficient that it actually appears to act as the real "technology bottleneck" of cluster computing. The reason for such inefficiency is twofold: industry-standard communication software is substantially older than LANs but too young compared to the history of OSs.

When the first implementations of what today we call an OS appeared in the Computer Science arena, the state of the art was completely different from today: hardware was extravagantly expensive as compared to software. As a consequence, for a long time priority number one for a "good OS" used to be the efficiency in the exploitation of (expensive) hardware resources. The structure of former OSs evolved according to this primary goal. Indeed most of the features that we find in a modern OS were designed in a price/performance perspective. This is the case, for instance, for processor scheduling in multiprogrammed time-sharing systems, virtual memory, file system primitives, disk drivers, etc.

Gradually, the state of the art advanced and eventually we reached a situation in which software is considered extremely expensive as compared to hardware, at least for the vast majority of computer applications and systems. This simple fact also determined a gradual change in the attitude of programmers with respect to code optimization. In the sixties a programmer was willing to devote a substantial effort to modifying a program in order to reduce memory request and/or CPU time. Nowadays if a program requires twice the RAM that is available on a machine we find it more convenient to buy a RAM extension (or to wait until next year in the hope that next year's machines will provide double RAM space for the same price) instead of spending time in code and data structure optimization. For this reason, modern priorities for a "good OS" are to be compatible with other systems, to allow easy re-use of (expensive) code, etc. Efficiency in the use of hardware resources is no longer a priority provided that sufficiently cheap and fast hardware can however be used to reach satisfactory levels of performance.

In this perspective of shifting the focus from efficient use of hardware resources to convenience of interconnection of standard components as one of the main OS concerns, LANs happened to become crucial in systems operation at the very time when the second trend was starting to prevail over the initial one.

At the beginning, the efficiency of LAN utilization was not perceived as a relevant problem. Inter-Process Communication (IPC) was only conceived as an event related to sporadic access to remote heterogeneous resources, and nobody was thinking about running parallel applications on NOWs or clusters of PCs. For these reasons, with LAN devices the viewpoint of *standardization for interoperation and portability* prevailed over the "old-fashioned" price/performance considerations. This is the reason why today's OSs happen not to support LAN devices with the same level of efficiency with which they support the operation of "older" devices such as CPU, virtual memory, disks, etc.

The prevailing needs of interoperability and standardization, combined with the prevalent view of communications as sporadic events, led to the establishment of industry-standard, inherently inefficient communication layers like the Unix IPC stack (namely, the Remote

Procedure Call (RPC) level built on top of BSD Sockets or System V Streams built on top of TCP or UDP protocols built on top of the IP protocol, etc.). Faster communications, whenever needed, are achieved by using faster interconnection hardware (such as Fast Ethernet, FDDI, ATM, Gigabit Ethernet, etc.) rather than optimizing the communication software. Even the idea of using NOWs and clusters as parallel computers could not change such a conservative approach to LAN communications. Indeed the first general purpose NOW-oriented implementations of the PVM and MPI parallel programming libraries (see Section 3.1.4) were based on software layers and even daemons invoking standard Unix IPC primitives, regardless of such primitives being not really committed to efficiency. It is no wonder that communication throughput and message delay experienced by parallel applications running on top of such implementations of PVM and MPI were very far from the potential of raw communication devices. However, as long as the peak computation performance of microprocessors was considered uninteresting for performance-demanding applications, there was no real need for efficient communication on NOWs and clusters. NOW platforms could only be exploited as low-cost tools for prototyping and debugging parallel software to be installed and run on a much more performing and expensive MPP.

Gradually, the evolution in microprocessor technology brought high-end workstations in the performance range of MPP computation nodes at a substantially lower cost. And, even more important, high-end PCs raised the performance range of entry-level workstations at very competitive cost. Such a technological revolution pushed off-the-shelf microprocessors inside MPPs as processing nodes, but with no corresponding revolution in prices. On the other hand, this very revolution turned commodity-based NOWs and clusters into full-fledged viable alternatives to MPPs.

Since current microprocessor performance is more than one thousand times greater than fifteen years ago, the only obstacle hampering an extensive diffusion of cluster computing is the cronic inefficiency of the industry-standard communication layers that became fixed in OS architecture during the past evolution of computer and communication technology.

1.3 The GAMMA project

In order to take advantage of clusters, existing communication software needs to be re-visited and possibly re-engineered in order not to become a more and more severe bottle-neck. Several research projects concerning efficient support for cluster-wide communications have been started so far, both from the hardware and the software standpoint. Many such research projects have a strong commitment to implementation, therefore many new communication protocols, mechanisms, abstractions, and devices have been made available. New concepts have been developed, and some of them start to be included in commercial products.

When the research action reported in this thesis started in the early 1996, most NOW projects could be grouped into two families, namely: those (mainly from USA) seeking best performance from high-end though quite expensive commodity interconnects and workstations, and those (much fewer) seeking reasonable performance from low-cost commodity clusters with reasonably low development effort. However there was no reason, at least in principle, not to seek for best performance from low-cost components like PCs and Fast Ethernet. Our choice of leveraging such cheap technology was motivated partly by a null budget and partly by the following considerations:

- Well-equipped PCs could provide workstation-like performance increasing at a fast rate and with a much lower cost; therefore PCs could be expected to replace workstations in every application domain, including NOW platforms.
- Shared Fast Ethernet was providing the best price/performance for LANs, whereas the aggregate bandwidth provided by switched Fast Ethernet appeared high enough to support a broad range of parallel applications; therefore Fast Ethernet technology appeared flexible enough to support a long-lasting research action in the field of cluster computing.

We therefore started experimental research aimed at identifying a successful approach to fast communications in a Fast Ethernet cluster of PCs. Thanks (or due) to previous experience with the CMAML Active Message library of the Thinking Machine CM-5 parallel computer, we immediately oriented our investigation toward an Active Message-like support for fast communication. The research prototype that was almost immediately (four months later) developed was thus called the Genoa Active Message MAchine (GAMMA).

Providing a commercial system for parallel processing was not our main motivation. The main goal of GAMMA, at least at the beginning of the project, was to understand the real obstacles to fast communications on low-cost clusters by experimentally playing with reliability versus performance versus abstraction trade-offs in a Linux environment, aiming at a successful approach under the following constraints:

- preserve the Linux multi-user, multi-tasking environment, extending it also to parallel applications;
- preserve the Linux network services across the LAN;
- minimize the need for modifications of the source code of the Linux kernel.

According to a typical experimental approach, this study was conducted by implementing a toy messaging system for Linux from scratch. Providing a small though reasonably high-level set of primitives supported by a very elementary communication protocol was the primary goal. Many optimization techniques were then applied in order to study their impact on performance, while a simultaneous extension of the set of supported functionalities was in progress aiming at a full-fledged though minimal messaging system for parallel processing. The excellent and in some cases surprising results achieved by GAMMA, which are the subject of this thesis, demonstrate the success of our approach.

Although the GAMMA programming interface has a fairly high abstraction level, offering some very efficient collective routines besides the most performing point-to-point communication calls currently available for Fast Ethernet, using GAMMA "as is" to write parallel applications is not that an easy task, mainly due to the adoption of an Active Message-like communication model. Rather, GAMMA is much more suitable as a support atop which higher-level industry-standard libraries like MPI may be implemented in an efficient way. Indeed this is currently one of the many on-going efforts in the GAMMA project.

1.4 About this book

This thesis is structured as follows: Chapter 2 introduces the main performance models used to evaluate communication systems, namely: latency-bandwidth, *LogP* and BSP. Chapter 3 reports the "state of the art" in the field of efficient support for communications in cluster computing. The main optimization techniques are briefly outlined, and a few selected experimental prototypes of fast messaging systems for clusters developed in the last few years are described and evaluated. Chapter 4 contains a detailed analysis of GAMMA, describing its basic architectural and programming principles, the reliability versus performance trade-offs and the optimization techniques adopted for its implementation together with the evaluation of their consequences in terms of communication performance. Chapter 5 reports about the design, implementation and performance evaluation of three different barrier synchronization algorithms for GAMMA, including one which is optimal on shared Fast Ethernet LAN. Chapter 6 describes a number of typical parallel benchmarks and applications, taken from linear algebra (Linpack, Matrix Multiply) and physics (Molecular Dynamics simulation), and reports about their implementation/porting atop GAMMA. The performance improvements shown by the GAMMA versions of all such programs over their respective MPI versions run atop TCP/IP on the same hardware platform provide evidence of the suitability of GAMMA as low-level support for the execution of parallel applications on low-cost clusters. Last but not least, Chapter 7 provides a brief outline of the on-going as well as planned activities of the GAMMA project.

Chapter 2

Evaluating communication performance

Predicting performance of applications going to run on a parallel/distributed platform is a key point for application developers. Unfortunately this is still a very hard business, mainly due to the weaknesses of the cost models available today for parallel and distributed machines.

Although predicting application performance is always difficult, a good performance characterization of the underlying communication system may be of great help, at least for a qualitative analysis. Features and limitations of the messaging system have different impacts on different applications. In the worst cases, it may at least help to compare different platforms in order to choose the most suitable one for a given application. Of course there are many performance models for communication systems, with various degrees of accuracy and complexity.

2.1 Latency/Bandwidth characterization

Most performance measurements of communication systems are given in terms of the two parameters *latency* L and *asymptotic bandwidth* B. The former deals with the synchronization semantics of a message exchange, while the latter deals with the data transfer semantics.

2.1.1 Latency

There is no general agreement about what "latency" is and how it should be measured. The purpose of a latency evaluation is to characterize the speed of the underlying system to synchronize two cooperating processes by a message exchange. The faster the system is with such a task, the better the performance is for highly synchronous applications. From the standpoint of a message-passing application, pure synchronization information can only be carried by a message exchange. However the payload of such a message carries no actual synchronization semantics, and as such it should be as small as possible, preferably empty.

For this reason, the most common definition of latency is as follows: *(one-way) latency* is the time needed to send a minimal-size message from a sender to a receiver, from the instant the sender starts a send operation to the instant the receiver is notified about the message arrival. Normally "sender" and "receiver" are application-level processes, however some questionable latency characterizations have been reported in the literature based on lower level (library, or even device driver) measurements.

It may be argued that the above definition of latency could make it very difficult, if not totally meaningless, to compare latency measurement from different platforms. Indeed the minimal allowed message size usually depends on the particular messaging system. Most systems allow a minimal size of one longword, others allow truly empty messages. However, given the above purpose of a latency measurement, it does not really matter whether the minimal-sized message carries a payload or not. A messaging systems that does not allow truly empty messages should be simply regarded as not optimal from the pure synchronization standpoint. Consistently, such non-optimality often translates into a greater latency time.

According to the above definition, the common technique to compute the latency parameter L is to use a *ping-pong* microbenchmark. This microbenchmark involves two process named "ping" and "pong". Process "ping" sends a message then receives it back from "pong". It also measures the time needed to execute the send/receive pair, that is the round-trip time. Process "pong" does the reverse. Actions are repeated a number of times to collect statistics, possibly discarding the first few measurements to exclude "warm-up" effects (memory allocations, start-up cache misses, unsynchronized processes, etc). L is computed as half the average round-trip time. Sometimes other statistics are computed instead of the average value.

2.1.2 End-to-end and one-sided asymptotic bandwidth

The asymptotic bandwidth B is (or at least should be) a characterization of how fast a data transfer may occur from a sender to a receiver. In order to isolate the data transfer from any other overhead related to the synchronization semantics, the transfer speed is measured for a very large (ideally infinite, hence "asymptotic") amount of data. If D is the time needed to send S bytes of data, measured from the instant the sender starts the "send" operation to the instant the receiver gets the complete message payload, then the asymptotic bandwidth B is computed as S/D when S is very large.

Any bandwidth measurement requires experiments with large data transfers. At least the following three techniques have been reported so far to accomplish such large data transfers:

- The *single-message* technique: send one single large message. This straightforward technique is feasible only if the underlying messaging system allows very large message sizes.

- The *burst* technique: send a long rapid sequence of fixed-size messages. This technique is used when the maximum message size allowed by the underlying messaging system is not large enough. However this is not correct in case we need a per-message characterization of the messaging system. Indeed in a per-message characterization of a system which does not allow big messages, one should emulate one single big message by fragmenting it into a long sequence of fixed-size messages to be re-assembled on the receiver side. A fair per-message bandwidth measurement should include the fragmentation as well as re-assembly overheads.

- The *stream* technique: send a long stream of bytes. This is feasible for stream-oriented communication systems like TCP/IP sockets.

Whatever the way a (large) data transfer is performed, there are in turn two techniques to perform the actual measurement:

- *End-to-end* measurement: use a ping-pong microbenchmark (see Section 2.1.1) to measure the average round-trip time. Process "ping" performs a data transfer of S bytes to process "pong", then receives the same data transfer back. The time needed to perform the two operations, called *round-trip* time, is measured. Process "pong" does the reverse, usually taking no measurement. Actions are repeated a number of times to collect statistics, possibly discarding the first few measurements to exclude initial "warm-up" effects. Then D is computed as half the average round-trip time, and the asymptotic bandwidth is computed as S/D. This microbenchmark measures the transfer rate of the whole end-to-end communication path.

- *One-sided* measurement: use a *ping* microbenchmark to measure the average send time. Also the ping microbenchmark involves the two processes "ping" and "pong". Process "ping" performs a data transfer of S bytes to process "pong". Process "pong" receives the data transfer then sends a short acknowledge message to process "ping" to inform it that data were successfully received. Process "ping" measures the data transfer time as the interval between the begin of its transmission and the reception of the acknowledge message. Actions are repeated a number of times to collect statistics, like with "ping-pong". D is computed as the average data transfer time (different from ping-pong, because here the time is not divided by two), and the asymptotic bandwidth is computed as S/D. This microbenchmark measures the transfer rate as perceived by the sender side of the communication path, thus hiding the overhead at the receiver side. This implies that the measured throughput is greater than the one evaluated by an end-to-end technique. A similar technique could be used to measure the transfer rate at the receiver side.

2.1.3 Throughput curve and half-power point

In the case of end-to-end latency/bandwidth characterization, the latency L and bandwidth B parameters are supposed to correctly characterize the end-to-end delay $D(S)$ of the messaging system for any message size S. This is because the following linear relation (derived from the one reported in [Hoc94]) is assumed as an analytical model of the communication performance:

$$D(S) = L + (S - S_m)/B \tag{2.1}$$

where S_m is the minimal message size allowed by the system.

Equation 2.1 assumes that the per-byte delivery cost $1/B$ does not depend on the actual message size. However this does not always hold. Especially with packet-oriented interconnects (like most LAN hardware used for clusters), which require data fragmentation and re-assembly, such a linear model does not capture the behaviour of the system when fragmentation starts to occur. Modelling performance of medium-size messages is important as they are frequently generated by medium-grained parallel applications.

By using the ping-pong and ping microbenchmarks introduced in Section 2.1.2, we can obtain more complete characterizations than simple latency/bandwidth estimation.

We define the *throughput curve* as the graph of the throughput function $T(S) = S/D(S)$ where S is the message size in bytes and $D(S)$ is the *(one-way) message delay* defined as one of the following:

- Half the round-trip time for a message of S bytes as measured by the ping-pong microbenchmark, using either single-message or burst data transfers. In this case we obtain the *end-to-end* throughput.

- The data transfer time as measured at the sender side by the ping microbenchmark, using either single-message or burst data transfers. In this case we obtain the *sender-side* throughput. A *receiver-side* throughput can be defined in a similar way.

It is worth noting that the asymptotic bandwidth is nothing but the throughput for a very large (ideally infinite) message.

Like with bandwidth measurements, sender-side throughput measurements lead to greater or equal results than end-to-end ones because they may actually characterize the transfer rate of just a subset of the communication path, namely from the sender process to the interconnection hardware.

A partial view of the entire throughput curve is given by the asymptotic bandwidth B together with the *half-power point*, defined as the message size S_h such that $T(S_h) = B/2$.

2.2 LogP

Even the most accurate latency/bandwidth characterization as well as complete throughput curve evaluation does not take into account an important parameter of any messaging system, that is the *software overhead* needed to perform a message exchange. Such a software overhead is to be intended as the time the CPU is involved in message processing, from interrupt-related overhead to protocol execution and error management. The impact of the software overhead on a running application is clear: the application loses CPU time in favour of the messaging system, and a worse overlapping between computation and communication is achieved. However a simple latency measurement cannot show the amount of software overhead, for the following two reasons: end-to-end latency results from a combination of hardware and software overheads, and the ping-pong test used to evaluate latency forces a synchronization between sender and receiver which leads to masking part of the software overhead at both sides of communication. Therefore it is no wonder that even a higher-latency communication system may yield better application performance than a lower-latency one, because of a reduced software overhead.

For a message of given size S, a better characterization of the messaging system would be one which breaks the one-way message delay time $D(S)$ (see Section 2.1.2) down into its relevant components, distinguishing the software overhead from other communication overheads. This is exactly the approach of *LogP* [CKP+93], a popular cost model for parallel platforms. Assuming S as a parameter of the model, the following four parameters characterize a message-passing parallel computer and especially its messaging system:

- The time L of physical flight of the message from the sender processor to the receiver processor along the hardware interconnection.
- The time o spent by the two CPUs engaged in communication to manage protocols and data transfers. Parameter o can be in turn broken down into two components: at the sender side o_s and at the receiver side o_r.
- The minimal time g that must elapse between the start of a send operation and the start of the next send operation issued by the same CPU. The ratio S/g is the one-sided throughput $T(S)$ of the messaging system as perceived at the sender side. A similar parameter can be defined for the receiver side.

- The number P of processors connected to the messaging system.

Measuring the LogP parameters is tricky business. The interested reader may look at [CLMY96]. An experience of such a measurement is reported in [ILM98].

2.3 BSP

The LogP model is more accurate than simple throughput curves, however it still lacks a precise methodology for deriving performance predictions from parallel programs in a reasonable way. The ability of making such predictions is a key point in a modern (and still lacking) development cycle of parallel applications, where any design decision should be driven also by performance considerations. As many authors have pointed out, the main weaknesses of existing cost models for parallel programs are related to a lack of structure in their underlying concurrency models. It is well known that a generic unstructured parallel program, be it message-passing or shared-memory, is NP-hard optimizable in terms of performance. More restricted paradigms allowing only structured parallel programs usually come with a much more effective and usable cost model, often including some simple performance model for the communication subsystem.

Bulk Synchronous Parallelism (BSP) [SHM96] is the best known of such structured models. It allows only parallel programs structured as a collection of processes, each composed of a sequence of so-called *supersteps*. Each superstep is a computational, sequential piece of code, containing calls to communication functions. The concurrency model assumes that communications initiated during a superstep are completed at the end of the superstep, and no process may proceed to the next superstep before other processes have completed the current superstep. In other words, the parallel computations involves repeating the following three phases: a parallel computation phase where communications may be initiated, a collective phase where communications are completed, and a barrier synchronization involving all processors.

It is intuitively clear that such a restricted model may allow an easy performance prediction, at least in principle, thanks to the extremely simple structure of BSP parallel programs. The main point here is the accuracy of the communication cost model.

In BSP, the messaging system is modeled by only two parameters, namely:

- The time g to deliver one data unit from one processor to another under continuous random traffic.
- The time l to issue a barrier synchronization involving all processors.

Parameter l is of intuitive meaning, given that each superstep ends with a barrier synchronization.

Parameter g is less obvious. In BSP all network traffic occurs in bursts at the end of each superstep. For well-designed programs and using state-of-art routing hardware it has been argued [SHM96] that a correct model for each BSP traffic burst is as if it were generated by all processors simultaneously sending messages to any other in a random order. In this case a more realistic characterization of the network is given by its "permeability", that is end-to-end throughput, under such traffic rather than the throughput in absence of any traffic. Giving only the per-unit delivery cost g amounts to assuming a linear model of the end-to-end throughput as a function of the message size. Even though this may not be realistic enough in some cases, it is simple to use indeed.

2.4 Evaluating collective communications

Almost all the existing models for communication and application performance deal with point-to-point communications. For collective communication routines (like broadcast, barrier synchronization, gather, scatter, total exchange, scan) there is not such a wide range of different metrics.

In [XH96] a simple extension of the linear model (Section 2.1.3) is proposed. The latency L and bandwidth B parameters become functions of the number of involved processors N. Intuitively, as more processors are engaged, we observe an increase of the group synchronization overhead, due to the increased number of involved processes, as well as an increase of the time needed to transfer data from a processor to any other, due to increased network traffic. In addition, if S is the total amount of data transferred by a collective communication and $D(N, S)$ is the time needed to accomplish the collective communication, then the *aggregate asymptotic bandwidth* $B_A(N)$ is defined as the ratio $S/D(N, S)$ when S approaches infinity. The relation between $B_A(N)$ and $B(N)$ depends on the particular collective pattern. In [HWW97] this model is used to evaluate collective routines of the MPI library on IBM SP2, Cray T3D and Intel Paragon.

A *naive* and widely used technique to evaluate performance of a given collective routine is as follows: each process measures the time needed to accomplish a predefined number N of calls to the collective routine, then computes the average time for one single call. The worst of such average measurements is assumed as the average completion time for one call to the collective routine. If the N calls are issued in a loop, the obtained measurement is significant only if the collective call is guaranteed not to return before each process in the group has finished the call. In many cases, such a rendezvous semantics is not implemented. With some MPI collective calls, this is not even specified by the standard, therefore a non-synchronizing implementation is perfectly legal. Whenever the collective routine does not synchronize processes upon return, a simple loop of repeated calls may lead to skewed measurements due to pipeline as well as network contention effects among consecutive not yet terminated calls. The pipeline effect may especially occur with multi-stage interconnections, which usually allow more messages to simultaneously travel along the network. In [NN97] a measurement technique that avoids such a risk is proposed. Basically it requires an *a priori* loop-based evaluation of which process in the group is always the last to terminate a given collective call. Then a measurement of one single collective call from that process may be issued. This way an effective worst-case measurement is taken while avoiding pipelining as well as contention effects due to repeated calls.

Chapter 3

Light-weight messaging systems for clusters

In this chapter we attempt to provide a picture of the "state of the art" in the field of cluster-wide communications. We claim that classifying existing prototypes, based on a description of their architectural approaches and tradeoffs accompanied by a performance comparison, is helpful to develop new ideas and prototypes, and may provide at least an insight, if not evidence, of the future trends of cluster computing. Since NOWs and clusters are distributed-memory architectures, the emphasis is deliberately put on message-passing communication systems. Since distributed emulations of shared memory abstractions on clusters are always built on top of message-passing systems, focusing on message-passing does not impact too much on generality.

The chapter is structured as follows: Section 3.1 gives a brief outline of the existing industry-standard communication environments, inherited by past eras of computer communications and now improperly but widely used for cluster computing. Section 3.2 describes the main optimization techniques exploited to develop more efficient communication support for cluster computing, and attempts to draw a broad classification of the various approaches into *kernel-level* and *user-level* communication architectures. Section 3.3 gives an account of some experimental messaging systems developed at the level of OS kernel. Systems developed to work outside the OS kernel, which therefore interact with OS and users by means of novel mechanisms, are described in Section 3.4. Finally, Section 3.5 gives a performance comparison of the messaging systems outlined in Sections 3.3 and 3.4 in order to attempt an analysis of the performance impact of the architectural and implementation issues discussed in this chapter.

3.1 Traditional communication mechanisms for clusters

3.1.1 TCP, UDP, IP, and sockets

Indeed the most widespread and accepted standard for communication at the network level is the Internet Protocol (IP), providing unreliable delivery of single packets. It is well known that packets sent by the IP protocol are not guaranteed to be delivered in their transmission ordering. Moreover the IP protocol is not intended to be directly exposed to applications: the scope of IP is to deliver packets to one-hop distant hosts, therefore it lacks the generality of a true end-to-end protocol. It might be argued that one-hop distances are exactly what we find in most clusters. However IP was developed far before LANs and clusters; this is a possible reason why nobody at that time could realize the potential advantage of exposing IP directly to application level. As a matter of fact, both the two basic kinds of communication Quality of Service (QoS), namely "connected" and "datagram", were implemented by two distinct additional layers atop IP, namely Transmission Control Protocol (TCP) and User Datagram Protocol (UDP).

Both TCP/IP and UDP/IP were made available to the application level through the same kind of Application Programming Interface (API), namely Berkeley Sockets. Like IP, sockets were developed far before people could even realize the impact of the particular network abstraction on communication performance, scalability and code optimization. Indeed sockets were designed with no goal other than to present a familiar Unix-flavoured abstraction: the network is perceived as a character device, and sockets are file descriptors related to that device. Processes could then send/receive messages by writing/reading such file descriptors. Once nicely fitted in the Unix architecture, the establishment of sockets and the IP protocol suite as an industry standard was only a matter of time during the long age when inter-process communication was only a sporadic event and interoperation between heterogeneous hosts connected by any kind of networks, including highly unreliable and multi-hop ones, was perceived as the main concern.

To summarize, sockets and the IP protocol suite owe their success to generality of purpose, similarity to other established abstractions, and initially favourable conditions (sporadic communications on slow networks). For an insight about how much and why TCP/IP sockets are inefficient on clusters see Section 3.2.1.

3.1.2 Active Messages

Active Messages [vECGS92] is a slight departure from the classical send-receive model. It is a *one-sided* communication paradigm, that is whenever the sender process transmits a message, the message exchange occurs regardless of the current activity of the receiver process. As an aside, there is no need for a receive operation.

The goal of Active Messages is to reduce the impact of communication overhead on application performance. The particular semantics of Active Messages may eliminate the need for much temporary storage for messages along the communication path, thus remarkably speeding up communications. Moreover, with proper hardware support it becomes easier to overlap communication with computation.

In traditional send-receive systems, messages delivered to a destination node may need to be temporarily buffered waiting for the destination process to invoke a "receive" operation which will consume them. The semantics of Active Messages is different: as soon as delivered, each message triggers a user-programmed function of the destination process, called the *receiver handler*. The receiver handler acts as a separate thread which promptly consumes the message, therefore decoupling message management from the current activity of the main thread of the destination process. Here "consuming" means integrating the message information into the ongoing computation of the main thread, notifying the message arrival to the main thread, and possibly seting up some data structures in order to promptly "consume" the next incoming message as soon as it arrives.

Many experimental high performance messaging systems described later on are Active Message-like systems. A well known commercial application is the Connection Machine Active Message Layer (CMAML) library by Thinking Machines Co. running on the CM-5 platform [TMC92]. CMAML yields user-level end-to-end throughput and latency close to the limit posed by the hardware interconnect [LC94].

With all current Active Messages-like messaging systems, the receiver handler is sender-based: each transmitted message is composed of two parts, namely the message body and an explicit pointer to the destination's receiver handler that will consume the message upon arrival. Usually it is the receiver handler that extracts the message and stores it into a data structure in the receiver address space. Such an RPC-like communication model requires the destination process to share its own code address space with the sender process, a condition which is easily fulfilled only with the Single Program Multiple Data stream (SPMD) paradigm.

