The influence of system calls and interrupts on the performance of a PC cluster using a remote DMA communication primitive

Olivier Glück, Jean-Luc Lamotte, Alain Greiner
University P. & M. Curie
LIP6 Laboratory
4 place Jussieu, 75252 Paris Cedex 05, France
{Olivier.Gluck, Jean-Luc.Lamotte, Alain.Greiner}@lip6.fr

Abstract

This paper presents an efficient MPI implementation on a cluster of PCs using a remote DMA communication primitive. For experimental purposes, the MultiPC (MPC) parallel computer was used. It consists of standard PCs interconnected through a gigabit High Speed Link (HSL) network. This paper focuses on communication software layers over the HSL network. Two implementations of MPI are described. The first one uses hardware interrupts for network events signaling and system calls in the communication critical path. The second one is based on full user-level communications. Measures show a latency of 15 μs on a Pentium II-350 with this optimized implementation. A quantitative analysis shows how system calls and interrupts impact on communication time. To tally performance in a realistic environment, experiments were run on the Gauss elimination method using a parallel implementation of a local numerical analysis computational package (CADNA).

1. Introduction

The MultiPC (MPC) parallel computer is a low cost, high performance cluster of PCs. The differentiating and original element of this cluster resides in its High-Speed Link (HSL) network designed at the University P. & M. Curie, Paris. This packet switched network uses one Gbits/s, point to point, serial links. From the application software point of view, the MPC computer provides an optimized Message Passing Interface (MPI) library [3]. Efficient software layers and a specific high-performance implementation of MPICH [11] has been developed on top of the HSL network.

The MPC parallel computer is a high performance cluster [8] with the following components:
- standard PC main-boards,
- a standard Unix-based operating system (LINUX or FreeBSD),
- an Ethernet control network,
- a high speed communication network: the gigabit HSL network,
- a FastHSL network controller on each node implementing the communication protocol,
- efficient software layers for using the HSL network,
- an efficient implementation of MPICH over the HSL network.

Communication performance is a critical and very important aspect of cluster computing. From the hardware point of view, gigabit high-speed networks (like Myrinet [7], SCI [12] or HSL [6]) are now very common. Nevertheless, an efficient hardware is not sufficient to reach good performance. The communication software overhead represents the main part of communication time [13]. The goal of projects like Active Messages [17] or Fast Messages [14] is to reduce this software overhead. The critical factors are well identified. We can quote intermediate data copies, the crossing of several communication layers, physical/virtual address translations and the use of system calls and interrupt signaling during communications.

This paper deals with an efficient implementation of MPICH over a high-speed network providing a remote DMA communication primitive, and the impact of using system calls and hardware interrupts during communications. Two implementations of MPICH over a basic "remote write" primitive have been realized: MPI-MPC 1 and MPI-MPC 2. Section 2 gives a brief description of the MPC parallel computer and its remote write primitive. Section 3 describes MPI-MPC 1, the first implementation of MPICH over MPC using system calls and interrupts during communication phases. Section 4 explains how user-level communication is achieved by the MPI-MPC 2 implementation. Section 5 is a comparison between these two implementations. Section 6 presents the computation time on the Gauss elimination method using a parallel implementation of a library that takes into account the floating point round-off error propagation: Control of Accuracy and Debugging for Numerical Applications (CADNA) [2]. Finally, Section 7 concludes and deals with future work aspects.

2. The MPC parallel computer

The MPC computer (MultiPC) is the result of a 6-year research project at LIP6. The goal was to design a cluster of PCs using a truly scalable and high speed interconnection network, providing an efficient remote DMA primitive to the software. This section gives a brief description of hardware and low-level software in the MPC computer. It is clearly a machine that falls within the class of low-cost Beowulf-like [4] high performance computers. A recent overview of High Performance Cluster Computing has been done in [8].

2.1. The MPC hardware

From the hardware point of view, MPC consists of several processing nodes interconnected by a custom high-speed communication network (HSL) which complies with the IEEE-1355 standard [6]. Processing nodes are standard PCs.