Other systems (e.g. GAMMA, see Section 3.3.2.1) exploit a slightly different Active Message-like model, more deeply inspired by the Thinking Machines' CMAML library. Each process owns a fixed number of *communication ports* and may attach a specific receiver handler to a given port in order to handle messages incoming through that port. This way the sender may specify a communication port of the destination process rather than a remote pointer to the receiver handler. This "port-oriented" variant of Active Messages is suitable for MIMD as well as SPMD programming, as it does not require cooperating processes to share a common address space.

3.1.3 RPC

The socket interface is the current industry-standard general purpose abstraction for communication and interoperation. Nevertheless its level of abstraction is particularly low. Although all the standard communication services and higher-level specialized protocols (e.g. SMTP for e-mail) were easily built atop sockets, there was a need for higher level but still general purpose abstractions in order to support the development of arbitrary distributed applications, especially for heterogeneous environments. Remote Procedure Call (RPC) by Sun is one such abstraction. Today RPC is the *de facto* industry-standard for communication in distributed client-server applications. Like with sockets and IP, the success of RPC is mainly due to familiarity and generality:

- RPC provides a familiar framework for the design of client-server applications, namely procedure call. Remote services are virtualized as procedures. Services are requested by calling such procedures with suitable parameters. The called service may also return a result. This semantic framework is perceived by programmers as very similar to sequential programming.
- In heterogeneous environments, one major difficulty is to cope with different formats for representing the same data type across different systems connected to the network. RPC hides any format difference, therefore providing a transparent view of an heterogeneous distributed system.

3.1.4 MPI and PVM

Sockets and the IP suite were born to support low-level general purpose interfacing between processes and a generic network. As more specific but still sufficiently general needs arise, new higher level protocols and abstraction are created. RPC was established as a standard by the community of developers of distributed client-server applications. The large community of parallel programmers was not yet involved, as traditional parallel computers were incomparably more powerful than distributed systems formed by networked uniprocessors at that time. As soon as it became clear that a collection of networked computers could well serve as a low-cost parallel platform for educational as well as prototyping purposes, there was an immediate need for a general-purpose system for message passing and parallel program management on distributed platforms.

Parallel Virtual Machine (PVM) [Sun90] was the first of such message passing systems to be established as a *de facto* standard for parallel programming. PVM provides an easy-to-use programming interface for process creation and inter-process communication, plus a run-time system for elementary application management. PVM runs nicely on the most general kind of distributed memory system, namely a generic IP network of generic host computers;

however there are also many implementations of PVM atop shared-memory systems and MPPs. This guarantees the widest portability of PVM programs across any range of parallel systems. Moreover, PVM processes may interrogate the PVM run-time system about the computational power of their local processors; this greatly helps designing self-balancing parallel applications. PVM owes most of its popularity, but also its inefficiency, to its run-time management system.

The Message Passing Interface (MPI) [The95, MPI] is the other established standard for message passing parallel programming. MPI offers a larger and more versatile set of routines than PVM but does not offer any run-time management system. Actually MPI owes its popularity more to the greater efficiency of its existing implementations compared to PVM rather than to its rich programming interface (indeed, MPI users tend to exploit a small fraction of the huge MPI routine set).

3.2 Light-weight communications for clusters

3.2.1 What we need for efficient cluster computing

Many people agree on the fact that Linux provides one of the most efficient implementations of the TCP/IP stack ever. The Linux TCP/IP socket one-way latency depends heavily on the specific network driver being used, which in turn depends on what Network Interface Card (NIC) has been leveraged. Let us consider a very basic cluster configuration comprising two Pentium II 300 MHz PCs, each running Linux 2.0.29 and equipped with a 3COM 3c905 Fast Ethernet NIC driven by one of the latest Linux drivers for this card. Suppose we connect the two PCs by a UTP class 5 crossover cable and work in half-duplex mode (this simulates the use of a Fast Ethernet repeater hub). Then run a simple ping-pong test to evaluate one-way latency and asymptotic bandwidth (after disabling the Nagle "piggy-backing" algorithm). Under such experimental conditions you should measure a one-way latency of 77.4 μs and asymptotic bandwidth of 10.8 MByte/s at socket level, on average. The average half-power point is found at 1750 bytes. Recalling that the maximum theoretical bandwidth of Fast Ethernet is 12.5 MByte/s and assuming a reasonable lower bound for latency of 7 μs, this means that with Linux TCP/IP and Fast Ethernet:

- Latency is one order of magnitude worse than the hardware latency.
- Efficiency in the range of short (single packet) messages is below 50%.
- Efficiency is good (86%) only with very long data streams. Sending long streams with half-duplex makes the packet flow contend for network access with the acknowledge flow running backward. Therefore we may expect better efficiency for long streams with full-duplex connections, which rules out repeater hubs.

Of course the overall impression is that Linux TCP/IP looks very good for traditional networking but not very good for cluster computing, especially looking at performance numbers in terms of CPU cycles rather than microseconds.

Figure 3.1 provides a comparison between throughput curves measured with slightly different system configurations. For instance, it is apparent that using the same hardware and OS but a different device driver leads to a substantial performance degradation. The reason is very simple. The 3c905 NIC can be programmed and driven in two different ways, that we call *descriptor-based DMA* (DBDMA) and *CPU-driven DMA*. When driven in DBDMA mode, the NIC itself performs DMA transfers between host memory and the network by simply scanning a precomputed list of so called "DMA descriptors" stored in host memory. Therefore the low-level memory-to-network and network-to-memory data transfers are operated by the NIC autonomously while the CPU is running the TCP/IP protocol code and building the necessary DMA descriptors for subsequent data transfers. This pipelines the end-to-end communication path and increases the communication throughput. The latest Linux drivers for the 3COM 3c905 NIC use the DBDMA mode. However, previous drivers used the CPU-driven DMA mode, according to which the host CPU itself starts a DMA operation of the NIC then waits for the DMA transfer to end before starting a subsequent DMA operation. This leads to a "store-and-forward" behaviour and lower throughput. A very similar 3COM card operated in CPU-driven DMA mode on Pentium 133 CPUs yields even lower performance. This provides an insight on the software overhead involved in running the TCP/IP stack: the larger this overhead, the larger degradation with slower CPUs.

Another consideration which does not emerge from Figure 3.1 is related to the communication performance on a congested LAN. It is well known that TCP uses a sliding-window flow control algorithm with "go-back-N" packet retransmission upon timeout. The flow control mechanism of TCP avoids packet overflow at the receiver side, however it cannot prevent overflow from occuring in a LAN switch in case of network congestion. When this occurs, some packets are discarded by the switch. Eventually the retransmission mechanisms of TCP on the sender hosts are triggered and start re-sending many more packets than needed, increasing network traffic and therefore making the LAN even more congested. Clearly LANs require better flow control algorithms or at least more sophisticated retransmission policies, which should take into account the fact that the only source of packet loss in a modern LAN is switch overflow and that this often occurs according to well-known patterns (packet losses due to switch overflow often happen in sequences called "clusters", allowing a selective retransmission algorithm to perform far better than "go-back-N").

A drawback of TCP/IP that has often been pointed out is that its layered structure implies a certain number of memory-to-memory data movements. This is a time-consuming task and pollutes the working set of the user process upon communication, leading to a higher number of cache misses with additional performance degradation. Another source of performance

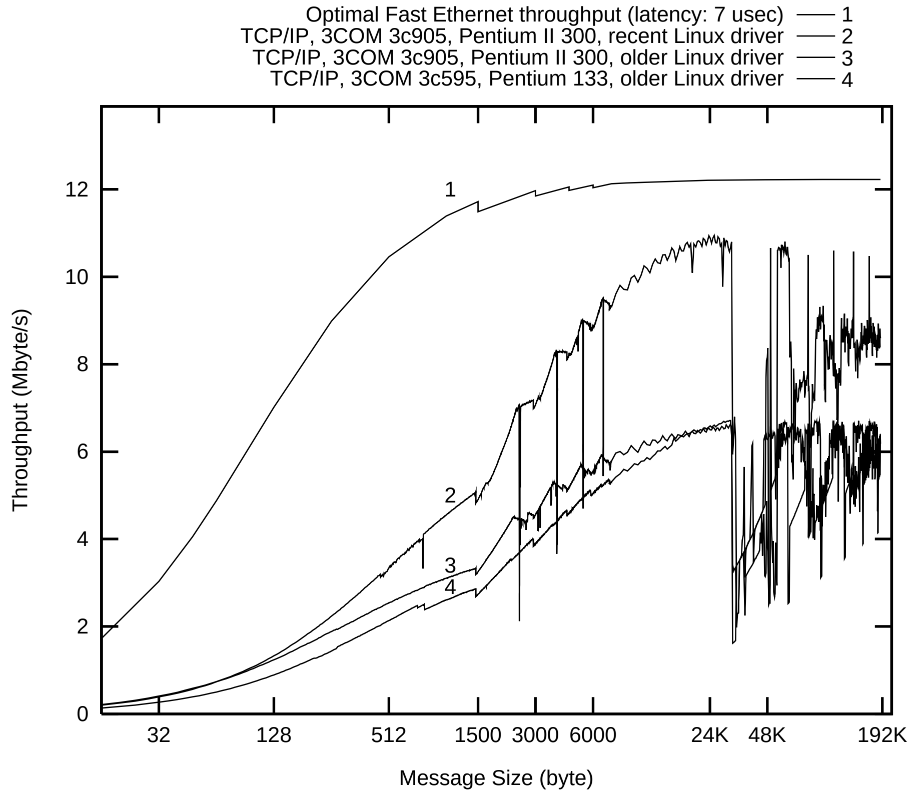

Figure 3.1: Linux 2.0.29 TCP/IP sockets: half-duplex "ping-pong" throughput with various NICs and CPUs (Nagle Disabled).

---

degradation is the poor code locality implied by the layered software structure.

Finally, all the throughput curves reported in Figure 3.1 show an erratic behaviour as a function of the stream length. This is due to the collision storms caused by contention between data and acknowledge packets over the shared LAN.

From the above analysis we can draw some general conclusions about what we should do to implement an efficient messaging system on NOWs and clusters:

- Choose an appropriate LAN hardware. A NIC which is "intelligent enough" to co-operate with the host CPU may be of great help for higher performance.
- Tailor the protocols to the underlying LAN hardware. The specific features of the LAN, its typical error rate and error patterns should be carefully considered when designing protocols.
- Target the protocols to the user needs. Different users and different application domains may need different tradeoffs between reliability and performance. Reliable, slower protocols should be offered together with performance-oriented, less robust ones. A fast protocol that is unreliable under generic operation conditions may be

reliable if certain assumptions on the network traffic patterns are fulfilled, and the user may prefer to explicitly enforce such conditions to obtain high performance.

- Optimize the protocol code and the NIC driver as much as possible. The software overhead of protocol execution must be kept as low as possible. In particular, layered software architectures are nice, elegant but often inefficient due to poor code locality and excessive parameter passing overhead.

- Minimize the use of memory-to-memory copy operations.

Indeed the above principles have inspired most of the common optimization techniques for cluster-wide messaging systems that are outlined in Section 3.2.2.

3.2.2 Typical techniques to optimize cluster-wide communication

Various attempts to address the inefficiency of standard inter-process communication mechanisms on clusters have been carried out in the last few years. Most messaging systems are committed to efficiency, at least to a certain degree. Although following different architectural approaches, most of them share a number of common performance-oriented features which we briefly outline below.

3.2.2.1 Using multiple networks in parallel

This is the most straightforward way to increase the aggregate communication bandwidth of the platform and reduce congestion. However this cannot help reducing latency.

3.2.2.2 Simplifying LAN-wide host naming

Indeed addressing conventions in a LAN might be much simpler than in a generic Wide Area Network (WAN).

3.2.2.3 Simplifying communication protocol

Long protocol functions are time-consuming, and their poor locality in the access to huge data structures generates a large number of cache misses and pollutes the working sets of user processes, thus leading to even more cache misses. General-purpose networks with a high error rate (like WANs and highly congested LANs) may require complex protocols to recover from packet losses in an efficient way. However, when using low error rate LANs where congestion is unlikely or impossible and end-to-end communication performance is the main concern, the trade-off between reliability and performance may change in favour of so called *optimistic protocols*. A typical optimistic protocol is one which performs very well in the supposedly usual case of no communication errors and no congestion. In the unlikely event of a communication error, an optimistic protocol may either recover from the error in an inefficient way or regard the event as catastrophic and limit itself to raise an error condition to the upper layers. Other fast protocols base their "optimism" upon an *a priori* global knowledge of the communication patterns allowed on the network (see Section 3.3.2.3 for an example).

3.2.2.4 Avoiding temporary buffering of messages

Making temporary copies of information along the communication path during the execution of complex protocols is a time-consuming task and pollutes the working sets of user processes. The ideal protocol would allow communications with no temporary message buffering. Protocols providing such a feature are called *zero-copy* protocols. A frequent answer to this challenge is remapping the kernel-level temporary buffers into user memory space. Such kernel-level buffers are usually directly accessed by the NIC when sending/receiving data, but remapping them into user space means that user processes also have direct access to them, so that the memory-to-memory transfers usually involved when crossing the user-kernel boundary are no longer needed. However this is often only a partial answer: data must still be moved back and forth between user data structures and communication buffers, no matter whether they are mapped into user space or not. Another solution is to lock the user data structures into physical RAM and let the NIC access them directly via DMA upon communication. This is also a partial answer, as only those data structures spanning contiguous virtual memory regions fit this method. For non-contiguous data structures, data transfers between host memory and NIC must be done in chunks. This implies a proper structure of the communication system (it must provide so called "gather/scatter" facilities).

3.2.2.5 Pipelining communication phases

Some NICs may be programmed to start transmitting data over the physical medium while the host-to-NIC DMA or programmed I/O transfer is still in progress. This allows the NIC to transfer data in a pipelined way during transmission. A similar behaviour can be programmed at the receiver side: the NIC-to-host data movement can be initiated while the data transfer from the physical medium to the NIC is still in progress. This way the end-to-end behaviour of the communication path is fully pipelined like in a wormhole routing. The performance improvements obtained by pipelining data transfers in this way may be impressive, in terms of latency as well as throughput.

3.2.2.6 Avoid system calls for communication

Invoking an OS system call is often considered too much a time-consuming task. Therefore one challenge is to minimize the need for system calls in the communication path. A frequent solution is to implement the communication system entirely at user level: all buffers and registers of the NIC are remapped from kernel space into user memory space so that user processes no longer have to cross the user-kernel boundary to drive the device. This is the basic idea of the so called *user-level* communication architectures (see Section 3.4). However this poses protection challenges in a multi-tasking environment where more processes may share the same communication devices. Protection can be enforced only by leveraging programmable NICs which can run code for protected device multiplexing, a feature currently offered only by expensive network devices.

3.2.2.7 Light-weight system calls for communication

As an alternative to eliminate the need for system calls, so called light-weight system calls are often implemented, which save only a subset of CPU registers and do not invoke the scheduler upon return.

3.2.2.8 Fast interrupt path

The code path to the interrupt handler of the network device driver is optimized in order to reduce interrupt latency in interrupt-driven receives.

3.2.2.9 Polling the network device

The usual method of notifying message arrivals by interrupts is time-consuming, therefore sometimes unacceptable especially in the case of short messages. The overheads of interrupt launch and service must always be payed, even if the incoming message processing overhead is low. Therefore most high-performance messaging systems provide the ability to explicitly inspect or poll the network devices for incoming messages, besides interrupt-based arrival notification.

3.2.2.10 Providing very low-level mechanisms

Following a kind of RISC approach, many high-performance communication systems actually provide only very low-level primitives. Such primitives can be combined in various ways to build many kinds of higher level communication semantics (blocking or not, reliable or not, buffered or not, etc) and APIs (standard, custom, etc.) in an "ad hoc" way especially tailored to the underlying platform as well as to the application/library needs. This solution can only be performance-effective if the calling overhead for the basic primitives is minimized. This is the reason why almost all the communication systems based on very low-level mechanisms are user-level also (see above).

3.2.3 The importance of efficient collective communications

Another crucial issue that must be addressed in order to turn the potential benefits of clusters into widespread, effective use is the development of parallel applications exhibiting high enough performance and efficiency with a reasonable programming effort.

Traditionally, high level standard communication libraries such as PVM or MPI have been introduced with the aim of facilitating parallel code development and porting. However, while for instance an MPI code is easily ported from one hardware platform to another provided that both provide an implementation of MPI, performance and efficiency of the code execution is not "ported" at all across platforms, because the same communication call implemented on different platforms according to different criteria may lead to quite different communication performance.

Therefore, whenever an MPI programmer seeks best performance for a given parallel program on the parallel machine at hand, he/she is induced to restructure his/her program in order to avoid the use of all those communication calls and patterns that turn out not to be efficiently implemented on the local machine.

Of course much lighter and effective porting effort would be required if parallel programs were written using high-level collective communications instead of *ad hoc* patterns of low-level point-to-point communications. Unluckily enough, collective routines often provide the most frequent and extreme instance of "lack of performance portability". In most cases, collective communications are implemented in terms of point-to-point communications arranged into standard patterns which do not take into account the potential benefits as well as the limitations of the underlying hardware interconnect. This is especially true with public-domain implementations of MPI and PVM for TCP networks. They assume nothing about the underlying interconnect, a LAN being treated the same way as a WAN. This implies very poor collective communication performance with clusters. As a result, parallel programs hardly ever rely on collective routines, even in cases where their exploitation would naturally lead to simpler or more modular code. The final consequence is that nobody gets trained with the usage of collective communications, and that message-passing parallel programming remains hard business.

A good tradeoff between the semantic expressiveness of a communication library and the efficiency of its actual implementation on clusters must be sought in order to turn clusters into really interesting environments for high performance parallel computing. In this perspective the efficient implementation of standard collective communications like barrier synchronization, gather-scatter, reduce, and total exchange, is of crucial concern.

3.2.4 A Classification of light-weight communication systems

From a hardware-oriented standpoint, existing messaging systems for clusters fall into two main families, namely: those based on *commodity off-the-shelf hardware*, and those based on *custom hardware*. From a software-oriented standpoint, they can be grouped into two main families as well, namely the *kernel-level* and the *user-level*. We classify and compare these different approaches.

3.2.4.1 Commodity versus custom devices

Using custom devices rather than commodity off-the-shelf ones (like in [DSP97, HKO+94, KOH+94, BLA+94, MK96]) may lead to very low latency and high bandwidth communication. The performance characteristics as well as the technological level of some custom devices are often much better than off-the-shelf ones. The performance superiority is often achieved through a closer integration between the communication device and the host CPU: many such communication devices could be plugged directly on the memory bus and tightly integrated in the memory hierarchy of the workstation. Indeed most such novel interconnects are targeted to distributed shared memory systems rather than message passing systems. We argue that in the long term such attempts are not likely to spread in the user community, as non-commodity interconnection devices are usually too expensive and do not enjoy the large R&D efforts that only commodity hardware vendors can sustain in the long term. Indeed many vendors of traditional massively parallel platforms have run into enormous difficulties in keeping their product up-to-date with technology, due to their extreme specialization to a narrow marketplace segment. Conversely systems exploiting only commodity components are guaranteed to keep up with the fast pace of the evolving hardware technology, as their marketplace segment is much broader. For this reason our choice is to leave all the communication systems based upon specialized hardware out of the scope of this chapter, although such systems are often quite interesting in many respects.

3.2.4.2 Kernel-level versus user-level

In the *kernel-level* approach the messaging system is supported by the OS kernel with a set of low-level communication mechanisms embedding a communication protocol. Such mechanisms are made available to user level through a number of OS system calls in order to ensure protected access to the devices in a multi-user, multi-tasking environment. Applications can either invoke these low-level mechanisms directly, or use higher level communication libraries built atop them. The former case is more typical for system applications (like the standard suite of Internet daemons and servers, which make explicit use of BSD sockets), whereas the second case is typical for parallel and distributed high-level applications.

The kernel-level approach fits nicely into the architecture of modern OSs, providing protected access to the communication devices with no need to limit or modify its multi-user, multitasking features even in the presence of commodity non-programmable devices. Although some minor additions to the OS kernel may be required, it is often possible to preserve both the standard OS interface and its functionality. For historical reasons all the current industry-standard communication APIs are currently implemented following the kernel-level approach. The kernel-level approach will be discussed in detail in Section 3.3.

A drawback of the kernel-level approach is that traditional protection mechanisms may require quite a high software overhead, due to the standard system call mechanism (scheduling activity upon return) as well as the kernel-to-user data movement. This poses hard challenges to implementors of high-performance messaging systems, forcing the exploitation of some optimization techniques (light-weight system calls, fast interrupt paths, see Section 3.2.2) while hampering other optimizations (e.g. the use of very low-level mechanisms).

The *user-level* approach aims at improving performance by minimizing the OS involvement in the communication path in order to obtain a closer integration between the application and the communication device. More precisely, access to the communication buffers of the network interface is granted without invoking any system calls. Any communication layer as well as API is implemented as a user-level programming library. In all the existing user-level systems, applications are provided with a non-standard API. The choice of the API itself is often determined by performance considerations, and there is no *de facto* standard API for such systems, although the specification of the Virtual Interface Architecture (VIA) [VIA98] promoted by CompaQ, Intel and Microsoft can be regarded as a first attempt to standardize user-level communication architectures.

In order for such an approach not to compromise protection in the access to the communication devices, either a single-user network access or strict gang scheduling are usually required. A third alternative is to leverage programmable communication devices which can run the necessary support for device multiplexing in place of the OS kernel. All three alternatives have their own drawbacks: single-user network access appears to be an unreasonable restriction in a modern processing environment; gang scheduling appears inefficient and requires major interventions at the level of OS scheduler; moving multiplexing support from the OS kernel to the device is infeasible with commodity low-cost components which usually are not programmable.

In the user-level approach, modifications to the OS kernel may range from simple addition of custom device drivers to deep interventions at the scheduler level, depending on the degree of protection to be preserved and the extent to which multi-user access to the communication device is to be allowed.

The user-level approach will be discussed in detail in Section 3.4.

3.3 Kernel-level light-weight communications

In the large family of kernel-level messaging systems running on commodity clusters we can identify two main sub-classes, namely *industry-standard API* systems and *best-performance* systems.

3.3.1 Industry-standard API systems

In the *industry-standard API* approach the main goal besides efficiency is to comply with an industry-standard for the low-level communication API. Retaining an industry-standard communication interface allows portability and reuse of existing applications and libraries developed for that interface. Usually this approach does not force any major modifications to the existing OS; rather, the new communication system is simply added as an extension of the OS itself (e.g. a custom device driver, a Linux kernel module, or a user-level programming library).

A drawback of this approach is that some optimizations in the underlying communication layer could be hampered by the choice of an industry-standard, like Berkeley sockets for instance, which was not originally conceived to address any performance issues.

3.3.1.1 Beowulf

The Beowulf project [SBS+95] runs on a Linux-based cluster of PCs. It follows a very conservative way, namely retaining the standard Unix protocol suites. Improved communication performance is achieved by exploiting two or more LANs in parallel, a technique called “channel bonding” which Beowulf supports at the level of Linux modified NIC drivers. Of course much better results would have been obtained had the communication software been optimized, in addition to using more LANs. However this system is currently commercialized in the USA: The *Extreme Linux* software package by Red Hat [Hat98] is nothing but a commercial distribution of the Beowulf system.

Besides the standard bus topology, a two-dimensional mesh topology has been investigated [ea96]. Each node in the mesh is connected to two distinct Ethernet LANs, namely the "horizontal" and the "vertical". Therefore each node is adjacent to every node in the same row as well as column. Moreover, each node acts as a software router in order to allow non-adjacent nodes to communicate. This way a node A can reach a non-adjacent node B by two distinct disjoint paths. The "channel bonding" technique is then applied to allow node A to communicate with node B using two such independent network paths in parallel, with two intermediate nodes engaged in parallel for routing. However a parallel disk I/O test conducted on such a 2-D mesh topology has shown throughput improvement over bus topology only in the case of small workloads.

3.3.1.2 Fast Sockets

Fast Sockets [RAC97] is an implementation of TCP sockets atop Active Messages. The raw Active Message layer provides good performance, whereas the upper socket interface, implemented entirely as a user-level library, provides industry-standard interfacing to the network. The protocol reverts to plain TCP/IP when the LAN boundary is crossed, e.g. through a gateway. In case of forking processes, Fast Sockets do not support the plain semantics of sockets according to which socket descriptors which are open at fork time are shared with child processes. Fast Sockets have been reported not to have an efficient connection phase. Measurements carried out on a pair of UltraSPARC 1's interconnected by Myrinet and running the Solaris OS have shown poor performance (57.8 μs latency, 32.9 MByte/s asymptotic bandwidth) compared to Myrinet raw communication performance. The relatively poor bandwidth is due to the UltraSPARC's SBus bottleneck, whereas the high latency is mainly due to the high (about 45 μs) latency of the underlying Active Messages layer.

3.3.1.3 PARMA^2

The PARMA^2 project [MRV$^+$97] is aimed at reducing communication overhead in a cluster of PCs connected by Fast Ethernet and running the Linux OS, by eliminating flow control and packet acknowledge from Linux TCP/IP and simplifying host addressing. The obtained protocol, called PaRma Protocol (PRP), has "datagram" QoS.

A distinctive goal of PARMA^2 is to retain a BSD socket interface in the standard multi-user Unix environment. The corresponding socket family is called `AF_PRPF`. Hence, the only thing to do for porting a stream socket application to PARMA^2 is to change the socket family to `AF_PRPF` in the source code. This allows porting all the socket-based Unix applications and libraries to the new protocol suite with very limited effort. For instance, the MPICH [GL96] implementation of MPI was able to run on PARMA^2 with only minor modifications.

At the lowest level, PRP preserves the Linux interface to the NIC driver. This allows PARMA^2 to run with whatever NICs are supported by Linux. Therefore no optimization was carried out at the level of NIC driver.

Very few data concerning PARMA^2 performance are available. PARMA^2 exhibits 74 μs one-way latency and 6.6 MByte/s asymptotic bandwidth at the socket level, that is about half the Linux TCP/IP latency and a 20% improvement over Linux TCP/IP maximum throughput on 3COM 3c595 NICs at the time of publication (1997). However the latency improvement over Linux UDP/IP sockets is not impressive (one-way latency of Linux UDP/IP sockets was about 100 μs on the same hardware platform), and the absolute communication performance of PARMA^2 appears still far from the raw Fast Ethernet performance.

Porting the MPICH library atop PARMA^2 resulted in a significant reduction of one-way latency at the MPI level (from 402 μs to 256 μs). A dedicated implementation of MPI called MPIPR, actually a simplified version of MPICH, has been developed and tested on PARMA^2. One-way latency has been further reduced for synchronous point-to-point MPI communications (from 256 μs to 182 μs). The only drawback of MPIPR is that it inherits the "datagram" QoS from the underlying PRP protocol, thus substantially violating the semantics of MPI communications.