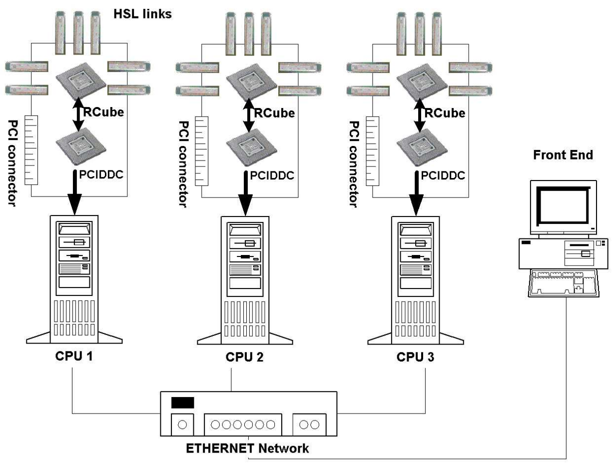

Figure 1. The MPC computer architecture.

Figure 1 presents the architecture of the MPC computer. Three nodes are represented. They are connected via an Ethernet control network to a front-end computer. Each node contains an HSL network controller board designed by the LIP6 laboratory. The FastHSL board carries 2 ASICs. The PCI-DDC chip [18] is a PCI controller that implements a "remote write" protocol. The RCUBE router [5] is a single chip 8*8 dynamic crossbar that offers 8 bi-directional HSL ports. Thanks to the highly integrated RCUBE router, there is no centralized switch in this architecture, as each node contains a routing capability. The HSL links are 1 Gbits/s bi-directional, full duplex, serial links. HSL connections between CPU1, CPU2 and CPU3 are not represented on figure 1. More information concerning the MPC hardware can be found in [10].

2.2. The remote write communication primitive

Each processor node is connected to an RCUBE router using a dedicated PCI to HSL network controller named PCI-DDC. This chip implements the Direct Deposit State Less Receiver Protocol (DDSLRP), developed at LIP6 to reduce the processor overhead. Classical data transfer protocols usually require several copies of data in intermediate buffers before and after transmission through the network. PCI-DDC can access directly the host memory. In order to enhance performance, PCI-DDC implements the "remote write" primitive described in figure 2. This can be seen as a DMA request where the local PCI-DDC directly fetches data from the local host memory and the remote PCI-DDC writes data directly into the remote memory.

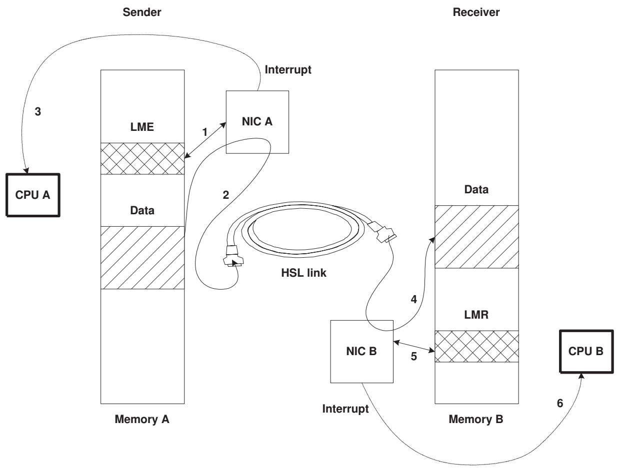

Figure 2. The remote write primitive of the MPC computer.

The descriptors of a message are pushed by the software into the LME, which is the "List of Messages to Send", located in host memory. This list contains the descriptors of buffers to be transmitted (local and remote physical addresses, length, destination node, etc.). A remote write communication proceeds as follows: (1) NIC A reads the message descriptor through DMA access on PCI-bus; (2) it starts data transmission, using again DMA accesses to the host memory relieving thus the sender processor from data transmission; (4) on the receiver side, as soon as PCI-DDC receives a packet, it starts writing incoming data at the corresponding memory location; (5) when the last packet is written, NIC B writes in the LMR, which is the "List of Received Messages" of the destination node; (3) (6) in emission/reception, the notification is achieved by a hardware interrupt signal to the host processor.

2.3. The PUT API

Our goal is to provide efficient software layers to access the HSL network from the application level. The lowest software layer, called PUT, offers basic kernel communication services using the remote write primitive described in section 2.2. This layer provides a kernel API that writes page descriptors into LME and handles event signaling using hardware interrupts. A zero-copy strategy is implemented to take advantage of the performance offered by the HSL network. To let multiple users call PUT simultaneously and to handle interrupts, this layer is designed inside the OS kernel.

The communication primitive supplied by the PUT API can only transfer a contiguous buffer located in the physical memory. It needs the following parameters: the receiver node identifier, the physical address of the local buffer, the physical address of the remote buffer, the data length and a set of flags. The latency of this layer is about 5 μs and the maximum bandwidth is 494 Mbits/s on a Pentium II 350MHz without using interrupts. For more details about the MPC low-level software layers, please consult [1].

3. MPI-MPC 1: the first implementation of MPICH on MPC

The MPC parallel computer runs MPI [3] applications. We have designed an efficient MPICH [11] implementation for PC clusters using a remote DMA communication primitive.

3.1. Description

This implementation is built over the PUT layer. The specification of PUT creates two main problems. Firstly, data transmitted by PUT must be physically located in a contiguous memory space. Secondly, the sender has to know where to write data in the remote physical memory.

There are two types of messages in MPICH: control messages and data messages. In our MPI implementation, control messages are used to transfer rapidly on the HSL network some control information or limited size user-data. The maximum size of a control message is set to 16 Kbytes in the current MPI-MPC implementation. There are four sub-types of control messages:
- SHORT: user-data (encapsulated in a control message),
- REQ: request of transmission of a large data message,
- RSP: reply to a request,
- CRDT: credits, used for flow control.

The MPC software allocates at boot time, on each node, an array of contiguous physical memory that is used to implement the destination buffers for the control messages. Each node gets the physical address of all remote slots that are allocated to him, through the control network (all nodes are connected by an Ethernet network for configuration). The transmission of control messages is done thanks to an intermediate copy in pre-allocated buffers on the sender and receiver side. Data messages are used to transfer user-data that can not be encapsulated in a control message. For those large messages, we want to keep a zero-copy transfer mode. This implies a rendezvous between the sender side and the receiver. The sender first sends a REQ control message to the receiver. Then, the receiver returns a RSP control message containing the physical description of the user destination buffer. Finally, the sender sends data without any intermediate copy.

For instance, figure 3 presents the way the user data are transmitted when the `MPI_Send` primitive is called by an MPI application. On the one hand, if the user data size is less than the maximum size of a control message (16 Kbytes in the current configuration), user data are encapsulated in a SHORT control message. Thus, two intermediate copies of user data are done but just one message is used for the transmission. On the other hand, if the user data size is larger than 16 Kbytes (case 2 of figure 3), a zero-copy strategy is used thanks to a rendezvous protocol. In this case, three messages are necessary to achieve the transmission.

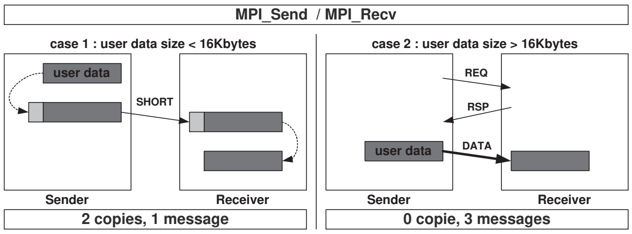

Figure 3. The MPI_Send primitive in the MPC parallel computer.

In the case of the zero copy transfer (case 2 of figure 3: large message), the Send/Receive operation requires locking data into memory and address translation on the sender and receiver side. Indeed, the sender/receiver supplies a virtual process address and the corresponding buffer is not necessarily contiguous in the physical memory. The receiver sends his buffer description in physical memory through the RSP control message to the sender. Then, the latter makes the matching of physical contiguous pieces between the sender and the receiver buffers. In the case of short messages (case 1 of figure 3), address translation is not necessary since intermediate pre-allocated buffers are contiguous in physical memory.