3.3.2 Best-performance systems

In the *best-performance* approach, the messaging system is supported by the OS kernel with a small set of flexible and efficient low-level communication mechanisms and by simplified communication protocols carefully designed according to a performance-oriented approach. Here challenges like choosing the right mechanisms and degree of virtualization as well as implementing such mechanisms efficiently are more important than choosing the level at which the communication layer is to be placed in the overall OS architecture.

3.3.2.1 Genoa Active Message MAchine (GAMMA)

In the Genoa Active Message MAchine (GAMMA) [CC97c, Cia98, CC], extensively described in Chapter 4, the Linux kernel has been enhanced with a communication layer implemented as a small set of additional light-weight system calls and a custom NIC driver with a fast interrupt path. Most of the communication layer is thus embedded in the Linux kernel, the remaining part being placed in a user-level programming library. The adoption of an Active Message-like communication abstraction called *Active Ports* allowed a minimal-copy optimistic protocol, with no need of either kernel-level or application-level temporary storage for incoming as well as outgoing messages. GAMMA implements pipelined communication paths among user processes. Multi-user protected access to the communication abstraction is granted. The GAMMA device driver is capable of managing both GAMMA and IP communication in the same 100base-T network. On 3COM 3c595 and 3c905 NICs, GAMMA yields very low one-way user-to-user latency (12.7 μs) and high asymptotic bandwidth (12.2 MByte/s, corresponding to 98% efficiency).

It must be said that the GAMMA communication protocol is unreliable: it detects communication errors (packet losses and corrupted packets) but then it simply raises an error condition without recovering. The GAMMA approach leaves to the user (application as well as library writer) the task of using the error detection mechanisms to build recovery policies of suitable complexity. However the very low latency delivered by GAMMA potentially allows a wide range of error recovery as well as explicit acknowledge policies to be implemented in a very efficient way.

3.3.2.2 Net*

Net* [RH98] is a communication system for Fast Ethernet based upon a reliable protocol implemented at kernel level. A user process can be granted network access by initially allocating kernel-space send and receive buffers and queues which are remapped into user-space. Only one user process per node can be granted network access. To send a message, a process copies the message itself into the "user-level image" of a send buffer, then invokes the kernel-level transmission protocol function through a dedicated system call. Such a system call is not necessary if another send operation was submitted a short enough time before, since the interrupt launched at the end of each physical transmission is serviced by a routine which tests send queues for new messages to transmit. To receive a message, a process may perform either a busy or a sleeping wait on the receive queue. Two levels of service are provided, namely a raw unreliable service for best performance, and a reliable service built atop the raw one.

Net* appears to share architectural features from both user-level and kernel-level approaches. The main drawbacks of Net* are that no kernel-operated network multiplexing is performed despite the kernel being involved in the protocol execution, and that user processes have to explicitly fragment and reassemble messages longer than the Ethernet MTU.

Very good performance is reported for Net*: the raw level exhibits 23.3 μs one-way latency and 12.1 MByte/s asymptotic bandwidth; the reliable level delivers 30.9 μs one-way latency with same bandwidth. However, the throughput performance of Net* has been evaluated with a one-sided technique, therefore, it does not model the end-to-end behaviour of the communication system.

3.3.2.3 Oxford BSP Clusters

A completely different approach to protocol optimization for LAN interconnects is to place some structural restriction on communication traffic by allowing only some well known patterns to occur. Such additional knowledge of the "shape of interaction" can be used to derive *a priori* assumptions to be exploited for optimized error detection and recovery strategies. This is the approach followed by the Oxford BSP Clusters [DHS98]. A parallel program running on a BSP cluster is assumed to comply with the BSP computational model (see Section 2.3). In the BSP model each process is shaped as a sequence of supersteps. Each superstep is a computation phase followed by a global and synchronizing communication phase. The pattern of such a global communication phase is an any-to-any total exchange followed by a final barrier synchronization. We briefly report here the main features and assumptions of the optimized protocol of BSP clusters.

- In the total exchange phases, the destination scheduling is different from processor to processor in order to avoid network hot spots. This greatly reduces the probability of network overflows, with corresponding very low packet loss rate.
- The interconnect is assumed to be a switched one, so that no network contention will ever occur.
- In a total exchange communication pattern, some of the messages transmitted can be used to piggyback both positive and negative acknowledgments for other messages. This reduces the need for acknowledgment packets as well as timeouts for error detection and recovery.

The protocol of BSP clusters has been implemented atop various lower level communication systems. Of course the most performing version is the one implemented atop the NIC directly, that is as a device driver called *BSPlib-NIC*. The BSP NIC driver implements the communication protocol and is placed at kernel level in the Linux OS, therefore BSPlib-NIC should be regarded as a kernel-level messaging system. However it shares some features and limitations with the user-level approach. For instance, the kernel-level transmit/receive FIFOs of the NIC are remapped into user memory space to allow user-level access to the FIFOs; such direct access occurs in a typically unprotected fashion forcing single-user mode. Moreover the particular NIC leveraged, that is the 3COM 3c905B Fast Ethernet adapter, has been programmed to automatically poll the transmit FIFO for packets to be sent; this makes it unnecessary to implement an explicit "start transmission" system call, thus no system calls are required along the whole end-to-end communication path.

This implementation of the BSP cluster protocol achieves a *minimum* (not average) one-way latency of 29 μs with 11.7 MByte/s asymptotic bandwidth. It is worth recalling that BSP cluster communications exhibit such performance while guaranteeing reliable delivery. Rather than trading reliability with performance, the BSPlib-NIC choice is to gain performance by placing restrictions on communication patterns and imposing single-user access.

3.3.2.4 Pupa

Pupa [VC98] is a messaging system providing reliable delivery and flow control. It is implemented as an extension of the FreeBSD OS kernel. The main features of Pupa are a smart buffer management policy and a light-weight communication protocol. However the Pupa communication performance on Fast Ethernet is comparable to Linux TCP/IP performance, maybe due to a layered software architecture with multiple temporary copies of messages.

3.3.2.5 Sender-Based protocols

Another instance is the implementation of Sender-Based protocols described in [SS96]. With this messaging system, the sender process specifies the location where the message will be stored in the memory space of the destination process. Sender-Based protocols have been reported to exploit about 82% of the raw communication bandwidth with latency below 24 μs on a FDDI-connected cluster of HP730 workstations. In [SS96] an excellent decrease (44%) in latency obtained with the fast interrupt path optimization is reported.

3.3.2.6 SSAM

SPARCStation Active Messages (SSAM) [vEABB95] is one of the first instances of kernel-level best-performance messaging systems for NOWs, besides being the first Active Message system running on a cluster, more precisely a 140 Mbps ATM network of SPARCStation-20 workstations. Communication performance of such an early prototype was not particularly brilliant though.

3.3.2.7 U-Net on Fast Ethernet

The basic principles of U-Net, described in Section 3.4.4, require a NIC's programmable on-board processor in order to implement device multiplexing without involving the host CPU. In the Fast Ethernet version of U-Net, called U-Net/FE [WBvE96], designed for a

Fast Ethernet network of Pentium PCs running Linux, the low-cost commodity NIC has no programmable on-board processor. The host CPU itself multiplexes the NIC over user processes by means of proper OS kernel support. For instance, the send operation requires invoking a light-weight system call to warn the NIC driver about the presence of a fresh outgoing message in an output buffer. For this reason we do not consider U-Net/FE as a user-level system, but only as a kernel-level emulation of the original U-Net concept yielding still good communication performance (30 μs one-way latency, 12.1 MByte/s asymptotic bandwidth) but with the drawback of presenting the very raw U-Net programming interface. One of the key points of the original U-Net is that it exposes very low-level mechanisms to user level. Indeed the U-Net programming interface is very similar to the programming interface of the NIC itself, and exhibits the same unreliability of communication as the NIC itself. In our opinion, implementing very low-level mechanisms at kernel level is a source of additional overhead for the upper communication layers, rather than an optimization technique.

3.4 User-level light-weight communications

The user-level approach to fast communication in a LAN environment derives from the assumption that OS communications are inefficient by definition. Hence the OS involvement in the communication path is minimized. System calls are avoided by allowing direct access to the NIC registers and storage areas, thus achieving a closer integration between applications and interconnection networks. This approach is rooted in the idea of F-Bufs [DP93], which was followed also in the Exokernel OS architecture [EKO95]. Indeed the user-level communication architecture shares the same fundamental idea of OS microkernels, that is moving OS services away from the kernel.

Often (but not always) a primary challenge besides efficiency is to provide "direct access" to the communication device without violating the OS protection model. Three solutions can be devised to guarantee protection, namely:

- Leverage programmable NICs which can run the necessary support for device multiplexing in place of the OS kernel. Currently this implies much higher hardware costs.
- Circumvent the problem by granting network access to one single trusted user. This is not always acceptable though.
- Implement some form of "network gang scheduling": a process has exclusive access to the network interface while it is running, and cannot be descheduled while any communication partner is still exchanging data with it. This eliminates the need

for the messaging system to operate run-time device multiplexing. However gang scheduling is inefficient and often unacceptable as a scheduling policy for many general purpose environments.

3.4.1 BIP

A recent achievement in the field of user-level messaging systems is the Basic Interface for Parallelism (BIP) [PT98], implemented atop a Myrinet network of Pentium PCs running Linux.

BIP offers both blocking and unblocking communication primitives following a send-receive paradigm implemented according to the rendezvous communication mode, with the exception of very short messages which are managed according to a buffered mode. The Myrinet hardware ensures in-order data delivery at a very low error rate, and controls the data flow through back-pressure. Under such conditions, Myrinet communication errors can be regarded as rare, catastrophic events. This is the reason why the BIP approach to communication errors is to offer a simple detection feature without implementing any recovery policy.

BIP pipelines the communication path by transparently fragmenting messages into packets whose size depends on the total message size. The Myrinet hardware does not require message fragmentation: the goal of BIP fragmentation is to allow the various DMA engines of the Myrinet adapters at both sides of a communication to work in a pipelined fashion. This is only possible if the message is fragmented into a stream of packets, where each packet traverses all the DMA engines in sequence. The fragmentation policy ensures best load balancing among such a pipeline of DMA engines.

Besides its many optimizations, BIP obtains best performance by getting rid of protected multiplexing of the NIC: the registers of the Myrinet adapter and the memory regions on which it operates are fully and unprotectedly exposed to user-level access.

BIP is able to deliver more than 96% of the raw Myrinet bandwidth (that is 126 out of 132 MByte/s) with a very low (4.3 μs) one-way latency time. It must be said that published BIP performance is computed as the *median* value of a number of measurements rather than average values. Porting the TCP/IP stack atop BIP provides an impressive demonstration of the poor efficiency of industry-standard protocol layers on fast interconnects (one-way latency 70 μs, asymptotic bandwidth 35 MByte/s). However porting of MPICH atop BIP [Tea] results in an acceptable degradation of communication performance at MPI level (median one-way latency is 12 μs and median asymptotic bandwidth is 113.7 MByte/s).

3.4.2 Fast Messages

Illinois Fast Messages (FM) [PKC97] is an Active Message-like system running on Myrinet-connected clusters, and providing reliable in-order message delivery with flow control and packet retransmission. FM works only in single-user mode, despite the Myrinet adapter providing a coprocessor (called LANai) which could have been programmed to multiplex the device among several processes.

The earlier versions of FM, namely FM 1.0 [PLC95] and FM 1.1, were designed for Myrinet clusters of SPARCStations. FM 1.1 exhibited good one-way latency (12 μs) but quite poor efficiency in terms of asymptotic bandwidth (16.1 MByte/s out of a raw Myrinet bandwidth of 132 MByte/s), due to the particularly slow SPARCStation's I/O bus.

Subsequent versions of FM, namely FM 2.x [LPC98] have been implemented on Pentium Pro clusters, where it delivers much better performance (11 μs latency, 77 MByte/s asymptotic bandwidth) thanks to the much faster PCI bus. Clearly FM remains quite far from the communication performance of the raw Myrinet hardware. However the main difference between FM 2.x and previous versions is in the programming interface. Porting MPICH atop FM [LC97] required a slight modification of the API in order to obtain a better match with the Abstract Device Interface layer of MPICH. Basically, FM has been enhanced with gather/scatter features in order to send/receive data from/to non-contiguous virtual memory regions without initiating a new send/receive operation for each region. With these enhancements to the programming interface, MPI atop FM could show a typical degradation of 6 μs in one-way latency at the MPI level with the same asymptotic bandwidth.

3.4.3 HPAM

Hewlett-Packard Active Messages (HPAM) [Mar94a] is one of the first user-level communication systems. It is an implementation of Active Messages on a FDDI-connected network of HP 9000/735 workstations. The FDDI interface is connected to the high-speed graphic bus of the HP workstation instead of the I/O bus. This leads to low (14.5 μs) latency and good (12 MByte/s) asymptotic bandwidth. Protected, direct access to the network is granted to a single process in mutual exclusion. An OS daemon schedules the processes to the network, and other major modifications to the OS allow the delivery of messages to non-scheduled processes. This way gang scheduling is avoided but the OS plays a non-negligible role in the communication path, at least in the general case of non-scheduled destination processes. HPAM provides reliable delivery with flow control and retransmission.

3.4.4 U-Net for ATM

Perhaps the most famous representative of the user-level approach is U-Net implemented on ATM networks [vEBBV95]. With U-Net, user processes are given direct protected access to the network device with no virtualization. Therefore the programming interface of U-Net is very similar to the one of the NIC itself. Any communication layer as well as any standard programming interface for communication is implemented in user-level programming libraries. Support to protect device multiplexing runs on the Intel i960 processor located on the Fore Systems SBA-200 ATM adapter.

The interconnect is virtualized as a set of "endpoints". Basically an endpoint is a kernel memory buffer plus a send queue and a receive queue for host-to-adapter synchronization. Endpoint buffers are used as intermediate storage for the send/receive operations, and correspond directly to portions of the NIC's send/receive FIFO queues. The role of the OS is limited to remapping one or more such endpoints to the memory space of a user process by means of dedicated system calls. After endpoint remapping, the process is granted direct, memory-mapped, protected access to dedicated portions of the adapter's send/receive FIFOs with no further involvement of the OS kernel. If the number of endpoints required by user processes exceed the available endpoints directly supported by the NIC, additional endpoints can be emulated by the OS kernel, providing the same functionality at reduced performance.

U-Net achieves communication performance very close to that of the raw 155 Mbps ATM hardware (one-way latency 44.5 μs, maximum throughput 15 MByte/s). The U-Net concept has also been implemented on Fast Ethernet LANs as a kernel-level emulation (see Section 3.3.2.7).

3.4.5 Virtual Interface Architecture (VIA)

The Virtual Interface Architecture (VIA) [VIA98], a large effort promoted by CompaQ, Intel and Microsoft, can be regarded as the first attempt to standardize user-level communication architectures. Basically VIA specifies an architecture which extends the basic U-Net interface with remote DMA (RDMA) services. It is oriented to System Area Networks (SANs), intended as high bandwidth, low latency, very low error rate, scalable and highly available interconnects. The VIA specification requires error detection as a feature of communication services. Protected multiplexing among user processes is also explicitly required, and recommended to be operated by the NIC itself for best performance. However reliability of communications is not mandatory, although explicitly recognized as a plus; this means that VIA is also suitable for unreliable media, at least in principle.

The VIA promoters expect NIC designers and vendors to develop network adapters supporting VIA mechanisms in hardware. However no “VIA inside” LAN/SAN adapter seems to be available in the marketplace at the time of writing (that is, about one year after releasing the VIA 1.0 specification [VIA98]). Of course emulating VIA at the level of OS kernel would provide much less impressive improvement in communication performance. Nevertheless an implementation of kernel-emulated VIA for Linux, called M-VIA [M-V98] is in progress. Only the unreliable delivery service has been implemented yet. Both Fast Ethernet and Gigabit Ethernet NICs are supported. Performance figures on Fast Ethernet are very good (one-way latency is 23 μs and asymptotic bandwidth is 11.9 MByte/s on a cluster of Pentium II 400 MHz PCs), but the presented API is very low-level as with U-Net.

3.5 A comparison among message passing systems for clusters

Table 3.1 reports a latency-bandwidth characterization of messaging systems running on a number of NOW prototypes, including the standard TCP/IP stack implemented in Linux.

A few communication layers running on commercial MPPs are reported as well, in order to provide a rough comparison with more traditional expensive parallel computers.

The table is divided into four sections, respectively for industry-standard API, user-level systems, kernel-level systems, and messaging systems running on some traditional MPPs.

It must be noted that the hardware platforms where measurements have been taken are often incomparable with one another in terms of network devices and/or CPU speed. Especially for latency comparisons, measurements taken with CPUs of different speeds may lead to misleading conclusions.

3.5.1 Clusters versus MPPs

The most performing messaging systems for NOWs and clusters appear indeed to be competitive w.r.t. MPPs, although comparing non-standard, low-level NOW programming interfaces with higher-level, standard message passing on MPPs is not completely fair. Of course NOW platforms cannot compete with MPPs in terms of asymptotic and bisection bandwidth, and lack one of the main MPP features, namely scalability w.r.t. the number of processors. On the other hand, the current cost of the scalability characteristics of an MPP is hardly justified by a large number of applications where parallel processing would in principle make sense. Indeed most of the MPP installations are “small” machines, equipped with only tens of processors and thus exploiting only a limited fraction of their scalability characteristics. A “small” MPP configuration is clearly outperformed in terms of

Platform	Latency (μs)	Max. throughput (MByte/s)
Linux 2.0.29 TCP/IP sockets, half-duplex		
(3COM 3c595 100base-T, Pentium 133 MHz CPU-driven DMA)	113.8	6.6
(3COM 3c905 100base-T, Pentium II 300 MHz DBDMA)	77.4	10.8
Fast Sockets (Myrinet,UltraSPARC)	57.8	32.9
PARMA^2 sockets (3COM 3c595 100base-T, Pentium 133 MHz)	74.0	6.6
GAMMA [Cia98] (3COM 3c905 100base-T, Pentium 133 MHz)	12.7	12.2
Net* (raw) (100base-T) [RH98]	23.3	12.1
Oxford BSP Clusters (100base-T) [DHS98]	29	11.7
Pupa (100base-T) [VC98]	198,0	7.8
Sender-Based protocols (FDDI) [SS96]	23.5	10.2
SSAM (ATM) [vEABB95]	30.0	5.6
U-Net/FE (DEC DC21140 100base-T) [WBvE96]	30.0	12.1
M-VIA (DEC DC21140 100base-T) [M-V98]	23	11.9
BIP (Myrinet,PPro) [PT98]	4.3	126
HPAM (FDDI) [Mar94a]	14.5	12
FM 2.x (Myrinet,PPro) [LPC98]	11.0	77
U-Net/ATM (FORE PCA-200 ATM) [WBvE96]	44.5	15
CM-5 CMAML ports [Cen]	10 - 15	8.5
SP2 MPL [MRV$^+$97, Table 2]	44.8	34.9
T3D PVMFAST [MRV$^+$97, Table2]	30.0	25.1

Table 3.1: "Ping-pong" comparison of message passing systems.

price/performance and even absolute performance by an efficient modern NOW or cluster.

3.5.2 Standard Interface approach versus other approaches

It is apparent from Table 3.1 that the standard interface architectural approach delivers quite poor exploitation of the interconnection hardware. This is evident with low-cost 100base-T (PARMA^2, Linux TCP/IP sockets) as well as a much more expensive and performing interconnect like Myrinet (Fast Sockets). This seems to be the price to pay for retaining low-level industry-standard programming interfaces, which hamper many possible optimizations because of a bad match with the underlying LAN hardware.

3.5.3 User-level versus kernel-level

It is apparent that the user-level approach has been successfully adopted only with higher performance interconnects (Myrinet, FDDI, ATM). This is due to two reasons, namely: higher-end network devices have on-board programmable processors which could run multiplexing/protection code, and an OS kernel mediation between user processes and network hardware is believed to become a communication bottleneck with fast enough LAN hardware.

From the reported performance numbers it is clear that user-level systems range from very efficient (BIP) to quite poor (FM). The choice of a workstation equipped with a slow I/O bus (like SUN SBus used in early prototypes of FM) is really a bad one. Indeed more and more NOW projects, although using high-speed interconnects, are leveraging high-end PCs instead of workstations due to unbeatable price/performance and good capabilities of the PCI bus. Also the choice of a particular communication abstraction may severely impact communication performance; this is a possible reason why BIP performs so much better than FM.

Kernel-level messaging systems almost exclusively run on low-cost commodity interconnected NOWs (with the exception of the early SSAM on ATM). This kind of platform, leveraging high-end PCs and Fast Ethernet LAN, is a low-cost alternative to fast NOWs, offering even better price/performance figures. The poorer network performance of low-cost interconnects poses harder efficiency challenges to the implementors of messaging systems. This should in principle induce implementors to follow the user-level approach. Nevertheless no plain user-level messaging architecture (either with unprotected device multiplexing or no multiplexing at all) has been implemented on low-cost NOWs so far. However GAMMA shows that excellent results on low-cost clusters can be obtained even with the mediation of the OS kernel and even with a fairly high-level and flexible programming interface (GAMMA delivers half the latency of the raw U-Net/FE), provided that communication protocol and device multiplexing be carefully developed according to a performance-oriented approach.

As a conclusion, it could be argued that user-level access to network devices should not be of primary concern for communication performance, at least in principle, given the loose integration allowed between high-speed commodity NICs and the host memory hierarchy in today's PC architectures. Porting efficient kernel-level communication systems on some next-generation interconnection devices (for instance Scalable Coherent Interface and Memory Channel), which are more tightly coupled with the CPU than today's off-the-shelf communication devices, would provide more insights about the actual impact of kernel-mediated NIC multiplexing on communication performance with fast networks. To the best of our knowledge, currently no research project is engaged in such a research task.

Chapter 4

GAMMA: the Genoa Active Message MAchine

4.1 Introduction

The Genoa Active Message MAchine (GAMMA) is a fast messaging system for 100base-T clusters of Pentium PCs running Linux and equipped with either 3COM 3C595-TX Fast Etherlink or 3COM 3C905-TX Fast Etherlink XL PCI 100base-T adapters.

Very recently, GAMMA has been ported to the DEC 21143 Fast Ethernet chipset [CC99]. The resulting implementation differs from the original one under many respects, and is not described in this dissertation.

GAMMA allows one to turn a cluster of PCs into a cost-effective system for parallel processing characterized by:

1. a cost per node (including communication devices) of less than 1.5 K US$ at the current market price;
2. one-way latency (12.7 $\mu s.$) as good as the leading-edge, expensive NOW prototypes, and better than many commercial messaging systems running on MPPs;
3. an asymptotic bandwidth corresponding to 98% of the 100base-T Ethernet theoretical throughput, which in absolute terms (12.2 MByte/s) happens to be higher than the one offered by Thinking Machines' CM-5 and by Transputers' channels and almost as good as the one offered by systems based on ATM and FDDI;
4. a very favourable throughput profile even for messages of short/medium size, with half-power point (50% efficiency) reached with 192 byte messages;

5. a fair scalability up to a few tens of computation nodes;
6. a fairly high-level, convenient programming interface to develop parallel applications as well as higher-level abstractions like MPI;
7. the availability of the full Linux OS development and run-time multi-tasking environment for all programming aspects (namely local and remote file systems, network services, public domain compilers, debuggers, etc.).

The novelty of GAMMA is the application of the Active Message communication paradigm to the case of a cluster of PCs, according to a kernel-level communication architecture but with only very marginal modifications to the OS kernel itself. This made it possible to obtain what is currently the most performant messaging system available for 100base-T Ethernet.

Although the GAMMA programming interface has a fairly high abstraction level, especially compared to much more radical approaches like U-Net and VIA, using GAMMA to write parallel applications is not an easy task due to the relatively poor set of communication calls. In this respect, GAMMA is much more suitable as a support atop which higher-level industry-standard libraries like MPI may be implemented in an efficient way.

As already pointed out in Section 1, providing a commercial system for parallel processing was not the main motivation of GAMMA, at least at the beginning of the project. The main goal of GAMMA was to develop a toy communication system for Linux to experimentally understand the real obstacles to fast communications on low-cost clusters and to devise a successful approach to overcome them under the following constraints:

- preserve the Linux multi-user, multi-tasking environment, extending it also to parallel applications;
- preserve the Linux network services across the LAN;
- minimize the need for modifications to the source code of the Linux kernel.

Therefore GAMMA was initially developed as a small prototype for performance experiments, with a gradual development towards a full-fledged though minimal messaging system for parallel processing.

In this chapter we describe in detail the programming model of GAMMA, its communication architecture and its programming interface. The many optimization techniques that have been adopted to implement GAMMA are described as well, together with an evaluation of their impact on end-to-end communication performance.

The results presented in this chapter appeared in [CC97b, CC97c, CC97a, Cia98, CC98a, CC98b]. Recent enhancements (e.g., the credit-based flow control introduced in the DEC

21143 version of GAMMA [CC99]), as well as some recent extensions to the GAMMA API, are not reported in this thesis. Up-to-date information about the GAMMA project activities is maintained in [CC].

4.2 Overall approach and key features of GAMMA

Following the "best-performance" kernel-level approach, with GAMMA the Linux kernel has been enhanced with a low-level messaging system providing the following basic mechanisms:

- static process grouping for SMPD programs;
- point-to-point inter-process communications;
- broadcast inter-process communications, directly exploiting the Ethernet broadcast services;
- barrier synchronization.

.

The GAMMA barrier synchronization primitive is treated separately in Chapter 5.

The basic mechanisms of the GAMMA communication system, with the exception of the barrier synchronization, are implemented as a small set of additional light-weight system calls (Section 3.2.2.7). A custom NIC driver implements an optimistic communication protocol (Section 3.2.2.3) and drives the NIC in such a way that the stages of the end-to-end communication path work in a pipelined way (Section 3.2.2.5). The adoption of a suitable network abstraction called Active Ports (Section 4.4) eliminates the need for any temporary copy of messages along the communication path, thus allowing what we call a *minimal-copy* protocol. The interrupt routine of the GAMMA driver is registered with the Linux kernel as a fast interrupt (Section 3.2.2.8), and a polling service is provided to wait upon receive (Section 3.2.2.9).

Most of the GAMMA communication layer is thus embedded in the Linux kernel at the NIC driver level. The remaining part (initialization routines, barrier synchronization, some collective routines, and stubs to basic services) is placed in a user-level programming library. Both C and FORTRAN programming interfaces have been developed.

Since NIC multiplexing is operated by standard OS kernel mechanisms, all the multi-user, multi-tasking functionalities of the Linux kernel have been preserved for sequential applications and granted to parallel applications as well. Moreover the GAMMA device driver is able to manage both GAMMA and standard IP communication on 100base-T Ethernet simultaneously, by preserving the Linux interface between driver and IP protocol and interpreting packet headers "on the fly" in order to separate GAMMA packets from IP packets and pass the latter to their standard IP code path. Allowing both IP and GAMMA network traffic to coexist on the same LAN is necessary in order to preserve useful network services (rshell, telnet, nfs, etc.) across the cluster, as well as to run mixed GAMMA/PVM/MPI parallel programs using socket-based implementations of MPI and PVM.