Every call to the PUT layer is done through a system call and event signaling relies on hardware interrupts in this first implementation. When an emission or reception is completed, the FastHSL board interrupts the local processor. Then, the interrupt handler of the PUT layer updates the kernel-level flag concerning the ended communication. At last, MPI-MPC layers have to poll flags to know which communication has completed.

3.2. Results and drawbacks

This section presents the performance obtained for the MPI-MPC 1 implementation described in section 3.1. The MPI application is a ping-pong involving two nodes. The corresponding pseudo code is presented in table 1. Each node alternatively acts as a sender or a receiver. Time is measured on node A starting from the `MPI_Send` call to the end of the `MPI_Receive` call. Thus, the resulting delay corresponds to the necessary time for a message to leave and return to node A. Such a measure is repeated several thousand times to obtain an average latency. In addition, this measurement is done for several sizes of message, n varying from 1 byte to 1 Mbytes.

Node A	Node B

`Forever do`
`Start_Timer`
`MPI_Send(n bytes)`
`MPI_Receive(n bytes)`
`Stop_Timer`
`Next`
	`Forever do`
	
	`MPI_Receive(n bytes)`
	`MPI_Send(n bytes)`
	
	`Next`

Table 1. MPI ping-pong pseudo code.

Measurements have been performed on a Linux MPC platform consisting of 2 standard PCs PII-350 connected through the HSL network. The latency (corresponding to half measured delay for a message size of one byte) observed with MPI-MPC 1 is 26 μs. The maximum throughput (obtained for 1 Mbytes messages) is about 419 Mbits/s.

These performances are quite good but are degraded by the systematic use of system calls and interrupts, as shown in the next section.

4. MPI-MPC 2: user-level communications

This section presents the MPI-MPC 2 implementation that achieves communications in user mode, and avoids the use of interrupts for signaling.

4.1. Description

MPI-MPC 1 implementation described in section 3 was built over the low-level PUT software layer of the MPC computer, that is implemented in the kernel. Thus, the MPI communications of this implementation used to involve system calls through a specific driver for accessing the HSL network interface. The MPI-MPC 2 implementation minimizes this overhead by performing user-level communications. Figure 4 illustrates how MPI-MPC 1 and MPI-MPC 2 access to the network interface.

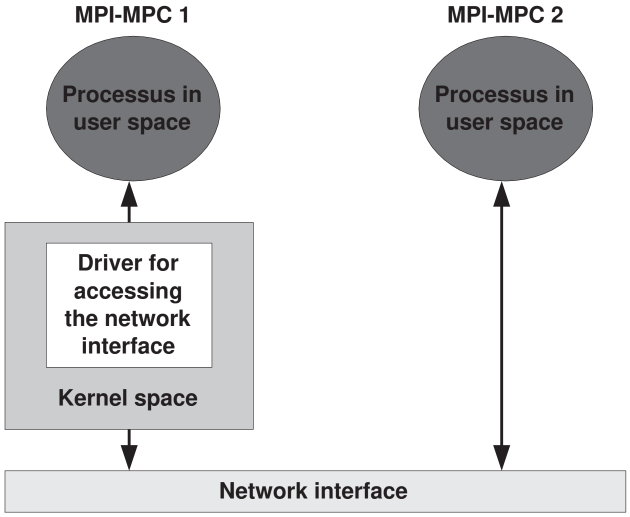

Figure 4. Differences between MPI-MPC 1 & MPI-MPC 2 implementations.

There are three major problems to solve in order to achieve communications in user mode. The first issue is related to the shared network resources such as the LME and LMR lists or some specific FastHSL registers. In a multi-task environment, those resources must be protected for exclusive accesses. The second issue is related to the signaling network events. In the first implementation, signaling is done by hardware interrupts. The drawback is that interrupts imply a switching context. Therefore, MPI-MPC

2 has to poll directly network resources in user mode. The third issue concerns virtual to physical address translations that are used for large messages.

Regarding the shared network resources, the adopted solution is to keep in the kernel the shared objects and to map them to the user space at starting time. The possible competing accesses are managed by the use of atomic locks (one per shared resource). Regarding network events, the idea is to poll directly in user mode the LME and LMR lists. This is possible because these lists are mapped to the user space of every MPI processes. This polling is not done on the whole list but only on the last modified entries. The polling can be very efficient, because MPI-MPC uses active polling only for blocking MPI calls (such as MPI_Recv). In case of non-blocking MPI calls (such as `MPI_Irecv`), MPI-MPC makes a non-blocking polling on both the LMR and LME lists. Moreover, when MPI-MPC has to poll a network event, all the completed communications are acknowledged at once.

Regarding the virtual/physical address translations, a solution to avoid system calls during communications is being studied and will be presented in a future paper. The general idea is to map the virtual memory of each process to a contiguous space of physical memory at application starting time.

There are no more system calls and signaling interrupts during the communication phases in MPI-MPC 2 (excepted address translations that are used for large message transfers). Of course, the initialisation phases still use system calls but only at starting time.

4.2. Results

This second implementation of MPI-MPC shows a latency of 15 μs and a throughput of 421 Mbits/s using the same MPI ping-pong as in section 3.2. From the latency point of view, MPI-MPC 2 is 11 μs better than MPI-MPC 1, which represents an improvement of more than 40%.

5. MPI-MPC1 / MPI-MPC2 comparison

The first part of this section analyzes both MPI-MPC implementations in terms of system calls, interrupts and intermediate copies. The second part shows the MPI-MPC 2 speed-up regarding MPI-MPC 1.

5.1. Recapitulative table

Table 2 recapitulates the number of system calls, interrupts and intermediate copies in both MPI-MPC implementation presented in section 3 & 4. Counting is carried out during a message transfer between two MPI tasks: one is doing a `MPI_Send` call and the other a `MPI_Recv` call. Case 1 & case 2 of figure 3 are taken into account. Case 1 involves one control message transfer whereas case 2 involves two control message and one data message transfers.

For instance, with MPI-MPC 1, a control message transfer generates one system call on the sender side, two intermediate copies and two interrupts (one on the sender & one on the receiver). In the same way, a data message transfer causes two interrupts, five system calls (one for adding entries in the LME list, four for address translations and locking data into memory on the sender & receiver) and no intermediate copy.

		MPI-MPC 1	MPI-MPC 2
System calls	Case 1	1	0
	Case 2	7	4
Interrupts	Case 1	2	0
	Case 2	6	0
Intermediate copies	Case 1	2	2
	Case 2	0	0

Table 2. Number of system calls, interrupts and intermediate copies during communications in both MPI-MPC implementations.

As expected, the main MPI-MPC 2 improvement will concern short messages (case 1): there is zero system call and zero signaling by interrupt. For long messages (case 2), there remain still system calls due to virtual/physical address translations. A solution has just been studied to avoid this overhead but it will be presented in a future paper.

5.2. User-level communications speed-up

Table 3 summarizes the latency and maximum throughput performances of both implementations.

MPI-MPC implementation	Latency (μs)	Throughput (Mbits/s)
MPI-MPC 1	26	419
MPI-MPC 2	15	421

Table 3. Latency and maximum throughput of both implementations.

Figure 5 shows that the MPI-MPC 2 implementation improves the latency, especially for short messages. The latency being the key point for short messages, this is a good result. Figure 6 presents the MPI-MPC 2 latency speed-up with regard to MPI-MPC 1 results. The MPI-MPC 2 improvement does not depend on the data size. The latency improvement is 11 μs per message sent. Thus, the MPI-MPC 2 speed-up decreases when the data size increases.

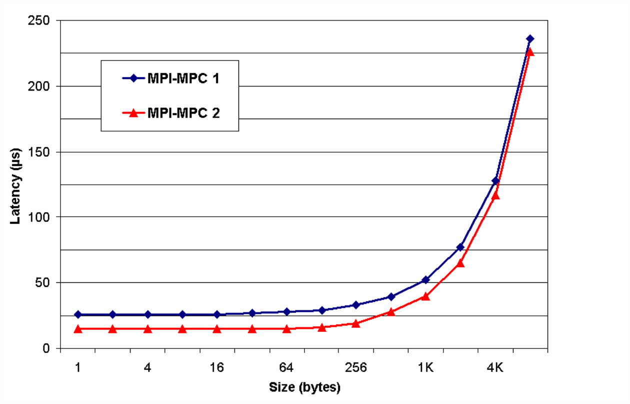

Figure 5. Latency comparison of MPI-MPC 1 & MPI-MPC 2 for short messages.