All the GAMMA communication software has been carefully developed according to a performance-oriented approach. Most of the GAMMA code has been written as carefully optimized C functions, with a few performance-critical pieces of code written in Intel 80x86 Assembler.

In the current prototype of GAMMA we can identify the following open points, some of which have recently been addressed, or are subject of ongoing work or scheduled as future activities (see Section 7.1):

- lack of flow control, which can lead to communication errors under particular circumstances;
- lack of support to multiple network connections on the same PC;
- lack of thread safety, needed to support multithreaded parallel applications on clusters of SMP multiprocessor PCs;
- lack of support to virtual parallelism, that is, more processes than processors.

4.3 The GAMMA computational model

In order to understand the implementation details as well as the programming interface of GAMMA, it is worth pointing out the computational model and communication abstraction we had in mind during the design phase.

4.3.1 The physical GAMMA

A *physical GAMMA* is a set of workstations or PCs forming a Fast Ethernet connected cluster. Each workstation is addressed by the MAC address of the corresponding 100base-T adapter.

4.3.2 The GAMMA program model: SPMD

The program model of GAMMA is SPMD. A GAMMA parallel program is a *process group* counting N *process instances* running in parallel, each on a distinct workstation of a physical GAMMA. Plain MIMD programming is allowed as well since sender and receiver need not share code address space and inter-group communication is allowed. GAMMA provides automatic mechanisms for process spawning on remote PCs in the cluster through standard remote shell services (Section 4.7.1).

Currently a GAMMA process group cannot include more processes than the number of workstations of the physical GAMMA, that is, each process instance runs on a distinct PC. However, GAMMA supports parallel multitasking, i.e., more than one GAMMA group may be active at the same time on the same physical GAMMA and each PC may be time-shared by more GAMMA processes belonging to distinct groups. GAMMA groups are distinct parallel programs running on behalf of potentially distinct users.

Each GAMMA group is identified by an number which is unique in the platform. We call such a number a *parallel PID*. Processes belonging to the same GAMMA group share the same parallel PID. The parallel PID is used to distinguish among processes belonging to different parallel applications at the level of each individual workstation. At most 256 different parallel PIDs may be simultaneously active on a physical GAMMA.

Process instances are numbered from zero on. This numbering provides a straighforward though sufficiently flexible naming convention at the application level. At the kernel level each of these numbers is mapped onto the MAC address of the workstation where the corresponding process instance is running.

4.3.3 GAMMA messages

With GAMMA, a message transmission means that the content of a contiguous virtual memory region in the application space of a sender process is copied to a contiguous virtual memory region in the application space of a receiver remote process. Therefore, an outgoing GAMMA message may only come from a contiguous region of virtual memory, and an incoming GAMMA message can only be stored into a contiguous region as well. Many data structures in common use (e.g. arrays, C `struct`) span contiguous regions in virtual memory space, therefore in most cases there is no need to provide dedicated buffers for incoming/outgoing messages. For non-contiguous data structures, suitable pack/unpack routines and temporary buffers should be explicitly programmed by the user.

4.4 The GAMMA network abstraction: Active Ports

Any GAMMA process of a given parallel application owns, and may activate and use thereof, 256 *Active Ports* [CC98b], numbered from zero to 255. GAMMA Active Ports were inspired by the Thinking Machines' CMAML Active Message library for the CM-5 [TMC92]. Each active port can be used to exchange messages with another process in the same as well as a different process group.

Each port may be used for *output*, *input*, or *input/output*. An output port can be used to send messages, an input port can be used to receive messages, and an input/output port can be used for both.

A port must be *bound* before any use, in order to specify its behaviour as an output, input, or input/output port. The GAMMA library provides a binding function (Section 4.7.2). The information passed to the binding function is partly kept by user-level data structures inside the GAMMA library and partly passed to the GAMMA driver at kernel level through a dedicated light-weight system call.

4.4.1 How GAMMA sends messages

In order to use an owned port for output, that is to send messages, a process should bind the port to the following items:

- The parallel PID and instance number of the receiver process.
- The number of a (supposedly input) port of the receiver process.

Currently GAMMA provides two send operations. The `gamma_send()` function (Section 4.7.3) will return upon successful as well as failed termination of physical transmission. However the `gamma_send_fast()` function (Section 4.7.4) will return immediately after the last byte of the message has been loaded into the NIC's transmit FIFO, without waiting for the physical transmission to complete. Therefore `gamma_send_fast()` will not necessarily detect a possible failure occurring in the final phases of the physical transmission. Clearly the communication semantics of both functions is blocking. However, for forward compatibility, the semantics of `gamma_send_fast()` is intended to be as non-blocking.

To send a message through an active port, the process has to invoke one of the two send operations described above. Both trap to the GAMMA device driver through a dedicated light-weight system call. Each message is then transmitted as an ordered sequence of one or more Ethernet frames. Following a minimal-copy policy, frames are arranged directly into the NIC's transmit FIFO, getting the frame bodies by DMA directly from the source data structure in user memory space with no intermediate buffering in kernel space. Frame headers have been precomputed at port binding time.

Since the DMA transfers operated on transmission act on physical addresses, the source data structure must have previously been pinned down and prefetched into physical RAM. For this purpose GAMMA provides a lock function (Section 4.7.15).

4.4.2 GAMMA broadcast service

GAMMA supports group broadcast: when binding a port for output, the destination node may be set to the constant `BROADCAST`, meaning that outgoing messages will be sent to every process in the group identified by the destination parallel PID. The sender may or may not belong to the destination group. Efficiency of the GAMMA broadcast service is guarenteed by the direct use of the native Ethernet hardware broadcast service. Therefore a GAMMA broadcast message is sent as a single physical transmission over Ethernet.

4.4.3 How GAMMA manages incoming messages

GAMMA Active Ports are inspired by Active Messages. Therefore GAMMA communications are one-sided, that is, there is a send but no receive operation. The receive semantics is user defined, and depends on the specific binding of the input port. Distinct input ports may have different user-defined receive semantics.

In order to use an owned port P for input, that is to receive messages, a process should bind port P to the following three entities:

- A *destination buffer*.
- A *receiver handler*.
- An *error handler*.

A port must be bound for input *before* being used to receive messages. Messages hitting a not yet input bound port are discarded.

The first operation carried out by the GAMMA driver upon each packet arrival is to extract the packet header from the NIC input FIFO and store it in a temporary structure in kernel space. This is needed for header inspection, which allows the driver to separate GAMMA packets from IP packets and to manage the payload subsequently (see next Section).

4.4.3.1 Destination buffer

The *destination buffer* is a contiguous region of virtual memory in the memory space of the user program (an array, for instance). Non-empty messages hitting port P will be stored into such a buffer. Both initial address and lengthh of the buffer must be specified. It is not mandatory to specify a destination buffer if only empty messages are expected to hit the port.

If a destination buffer has been specified, it must be pinned down and prefetched into physical RAM (Section 4.7.15).

After a pinned-down destination buffer is bound to a port P, the local instance of the GAMMA driver knowns the place in physical RAM where non-empty messages hitting port P must be stored. Therefore, when the GAMMA driver detects frame arrivals for port P, the frame payloads can be copied directly from the NIC's receive FIFO to their final destination with no temporary storage in kernel space and regardless of the schedule state of the receiver process at the arrival time.

If a message hitting port P fits the destination buffer of P exactly, the next message hitting port P will be stored at the beginning of the same destination buffer, thus overwriting the current one, unless the port has been re-bound to a different destination buffer meanwhile. If the message is larger than the destination buffer, then the GAMMA driver truncates it to fit the buffer and runs the error handler to raise an error condition (see below). If the message is shorter than the destination buffer, then the next message hitting port P will be stored contiguously after the current message, using the remaining part of the same destination buffer, unless the port has been re-bound to a different destination buffer meanwhile. This behaviour helps build gather-like communication patterns. However, if such a new message is larger than the available remaining room in the destination buffer, then the GAMMA driver truncates it to fit the buffer and runs the error handler to raise an error condition.

4.4.3.2 Receiver handler

The *receiver handler* is an application-defined function. It will be executed each time a message hits port P, more precisely after the message body (if any) has been copied to the destination buffer (if any). Empty messages hitting port P will trigger the receiver handler as well.

It is not mandatory to specify a receiver handler.

As soon as the last packet of a message hitting port P has been stored into the corresponding destination buffer, the GAMMA driver can immediately run the receiver handler bound to P so as to allow a real-time management of the message itself. The receiver handler is run on the interrupt stack in non-interruptable mode and at kernel privilege level, as if it actually were a user-defined piece of the device driver itself. As a consequence, new messages for the port will not be moved to the destination buffer until the current run of the receiver handler has completed. Moreover, every data structure referred to by any handler must have been previously locked and prefetched into physical RAM (see Section 4.7.15).

If the receiver process is not currently running, a temporary context switch is performed by the GAMMA driver so as to ensure that the handler runs in the right context.

Clearly if an input port is bound to a receiver handler, the port is to play an active user-defined role upon each message arrival. This is the reason why GAMMA ports are called "active ports".

The possibility of running the receiver handler in real time upon message arrival is crucial in order to prevent a subsequent message hitting port P from overlapping the current one in the same destination buffer, if this is to be avoided. Indeed a typical receiver handler:

- integrates the current message into the main thread of the destination process;
- notifies the message reception to the main thread itself;
- selects a fresh destination buffer in user space for the next incoming message and re-binds port P to it.

However, due to the uninterruptability of GAMMA receiver handlers, a network overrun may occur on a receiver NIC as a result of running a long-lasting handler. While it is up to the user not to program long-lasting receiver handlers in the current prototype of GAMMA, it is clear that the only way to overcome the obvious reliability problem is to enhance the GAMMA protocol with some form of flow control (see Section 7.1 and [CC99]).

4.4.3.2.1 The security breach of GAMMA receiver handlers Another open issue is related to the security breach caused by running user-programmed handlers at kernel privilege level in non-interruptable mode. It is clear that few people would use GAMMA "as is" with this security hole. However, this mechanism allows a "research user" to experiment various user-programmed receive policies/algorithms in a flexible and low-overhead way. A "real user" would like to use an efficient but high-level and industry-standard API like MPI. The GAMMA mechanisms allow us to implement MPI atop GAMMA by writing some time-critical pieces of code as handlers. Once everything works, there is no reason to keep such pieces of code at user level: They should go down to driver level, and then it would become unnecessary to expose the handler mechanism to user level.

4.4.3.3 Error handlers

GAMMA communications rely on a very simplified communication protocol with neither error recovery nor flow control. This protocol is able to do some error detection but will not perform automatic (and time-consuming) error recovery. The choice of GAMMA for communication errors is to allow applications to be notified about communication errors and have their own specific error management policies, if required. This is supported by allowing applications to bind each input port to an *error handler*. An error handler is an application-defined function. It is very similar in concept to the receiver handler, but runs each time a corrupted packet hits the port, or when a send operation issued from the port fails, or when a packet loss is detected.

It is not mandatory to specify an error handler.

4.4.4 How a GAMMA process waits for incoming messages

Like with any Active Message-like communication system, a GAMMA receiver handler acts as an independent thread of the application triggered by message arrivals rather than being invoked by receive operations. This means that the programmer has to spend an additional effort to ensure that receiver handlers correctly cooperate with the main process thread. A very frequent problem is when the main thread needs to synchronize with a message arrival before continuing computation (e.g. when the process needs to receive data before processing them). A general solution is to use application-defined synchronization flags as follows:

1. A flag F of the application is initially reset.

2. In order to wait for one incoming message from a port P, the receiver process starts a busy wait in a loop until F is set.

3. The receiver handler bound to port P sets F upon message arrival.

GAMMA offers a more flexible and reliable solution in the form of two semaphore-like library functions, namely `gamma_wait()` (Section 4.7.8) and `gamma_signal()` (Section 4.7.6). These functions give safe access to per-port semaphores embedded into the GAMMA library. The example above becomes as follows:

1. In order to wait for one incoming message from port P, the receiver process issues `gamma_wait(P,1)`

2. The receiver handler bound to port P issues `gamma_signal(P)` upon message arrival.

4.5 The optimization story of GAMMA

4.5.1 Basic optimizations: light-weight system calls and fast interrupt path

As already discussed above, with GAMMA the user-kernel boundary is crossed through a small set of additional (w.r.t. the standard Linux kernel) light-weight system calls. All GAMMA calls are built on top of such system calls, activated using the trap address `0x81` (like with the Fast Ethernet implementation of U-Net [WBvE96]) which traps down to kernel in the GAMMA device driver through a short and fast code path. Moreover, upon Interrupt Request (IRQ) the CPU calls the interrupt routine of the GAMMA driver through a fast interrupt path.

4.5.2 Pre-computed packet headers

As an additional optimization, as soon as a port P is bound for output to a destination, GAMMA precomputes the header for Ethernet frames that are to be transmitted upon send operations issued through P. This way the overhead of the GAMMA protocol for segmenting long messages into a sequence of frames upon transmission is greatly reduced. Moreover, sending a GAMMA message only requires specification of a previously bound output port, besides message pointer and size, with a saving in latency time due to fewer parameters being passed.

4.5.3 Non-standard GAMMA header versus IEEE 802.3 compliant header

With GAMMA, the standard 14 byte long IEEE 802.3 frame header is augmented with GAMMA specific information (parallel PID, destination port number, sequence number, size of the carried payload fragment, whether it is the last packet of a fragmented message or not), peppe completare encoded into additional six bytes. The layout of the resulting 20 byte long header is represented in Table 4.1.

The IEEE 802.3 header carries the MAC address of the sending NIC, besides the MAC address of the destination NIC. Such information is exploited by most modern switches, which may "learn" about what NIC is connected to what port by inspecting the headers of all packets they switch. However, this information is useless with repeater hubs. Saving on the transmission of the six byte source MAC address may be a significant optimization with repeater hubs, since the time needed to send it is comparable to the hardware latency of

Size (bit)	Meaning
48	source MAC address
48	destination MAC address
16	GAMMA protocol id
8	parallel PID
8	destination port
11	size of payload in byte
2	unused
1	credit request (flow control only)
1	is this the first packet of a message?
1	is this the last packet of a message?
8	packet sequence number
8	instance number of sender (flow control only)

Table 4.1: GAMMA header, IEEE 802.3 compliant version (20 bytes). The middle line separates the IEEE 802.3 part from the GAMMA specific part.

the hub itself (at least if the NIC is driver by Programmed I/O). Omitting the source MAC address does not affect packet deliverability, as the destination MAC address is sufficient to this end. This leads to the more compact, non-standard 12 byte header currently used by GAMMA, whose layout is represented in Table 4.2. The choice between standard and optimized GAMMA header can be done when compiling the GAMMA driver.

4.5.4 Minimal-copy communication path

Any messaging system based on the send-receive paradigm, as well as any implementation of Active Messages which does not address issues related to the schedule state of the receiver process (see [PLC95] for instance) must provide at least one level of temporary message buffering on the receiver side of a communication. The reason for this is that message consumption may not promptly occur at message arrival: the receiver process may not be running at that very time and thereof the execution of a receive operation/handler may have to be deferred. Usually such intermediate buffering is hidden in the system-level communication software layers.

However, Active Messages would in principle eliminate the need for *any* intermediate message buffering, even on the receiver side, if the execution of the receiver handler could be triggered immediately upon message arrival regardless of the schedule state of the receiver process. Indeed in this case the incoming message would be promptly consumed, and tem-

Size (bit)	Meaning
48	source MAC address
2	bits 0 and 1 of GAMMA protocol id
6	bits 2 to 7 of instance number of sender (flow control only)
8	packet sequence number
8	parallel PID
8	destination port
11	size of payload in byte
2	bits 0 to 1 of instance number of sender (flow control only)
1	credit request (flow control only)
1	is this the first packet of a message?
1	is this the last packet of a message?

Table 4.2: GAMMA optimized header (12 bytes). The middle line separates the only surviving chunk of the IEEE 802.3 header from the GAMMA specific part.

porary buffering on the receiver side would be no longer needed. Such optimization would allow a minimal-copy communication path, which ideally could even be a true zero-copy one (however, most NICs store incoming packets into receive FIFOs which may or may not be regarded as a temporary storage for message chunks; this is the reason why we prefer the term "minimal-copy" instead of "zero-copy" or "one-copy").

With GAMMA Active Ports, both the user-space destination buffer and the program-defined receiver handler are made known to the NIC driver before communications take place. Upon each packet arrival, after having extracted the packet header for inspection, the NIC driver copies the payload from the NIC's receive FIFO to the user-space destination buffer directly, and activates the receiver handler on behalf of the receiver process, possibly issuing a temporary context switch in case it were not running at arrival time. A few drawbacks may affect this approach, namely:

- The memory region corresponding to the user-space destination buffer and the code of the receiver handler must be marked as non-swappable and must be prefetched into physical RAM before any message arrival.
- Receiver handlers must be really short pieces of code. If receiver handlers take too long a time for execution, they could become the bottleneck of the messaging system and message overflow would occur, unless explicit flow control is provided.

4.5.5 Optimistic protocol

GAMMA communications rely on an optimistic communication protocol which does not provide flow control. The "optimism" of the GAMMA protocol is partly justified by the following considerations:

- In a LAN environment with good quality wiring, the only source of frame corruption is collision with other frames in case of shared Ethernet. In any case, frames corrupted by collision have bad CRC and are discarded by the receiver GAMMA device driver. In case of collision, frame retransmission is automatically performed by the NICs up to sixteen times, after which an error condition is raised by the NIC. Therefore a simple test on send failures allows the driver to immediately retry a transmission which has failed due to excessive collisions, not to mention that using a switch instead of a repeater hub eliminates the problem altogether.

- If the handler on the receiver side of a GAMMA communication runs quickly enough (as usually is the case) both the RAM-to-NIC transfer rate on the sender side and the NIC-to-RAM transfer rate on the receiver side are much greater than the Fast Ethernet transfer rate. Therefore the flow is implicitly controlled by simply testing whether there is enough room in the transmit FIFO of the sender's NIC before writing a frame. Indeed GAMMA performs such a test and saturates the physical channel without any receiver overflow in case of "quick enough" receiver handlers.

- Frames are guaranteed to be delivered according to their transmission order.

Summarizing, a "well-written" GAMMA parallel program running on a "well-cabled" switched cluster should not incur communication errors as long as a single GAMMA application is run in the cluster. However nothing prevents the programmer from writing a time-consuming receiver handler which slows down the GAMMA driver upon receive, or from running an application which generates communication hot-spots on a LAN switch, or from running multiple GAMMA applications simultaneously in the same cluster. In the first case, packet overflow may occur at the receiver NIC. In the second case, packet overflow may occur at the switch itself. In the last case, overflow may occur at both devices. In all cases a communication error may occur.

The choice of GAMMA for communication errors is to implement some error detection mechanisms and allow applications to have their specific error management policies, if required. This is supported by allowing applications to bind their input ports to error handlers (Section 4.4.3.3). However, it is clear that some form of flow control [CC99] is the only safe answer to the lack of reliability of GAMMA, especially in the case of multiple GAMMA applications running simultaneously on the same cluster. This is currently under investigation (Section 7.1). However, the case of one single "well-written" GAMMA application running on a cluster at full communication speed is not that unrealistic; this is the reason why newer prototypes of GAMMA provide both flow-controlled and "unreliable" full-speed communications.

4.5.6 Performance results of earlier prototypes

The communication performance of the earliest prototype of GAMMA [CC97c], called GAMMA 0.1, was quite good although not excellent. On Pentium 133 MHz clusters, one-way user-to-user latency as measured by a simple "ping-pong" GAMMA program was 33.5 μs, and asymptotic bandwidth was 9.9 MByte/s, significantly better than Linux TCP/IP sockets at that time but not optimal. GAMMA 0.1 exploited Programmed I/O instead of DMA for moving messages between host memory and NIC.

The subsequent version of GAMMA, namely GAMMA 0.2, exploited CPU-driven DMA transfers (Section 3.2.1) plus precomputed headers and more careful code and data alignments. GAMMA 0.2 yielded 12.1 MByte/s asymptotic bandwidth, a great performance improvement for large messages. One-way user-to-user latency was 23.5 μs, about six times smaller than Linux TCP/IP sockets, and yet in the range of many messaging systems running on NOWs and MPPs. The throughput curve of GAMMA 0.2 is reported in Figure 4.1, curve number 2.

It must be noted that the particularly poor performance of Linux TCP/IP at that time was due to the fact that the "old" 3c595 Fast Ethernet NICs allowed only Programmed I/O and CPU-driven DMA transfer modes. Almost all today's NICs support DBDMA (Section 3.2.1), thus allowing the same Linux TCP/IP implementation to perform much better. However, the Linux TCP/IP stack has recently been re-engineered for even better performance. Clearly the TCP profile depicted in Figure 4.1 is no longer valid.

Based on the DMA version of the first GAMMA prototype we have implemented a "ping-pong" application where incoming messages are received in one buffer then moved to a different buffer in order to simulate a temporary copy of each message on the receiver side under the assumption of CPU-driven DMA transfers. The throughput curve of such simulated buffered messaging system is reported in Figure 4.1, curve number 1. The performance degradation is clearly apparent: a throughput loss of more than 3 MByte/s (24% of the raw 100base-T bandwidth) is experienced by large messages, and the throughput curve is much closer to the one of Linux TCP/IP. This provides evidence of how much better a minimal-copy solution may perform on 100base-T LAN with respect to more traditional, buffered solutions adopted by most industry standard protocols.

4.5.7 Filling up the communication pipeline

From curve 2 of Figure 4.1 the throughput profile of GAMMA 0.2 appears to increase quite slowly for messages of increasing size up to 1500 bytes, where the throughput achieves 6.6 MByte/s; then the throughput for larger and larger messages appears to increase much faster. The reason for such behaviour is simple: the communication path between the sender and the receiver nodes is a pipeline comprising the following three stages:

1. The RAM-to-NIC path on the sender node, operated by a DMA engine.
2. The path from the sender NIC to the receiver NIC, where packets flow along the physical LAN according to the Ethernet protocol.
3. The NIC-to-RAM path on the receiver node, operated by another DMA engine which moves data to the destination buffer.

In the GAMMA protocol, messages longer than 1500 bytes are segmented on send and reassembled on receive in order to fit the maximum allowed size of Ethernet frames. A long, segmented message counting many Ethernet frames fills up the pipeline better than a short, unsegmented one. Consequently the throughput increases rapidly for increasing message size beyond 1500 byte due to pipeline filling. We could also say that as messages become longer than 1500 bytes the behaviour of the communication path turns from "store-and-forward" to "cut-through", due to message fragmentation.

A better exploitation of the above pipeline structure would lead to better throughput for small to medium size messages. A good communication throughput for small messages is of great importance, as it would allow a finer grain of distribution in parallel applications.

One possible approach to improve the exploitation of the communication pipeline is to implement a finer-grain message fragmentation, by allowing even messages shorter than 1500 bytes to be fragmented into small frames (as already proposed in [CC97b]). Another possibility is to properly program the NIC driver so as to enable what we call the "early send" and "early receive" capabilities that virtually all modern NICs offer. With "early send" enabled, a NIC starts transmitting an Ethernet frame as soon as the first few bytes of a packet are uploaded to its own transmit FIFO, rather than waiting for the whole packet to be completely uploaded. With "early receive" enabled, a NIC raises an IRQ on frame reception as soon as the first few bytes of the frame entered its own receive FIFO, rather than waiting for the whole frame to enter. By enabling "early send" on the sender side and "early receive" on the receiver side, the behaviour of the whole communication path turns from "store-and-forward" to "cut-through" even with very short packets, therefore the throughput curve is expected to improve, especially for small to medium size messages. The latency time is expected to decrease as well.

Enabling the "early receive" feature poses a number of additional challenges. Indeed, early receiving a packet header means that the GAMMA driver trusts the header information at a time when the correctness status of the packet is not yet known. Eventually, the driver will realize that the packet was to be discarded, but this will occur when the (possibly corrupted) payload has already been processed and stored at some (possibly wrong) destination. In this unlucky case, the GAMMA protocol implemented by the driver can "backtrack" most of the current packet processing, with the obvious exception of a possible memory copy carried out to a wrong destination buffer (a wrong information about message size may not cause message overflow in a destination buffer; see Section 4.4.3.1). For safety reasons, a receiver handler can only be run if the correctness check for the last receiver packet was successful; indeed, it would be impossible to backtrack from a complete run of a user-programmed receiver handler.

The communication performance of GAMMA noticeably improved after enabling the "early send/receive" capabilities of the NICs. With the next version of GAMMA, namely GAMMA 0.3, one-way user-to-user latency dropped from 23.5 to 16.8 μs. Curve 3 in Figure 4.1 reports the GAMMA 0.3 throughput profile. The improvement of throughput over GAMMA 0.2 for small to medium size messages after enabling "early send/receive" is clearly apparent. For instance, throughput for 1500 byte messages increased from 6.6 to 10.9 MByte/s, an improvement of more than 65%. The half-power point shifted down from 670 to 192 bytes.

It must be said that the favourable impact of "early receive" on the overall communication performance is expected to loose its importance in the future, due to expected improvements in the memory architecture of future PCs which will greatly reduce the overhead of small data movements.

4.5.8 Polling the network device for incoming messages

Even when using fast interrupt paths (Section 3.2.2), the time overhead resulting from IRQ management is quite high. In most off-the-shelf PC architectures the hardware latency for IRQs is not negligible, on the order of a few microseconds. Such *IRQ latency* adds to communication latency. With proper arrangement of the parallel program, such additional latency may be hidden by overlapping it with the ongoing computation on the receiver process. However there are some cases in which this is not possible, especially when a synchronous receive must occur.

The only possible way to avoid the IRQ latency is to explicitly poll the communication device for incoming messages instead of relying on IRQ-based asynchronous mechanisms (see Section 3.2.2.9). Of course, polling the NIC may be done only on synchronous receives, which, however, are the only cases when IRQ latency may not be hidden.

The delay in interrupt servicing due to IRQ latency results in buffering the initial part of the incoming packet into the NIC's receive FIFO. However the DMA transfer started by the interrupt service routine of the NIC driver may consume such buffered chunk at the full speed allowed by the DMA engine. If the maximum DMA transfer rate is higher than the physical LAN throughput and the "early receive" feature of the NIC is enabled, the initial delay due to the IRQ latency is rapidly recovered by the fast DMA transfer even with short messages. This is the reason why by implementing a polling mechanism in GAMMA we were expecting only a latency improvement, with a very limited influence on the overall throughput profile.

Our latest version of GAMMA, namely GAMMA 0.4, implements a polling mechanism partly at kernel level and partly at library level. At kernel level, the polling loop runs uninterruptably but only for a short-duration (50 μs) time interval. Such short-duration uninterruptable polling loop is issued by invoking a dedicated light-weight system call. The `gamma_wait()` library function, issued by applications on synchronous receive phases, invokes the polling system call repeatedly in a library-level, and hence interruptable, loop.

As expected, the impact of polling on the throughput curve has been negligible. The most important achievement is that one-way user-to-user latency has dropped from 16.8 to 12.7 μs, due to saving the IRQ latency (about 4 μs). With this last optimization GAMMA becomes the fastest messaging systems currently running on 100base-T Ethernet, with unprecedented performance in terms of latency as well as throughput. The closeness of the throughput curve to the optimal 100base-T Ethernet throughput curve (curve 4 in Figure 4.1), computed assuming a minimum one-way user-to-user latency of 7 μs, accounts for the extreme degree of efficiency of the GAMMA communication layer.

4.6 Conclusions and open issues

We have shown the impressive communication performance achieved by a carefully optimized OS-level implementation of Active Message-like messaging in low-cost clusters of PCs running Linux. In terms of latency and bandwidth as well as throughput for intermediate size messages, GAMMA delivers communication performance which is unprecedented on 100base-T Ethernet. Especially the very good throughput profile in the region of short messages allows GAMMA to exploit a finer grain of parallelism than any other cluster platform based on Fast Ethernet technology.

An unexpected result of GAMMA is that it contradicts the common belief that only a "user-level" communication architecture may yield a satisfactory answer to the demand for efficient inter-process communication, at least with low-cost LANs. The GAMMA architecture based on a communication layer largely embedded in the kernel of a standard OS outperforms any other low-cost NOW architecture. This happens because we have

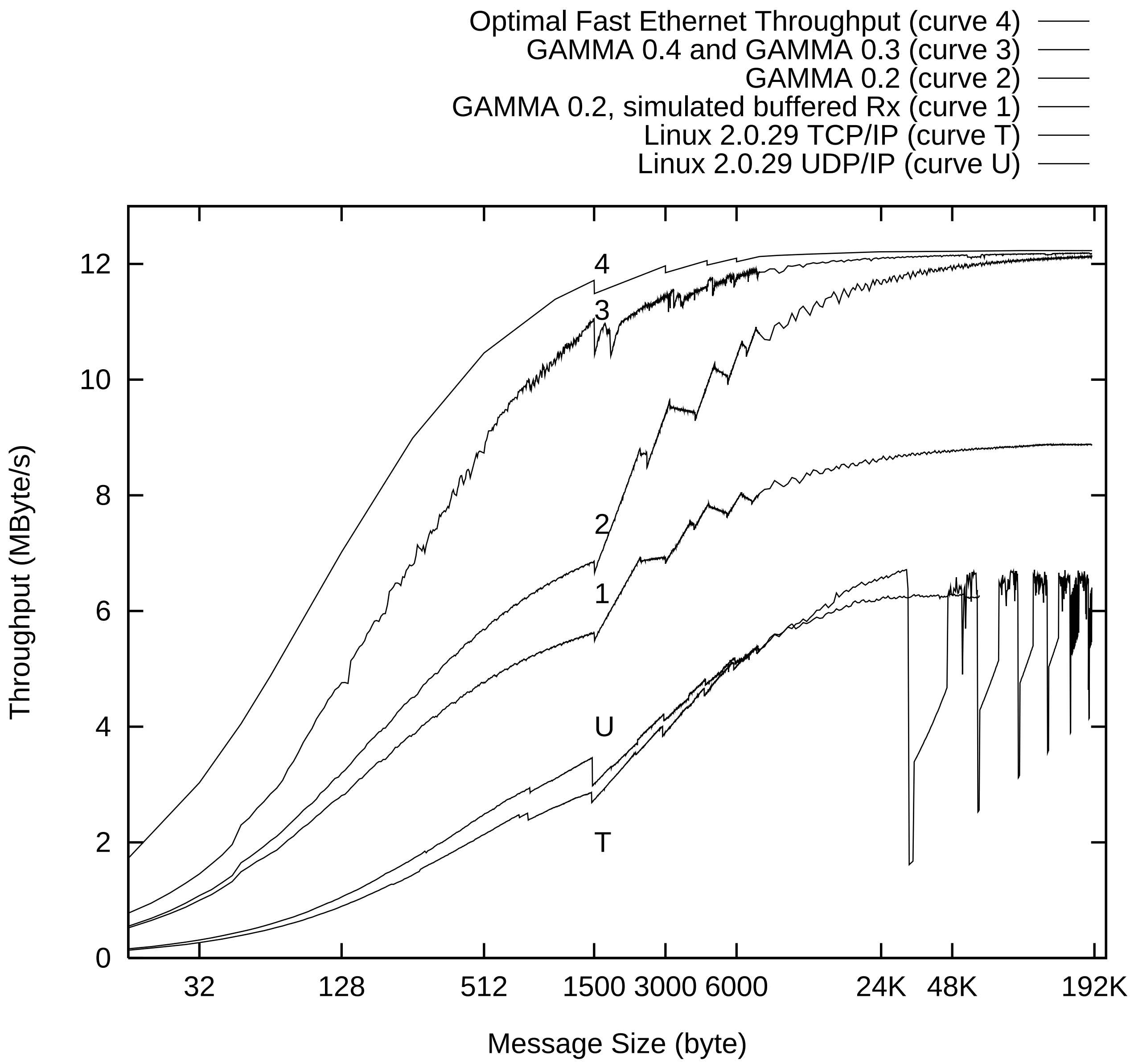

Figure 4.1: "Ping-pong" throughput curves of various versions of GAMMA compared to Linux TCP/IP and Linux UDP/IP on the same hardware platform.

found the answer to what is by far the most critical issue in commodity-based NOWs, namely to provide an optimized implementation of the "right" communication mechanisms providing the proper virtualization of network devices. The choice of level (user-level or kernel-level) that such mechanisms are to be placed at in the overall OS architecture is only of secondary concern when leveraging commodity components, given that such components cannot currently be closely integrated with the host CPU and memory hierarchy.

Currently a GAMMA cluster cannot compete with massively parallel platforms in terms of aggregate throughput as well as scalability. However, we argue that the cost of scalability characteristics of commercial MPPs is hardly justified in the vast majority of applications where parallel processing would, in principle, make sense. Only very few special applications would really be worth the cost of the scalability characteristics that a typical MPP interconnect can provide. The excellent achievements of the GAMMA project show that low-cost clusters of PC may become a valid support for parallel processing in a much wider range of application domains. Moreover, in the mid-term Gigabit Ethernet is expected to replace the current 100base-T Ethernet technology, with a substantial improvement in aggregate throughput that GAMMA should immediately provide after a low-effort porting of the GAMMA device driver to a new NIC.

Many of the optimization techniques that we have discussed before are of general interest in implementing communication layers and NIC drivers. Indeed techniques like a finer segmentation of messages as well as the exploitation of "early send/receive" features of the network devices help increase the communication performance of any messaging system running in a LAN environment, especially for small to medium size messages. The adoption of light-weight communication protocols, made possible by the reliability and the simplified device addressing of modern LANs, has been already explored by many authors (see [MRV+97, RAC97] for instance). Minimal-copy protocols are not a new idea, although they require suitable low-level communication mechanisms in order to be effectively implemented and work in a multi-tasking OS. Nevertheless, in our opinion the GAMMA research action has the merit of having tackled performance issues of cluster-wide communications in quite a deep and thorough way, providing new insights and a successful approach to efficient cluster computing.

Here is a short outline of some open issues in the GAMMA project, which will be discussed more in detail in Chapter 7:

- Studying and implementing efficient, low-overhead flow control policies to improve the reliability of GAMMA communication [CC99].
- Implementing multicast communications in addition to broadcast.
- Enhancing the GAMMA programming library with a fairly complete suite of collective communication routines (gather, scatter, reduce) and additional basic mechanisms

(N-way gather/scatter communication) in order to support a performance-oriented implementation of MPI for Fast Ethernet atop GAMMA.

- Porting GAMMA to more modern 100base-T Ethernet NICs supporting the DBDMA transfer mode. This is currently in progress for 100base-T NICs based upon the DEC 21143 chipset and the Intel EtherExpress Pro NIC [taUdRLS98, CC99].
- Implementing thread safety at the level of GAMMA device driver.
- Experimenting with Gigabit Ethernet technology.

4.7 The GAMMA programming interface

The GAMMA communication library provides functions and variables for process grouping, point-to-point communication, and collective communications at the application level. Both C and FORTRAN calls are provided.

This section describes the functions and variables of the GAMMA communication library. Only the C interface is described.

Each library function, with the exceptions of `gamma_time()` and `gamma_time_diff()`, returns a negative value in case of error, and a non-negative one in case of successful completion. Currently the negative value possibly returned by a GAMMA routine is not informative of what kind of error condition has been detected.

The current implementation is a static library named `libgamma.a`.

4.7.1 gamma_init()

```
int gamma_init( int num_nodes, int argc, char **argv );
```

A parallel computation is started.

As a sequential user process P invokes it, a GAMMA process group is activated. The group is composed of process P, running on the local workstation, plus additional `num_nodes`-1 processes identical to P launched on `num_nodes`-1 distinct remote workstations (chosen by those found in file `/etc/gamma.conf`) via the `rsh` command.

Hence after having invoked `gamma_init()` the invoking user process P is replicated on `num_nodes` workstations in the cluster, thus forming a running SPMD parallel application.

The process replicas themselves eventually invoke `gamma_init()`, but this time the effect is that of registering themselves with the created group, without creating new ones.

A positive number called *parallel PID* uniquely identifies the newly created process group in the cluster. The parallel PID is *not* returned as a result of `gamma_init()`; rather, it is returned by the `gamma_my_par_pid()` function.

Note that nothing prevents two independent user processes P and Q to issue a call to `gamma_init()` separately from one another. This will result in the creation of two distinct GAMMA process groups in the same cluster, each with a distinct parallel PID. The two groups may share some or even all the available workstations in the cluster, but cannot share processes.

Currently invoking `gamma_init()` with `num_nodes` less than or equal to zero or greater than the total number of workstations connected to the cluster has the same effect as if `num_nodes` were equal to the total number of workstations connected in the cluster.

4.7.2 gamma_setport()

```
int gamma_setport(
                    unsigned char activated_port,
                    unsigned char dest_node,
                    unsigned char dest_par_pid,
                    unsigned char dest_port,
                    void (*handler_in)(),
                    void (*handler_err)(),
                    void *buffer_in,
                    unsigned long buffer_len
);
```

Binding of one of the 256 *active ports* of the calling process, numbered from 0 to 255. Port numbers from 247 on are currently reserved for GAMMA collective routines.

The calling process must have previously invoked `gamma_init()`.

Port `activated_port` may be bound for output, input, or input/output.

Binding port `activated_port` for output requires specifying a port of a remote process acting as receiver side. The remote port should be bound for input or input/output. The remote port is fully specified by the triple `dest_node` (instance number of the receiver process), `dest_par_pid` (parallel PID of the process group the receiver process belongs to), and `dest_port` (a specific input port of the receiver process). Parameter `dest_node` may be set to the constant `BROADCAST` (defined as the conventional instance number 255). In this case, each message transmitted through `activated_port` will be broadcast to each process in the group specified by `dest_par_pid` (excluding the sender itself). In this case each receiver process will get the message through its locally owned port specified by `dest_port`.

Communication between distinct GAMMA groups is allowed.

Binding port `activated_port` for input may require specifying a *destination buffer*, a *receiver handler* and an *error handler*.

The *destination buffer* is a contiguous virtual memory region in application space (e.g. an array), specified by start pointer `buffer_in` and size in bytes `buffer_len`. A non-empty message hitting port `activated port` will be stored into such buffer. It is not mandatory to specify a destination buffer if only empty messages are expected to hit the port.

If the currently arriving message fits the destination buffer exactly, the next message hitting the same port will be stored at the beginning of the same destination buffer, thus overwriting the current one (unless the port has been bound to a different destination buffer meanwhile).

If a destination buffer was not specified and nevertheless a non-empty message hits the port, then the error handler (see below) is executed to raise an error condition (see `gamma_errno`).

If the currently arriving message is larger than the destination buffer, then the message is truncated to fit the buffer and the error handler (see below) is executed to raise an error condition (see `gamma_errno`).

If the currently arriving message is shorter than the destination buffer, then the next message hitting the port will be stored contiguously after the current message, using the remaining part of the same destination buffer (unless the port has been bound to a different destination buffer meanwhile). This helps build gather-like communication patterns. However, if such a new message is larger than the available remaining room in the destination buffer, it is truncated to fit the buffer and the error handler (see below) is executed to raise an error condition (see `gamma_errno`).

Currently the destination buffer must have been locked and prefetched into physical RAM before use (see Section 4.7.15).

The *receiver handler* specified by `handler_in` will be executed each time the last chunk of a new message has been successfully copied to the destination buffer. Empty messages will trigger the receiver handler as well. New messages for `activated_port` will *not* be delivered to the destination buffer until the receiver handler has run to completion. It is not mandatory to specify a receiver handler.

The *error handler* specified by `handler_err` is very similar in concept to the receiver handler, but is run each time a corrupted message hits the port, or when a `gamma_send()` operation issued from `activated_port` fails. It is not mandatory to specify an error handler.

Handlers are launched by the GAMMA driver and run with interrupts disabled at kernel privilege level. This imposes the constraint that every data structure referred to by any handler must have been previously locked and prefetched into physical RAM (see Section 4.7.15). Handlers may invoke any GAMMA call.

4.7.3 gamma_send()

```
int gamma_send(
                 unsigned char output_port,
                 void *data,
                 unsigned long len
);
```

A message is sent through the port `output_port`. The port is supposed to have previoulsy been bound for output. (see function `gamma_setport()`).

The message is supposed to be stored in a contiguous region of virtual memory in application space (e.g. an array), specified by start pointer `data` and size in bytes `len`. The region must have been locked and prefetched into physical RAM before use (see Section 4.7.15).

The content of the buffer is written in the NIC's transmit FIFO as one or more packets, without any intermediate copy in kernel space.

`gamma_send()` returns upon (successful as well as failed) end of physical transmission. Therefore its semantics is blocking.

4.7.4 gamma_send_fast()

```
int gamma_send_fast(
                      unsigned char output_port,
                      void *data,
                      unsigned long len
);
```

A message is sent through the port `output_port`.

This call is very similar to `gamma_send()`. The only difference is that it returns immediately after the last chunk of the message body has been loaded into the NIC's transmit FIFO, without waiting for the physical transmission to occur. Therefore `gamma_send_fast()` will not necessarily detect a possible failure occurring in the last phases of a physical transmission.

For forward compatibility, the semantics of `gamma_send_fast()` is intended to be non-blocking, although it is currently implemented as blocking.

4.7.5 gamma_attach_buffer()

```
int gamma_attach_buffer(unsigned char input_port, void *buffer_in);
```

Port `input_port` is bound to the destination buffer specified by pointer `buffer_in`. The next messages hitting the specified port will be stored into this buffer. The buffer must have been previously locked and prefetched into physical RAM (see Section 4.7.15).

This is a low-overhead alternative to the `gamma_setport()` function. It does not require any system call to be invoked since the buffer address is actually kept in the user data segment. Its intended use is within receiver handlers, in order to prepare a fresh destination buffer after having consumed the previous one. The size of the destination buffer cannot be changed with this function. It is an error to use `gamma_attach_buffer()` without having specified the buffer size by a previous call to `gamma_setport()`.

4.7.6 gamma_signal()

```
int gamma_signal(unsigned char sem);
```

In order to allow a receiver process to cooperate and synchronize with receiver handlers in a safe way, GAMMA provides 256 per-process *semaphores* numbered from 0 to 255. Semaphores from 247 on are reserved for GAMMA collective routines.

GAMMA semaphores are initialized to zero by `gamma_init()`.

`gamma_signal(sem)` causes semaphore `sem` to be atomically incremented by one.

Typically this function is issued by a receiver handler in order to signal the arrival of a message to the main thread of the receiver process.

4.7.7 gamma_sigerr()

```
int gamma_sigerr(unsigned char sem);
```

In order to allow a receiver process to cooperate and synchronize with error handlers in a safe way, GAMMA provides 256 per-process *error semaphores* numbered from 0 to 255. Error semaphores from 247 on are reserved for GAMMA collective routines.

GAMMA error semaphores are initialized to zero by `gamma_init()`.

`gamma_sigerr(sem)` causes error semaphore `sem` to be atomically incremented by one.

Typically this function is issued by an error handler in order to signal the arrival of a corrupted message or the failure of a send operation to the main thread of a process.

4.7.8 gamma_wait()

`int gamma_wait(unsigned char sem, unsigned long n);`

The invoking process busy-waits until semaphore `sem` raises value `n`. Semaphore `sem` is atomically decremented by `n` upon return.

Typically this function is invoked by a process waiting for message arrivals. Semaphore `sem` is typically incremented by some receiver handler issuing `gamma_signal()`. During the busy wait, the NIC is polled for incoming frames so as to speed up message arrivals by avoiding IRQ overheads. However this is only an optimization, which does not change the semantics.

On return, `gamma_wait()` yields zero if no receive errors were encountered, otherwise it yields a negative number whose absolute value is the count of how many times the function `gamma_sigerr` has been issued on error semaphore `sem` since last run of `gamma_wait`.

4.7.9 gamma_test()

`int gamma_test(unsigned char sem);`

Returns the current value of semaphore `sem`. The value of `sem` is left unchanged.

4.7.10 gamma_atomic()

`int gamma_atomic(void (*funct)(void));`

Function `funct` is executed atomically, that is it will not be interrupted by either the scheduler or any receiver or error handler.

This allows for any function of the user program to be issued safely in case it shares data structures with receiver/error handlers. Currently each function to be executed atomically is constrained to the same restrictions as receive and error handlers are.

4.7.11 gamma_sync()

`int gamma_sync(void);`

Barrier synchronization among all processes within a process group.

After calling `gamma_sync()`, the caller process resumes execution successfully (that is, without error code) only when all other processes in the same group of the caller have reached the `gamma_sync()` function.

Exploiting a two-token synchronization mechanism, the GAMMA implementation of this collective communication primitive achieves best performance over shared Fast Ethernet channels [CC98c].

4.7.12 gamma_my_par_pid()

```
int gamma_my_par_pid(void);
```

Returns the parallel PID of the GAMMA process group of the caller, as assigned by the previous call to function `gamma_init()`.

4.7.13 gamma_my_node()

```
int gamma_my_node(void);
```

Returns the instance number of the calling process, relative to the GAMMA process group of the caller.

If the group contains `num_nodes` processes, the returned value will be in the range from 0 to `num_nodes`-1. The process which created the process group always has instance number zero.

The programming paradigm supported by GAMMA is SPMD. In this paradigm, each process may differentiate its behaviour by testing its own instance number.

4.7.14 gamma_how_many_nodes()

```
int gamma_how_many_nodes(void);
```

Returns the number of process instances belonging to the GAMMA process group of the caller process.

4.7.15 gamma_mlock()

```
int gamma_mlock(void *buffer, unsigned long len);
```

This function prefetches and locks into physical RAM a contiguous region in the virtual memory of the calling process starting from address `buffer` and counting `len` bytes.

Usually such a contiguous memory region is either a store for incoming messages or a global variable accessed by a receiver or error handler. It must be prefetched and locked into physical RAM in order for the GAMMA driver not to incur a page fault while storing an incoming message or running a handler.

`gamma_mlock()` adds the prefetch functionality to the standard UNIX `mlock()` function.

4.7.16 gamma_munlock() and gamma_munlockall()

```
int gamma_munlock(void *buffer,unsigned long len);
int gamma_munlockall(void);
```

These functions unlock previously locked memory regions.

They are very similar to the standard UNIX `munlock()` and `munlockall()` calls.

4.7.17 gamma_exit()

```
int gamma_exit(void);
```

The invoking process terminates the parallel computation, exiting from its process group. The process who created the group (and got instance number 0) will destroy the group as soon as every other process instance has left the group.

4.7.18 gamma_time()

```
void gamma_time(time_586 t);
```

The contents of Pentium's register TSC is copied to variable `t`. Type `time_586` is defined as

```
struct { unsigned long hi; unsigned long lo; }
```

Register TSC is incremented by one at each CPU clock tick, so this function is useful for time measurements involved in performance evaluations.

4.7.19 gamma_time_diff()

```
double gamma_time_diff(time_586 b, time_586 a);
```

The time interval between instants `b` and `a` (possibly recorded by means of the `gamma_time` function) is computed in microseconds and returned as the result.

Currently the conversion from CPU clock ticks to microseconds requires a constant named `CLOCK` to be set to the CPU clock frequency in MHz before compiling the GAMMA library. More information is in the `README` file enclosed with the GAMMA source code.

4.7.20 gamma_active_port

```
unsigned char gamma_active_port;
```

During the execution of a receiver handler or an error handler, this variable holds the number of the port which has triggered the execution of the handler itself.

4.7.21 gamma_active_errno

```
unsigned char gamma_errno;
```

During the execution of an error handler, this variable holds a number corresponding to the error condition which has triggered the execution of the error handler.

The values that `gamma_errno` may hold are:

- `GAMMA_ERR_FRAMELOST` (2) in case a chunk of a message was lost;
- `GAMMA_ERR_MSGTOOLONG` (3) in case the user-level memory region for storing an incoming message is either missing (that is, no buffer is registered with the receiving port) or no longer valid (that is, the buffer was filled by a previous message and not replaced with a fresh one).
- `GAMMA_ERR_SENDFAILURE` (4) in case the transmission of a message has failed, for instance due to network congestion.

Chapter 5

The GAMMA barrier synchronization algorithm

5.1 Introduction

The issue of efficiently supporting inter-process communication in NOW platforms has been addressed by a large number of research projects. To the best of our knowledge a comparable effort has not been devoted to the challenge of efficiently supporting collective communication and especially barrier synchronization, despite such cooperation mechanisms being of great importance in parallel processing applications.

A first approach is to port existing collective algorithms atop more efficient low-level communication protocols. The work [BDH$^+$97] is a typical representative of such an approach in the case of some MPI collective communication primitives. Such primitives were ported on a custom reliable point-to-point low-level communication protocol built just atop the IEEE 802.3 Ethernet protocol and offering efficient broadcast service directly mapped onto the Ethernet hardware broadcast. The implementation is based on “slow” 10 Mbps Ethernet. Absolute communication as well as synchronization performance appears to be not particularly brilliant. However their results are by no means comparable to experiments carried out on Fast Ethernet.

Another approach is to implement more efficient collective algorithms atop existing low-level communication protocols. The work [BI95] describes the experience of re-implementing some IBM PVMe collective routines on the IBM SP-2 using a simplified version of a complicated broadcast algorithm proved optimal under the “postal” performance model [BNK92]; the obtained performance improvement is significant only for small-size messages. The work [DM95] reports about an enhancement of the PVM barrier synchronization primitive based on re-implementing it atop broadcast UDP sockets. In this way, performance of PVM barrier is increased by 36% with as many as 32 processors on 10Mbps Ethernet; however the absolute performance remains quite unsatisfactory for real parallel processing purposes (more than 280 milliseconds to synchronize 32 processors).

The two approaches recalled above are mutually complementary in order to achieve best performance with collective communications. Unfortunately much of the efficiency may be lost if the implementation of a collective mechanism does not properly take into account the features and limitations of the underlying communication hardware, which hardly are properly represented in the existing performance models.

In this chapter we address the problem of efficiently supporting barrier synchronization in a network of PCs leveraging lowest-cost, shared 100base-T Ethernet technology. Although the trend is towards a continuous decrease in the price of switching devices, non-switched Fast Ethernet technology is still substantially cheaper than the switched one, and as such it is very frequently found in today's LANs. Moreover non-switched Fast Ethernet exhibits slightly lower latency time than the switched one, at the expense of aggregate throughput and scalability. Thus the cost/performance ratio of a shared Fast Ethernet-based NOW may be very appealing for the wide range of parallel applications which make use of short messages and need frequent synchronization among processes.

The results presented in this chapter appeared in [CC98c].

5.2 Two *naive* barrier protocols atop GAMMA

The most straightforward way to implement a barrier synchronization mechanism is to devise a distributed synchronization protocol and to implement it atop a point-to-point inter-process communication layer. If the native inter-process messaging system provides broadcast features, such features may be exploited to increase the synchronization performance. GAMMA provides both point-to-point and broadcast low-latency communication. All the barrier protocols that we have implemented atop GAMMA take great advantage from exploiting the GAMMA broadcast services. All the barrier protocols that we have implemented are based on the exchange of point-to-point as well as broadcast "empty" messages among processes in a SPMD static process group.

Hereafter N will denote the number of members in a GAMMA SPMD process group. Moreover the *master* process in the group (usually labelled by instance number zero) will play a role which is different from all the remaining processes in the group, called *slaves* from now on.

5.2.1 Concurrent barrier synchronization

In the *concurrent* barrier synchronization protocol each slave issuing a barrier primitive sends an "enter barrier" message to the master, independently of the activity of all the other slaves. Then it waits for an "exit barrier" message from the master. The master issuing the barrier primitive knows that N processes belong to the group and waits for $N-1$ "enter barrier"messages from slaves. The counter for such messages is implemented as a GAMMA semaphore primitive. As soon as the master gets the last of such $N-1$ messages it broadcasts a single "exit barrier" message to all the slaves in the process group, then it terminates the barrier synchronization and resumes normal activity. Each slave eventually receives the "exit barrier" message, then resumes its normal activity as well.

Our concurrent protocol involves sending N small frames over the shared Fast Ethernet channel, thus being theoretically optimal in terms of communications. Indeed it works fine when processes join the barrier synchronization at sufficiently different time instants from one another, that is when they are not already synchronized.

However in the execution of many parallel algorithms with well-balanced load among nodes and frequently issued barrier synchronizations the opposite occurs. Namely, processes tend to be already synchronized, or at least they happen to join a barrier synchronization during a quite short time interval. In this case the behaviour of the concurrent barrier protocol depends on the underlying LAN hardware: collisions may well occur among many "enter barrier" messages when leveraging shared Fast Ethernet LAN technology. Ethernet contention forces random waits and frame retransmissions, according to the standard CSMA/CD protocol. As a consequence, the barrier synchronization time becomes unacceptably large even with relatively few processors and a very light-loaded channel.

5.2.2 Serialized barrier synchronization

In the *serialized* barrier synchronization protocol all slaves but the one with highest instance number (namely $N-1$) behave the same, namely: when issuing the barrier primitive the slave with instance number i first waits for an "enter barrier" message from slave $i+1$, then it sends an "enter barrier" message to the process (slave or master) whose instance number is $i-1$. The slave with instance number $N-1$ simply sends an "enter barrier" message to slave $N-2$ right away upon issuing the barrier primitive. After sending its "enter barrier" message, each slave waits for the "exit barrier" message from the master (like in the concurrent barrier protocol). The master issuing the barrier primitive must first wait for a single "enter barrier" message from slave numbered 1, then it broadcasts a single "exit barrier" message to all the slaves in the process group, after which it resumes normal activity. Each slave eventually receives the "exit barrier" message, then resumes its normal activity as well.