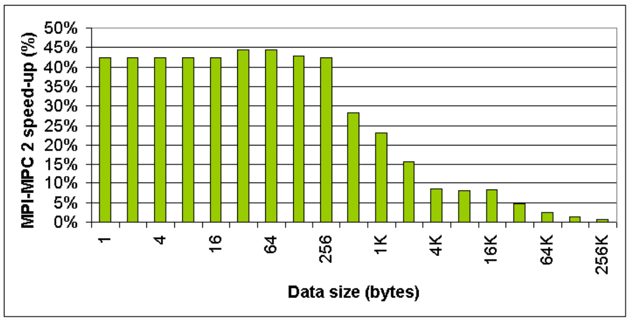

Figure 6. The MPI-MPC 2 latency speed-up with regard to MPI-MPC 1 latency.

6. A realistic application

To tally performance with a realistic application, experiments on the solving of a linear system by the Gauss elimination method are presented. The particularity of this application is to take into account the floating-point round-off error propagation with the CADNA software that is based on the CESTAC method, developed in the LIP6 laboratory of the University Paris 6. Regarding the communication scheme, this application generates a lot of short MPI messages.

6.1. The CESTAC method and the CADNA software

When a programmer creates a new numerical algorithm, he or she uses mainly arithmetic based on real numbers.

The main difference between the mathematical and the computer implementation of an algorithm is due to the representation of real numbers. Given that most real numbers cannot be exactly represented in computer memory, each elementary operation creates a little round-off error because of the limited coding of real numbers. Thus, the fundamental properties underlying the exact arithmetic (for example the associativity) are no longer satisfied. What are the effects of the round-off error propagation on the results? Is the trace of the mathematical algorithm equal to the trace of the computer algorithm?

6.1.1. The CESTAC method. Based on the probabilistic approach, the CESTAC method created in 1974 by M. La Porte & J. Vignes proposes a new method to control and estimate the round-off error propagation. We briefly present its principles. More information can be found in [16] [9] [15].

Practically, this is achieved by using a random arithmetic which rounds with a probability of 0.5 to the value by excess or by default after each elementary operation. Running N times the same program with this new arithmetic yields N different results: $R_1, ..., R_N$. Then, the computed result $\overline{R}$ is the average of the R_i and its number of significant digits is $C_{\overline{R}}$:

$$\overline{R} = \frac{1}{N} \times \sum_{i=1}^{N} R_i \text{ and } C_{\overline{R}} = \log_{10}\left(\frac{\sqrt{N} \times |\overline{R}|}{s \times t_\beta}\right)$$

with $s^2 = \frac{1}{N-1} \times \sum_{i=1}^{N} \left(R_i - \overline{R}\right)^2$ where t_β is a statistical constant.

A new theoretical arithmetic [9] based on the CESTAC method and named the stochastic arithmetic has been defined. It proposes new definitions for the equality and order relations taking into account the accuracy of each operand and a new definition for the zero number. In this arithmetic, a computed zero is a number with no significant digit or that is a mathematical zero (all the values of a number are equal to zero) [15].

The advantage of the CESTAC method is the value of N, which can be small. Practically, N is equal to three.

6.1.2. The CADNA software for sequential architectures. The Control of Accuracy and Debugging for Numerical Applications (CADNA) [2] [9] library is a SYNCHRONOUS implementation of the CESTAC method.

The control of the round-off error propagation is only performed on variables of stochastic types with N=3. CADNA implements two new types: `single_st` (stochastic single precision), `double_st` (stochastic double precision). Every arithmetic operator and every order relation have been overloaded for the stochastic types. With these new definitions, CADNA allows to provide the number of significant digits of all the intermediate or final results and to control the branching. If a result has no significant digit, @.0 is printed. Typically, all the floating-point types of a code are replaced with the stochastic types.

The CADNA implementation uses special functions that can change the round-off mode of the IEEE floating point arithmetic. With a probability 0.5, CADNA selects randomly the rounding mode towards $-\infty$ or $+\infty$. An operation consists in performing:

```
for i=1 to N
 choose randomly the rounding mode
 compute the operation result with the field i
end
```

With these new possibilities, CADNA also allows a numerical debugging of any code written in Fortran, C or C++. All the instabilities that appear during the running are recorded in a log file and can be traced with a classical debugger.

The use of CADNA in a numerical code has a cost. The computation time is multiplied by a factor from 5 to 10 when only the estimation of accuracy is performed with $N = 3$ (i.e. without the numerical debugging functions). The memory size of the data is multiplied by a factor 3.

CADNA is copyrighted and is the property of the University Pierre and Marie Curie, Paris.

6.1.3. The parallel implementation of the CESTAC method. The aim of the parallel implementation of the CESTAC method on the MPC parallel machine is to decrease the computation time of the validation tool, which concerns only the estimation of the round-off error propagation. The simple synchronous implementation of the sequential version involves a simple parallel implementation: three processors performing each one a computation. Thus, the application is divided into three processes. Most of arithmetic operators do not use communication between the three processes excepted four kinds of functions that need to exchange their results to be sure that the executions take the same way on the three processes:
- the test functions: the three tests should give the same answer TRUE or FALSE,
- the absolute value: the three parts of code should take the same value $-a$ or a,
- the conversion of the stochastic type to IEEE type: the result is the mean value of the three values,
- the printing functions: the result is the mean value of the three values printed only with the significant digits.

A distributed algorithm is used to synchronize and exchange data between the three processes. Each process sends its value (less than 32 bytes) to the two others (brother1 & brother2). The corresponding pseudo-code is the following on each process:

```
Local variable T1, T2
Send A to my brother1
Send A to my brother2
Receive T1 from my brother1
Receive T2 from my brother2
Compute a result RES with A, T1, T2
```

To test the parallel implementation of the CESTAC method, a program for solving linear systems using the Gauss elimination method has been used. Such a program is interesting because the problem is easily scalable and the algorithm can perform with or without complete pivoting. In the first case, the three processes exchange a lot of data due to the search of the maximum pivot (test and absolute value functions). In the second case, the estimation of the round-off error propagation does not need communication between the processes.

6.2. Results and speed-up

Experiments have been run on the same MPC parallel computer as the one used in section 3 (three Pentium II-350 with 128 MB of memory). The goal of these tests is to compare MPI-MPC 1 & MPI-MPC 2. The computation time for solving a linear system with the Gauss elimination method has been measured for the following system sizes (800, 1200, 1600, and 2000 equations), with and without pivoting. When solving the system without pivoting, the computation of the system solution with CADNA does not involve communications at all, because there are no test and absolute value functions used. The pivoting algorithm adds a lot of communications and CADNA computation due to the search of the maximum pivot and line permutations (insignificant cost in time compared to communication times). The measured results are presented in table 4.

As expected, there is no difference between MPI-MPC 1 and MPI-MPC 2 in the non-pivoting case. In presence of communications, the global computation time is better with MPI-MPC 2 than MPI-MPC 1 whatever the system size.

	Gauss without pivoting	Gauss with pivoting		With pivoting minus without pivoting	
System size	Execution time (s)	Execution time (s)		Pivoting time including com.	
	No com.	MPI-MPC 1	MPI-MPC 2	MPI-MPC 1	MPI-MPC 2
800	100	151	131	51	31
1200	348	449	414	101	66
1600	786	977	914	191	128
2000	1580	1868	1757	288	177

Table 4. Computation time (in seconds) for solving linear systems using the Gauss elimination method with the parallel version of the CESTAC method on the MPC parallel computer.

A more precise analysis of these computation times has been done. The number of distributed exchanges has been counted. The mean time of one exchange has been calculated in table 5.

System size	Number of exchanges	With pivoting minus without pivoting (s)		One exchange time (μs)	
		MPI-MPC 1	MPI-MPC 2	MPI-MPC 1	MPI-MPC 2
800	646682	51	31	79	48
1200	1450450	101	66	70	46
1600	2574140	191	128	74	50
2000	4018285	288	177	72	44
Mean value				74	47

Table 5. Cost of one distributed exchange.