The serialized barrier protocol is theoretically optimal in terms of communications, exactly as the concurrent protocol is. However, the former is aimed at avoiding contention over the Fast Ethernet bus in case of already synchronized processes, thus behaving in a totally different way from a practical point of view when run on our platform of choice. Contention is completely avoided by imposing a total temporal order among messages during the execution of the synchronization protocol. Of course the behaviour of the serialized barrier protocol is arguably not optimal in the case of non-synchronized processes as well as with switched LAN technology.

5.3 Towards an efficient barrier protocol for shared Fast Ethernet

In between the concurrent and the serialized barrier protocols a number of hybrid protocols may be devised with intermediate degrees of serialization among processes.

The general algorithm for such hybrid protocols works as follows. The set of slaves in a process group is equally partitioned into K subsets called *chains*, each comprising $(N-1)/K$ processes (here we assume that $N-1$ is a multiple of K for the sake of presentation simplicity). Slaves in each chain behave according to the serialized protocol. However the chains work independently of (hence concurrently with) one another during the execution of the barrier protocol. Each chain has a *leader*, which is the slave with lowest instance number in the chain. The leader of each chain cooperates with the master in the same way as the slave with least instance number in the plain serialized protocol. The master issuing the barrier primitive first waits for an "enter barrier" message from each of the K chain leaders, then it broadcasts one "exit barrier" message to the whole process group.

We shall call the above algorithm *K-chain* barrier synchronization protocol. The K-chain protocol trivially collapses to the plain concurrent protocol when $K = N-1$ and to the plain serialized protocol when $K = 1$.

With K ranging between 1 and $N-1$ one could in principle expect performance profiles intermediate between the profiles of the concurrent and the serialized protocols. Our original idea was to find a suitable value for K such that the obtained barrier protocol worked "fine" with synchronized as well as non-synchronized processes, by reducing the frequency of collisions without imposing an excessive degree of serialization in the protocol. Clearly we expected each parallel application would require its own optimal value for K, depending on the average degree of synchronization already exhibited by the processes when they join barrier synchronizations.

However, for some small values of K (say, in the range between two and four) we could expect better performance than the plain serialized protocol itself (corresponding to setting $K = 1$) even with already synchronized processes. In the case of $K = 2$ the reason for such expectation is the following. With already synchronized processes, the two process chains start their activity nearly at the same time. As a consequence a contention and possibly a collision might occur between the first two "enter barrier" messages arising from the two chains starting their activity. Such contention would force one of the two transmissions to be delayed w.r.t. the other one (in the case of an Ethernet collision both transmissions are delayed by a random time interval, and one of them will occur after the other with high probability). At this point the activities of the two chains turn out to be slightly desynchronized w.r.t. each other. This little amount of desynchronization is maintained along the entire execution of the barrier protocol, thanks to the Ethernet Carrier Sense feature which can now serialize transmissions concurrently performed by the two chains with no possibility of further collisions. Since each chain's activity is a sequence of physical transmissions interleaved with execution of low-level communication software, delaying one chain w.r.t. the other should allow overlapping physical transmission from one chain with software overhead from the other chain and vice-versa. Such "computation-to-communication" overlap cannot occur with the plain serialized protocol by definition, hence a possible advantage of 2-chain protocols over the plain serialized one. The same argumentation might in principle work also with $K > 2$, although in this case the Carrier Sense mechanism is not sufficient to prevent the occurrence of collisions.

5.4 Performance measurements and comparisons

We have implemented the concurrent as well as the serialized and the K-chain barrier synchronization protocols atop a working prototype of GAMMA equipped with 16 Pentium 133MHz PCs networked by a 100base-TX Ethernet repeater hub. On this system we carried out measurements of barrier synchronization time for each protocol in the case of already synchronized processes.

Time measurements were accomplished by means of the TSC register of the Pentium CPU which is incremented by one every clock tick. By reading the TSC content before and after the execution of the barrier primitive we were able to measure the execution time of that primitive with the accuracy of 1/100 of a microsecond.

Each protocol has been implemented as a function in a user-level C library. For each protocol we have performed 104 trials in a C program loop. Each trial resulted in a single time measurement. The measurement loop of our micro-benchmark looks as follow:

```
#define N_TRIALS 104
```

```
#define N_NODES 16

void main(int argc, char **argv) {
    TimeStamp start, end;

    gamma_init(N_NODES,argc,argv);

    for (i=0;i<N_TRIALS;i++) {
        gamma_sync();
        take_time(start);
        gamma_sync();
        take_time(end);
        record(time_interval(start,end));
    }

    gamma_exit();

    compute_mean();
}
```

The program is launched as a single sequential process on one node of the network. As soon as `gamma_init()` is issued, the process creates a process group, then launches `N_NODES-1` additional copies of the same program on distinct remote nodes via the UNIX `rsh` service, then joins the process group with instance number zero and issues a first (concurrent) barrier synchronization embedded in the `gamma_init()` function. Each copy starts execution from the beginning of the program, and as soon as it reaches the `gamma_init()` statement it joins the process group, issuing the embedded (concurrent) barrier synchronization. As soon as the last process has joint the process group the barrier synchronization succeeds, and execution of each process can continue in the framework of a SPMD process group. `gamma_sync()` is the GAMMA library function for barrier synchronization.

The first two measurements as well as the worst two ones are always discarded. The average execution time of barrier synchronization is then computed as the arithmetic mean among the remaining 100 measurements. From the code fragment above it is apparent that for each trial a preliminary barrier synchronization is issued before the actual barrier measurement. This way we could isolate and measure the overhead of GAMMA barrier synchronization, since processes were always already synchronized.

Figure 5.1 depicts the execution time curves of concurrent, serialized, 2-chain, 3-chain and 4-chain barrier synchronization protocols implemented atop GAMMA and issued by up to 16 already synchronized processors connected by a shared Fast Ethernet.

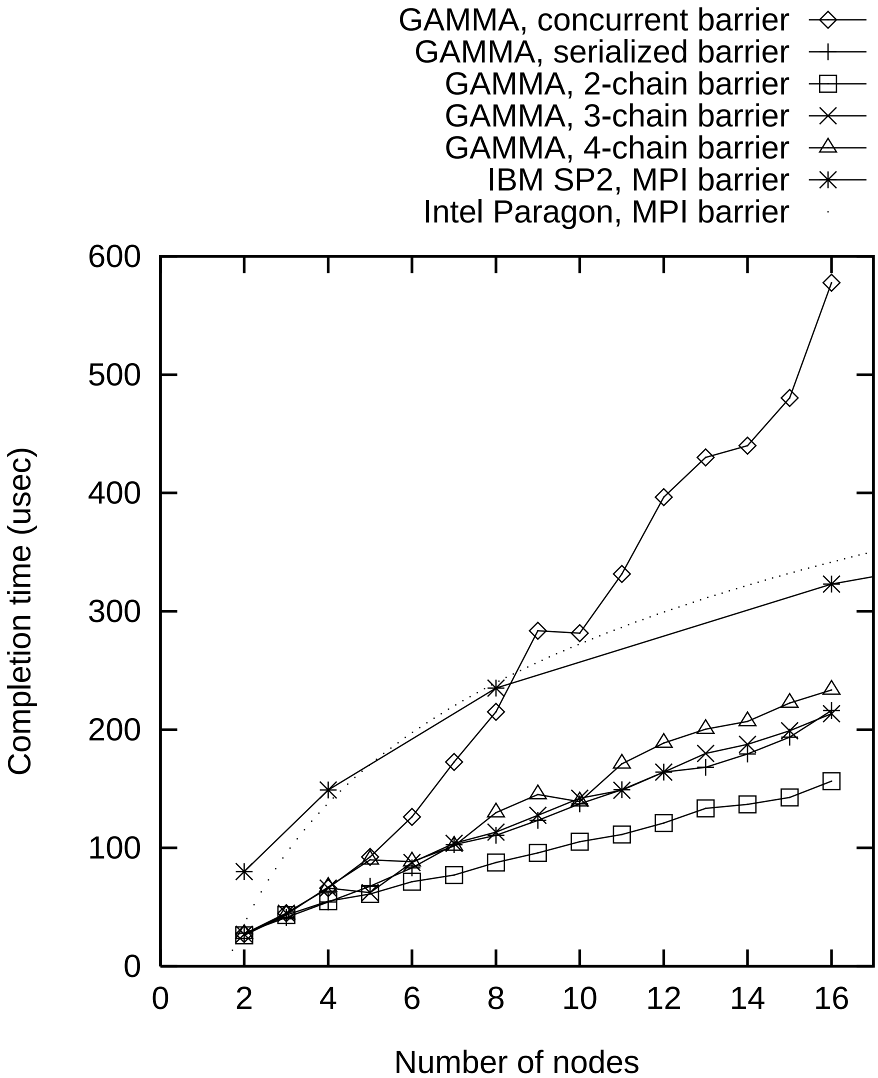

Figure 5.1: Execution time of some barrier synchronization protocols on GAMMA compared with MPI barriers on IBM SP2 and Intel Paragon.

The exponential behaviour exhibited by the concurrent barrier protocol is clearly apparent. It is an expected consequence of the great number of collisions arising on the Ethernet bus when slaves try to communicate with the master without coordinating with one another.

The performance profile of the serialized barrier protocol was expected to be a linear function of the number of nodes. Such linear behaviour is clearly apparent and well fitted by the following linear function of the number of nodes N (time is in μs):

$$T(N) = 13.6 \cdot N \tag{5.1}$$

which simply models the round-trip time among a ring of N nodes on GAMMA assuming a one-way latency time of 13.6 μs (given by the sum of GAMMA latency time plus the 0.9 μs latency of our Fast Ethernet repeater hub).

The performance curve for the 2-chain barrier synchronization protocol in the case of already synchronized processes on a shared Fast Ethernet appears to be linear as well. It nicely fits the following linear equation when $N \geq 3$:

$$T(N) = 20 + 8.5 \cdot N \tag{5.2}$$

As expected, the 2-chain barrier protocol not only outperforms the (inefficient) concurrent barrier protocol, but also performs better than the plain serialized protocol. This means that the 2-chain protocol achieves better exploitation of the Fast Ethernet bus without generating collisions, thanks to a greater amount of parallelism compared to the serialized protocol achieved at a very low collision rate (at most one initial collision). Of course the exploited amount of raw Fast Ethernet bandwidth is low, due to the very small size of packets involved in the protocol, but this is not of concern when dealing with inter-process synchronization.

The 3-chain barrier protocol appears to behave the same as the serialized protocol. This means that the contention rate of the 3-chain protocol is high enough to compensate for the potential performance improvement due to an even greater degree of parallelism. This behaviour can be read as a wrong choice of granularity in exploiting parallelism: The communication overhead (here due to Ethernet contention) becomes too large compared to the potential performance gain when more than two chains are engaged in the barrier protocol. Accordingly, the 4-chain protocol behaves even worse than the 3-chain protocol, denoting an even higher contention rate.

Since the 3-chain protocol performs significantly worse than the 2-chain protocol, we can argue that also a tree-based barrier protocol on shared Fast Ethernet would perform worse than the 2-chain protocol. This claim is supported by the consideration that a tree-based protocol has a greater average degree of parallelism compared to the 3-chain protocol in the case of more than eight processors.

A comparison with the performance profile of MPI barrier primitives implemented on IBM SP2 and Intel Paragon is provided. Performance measurements for such two commercial MPPs are quoted from [HWW97]. We argue that the comparison between MPI and GAMMA in the case of barrier synchronization primitive is fair. Indeed the MPI barrier primitive could in principle be implemented as a straightforward mapping onto the GAMMA barrier primitive with very limited additional software overhead. It is quite evident that both the serialized and the 2-chain barrier protocols implemented atop GAMMA perform significantly better than MPI barrier on both MPPs, at least with up to 32 processors. Much of the superiority of GAMMA barrier synchronization over other MPPs is due to the exploitation of the broadcast service offered by the underlying Fast Ethernet LAN. Indeed the logarithmic profile of MPI barrier on both MPPs will eventually win over the linear behaviour of GAMMA barriers, but it is apparent from the execution time curves that the GAMMA 2-chain barrier protocol is able to perform significantly better than the MPI barrier on IBM SP2 and Intel Paragon even with a significant number of processors (at least about 48), not to mention the much better price/performance ratio.

Chapter 6

Benchmarking GAMMA

6.1 Introduction

In order to evaluate the impact of excellent GAMMA communication performance on some typical parallel applications, we developed and measured performance of two benchmarks from linear algebra, namely Linpack and Matrix Multiply, as well as a "real" application from physics, namely a Molecular Dynamic simulation code.

The Linpack benchmark has been ported from MPI to GAMMA, whereas Matrix Multiply has been developed on GAMMA from scratch. Performance has been measured on a small cluster of four PCs, each comprising a Pentium II 300 MHz CPU, 64 MByte of RAM and 512 KByte of secondary cache. The performance improvement with respect to standard MPICH is considerable.

The Molecular Dynamics simulation program has been ported from PVM. It required some re-structuring of the communication patterns in order to adapt to the characteristics of GAMMA and run efficiently on a shared 100base-T Ethernet cluster of sixteen Pentium 133 MHz PCs. The programming effort for the porting was a few man-weeks to debug and tune the code for best performance. A similar porting experience, related to a Flame Front Propagation simulator, appeared in [CML98].

The material reported in this chapter appeared in [CC97a, CM98b, CM98a, CC98d].

6.2 Linpack

The Linpack benchmark [Don98] consists of resolving linear systems of equations by LU factorization with partial pivoting. Such a benchmark is one of the *de facto* standards for benchmarking both sequential and parallel computers. For parallel platforms, Linpack is by far the most accepted benchmark. Parallel Linpack is used to rank parallel platforms by computation speed in the famous *Top 500 Supercomputers* list [top98].

The core of any implementation of Linpack is a library of FORTRAN basic subroutines called BLAS (Basic Linear Algebra Subroutines). BLAS includes a subroutine for matrix multiply (DGEMM), the most critical and time-consuming Linpack operation. Machine-oriented versions of BLAS exist for most modern computers, including Intel-based PCs.

A general-purpose parallel implementation of Linpack has already been developed in the context of the ScaLAPACK project [sca98]. In that implementation, the communication phases are assumed to be separately implemented as a library called BLACS (Basic Linear Algebra Communication Subroutines). Both MPI and PVM implementations of BLACS are available. BLACS assumes that processes are arranged into a two-dimensional grid mapped onto processors in an arbitrary way, and matrices are distributed across such a grid according to a so called "block cyclic" decomposition. Broadcast communications occur along rows as well columns in the process grid. Clearly the best arrangement when using an Ethernet cluster is a single-row process grid where each process is run on a distinct processor. In this case the matrix decomposition turns from "block cyclic" into "block-column cyclic": matrices are partitioned into vertical strips which are distributed cyclically among processors. Such an arrangement minimizes communications and requires only row-wise broadcast communications. Indeed the GAMMA version of Linpack follows the "block-column cyclic" strategy. In addition, the original MPI version of Linpack has been tuned to follow the "block-column cyclic" decomposition in order to achieve best performance on bus-based interconnects.

6.2.1 Direct broadcast for efficient data decomposition

Using the BLAS and BLACS libraries permits isolation of all the machine-dependent parts (respectively the computation part and the communication part) of the program in a modular way. To implement Linpack atop GAMMA we only would need a GAMMA implementation of BLACS plus an Intel-oriented implementation of BLAS.

Indeed we exploited the very efficient BLAS developed by Intel for the Pentium Pro CPU in the framework of the ASCI project [asc98], which makes excellent use of the cache hierarchy and pipelined FPU of the Pentium architecture. However, in order to maximize performance we actually started with the FORTRAN 77 sequential version of Linpack developed in the framework of the LAPACK project [lap98] and parallelized it from scratch. This way we took into careful account all the features and limitations of the (cheap) networking hardware available.

One crucial feature of GAMMA, besides its very low latency, is the possibility of sending multicast messages directly exploiting the Ethernet broadcast capabilities. No current general-purpose implementation of either MPI or PVM exploits Ethernet broadcast directly. Direct broadcast saves a lot of messages and makes the "block-column cyclic" matrix decomposition even more convenient, since this permits minimization of the overhead for all the row-wise broadcast communications occurring along the single-row process grid.

6.2.2 Contention-free communication patterns

A serious potential drawback of our cluster is the shared Ethernet bus contention, which may thrashing of the channel due to collisions and re-transmissions. A trivial solution to this problem would be to replace our repeater hub by a switch, but at a higher price. The price of a good "cut-through" switch delivering low frame latency is still significantly high compared to off-the-shelf "store-and-forward" switches characterized by substantially higher frame latencies.

Another solution to the contention problem is to explicitly program applications in order to follow contention-free communication patterns. Reasonably good performance is expected provided that the communication pattern is carefully designed not to introduce excessive synchronization in the program. In the GAMMA version of Linpack, the "block-column cyclic" decomposition naturally implies that only one processor at a time sends data (the current pivot columns, to be broadcast to every other process in the grid), thus implicitly forming a contention-free communication pattern. Therefore, a straightforward implementation of the algorithm avoids simultaneous multiple access attempts to the Ethernet medium by different processing nodes, thus yielding optimal exploitation of the available bandwidth even with the lowest cost interconnection technology.

6.2.3 Avoiding application-level message buffering

A common source of performance degradation in message-passing programs is the need for application-level temporary storage for incoming messages, especially if the underlying messaging system is a minimal-copy one. In other words, usually providing a minimal-copy protocol simply means pushing message copy overhead to application level rather than avoiding the copy overhead alltogether (as one would expect). Depending on the application program rather than on the implementation of the communication support, such temporary buffers are needed whenever a process may be forced to receive a new message in a final destination area before having completely processed the previous message stored in the same area.

Indeed this problem could also affect the performance of GAMMA applications: the receiver is forced to accept a message in its final destination at any time the sender starts a communication (otherwise the incoming message would be lost). In the GAMMA version of Linpack, this might cause a race condition whenever a processor sends pivot columns for the next step of the algorithm before the other processors have completed the current step. A possible solution would be to introduce a barrier synchronization at the beginning of each step of the LU decomposition algorithm, with a possible performance degradation due to the additional synchronization. Another possible solution would be to receive incoming messages into a temporary area and copy them to their final destination only when needed, with a possible performance penalty.

We solved this problem in a different (and more efficient) way. We implemented application-level circular queues of arrays for storing incoming messages in their arrival order. Computations on received pivot columns are carried out directly on the arrays at the queue head, whereas fresh incoming data are stored in the arrays at the queue tail. Head pointers of the queue are accessed and modified by the main thread of the receiver process, whereas tail pointers are accessed and modified by the handler devoted to receiving remote pivot columns. This way subsequent incoming pivot columns will not overwrite previous not yet processed columns. In any case a single node cannot advance to the computation of the $n+2$-th step until all other processors have completed step $n+1$ and broadcast their results, therefore only two array buffers are needed for the implementation of the circular queues. Hence this solution efficiently avoids temporary copies of messages without introducing unnecessary synchronizations.

6.2.4 Developing the GAMMA Linpack code

The design, coding and debugging of the Linpack application for GAMMA according to the design principles already discussed required a total effort of about one man-week. The source code of the program is available on the Web through the GAMMA project home page.

Matrix entries were stored in double precision. Matrices were partitioned into "block-column" vertical strips, each counting 64 columns. The BLAS library used in our experiments was the Intel ASCI Option Red BLAS 1.1c [asc98]. All the programs were compiled by first translating them from FORTRAN to C, with `ftoc` then compiling the C code with an experimental optimizing compiler for Pentium architectures derived from GNU gcc [pgc98]. The driver program (that was written in plain C) also performs accuracy tests on the final result.

Our experimental platform was a 100base-T cluster of four PCs, each equipped with Pentium II 300 MHz CPU, 64 MByte SDRAM, 512 KByte L2 cache, 3COM 3c905 rev.B network adapter, and Linux 2.0.29 operating system with both the GAMMA driver and a recent Linux driver for 3COM 100base adapters.

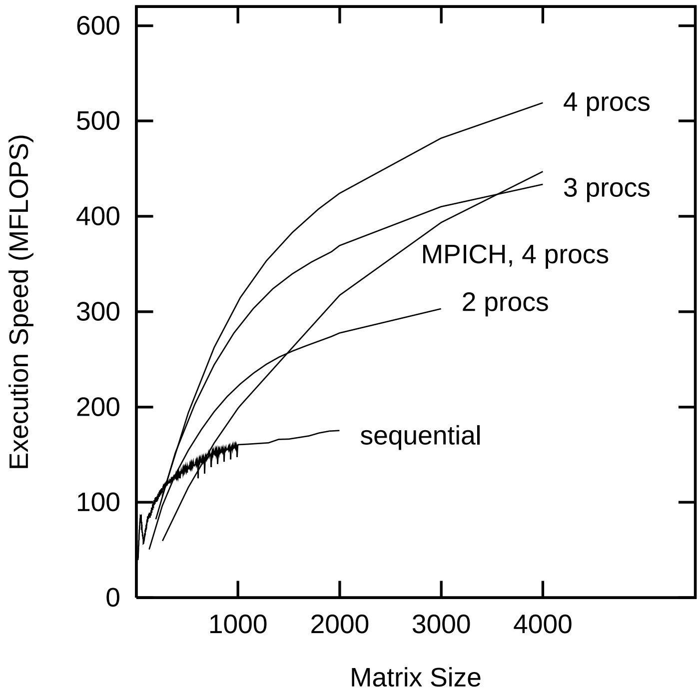

Figure 6.1: Double precision Linpack performance on GAMMA. Performance from the MPICH version with 4 processors is reported as a reference.

6.2.5 Performance results

Performance achieved by the original LAPACK sequential version of the benchmark with the same BLAS and processor is reported as a reference in each performance graph. The theoretical peak floating point speed of Pentium II 300 MHz is 300 MFLOPS, as the pipelined FPU of the Pentium architecture is able to start one floating point operation per clock cycle. Of course this is only an upper bound of what a Pentium CPU can actually do.

Due to lack of RAM (64 MBytes per node) we could not measure the sequential Linpack with very large matrices. However by extrapolation it seems that performance asymptotically reaches 180 MFLOPS, which is good though not excellent efficiency for such a computation-intensive task.

Figure 6.1 reports the execution speed curves of the GAMMA version of Linpack. The behaviour up to four processors indeed appears satisfactory. More than 500 MFLOPS (519 to be precise) are achieved with four processors even with matrices as small as 4000×4000. We expect greater performance with larger matrices, however this would require more RAM on each PC. Assuming 180 MFLOPS as an upper bound on the sequential performance, the speed-up achieved by four processors with 4000×4000 matrices is 2.88.

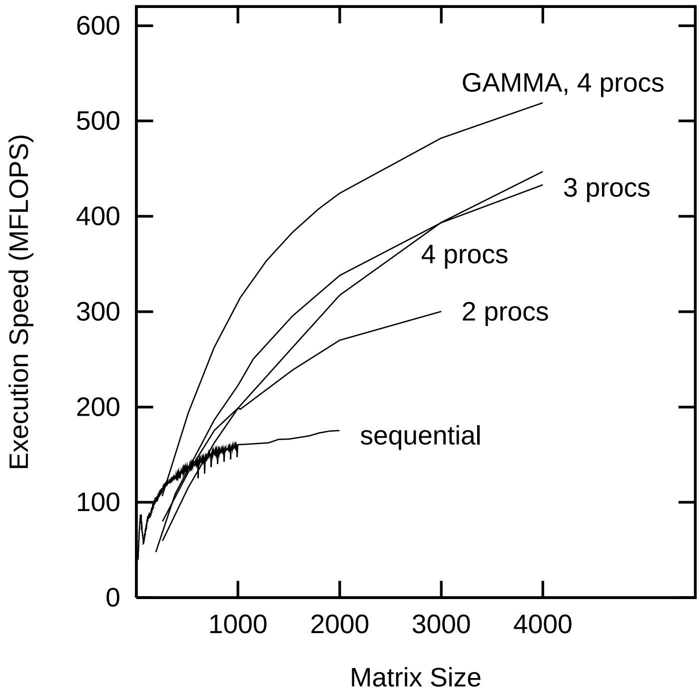

Figure 6.2: Double precision Linpack performance using MPICH atop Linux TCP/IP. Performance from GAMMA 4 processors is included as a reference.

Figure 6.2 reports the execution speed curves of the MPI version of Linpack obtained using the MPICH implementation of MPI [mpi98]. Here the situation appears less satisfactory. The larger communication overhead of MPICH prevents performance from exceeding 450 MFLOPS even with 4000×4000 matrices on four processors, and, even more important, the performance curves with smaller matrices are by far worse than the GAMMA ones.

Of course with larger and larger matrices the communication overhead (which grows quadratically with matrix size) becomes less and less important compared to the computation time (which grow cubically with matrix size). Therefore if we want to use this benchmark to compare different messaging systems on the same processing nodes results are significant only with rather small problem sizes.

It is worth noting that with GAMMA the performance with $N+1$ processors consistently exceed the performance with N processors for all $N > 1$. Indeed using "block-column cyclic" matrix decomposition plus direct broadcast communications instead of "block cyclic" decomposition plus point-to-point communications makes the overall communication overhead almost independent of the number of processors engaged in the computation. Hence for any matrix size, engaging more processors reduces the computation time without practically increasing the communication time.

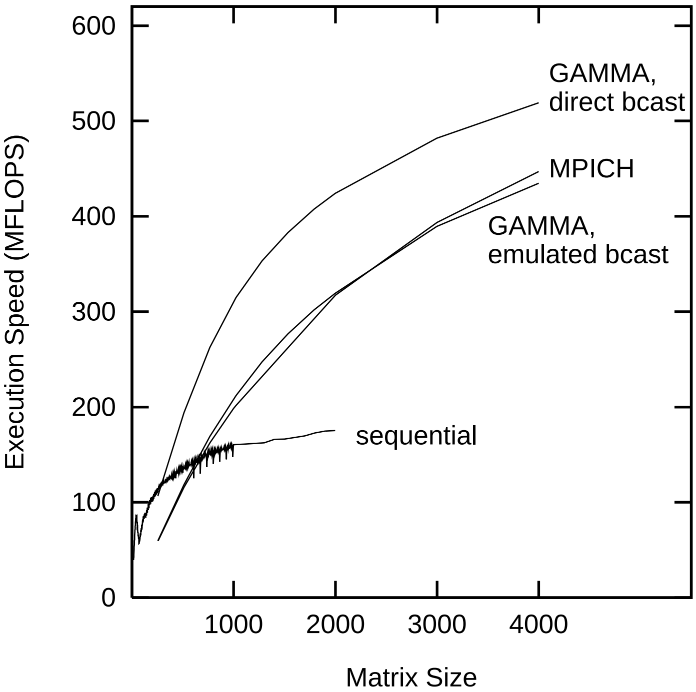

Figure 6.3: Double precision Linpack performance with 4 processors: GAMMA with direct broadcast, GAMMA with emulated broadcast, and MPICH.