One exchange includes two communications (see distributed algorithm above) and one CADNA computation. The mean time of one exchange is 74 and 47 microseconds respectively on MPI-MPC 1 and MPI-MPC 2. Thus, the mean speed-up is 36%. By dividing these two numbers by two (two communications), the result is 37 and 23.5 microseconds for one communication and a part of CADNA computation. The difference between these two values is 13.5 microseconds that is coherent with the latency measurements presented in section 5.

7. Conclusion and future work

In this paper, two implementations of the MPI communication library on a remote DMA communication primitive have been presented. The first implementation is built on top of the kernel-level PUT API of the MPC parallel computer. As a consequence, system calls and network events signaling by interrupts are used during the communication phases generating a significant latency overhead. The second implementation achieves the MPI communications in user mode on the same hardware platform, avoiding the use of system calls and interrupts for signaling network events. An important improvement on the communication time has been shown with this optimized implementation. A latency reduction greater than 40% has been measured for the short messages. This performance improvement has been confirmed for a real application as presented in section 6.

System calls and interrupts have a significant impact on the performances, but we described a method to avoid them, without compromising the sharing of the network between several processes running on the same node.

The presented mechanisms have been run on the LIP6 MPC computer, but can be implemented on any cluster of PCs using a remote DMA primitive, such as Myrinet clusters for example.

References

[1] A. Fenÿo. Conception et réalisation d'un noyau de communication bâti sur la primitive d'écriture distante, pour machines parallèles de type "grappe de PCs". PhD. thesis of University Pierre et Marie Curie, Paris, France, July 2001.

[2] The CADNA software: `http://www-anp.lip6.fr/chpv/english/cadna/`.

[3] The MPI forum: `http://www.mpi-forum.org`.

[4] The Beowulf project: `http://www.beowulf.org`.

[5] V. Reibaldi. Conception et réalisation d'un routeur de paquets à hautes performances. PhD. thesis of University Pierre et Marie Curie, Paris, France, 1997.

[6] IEEE Standards Department. IEEE 1355 Standard for Heterogeneous Interconnect (HIC) Low Cost Low Latency Scalable Serial Interconnect for Parallel System Construction. August 1994.

[7] N. Boden et al. Myrinet: A Gigabit-per-Second Local-Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[8] R. Buyya. *High Performance Cluster Computing*. Prentice Hall, NJ, USA, 1999.

[9] J.-M. Chesneaux. Study of the computing accuracy by using probabilistic approach. *In contribution to Computer Arithmetic and Self-Validating Numerical Methods, ed. C. Ulrich, (J.C. Baltzer)*, pages 19–30, 1990.

[10] O. Glück et al. Protocol and Performance Analysis of the MPC Parallel Computer. *Proceedings of 15th International Parallel & Distributed Processing Symposium (IPDPS'2001)*, page 52, San Francisco, CA, USA, April 2001.

[11] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, September 1996.

[12] D. B. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro*, 12(1):10–22, February 1992.

[13] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. *Proceedings of 24th Annual International Symposium on Computer Architecture*, pages 85–97, Denver, CO, USA, June 1997.

[14] S. Pakin, V. Karamcheti, and A. A. Chien. Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors. *IEEE Concurrency*, 5(2):60–73, April-June 1997.

[15] J. Vignes. A stochastic arithmetic for reliable scientific computation. *Mathematics and Computers in Simulation*, 35:233–261, July 1993.

[16] J. Vignes and M. La Porte. Error analysis in computing. *Information Processing 74*, North Holland, 1974.

[17] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. *Proceedings of 19th Annual International Symposium on Computer Architecture*, 20(2):256–266, Gold Coast, Australia, May 1992.

[18] F. Wajsbürt, J. Desbarbieux, A. Greiner, C. Spasevski, and S. Penain. An Integrated PCI component for IEEE 1355 Networks. *Proceedings of EMMSEC'97*, Florence, Italy, 1997.