This should guarantee good scalability with a much larger number of processors even using very low-cost interconnects like shared 100base-T Ethernet as compared to other 100Mbps interconnects. Of course the upper bound for scalability remains the number of "block-column" vertical strips in which the matrix must be partitioned, however this limit is not due to the interconnection hardware. The situation with MPICH is very different: due to greater communication overhead and inefficient emulation of broadcast communications, performance with $N+1$ processors starts exceeding performance with N processors only when the matrix becomes large enough so that computation time overwhelms communication time.

In order to evaluate how much the better performance delivered by GAMMA depends on better throughput and how much it depends on the availability of direct broadcast, we also measured performance of a different GAMMA version where broadcast communications were emulated as sequences of point-to-point communications, in much the same way as MPICH currently does. The results obtained with four processors are reported in Figure 6.3. Apparently the message-passing version of the Linpack benchmark is not very sensitive to communication throughput: indeed the point-to-point GAMMA version obtains performance similar to the MPICH version, despite the much better GAMMA throughput.

However the difference between using direct and emulated broadcast is clearly apparent. Using direct broadcast with four processors is equivalent to using point-to-point communications with four times larger a communication bandwidth, which means much greater performance and scalability with the same low-cost interconnection hardware.

6.3 Parallel Matrix Multiplication

Actually, the Parallel Matrix Multiplication (PMM) benchmark was the first one developed on GAMMA [CC97a]. Originally the program was developed in C, and run on a GAMMA cluster of sixteen Pentium 133 MHz PCs. The absolute execution speed in MFLOPS was restricted due to the limited power of the CPUs and the poor level of compiler optimization. The main result obtained in that case was an almost linear speed-up compared to sequential code.

We recently re-implemented the benchmark exploiting the same Intel ASCI Option Red BLAS that we used for the Linpack benchmark (Section 6.2) and ran the code on a smaller cluster of four Pentium II 300 MHz. In order to obtain a more stressful benchmark from the point of view of communication performance as compared to the Linpack benchmark, we organized our PMM benchmark according to a master/slave paradigm. The computation starts as a sequential process running on one of the processors of the cluster. Square matrices A, B and C of equal size m are all allocated within the address space of this sequential process. Then this process becomes the "master" of a GAMMA group composed of n processes running on different processors. Matrix A is broadcast to the $n-1$ slave processes, while matrix B is scattered assigning m/n columns to each process of the GAMMA group. Each processor then computes its own portion (comprising m/n columns) of the resulting matrix $C = A \cdot B$ autonomously. Then each slave sends its columns to the master that can thus gather the complete resulting matrix C. A token passing protocol is superimposed on the result gathering algorithm, in order to avoid shared Ethernet bus contention.

Broadcast, scatter and gather times are included in the measurement of the parallel version of the code, as well as the local computation time. Figure 6.4 reports the execution speed curves of the GAMMA version of PMM. The sequential version of the code achieves about 180-185 MFLOPS on a single node for matrices of dimension significantly greater than the size of the internal L1 cache. The GAMMA parallel version achieves speed-up greater than 1 for matrices larger than 200×200, and appears to be increasing for larger matrix sizes up to the point of filling the available RAM on the master processor completely. For example, with 4 processing nodes the actual speed is 292, 363, 461, and 520 MFLOPS, respectively in cases of $m = 400$, 600, 1000 and 1500. The speed-up with respect to the sequential case is therefore 1.61, 1.98, 2.51, and 2.84, respectively. Matrices larger than

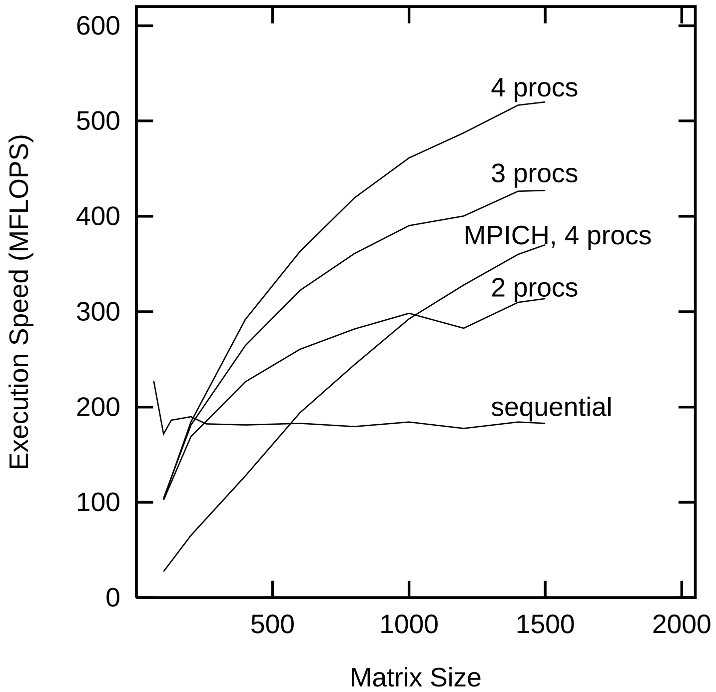

Figure 6.4: Double precision Matrix Multiplication performance on GAMMA. Performance from the MPICH version with 4 processors is reported as a reference.

1500×1500 do not fit the 64 MBytes RAM available on our prototype. In the largest case of 1500×1500 matrices, the processor efficiency (defined as the speed-up divided by the number of processors) is 86% for two processors and 71% for 4 processors.

Like in the Linpack benchmark, performance is monotonically increasing as a function of the number of processing nodes (thanks to the efficient exploitation of the direct broadcast capabilities of the GAMMA platform), and substantially higher than the MPICH version of the same program.

Figure 6.5 reports the execution speed curves of the MPI version of Matrix Multiplication obtained using the MPICH implementation of MPI. Again the poor simulation of broadcast kills performance in the case of more than two processing nodes and average size matrices (huge matrices in this case simply don't fit in the single node RAM). The 2 processor version (which is the best one for smaller matrices) achieves speed-up greater than 1 only with 400×400 matrices. However, scalability to more than 2 processing nodes is practically prevented by the communication bottleneck in this case. The highest speed of 370 MFLOPS is achieved with 4 processors and 1500×1500 matrices, but with a speed-up of only 2.02.

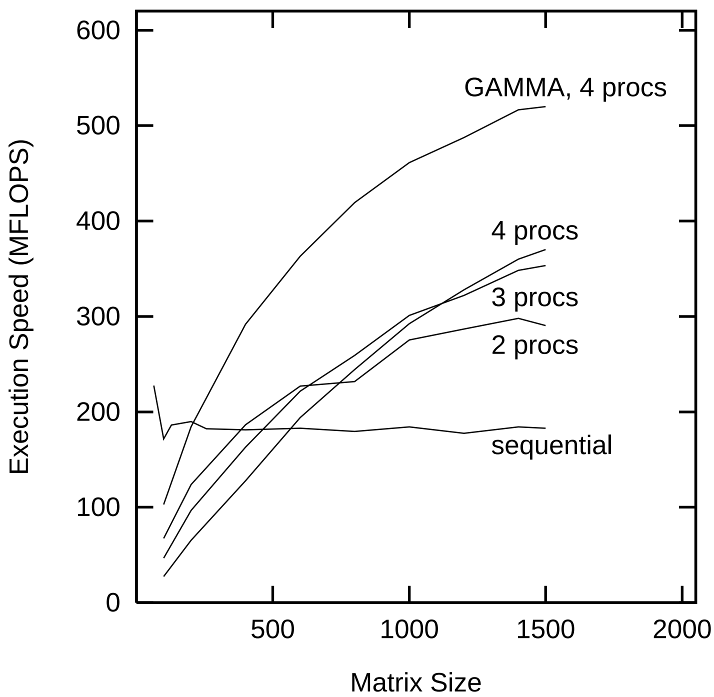

Figure 6.5: Double precision Matrix Multiplication performance using MPICH atop Linux TCP/IP. Performance from GAMMA 4 processors is also included.

6.4 Molecular Dynamics

Molecular Dynamics (MD) is one of the most frequent parallel applications in the scientific community. MD typically exhibits fairly good speed-up figures on a wide range of parallel computers with good intrinsic load balancing. This offers the opportunity to investigate the behaviour of large size samples of material by numerical simulation.

A serious obstacle to running MD on a cluster of PCs is the high communication latency exhibited by standard parallel programming environments like PVM and MPI running atop industry-standard communication protocols like TCP and UDP. Recently several teams have been engaged in producing efficient solutions using faster networks and optimized communication software to keep latency as low as possible. Many such attempts gave rise to non-standard programming interfaces for high-performance communication. Porting a non-trivial parallel application on a non-standard communication layer may be an expensive task. However a better price/performance ratio and a satisfactory absolute performance level on a cluster of PCs may justify the porting effort.

In this section we discuss three experiences of porting an existing MD parallel application on a low-cost cluster of PCs. The original MD code is a FORTRAN program with calls to PVM communication routines. The low-cost cluster is a pool of sixteen Pentium 133 MHz PCs, each equipped with 32 MByte of RAM and 256 KByte of second-level cache, networked by a shared 100base-TX Ethernet LAN. Each PC runs Linux, a POSIX-compliant Unix operating system.

The first experience [CM98b] consists of migrating MD from PVM to the the Genoa Active Message MAchine (GAMMA) [CC97c, Cia98], an efficient communication system based on Active Messages [vECGS92] and designed for best efficiency on 100base-T clusters of PCs. Porting MD to GAMMA required replacing PVM calls with calls to communication routines from the GAMMA library, as well as changing some communication patterns in order to achieve better exploitation of the capabilities of the underlying network hardware fully exposed by GAMMA. Therefore the corresponding porting effort was not negligible. The obtained MD application shall be called MD-GAMMA hereafter.

The second porting experience (also described in [CM98b]) consists of running the original PVM version of MD "as is" on our cluster. This corresponds to a zero porting effort.

The third porting experience consists of trying to tune the communication patterns of the original PVM version of MD in order to increase the match with the network architecture of our cluster. This implies a very limited porting effort. The obtained application shall be called MD-TOKEN hereafter, as a circulating token has been added to reduce network contention.

6.4.1 The Molecular Dynamics problem

Our MD application [Mar94b, MRS95] is a typical Molecular Dynamics code used for simulating the behaviour of polarizable fluids. The current release of MD is written in FORTRAN with calls to PVM routines, and is structured as a MIMD application.

The simulation of material samples with a larger number of molecules turns the behaviour of MD from communication intensive to computation intensive. In our investigations the number of molecules has been kept as low as 4000 to stress the communication side.

MD performs a standard Lennard-Jones calculation plus the solution of the induced polarizability on each molecule taking in account first dipole momentum. Each step of MD consists of evaluating the induced dipoles $\bar{p}_i$ consistent with the values of $\bar{E}_i^{(q)}$ due to a given distribution of the point charges. This part of the calculation requires an iterative procedure with small computation time and many communications to exchange the values of the induced polarizability at each iteration among all processors. For a small number of molecules the cutoff radius is of the same size as the replicated box and the number of force vectors between molecule pairs grows almost quadratically with the total number of molecules. In such a situation any domain decomposition technique based on the spatial position of each molecule in the box is not feasible.

In the parallel implementation each processor maintains a copy of the position of each molecule. However each processor will compute force pairs only on a predefined subset of molecules which has been previously assigned to it. In this way the list of interacting particles, which is by far the largest data structure of MD, could be partitioned among the computation nodes and the total memory occupancy per processor is expected to decrease with increasing number of computation nodes.

When using high-latency communication systems like PVM, an important optimization is to keep the number of distinct messages as low as possible in order not to pay too much for the communication start-up costs. This is achieved by packing all the variables to be communicated (i.e. forces,virial,energy) in a single outgoing message whenever possible. Keeping the number of distinct messages as small as possible reduces the possibility of using multicast/broadcast communication primitives, since in PVM such collective communications are implemented as bare repetitions of point-to-point communications. Almost all communications were point-to-point except for a few of them, i.e. the exchange of the new coordinates of the molecules.

6.4.2 Migrating the application from PVM to GAMMA

In order to migrate MD from PVM to GAMMA to obtain the MD-GAMMA application, the GAMMA programming library has been extended with FORTRAN stubs to the original

GAMMA communication C functions in a straighforward way.

Our PC cluster is equipped with low-cost shared 100base-T Ethernet hardware. This implies that the communication patterns of MD may cause lots of Ethernet collisions, with heavy communication delays. This could be partially avoided if the Fast Ethernet hub were replaced by a switch, but at a higher price. The alternative is to explicitly program a proper serialization of network accesses at the application level and to take best advantage of the Ethernet's hardware broadcast facility that the GAMMA programming interface directly exposes. The serialization of communications during collective all-to-all data exchanges has been obtained in MD-GAMMA by considering all processes as circularly ordered by instance number and implicitly granting the broadcast transmission right to a process after it has received broadcast messages from all its predecessors.

Another source of performance degradation with MD is the need for application-level temporary storage for incoming messages. Even with a minimal-copy messaging system like GAMMA, MD-GAMMA must implement a temporary storage for received messages, because some broadcast messages carry information to be scattered among many processors and summed component-wise with existing local information arranged as arrays.

A potential problem with GAMMA is that the receiver is forced to accept messages in their final destination at any time the sender starts a communication. This may cause race conditions in the memory of the receiver process during the all-to-all exchange phase of MD. Such all-to-all exchange is a two-step operation structured as two communication phases interleaved by one computation phase. In the computation phase the fresh data from the first communication phase are manipulated i.e. summed to previous data. If data from the second communication were delivered in the same data structure as data from the first communication, an inconsistency would arise if the second communication occurs before the intermediate computation step is complete. To avoid such race conditions in MD-GAMMA we had to implemented FIFO queues of application receive buffers for storing incoming GAMMA messages. Computations are carried out directly on the FIFOs' head arrays, whereas fresh incoming data are stored in the FIFOs' tail arrays. This way data from the second communication phase do not overwrite data from the first phase which have not yet been processed.

Migrating MD from PVM to GAMMA required one week of work to replace PVM calls with GAMMA calls, change some communication patterns and implement Active Messages-like receive policies, plus an additional week of work to debug and run the obtained MD-GAMMA application.

6.4.3 Tuning the existing PVM application

Another possibility for porting an existing PVM application on a given target platform is to retain the original message passing interface and to tune the communication patterns of the application in order to increase performance by matching the target architecture.

In the case of MD, an obvious drawback of the original version when running on a bus-interconnected pool of processing nodes like a PC cluster with shared Fast Ethernet is bus contention, which may cause unacceptably large communication delays due to collision storms. The easiest way to overcome this problem is to serialize processes when accessing the network by adding a circulating token, implemented by ordered exchanges of null PVM messages.

In our preliminary study we added a circulating token only in one subroutine of MD, which turns out to be heavily used in the program run. The obtained MD-TOKEN application required a very limited working effort. The token overhead is negligibe compared to the overall communication overhead as well as the MD computation time.

6.4.4 Performance results

Let us consider the speed-up curves depicted in Figures 6.6. The slow-down exhibited by MD as the number of processors increases beyond eight is clearly apparent. Given the low computational power of Pentium 133 MHz CPUs, such behaviour accounts for the poor efficiency of the PVM messaging systems involving many temporary copies of messages during the traversal of many layers of communication protocols, as well as the collision storms arising from processes simultaneously accessing the shared LAN during the exchange phases of the program execution.

However the excellent speed-up curve of MD-GAMMA up to 16 nodes, with the promise of good scaling over even more processors, is mainly due to the following reasons:

- the relatively poor floating-point computational power of Pentium 133 MHz CPUs
- the high efficiency of GAMMA inter-process communications
- the fine tuning of the communication patterns in the GAMMA version of the application, based on the knowledge of features (broadcast) and limitations (shared LAN) of the underlying communication hardware.

In spite of its lower collision rate, MD-TOKEN shows a speed-up curve which is even worse than MD. The reason is that serializing network accesses by a circulating token implies serializing the software overhead of communications as well. When communication overhead is high, as with ordinary PVM, the potential advantage of eliminating collisions is cancelled by the loss of parallelism in the execution of low-level communication software. Thus, coordinating processes at application level in the hope of making better use of the network may result in a counter effect with high-latency messaging systems. It is worth noting that the overhead of the circulating token itself is negligible (less than 5% with 16 nodes).

Figure 6.7 reports the average completion time per time-step for MD as well as MD-GAMMA and MD-TOKEN on our PC cluster. The curve of average completion time per time-step of MD on an eight-"thin-nodes" IBM SP2 is reported too. MD-GAMMA appears to outperform the IBM SP2 if more than twelve processors are engaged in the computation, besides performing better than the other two MD versions. When reading such curves it is important to pay attention to both the absolute performance and the cost of the hardware platform. It is worth pointing out that the current cost on the marketplace of a 16-node GAMMA leveraging shared 100base-T Ethernet and Pentium 133 MHz CPUs is comparable to the cost of one single high-end workstation.

In the case of MD it is apparent that exploiting a low-latency messaging system like GAMMA is the only way to turn a low-cost cluster of PCs into a cost-effective solution for parallel processing. The same holds for the large class of "non-embarassingly parallel" well-balanced parallel applications. The gain in price/performance as well as the good absolute performance level obtained on such kinds of inexpensive platforms makes the porting effort worthwhile, at least in the case of well documented applications. Moreover, the transition from Pentium 133 MHz to Pentium II (400MHz or higher) is expected to make a GAMMA cluster also competitive with respect to leading-edge NOW platforms like the DEC Memory Channel cluster.

6.5 Conclusions

In this chapter we focused on the use of lowest-cost, off-the-shelf hardware components and a best-performance messaging system in order to assemble clusters of PCs for high-performance parallel applications. Our results prove that even shared 100base-T Ethernet can provide a sufficiently fast communication support to run significant applications on a cluster, provided that the OS support for communication is properly enhanced and the application itself is structured in order to take the hardware communication constraints into proper account.

In all three cases described above, contention-free collective communication patterns were almost immediately identified and implemented. Low-latency plus direct broadcast communications were crucial in order to implement such contention-free collective communication patterns efficiently. Similar patterns, which usually require an explicit serialization of send

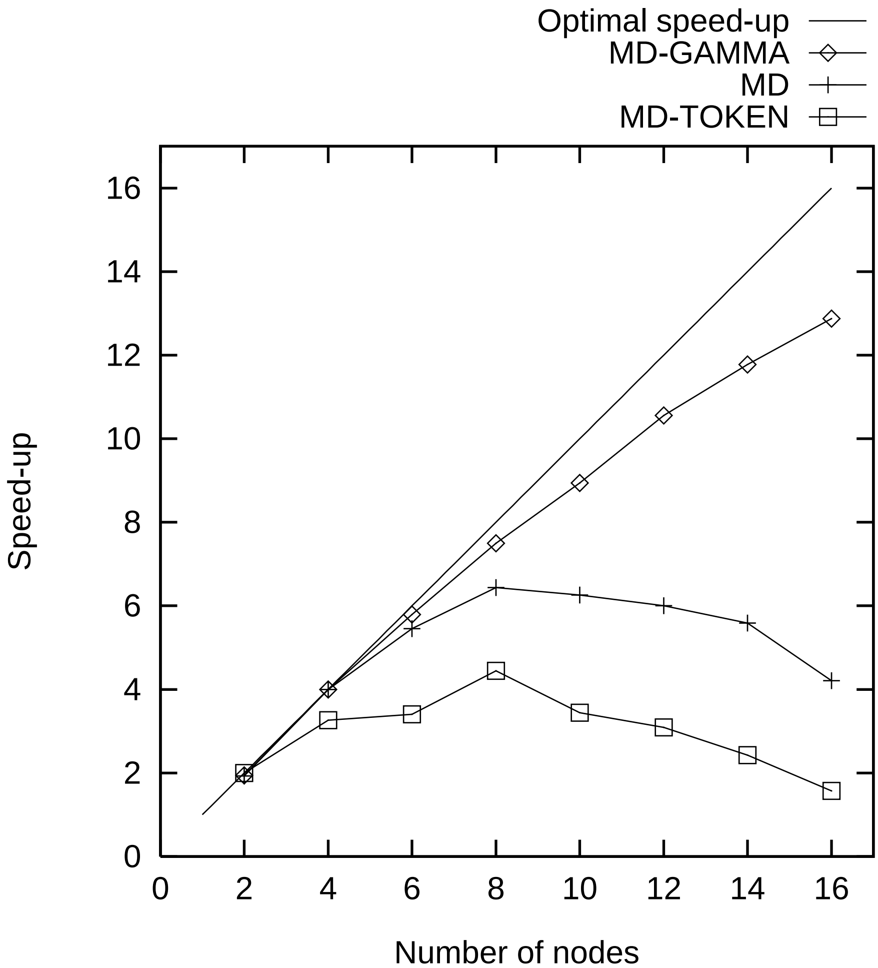

Figure 6.6: Molecular Dynamics, GAMMA vs. PVM: speed-up comparison with same hardware platform (shared 100base-T Ethernet network of Pentium 133 PCs).

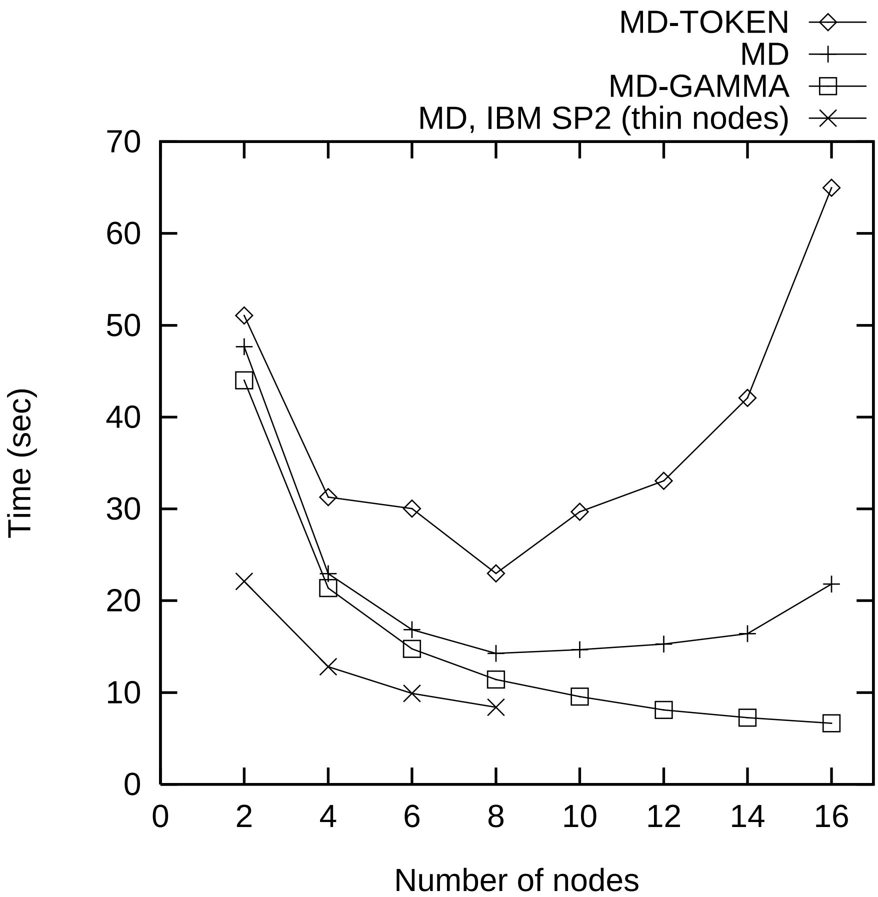

Figure 6.7: Molecular Dynamics: average completion time per time-step on various parallel platforms including GAMMA.

operations, are not feasible with today's public-domain implementations of MPI and PVM because of their poor emulations of broadcast and high software overhead. This strongly suggests that a good messaging system should offer a complete set of high-level collective routines, and that the only way to achieve a satisfactory degree of efficiency for such routines is to take into careful account all the features and limitations of the underlying interconnect as well as the existing point-to-point communication calls in order to design *ad hoc* performance-oriented communication patterns, a somehow opposite approach compared to what is currently done inside public-domain implementations of MPI and PVM. This important point is discussed more deeply in Chapter 7.

Chapter 7

Extensions of the work

In the previous chapters of this book, the main results achieved by the GAMMA project have already been outlined. It is worth reiterating very briefly the most important ones:

- For communication systems running on low-cost commodity interconnects like Fast Ethernet, following a user-level architecture rather than a more traditional kernel-level one is not the main concern for best performance. Indeed today's commodity interconnects cannot be closely coupled with either the CPU or the memory hierarchy of the host computer, as the cooperation between PC and NIC is currently done by means of an I/O bus. Given this constraint, the elimination of the OS kernel from the communication path cannot substantially improve performance. On the other hand, it poses hard challenges in multi-user multi-tasking environments. Indeed GAMMA achieves unprecedented performance on Fast Ethernet following a kernel-level approach which preserves the compatibility with the existing Linux environment (multi-tasking and Unix IPC stack).

- Efficient collective routines and especially barrier synchronization and broadcast communication can be implemented in a very efficient way on low-cost interconnects like shared Fast Ethernet, provided that the features and limitations of the network hardware are taken into careful account. Using such efficient routines in typical parallel applications can lead to dramatic benefits in terms of performance.

GAMMA is still an on-going research project. For this reason, giving an outline of the main ideas for on-going and planned activities of this project is even more important than drawing conclusions. This chapter outlines the expected developments of the GAMMA project.

7.1 Low-level extensions of GAMMA

Here is a short outline of some low-level open issues in the GAMMA project. Most of the extensions described below would be very useful in the perspective of supporting an efficient implementation of MPI, which is one of the most important developments planned in the near future.

- Studying efficient, low-overhead flow control policies to improve the reliability of GAMMA communication. The approach would be experimental again. Preliminary work has already been done [CC99]. The basic idea is that flow control should imply only minimal information from the receiver back to the sender. This rules out per-packet acknowledgments. Another basic point is that flow control does not necessarily imply reliability. It should simply avoid packet losses due to overflow at the receiver end as well as in the LAN switch. In a "well-cabled" switched LAN, no packet losses means no communication errors up to cable problems, which virtually means reliable communication. We think this is enough for most cluster applications, and necessary to support MPI (whose semantics requires reliable communications).

- Implementing light-weight multicast communication. Currently GAMMA supports only group broadcast based on Ethernet hardware broadcast. The same hardware mechanism could be exploited to implement multicast, that is broadcast messages to only a subset of the process group called a *multicast set*. Multicast sets could be useful to implement collective calls for MPI process subgroups (called *contexts* in MPI jargon).

- Implementing a fairly complete suite of collective communication routines (gather, scatter, reduce) in order to efficiently support MPI collective communication calls. This point is detailed in Section 7.2.

- Porting GAMMA to more modern NICs supporting the DBDMA transfer mode (Section 3.2.1). This is currently in progress for 100base-T NICs based upon the DEC 21143 chipset and the Intel EtherExpress Pro NIC [taUdRLS98, CC99]. As already pointed out in Section 3.2.1, DBDMA-working NICs use a fixed region of the host physical memory to store incoming as well as outgoing packets arranged into circular queues of so called "descriptors". These descriptor rings replace the on-board transmit/receive FIFOs equipping older, CPU-driven NICs, using host memory instead of on-board memory to hold incoming/outgoing packets. Each descriptor points to a memory buffer holding the packet content. With most DBDMA-working NICs, it is usually allowed to split a packet into two separate chunks, stored in two different buffers, and still having one single packet descriptor with two buffer pointer. Send operations through DBDMA-working NICs do not need any extra copy of messages in kernel space: Using a two-pointer descriptor allows to enqueue outgoing packets in the output ring with no need to compose the header with the payload withing a single temporary buffer to form an outgoing packet. For message receives, a DBDMA-working NIC autonomously stores incoming packets into predefined buffers pointed to by the descriptors enqueued in the receive ring. This rules out the possibility of storing the payload to its final destination directly, as the NIC cannot take real-time decisions based on header inspections (only a programmable NIC could do this). Consequently, each descriptor in the receive ring has only one pointer to a single kernel-level buffer, which will hold both the header and the payload fragment. It is up to the CPU, when running the receive routine of the GAMMA driver, to inspect the header and then copy the payload fragment to the final destination. However, this extra copy of messages from kernel to user space does not change anything to the CPU point of view: The DMA bus-master operation, that GAMMA should drive with CPU-driven NICs in order to extract each payload fragment from the on-board NIC's receive FIFO, is simply replaced by a memory copy from a kernel buffer, an even faster operation indeed.

- Implementing thread safety at the level of GAMMA device driver, needed to support multithreaded parallel applications on clusters of SMP multiprocessor PCs.
- Experimenting with Gigabit Ethernet technology.

The following issues of minor concern are still open:

- Lack of cooperation between the GAMMA device driver and the Linux memory manager at kernel level, needed to allow message delivery to virtual memory pages that are not yet mapped into physical RAM. This would eliminate the need of explicitly pinning/unpinning user-level data structures into physical RAM upon communication, a non-trivial task with serious implications on memory occupancy. Here the basic idea is that the GAMMA driver should interrogate the Linux memory manager tables upon sending/receiving the first packet of a message, and possibly invoke kernel-level page fault routines to fetch the virtual pages corresponding to the source/destination buffer of the message. Since interrogating the memory manager tables and forcing page fetch may be a time-consuming task for large buffers, this activity should be performed by the driver on a per-page basis in parallel to the NIC send/receive activity, so as to partially hide the additional end-to-end communication overhead.
- Lack of support to multiple network connections on the same PC.
- Lack of support to virtual parallelism, that is, more processes than processors.

7.2 High-level extensions: efficient collective routines

From the porting/implementation activities reported in Chapter 6 it emerges that a good matching between efficient communication primitives offered by the system and "communication needs" of the application may have a larger impact on overall performance than mere bandwidth, latency, and throughput numbers. In all the three reported case studies, some typical collective communication phases were almost immediately identified during the porting/implementation effort. The proper implementation of such collective phases on shared Fast Ethernet by means of *ad hoc* contention-free communication patterns led to an immediate benefit in terms of performance. Low-latency plus direct broadcast communications were crucial to obtain this result.

A similar conclusion can be drawn from the experience of implementing barrier synchronization algorithms (Chapter 5). In that case, best performance on shared Fast Ethernet was obtained adopting an *ad hoc* communication pattern using GAMMA low-latency point-to-point and broadcast messages, together with an explicit knowledge of the features of the underlying interconnect.

Similar *ad hoc* patterns are not feasible with point-to-point and broadcast communication calls of today's public-domain implementations of MPI and PVM because of their high software overhead, which cancels all the potential advantages of this performance-oriented approach to collective routine implementation.

The overall lesson learned is that a good messaging system should offer a complete set of *efficient* high-level collective routines implemented using the following basic "ingredients":

- low-latency point-to-point communications;
- low-overhead multicast/broadcast;
- *ad hoc* communication patterns taking into serious account all the features and limitations of the underlying interconnect.

The above "recipe" is opposite to what is currently done inside public-domain implementations of MPI and PVM, where collective routines, including multicast and broadcast, are implemented as careless patterns of point-to-point high-overhead communications.

A complete set of efficient collective routines is what we need to support the collective calls of a performance-oriented implementation of MPI. Indeed this is one of the on-going activities in the GAMMA project. It could be argued that an *ad hoc* implementation of the whole set of collective calls on a given interconnect may require large porting efforts when moving to different interconnects. This is indeed true, but is also the only way to guarantee "performance portability" across platforms at the level of user applications.

From the viewpoint of efficient collective routines, the availability of direct hardware broadcast/multicast makes the Ethernet technology and especially the forthcoming Gigabit Ethernet very attractive as a cluster interconnect for parallel processing, provided that this feature is directly exposed to the user level and properly exploited by high-level library writers.

7.3 Porting MPICH atop GAMMA at ADI level

The GAMMA programming interface is not an industry-standard. As such, it is necessary to spend a non-negligible porting effort in order to run existing parallel applications on top of it. Indeed this affects the usability of the optimal performance profile of the GAMMA communication system. In this respect, one of our short-term goals is to address application portability issues by providing an industry-standard programming interface like MPI.

The implementation approach has been to start from the public-domain implementation MPICH [GL96], and gradually modify its intermediate layer called Abstract Device Interface (ADI) in order to re-implement it on GAMMA. With MPICH, all the MPI calls are implemented in terms of ADI functions. Therefore, porting the ADI layer to GAMMA means running the whole MPICH atop GAMMA.

We already have developed a prototype implementation of MPI/GAMMA. Work is still in progress to test it with real-world MPI applications. However, the current prototype passes all the tests included in the MPICH distribution, and successfully runs the test of the BLACS (Basic Linear Algebra Communication Subroutines) library [bla98].

7.3.1 Some ADI concepts

In the ADI jargon, an arriving message which finds a pending receive to match at the receiver side is called an *expected message*. In principle, expected messages do not need any temporary storage since they could delivered directly to the user-level destination data structure provided by the pending MPI receive operation which matches it. Otherwise the message is called *unexpected*, and needs to be temporarily stored inside the ADI layer at the receiver side, waiting for a matching receive to be posted. This distinction appears more concrete when looking at the ADI architecture, where we can find two main distinct queues, namely the queue of not yet matched, pending receives, and the queue of unexpected messages waiting for a matching receive.

The original ADI inside MPICH uses three protocols for message delivery, namely the "short" and the "eager" protocols, which are substantially the same (the "short" protocol is an optimization of the "eager" one in the case of very short messages), and the "rendezvous" protocol. By the short and eager protocols, a message is sent regardless of any knowledge about the state of the receiver; at the receiver side, every effort is done in order to store the arriving message, be it expected or not, including dynamic allocation of temporary storage. As a matter of fact, under some circumstances the receiver side may fail hosting an unexpected message due to insufficient memory, thus violating the MPI semantics. However, the rendezvous protocol forces a synchronization between sender and receiver before transmitting data, in order to ensure the availability of receiver resources; this is achieved at the expenses of latency, what makes the rendezvous protocol suitable for long messages. Indeed, ADI poses a threshold on the message size beyond which the rendezvous protocol is used instead of the eager one, in order to avoid buffering long messages. Clearly the rendezvous protocol is also used with short messages whenever an MPI synchronous send routine is called.

7.3.2 Rewriting ADI atop GAMMA

In order to make best use of the GAMMA programming interface while providing an MPI interface to the user, we have substantially rewritten the original ADI layer atop GAMMA. The obtained messaging system will be called GADI (GAMMA ADI) in the sequel.

In GADI only two protocols are used for message delivery, namely, an eager protocol, and a rendezvous protocol.

For contiguous MPI data types, point-to-point GADI communication is minimal-copy, that is, it does not require any additional temporary storage at the receiver side, in the following cases:

- the arriving message is expected;
- an MPI synchronous send is invoked, which implies that the message is certainly expected thanks to using the rendezvous protocol;
- the message is longer than a given threshold currently set at 30000 bytes, which forces GADI to use the rendezvous protocol; this avoids storing long messages in temporary buffers.

In the remaining case, that is, whenever a unexpected message arrives (this implies the message is shorter that 30000 bytes), one more temporary copy of the message payload is performed by the receiver, using GADI temporary buffers. To minimize overhead, our choice is to use fixed preallocated storage for unexpected messages. Each receiver provides N bytes of total preallocated room per sender at GADI level. In order to prevent this resource to run low in the case of too many unexpected message arrivals, a credit-based flow control has been implemented inside GADI. This consists of initially giving each sender a credit for transmitting N bytes to each destination when using the GADI eager protocol, then forcing GADI to switch to rendezvous protocol at the sender side when its credit to a required destination has been consumed. As a result, the sender will synchronize with the destination before proceeding, and the destination will possibly refill the sender's credit during this synchronization phase.

For non-contiguous MPI data types as well as with buffered send/receives, two additional temporary copies of messages are carried out by GADI. This is unavoidable consequence of the MPI semantics when using MPI buffered mode. For non-contiguous MPI data types, temporary copies might only be avoided if the underlying messaging system (GAMMA in this case) would provide optimized, minimal-copy support to transmit from/to non-contiguous memory regions (so called "gather/scatter" operations, not to be confused with the well-known homonymous collective routines).

7.3.3 "Ping-pong" performance

Our preliminary "ping-pong" tests of MPI/GAMMA on a pair of Pentium II 350 MHz connected by a Fast Ethernet hub show a one-way latency of 17.7 μs, including 1 μs hardware latency from the repeater hub, with an excellent asymptotic bandwidth of 11.3 MByte/s which however could reach 12 MByte/s on a collision-free LAN. The half-power point is reached at 250 bytes, and the overall throughput profile, as depicted in Figure 7.1, accounts for a really low-overhead implementation when compared to the raw GAMMA throughput curve. The substantial improvement of MPI/GAMMA over MPICH running atop Linux TCP/IP on the same hardware platform is clearly apparent. Of course, the "ping-pong" program forces an implicit synchronization between sender and receiver which allows minimal-copy communications most of the time during the test. On the other hand, for messages longer than 30000 bytes the communication path is minimal-copy all the time. Therefore the depicted throughput curve of MPI/GAMMA virtually corresponds to the optimal case.

7.3.4 Ongoing work

Porting the ADI layer to GAMMA greatly speeds up point-to-point MPI communications, but is not as much a satisfactory answer for collective calls. Speeding up MPI collective calls requires providing collective services at ADI level, possibly mapped onto efficient collective routines at lower level. Indeed GAMMA already provides efficient support to some collective patterns, by taking best advantage of the Ethernet hardware broadcast service. Once the GAMMA library is extended with a wider set of collective routines, and after modifying the ADI layer in order to deliver GAMMA collective routines to the

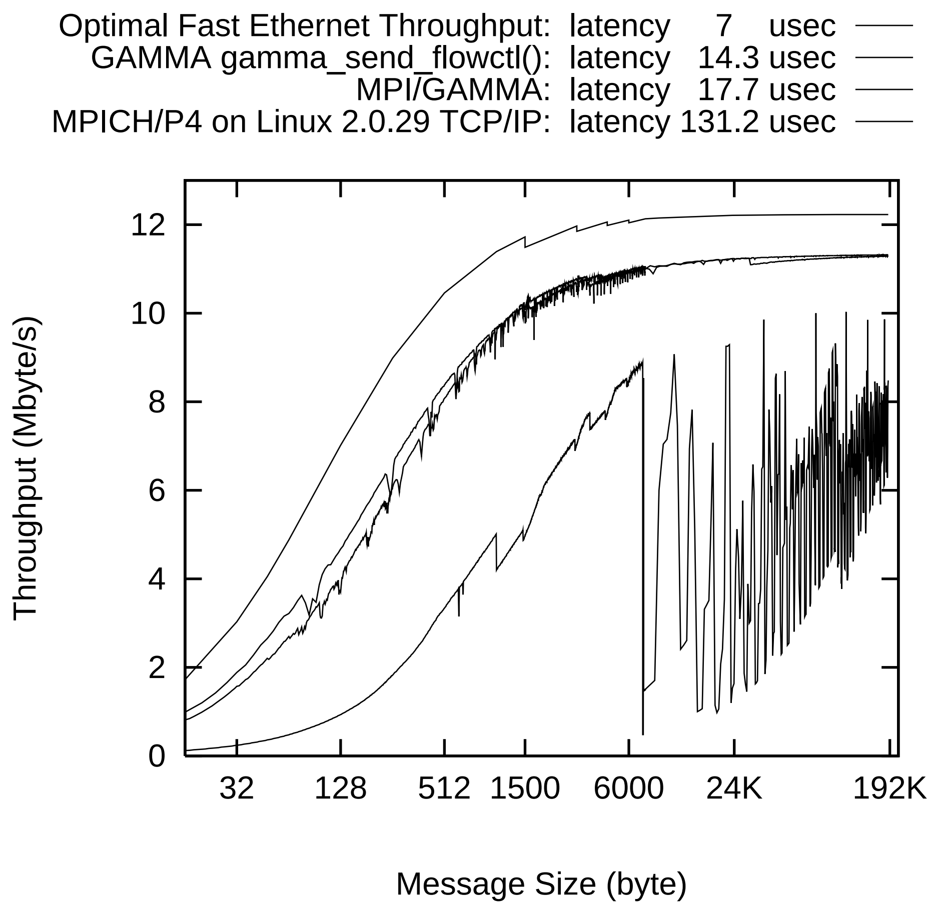

Figure 7.1: Preliminary "ping-pong" evaluation of MPI/GAMMA on a pair of Pentium II 350 MHz connected by a Fast Ethernet repeater hub. Performance of Linux TCP/IP and MPICH atop TCP/IP on the same hardware platform is reported as a reference.

upper MPI level, we may expect a performance gain for all the MPI collective routines far beyond the yet impressive improvement exhibited by MPI/GAMMA with point-to-point communications.

Bibliography

[ACPtNt95] T. Anderson, D. Culler, D. Patterson, and the NOW team. A Case for NOW (Networks of Workstations). *IEEE Micro*, 15(1), February 1995.

[asc98] ASCI project, `http://www.sandia.gov/ASCI/`, 1998.

[BDH+97] J. Bruck, D. Dolev, C. Ho, M. Rosu, and R. Strong. Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations. *Journal of Parallel and Distributed Computing*, 40(1):19–34, January 1997.

[BI95] M. Bernaschi and G. Iannello. Efficient Collective Communication Operations in PVMe. In *2nd EuroPVM Users' Group Meeting*, Lyon, France, September 1995.

[BLA+94] M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94)*, pages 142–153, April 1994.

[bla98] BLACS homepage, `http://www.netlib.org/blacs/`, 1998.

[BNK92] A. Bar-Noy and S. Knipis. Designing Broadcasting Algorithms in the Postal Model for Message-Passing Systems. In *Proc. of the 4th ACM Symp. on Parallel Algorithms and Architectures (SPAA'92)*, June 1992.

[CC] G. Chiola and G. Ciaccio. GAMMA home page, `http://www.disi.unige.it/project/gamma/`.

[CC97a] G. Chiola and G. Ciaccio. Architectural Issues and Preliminary Benchmarking of a Low-cost Network of Workstations based on Active Messages. In *Proc. 14th ITG/GI conference on Architecture of Computer Systems (ARCS'97)*, Rostock, Germany, September 1997. VDE.

[CC97b] G. Chiola and G. Ciaccio. GAMMA: a low cost Network of Workstations based on Active Messages. In *Proc. Euromicro PDP'97*, London, UK, January 1997. IEEE Computer Society.

[CC97c] G. Chiola and G. Ciaccio. Implementing a Low Cost, Low Latency Parallel Platform. *Parallel Computing*, (22):1703–1717, 1997.

[CC98a] G. Chiola and G. Ciaccio. A Performance-oriented Operating System Approach to Fast Communications in a Cluster of Personal Computers. In *Proc. 1998 International Conference on Parallel and Distributed Processing, Techniques and Applications (PDPTA'98)*, volume I, pages 259–266, Las Vegas, Nevada, July 1998. CSREA Press.

[CC98b] G. Chiola and G. Ciaccio. Active Ports: A Performance-oriented Operating System Support to Fast LAN Communications. In *Proc. Euro-Par'98*, number 1470 in Lecture Notes in Computer Science, pages 620–624, Southampton, UK, September 1998. Springer.

[CC98c] G. Chiola and G. Ciaccio. Fast Barrier Synchronization on Shared Fast Ethernet. In *Proc. of the 2nd International Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC'98)*, number 1362 in Lecture Notes in Computer Science, pages 132–143. Springer, February 1998.

[CC98d] G. Chiola and G. Ciaccio. Porting and Measuring the LINPACK Benchmark on GAMMA. In *Proc. DAPSYS'98*, Budapest, Hungary, September 1998. Institute of Applied Computer Science and Information Systems, University of Vienna.

[CC99] G. Chiola and G. Ciaccio. GAMMA on DEC 2114x with Efficient Flow Control. In *Proc. 1999 International Conference on Parallel and Distributed Processing, Techniques and Applications (PDPTA'99)*, Las Vegas, Nevada, June 1999.

[Cen] Center of Computational Science of the U.S. Naval Research Laboratory. On-line documentation,`http://www .nrl.navy.mil /CCS /help /cm5 /products /cmmd.html`.

[Cia98] G. Ciaccio. Optimal Communication Performance on Fast Ethernet with GAMMA. In *Proc. Workshop PC-NOW, IPPS/SPDP'98*, number 1388 in Lecture Notes in Computer Science, pages 534–548, Orlando, Florida, April 1998. Springer.

[CKP+93] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proc. of the Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'93)*, S. Diego, CA, 1993.

[CLMY96] D. Culler, L. Liu, R. Martin, and C. Yoshikawa. Assessing Fast Network Interfaces. *IEEE Micro*, 16(1):35–43, February 1996.

[CM98a] G. Ciaccio and V. Di Martino. Efficient Molecular Dynamics on a Network of Personal Computers. In *Proc. VecPar'98*, Porto, Portugal, June 1998.

[CM98b] G. Ciaccio and V. Di Martino. Porting a Molecular Dynamics Application on a Low-cost Cluster of Personal Computers running GAMMA. In *Proc. Workshop PC-NOW, IPPS/SPDP'98*, number 1388 in Lecture Notes in Computer Science, pages 524–533, Orlando, Florida, April 1998. Springer.

[CML98] G. Ciaccio, V. Di Martino, and P. Lanucara. Porting the Flame Front Propagation Problem on GAMMA. In *Proc. International Conference and Exhibition on High-Performance Computing and Networking 1998 (HPCN Europe '98)*, number 1401 in Lecture Notes in Computer Science, pages 884–886, Amsterdam, The Netherlands, April 1998. Springer.

[CSG98] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture - A Hardware/Software Approach.* Morgan Kaufmann, 1998.

[DHS98] S. Donaldson, J. M. D. Hill, and D. B. Skillicorn. BSP Clusters: High Performance, Reliable and Very Low Cost. Technical Report PRG-TR-5-98, Oxford University Computing Laboratory, Programming Research Group, Oxford, UK, 1998.

[DM95] G. Davies and N. Matloff. Network-Specific Performance Enhancements for PVM. In *Proc. of the Fourth IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-4)*, 1995.

[Don98] J. Dongarra. Performance of Various Computers Using Standard Linear Equations software. Technical report, `http://www.netlib.org/benchmark/performance.ps`, May 1998.

[DP93] P. Druschel and L. Peterson. Fbufs: A High-bandwidth Cross-domain Transfer Facility. In *Proc. of the 14th ACM Symp. on Operating Systems Principles (SOSP'93)*, December 1993.

[DSP97] A. Davis, M. Swanson, and M. Parker. Efficient Communication Mechanisms for Cluster Based Parallel Computing. In *Proc. of the 1st International Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC'97)*, number 1199 in Lecture Notes in Computer Science. Springer, February 1997.

[ea96] C. Reschke et al. A Design Study of Alternative Topologies for the Beowulf Parallel Workstation. In *Proc. of the Fifth IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-5)*, August 1996.

[EKO95] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel, an Operating System Architecture for Application-level Resorce Management. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP'95)*, December 1995.

[GL96] W. Gropp and E. Lusk. User's Guide for MPICH, a Portable Implementation of MPI. Technical Report MCS-TM-ANL-96/6, Argonne National Lab., University of Chicago, 1996.

[Hat98] Red Hat. Extreme Linux, `http://www.redhat.com/extreme/`, 1998.

[HKO+94] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J. P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proc. of the 6th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 274–285, October 1994.

[Hoc94] R.W. Hockney. The Communication Challenge for MPP: Intel Paragon and Meiko CS-2. *Parallel Computing*, 20(3):389–398, March 1994.

[HWW97] K. Hwang, C. Wang, and C-L. Wang. Evaluating MPI Collective Communication on the SP2, T3D, and Paragon Multicomputers. In *Proc. of the 3th IEEE Symp. on High-Performance Computer Architecture (HPCA-3)*, February 1997.

[ILM98] G. Iannello, M. Lauria, and S. Mercolino. Cross-platform Analysis of Fast Messages for Myrinet. In *Proc. of the 2nd International Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC'98)*, pages 217–231, February 1998.

[KOH+94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy. The Stanford FLASH Multiprocessor. In *Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94)*, pages 302–313, April 1994.

[lap98] LAPACK project, `http://www.netlib.org/lapack/`, 1998.

[LC94] L.T. Liu and D.E. Culler. Measurement of Active Message Performance on the CM-5. Technical Report CSD-94-807, Computer Science Dept., University of California at Berkeley, May 1994.

[LC97] M. Lauria and A. Chien. MPI-FM: High Performance MPI on Workstation Clusters. *Journal of Parallel and Distributed Computing*, 40(1):4–18, January 1997.

[LPC98] M. Lauria, S. Pakin, and A. Chien. Efficient Layering for High Speed Communication: Fast Messages 2.x. In *Proc. of the Seventh IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-7)*, Chigago, Illinois, July 1998.

[M-V98] M-VIA Home Page, `http://www.nersc.gov/research/FTG/via/`, 1998.

[Mar94a] R. P. Martin. HPAM: An Active Message layer for a Network of HP Workstations. In *Proc. of Hot Interconnect II*, August 1994.

[Mar94b] V. Di Martino. Computer Simulation of Polarizable Fluids. In *First European PVM Meeting*, Rome, October 1994.

[MK96] E. P. Markatos and M. G. H. Katevenis. Telegraphos: High-Performance Networking for Parallel Processing on Workstation Clusters. In *Proc. of the 2nd IEEE Symp. on High-Performance Computer Architecture (HPCA-2)*, February 1996.

[MPI] Message Passing Interface Forum, `http://www.mpi-forum.org`.

[mpi98] MPICH - A Portable MPI Implementation, `http://www.mcs.anl.gov/mpi/mpich/`, 1998.

[MRS95] V. Di Martino, G. Ruocco, and M. Sampoli. Molecular dynamics of polarizable fluids on parallel systems. In *HPC-ASIA '95*, Taipei, Taiwan, September 1995.

[MRV+97] P. Marenzoni, G. Rimassa, M. Vignali, M. Bertozzi, G. Conte, and P. Rossi. An Operating System Support to Low-Overhead Communications in NOW Clusters. In *Proc. of the 1st International Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC'97)*, number 1199 in Lecture Notes in Computer Science, pages 130–143. Springer, February 1997.

[NN97] N. Nupairoj and L. M. Ni. Performance Metrics and Measurement Techniques of Collective Communication Services. In *Proc. of the 1st International Workshop on Communication and Architectural Support for Network-Based Parallel Computing (CANPC'97)*, number 1199 in Lecture Notes in Computer Science. Springer, February 1997.

[pgc98] Pentium Compiler Group, `http://www. iti.cs.tu-bs.de /soft /www.goof.com /pcg/`, 1998.

[PKC97] S. Pakin, V. Karamcheti, and A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 1997.

[PLC95] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proc. Supercomputing '95*, San Diego, California, 1995.

[PT98] L. Prylli and B. Tourancheau. BIP: a new protocol designed for high performance networking on Myrinet. In *Proc. Workshop PC-NOW, IPPS/SPDP'98*, number 1388 in Lecture Notes in Computer Science, pages 472–485, Orlando, Florida, April 1998. Springer.

[RAC97] S. Rodrigues, T. Anderson, and D. Culler. High-performance Local-area Communication Using Fast Sockets. In *Proc. USENIX'97*, 1997.

[RH98] R. D. Russel and P. J. Hatcher. Efficient Kernel Support for Reliable Communication. In *Proc. 1998 ACM Symp. on Applied Computing (SAC'98)*, Atlanta, Georgia, February 1998.

[SBS+95] T. Sterling, D.J. Becker, D. Savarese, J.E. Dorband, U.A. Ranawake, and C.V. Packer. BEOWULF: A Parallel Workstation for Scientific Computation. In *Proc. 24th Int. Conf. on Parallel Processing*, Oconomowoc, Wisconsin, August 1995.

[sca98] ScaLAPACK project, `http://www. netlib.org /scalapack/`, 1998.

[SHM96] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. Technical Report PRG-TR-15-96, Oxford University Computing Laboratory, Programming Research Group, Oxford, UK, 1996.

[SS96] M. R. Swanson and L. B. Stoller. Low Latency Workstation Cluster Communications Using Sender-Based Protocols. Technical Report UUCS-96-001, Dept. of Computer Science, University of Utah, January 1996.

[Ste96] T. Sterling. The Scientific Workstation of the Future May Be a Pile of PCs. *Comm. of ACM*, 39(9):11–12, September 1996.

[Sun90] V. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience*, pages 315–339, December 1990.

[taUdRLS98] The GAMMA team at Università di Roma "La Sapienza". On-line documentation, `http://gamma.dsi.uniroma1.it`, 1998.

[Tea] The BIP Team. MPI-BIP: An Implementation of MPI over Myrinet, `http://lhpca.univ-lyon1.fr/mpibip.html`.

[The95] The Message Passing Interface Forum. MPI: A Message Passing Interface Standard. Technical report, University of Tennessee, Knoxville, Tennessee, 1995.

[TMC92] Connection Machine CM-5 Technical Summary. Technical report, Thinking Machines Corporation, Cambridge, Massachusetts, 1992.

[top98] Top 500 Supercomputer Sites, `http://www.netlib.org/benchmark/top500.html`, 1998.

[VC98] M. Verma and T. Chiueh. Pupa: A Low-Latency Communication System for Fast Ethernet. Technical report, Computer Science Department, State University of New York at Stony Brook, April 1998.

[vEABB95] T. von Eicken, V. Avula, A. Basu, and V. Buch. Low-latency Communication Over ATM Networks Using Active Messages. *IEEE Micro*, 15(1):46–64, February 1995.

[vEBBV95] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP'95)*, Copper Mountain, Colorado, December 1995.

[vECGS92] T. von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, Gold Coast, Australia, May 1992.

[VIA98] Virtual Interface Architecture Home Page, `http://www.viarch.org/`, 1998.

[WBvE96] M. Welsh, A. Basu, and T. von Eicken. Low-latency Communication over Fast Ethernet. In *Proc. Euro-Par'96*, Lyon, France, August 1996.

[XH96] Z. Xu and K. Hwang. Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2. *IEEE Parallel and Distributed Technology*, pages 9–23, Spring 1996.