Asynchronous Communications in MPI – the BIP/Myrinet Approach

Frédérique Chaussumier[1], Frédéric Desprez[2], and Loïc Prylli[2]

[1] LHPC, ENS-Lyon, F-69364 Lyon cedex 07, France
[2] LIP, ENS-Lyon, F-69364 Lyon cedex 07, France
[fchaussu,desprez,lprylli]@ens-lyon.fr

Abstract. In this paper, we present our experiments on asynchronous communications using the BIP and MPI interfaces on a cluster of PCs connected with a Myrinet network. We describe the implementation details of those communications and point the problems and solutions.

1 Introduction and motivations

The implementations of the Message Passing Interface (MPI)[8] are now available on every kind of platforms, from SMP to clusters of PCs. This ensures a very good portability. However, the use of distributed memory machines or network of workstations adds an overhead due to the communications. To hide this overhead, non-blocking communications can be used to overlap computations and communications. However, the assumption that the communication layer provides a "real" overlap and an asynchronous execution of the communication is not obvious. Several papers have presented some ways to hide communication latency [1] or to use asynchronous communications to improve the implementation of parallel algorithms [3, 5]. In [4], a good presentation of communication latency hiding is presented, including active messages [6].

In this paper, we study the possibility of communication overlap on a cluster of PCs interconnected with the high speed Myrinet network through two different communication layers: BIP (Basic Interface for Parallelism) [7], an optimized communication layer for the Myrinet network and MPI over BIP. We first explain the difference between non-blocking and asynchronous communications. The second and third sections show that on the targeted platform, with the available hardware and the basic communication layer used, overlapping communication with computation is possible. In the two last sections, and before a conclusion, we explain why, despite of the possible overlap provided by hardware and basic software, the available MPI interface does not provide overlap.

2 Non-blocking versus asynchronous communications

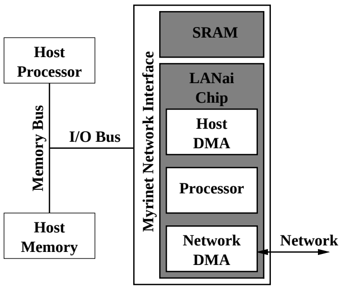
A communication call is said *non-blocking* if it may return before the operation completes. A communication is said *asynchronous* if its execution proceeds as the same time as the execution of the program. Both kind of routines allow the program to continue its execution but prevent the user from re-using resources (such as buffers) specified in the call. A non-blocking communication is not necessarily asynchronous. The data to be communicated can be copied to a temporary buffer and the communication itself can be delayed. If the architecture of the machine has separate communication and computation processors, the communication can be started by the computation processor which in turn gives the task of sending the data over the network to the communication processor.

Both asynchronous and non-blocking routines need a separate "wait" or "test" call to make sure that the communication has completed and that resources can be safely reused. MPI provides non-blocking send (`MPI_Isend`, `MPI_Issend`, `MPI_Irsend`, and `MPI_Ibsend`) and receive (`MPI_Irecv`) communication routines. They can be implemented as asynchronous depending of the target platform.

3 The BIP basic communication layer

We consider the semantics and implementation of the simple problem of sending a message from a host memory to a remote memory with the BIP basic communication layer using non-blocking communication calls.

BIP has two different mechanisms for short and long messages. Long messages sends and receives have a rendez-vous mechanism where a receive needs to be posted before the matching send has started. On the opposite, short messages are stored into a circular queue so that the send calls will not block even if no matching receive has been posted. Moreover, only one non-blocking send or receive may be posted at one time.



The targeted platform is the LHPC PoPC which is a cluster of Pentium Pro 200 Mhz running Linux. Each workstation is equipped with a Myricom/PCI network interface card [2] on the PCI Bus. The figure shown here shows the details of the interface card. It contains a host DMA engine, which moves data from host main memory to the SRAM on the network interface, a network DMA engine, which moves the data from the SRAM into the network, and a LANai processor, which executes the low-level messaging protocol and is responsible for both coordinating the actions of the DMA engines and interfacing with the host.

To transfer a message, the host DMA moves the data from host memory into the on-board SRAM, and the network DMA moves the data from the SRAM into the network.

3.1 Interaction between the compute and the network processors

During a communication using the BIP interface, interactions between compute and network processors to send data are minimum. For a short message, only one low-level communication step is necessary on the network, and the interaction between host and network card sums up to filling a small request structure in the network card memory, appending the payload at the end. For long messages which are splitted into chuncks [7], the interaction between host and network

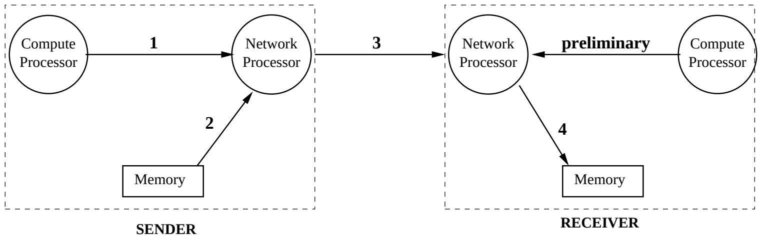


Fig. 1. Different Steps in a Communication.

processor is similar, but the payload will be fetched by DMA by the network processor, using a gather list provided by the compute processor. Although there are several communications on the network, they are handled by the network processor without any host intervention. This means that the compute processor can execute some other computation with no necessary interruption while the whole communication proceeds. Figure 1 illustrates the different steps during a communication using the BIP interface. Before the communication actually occurs, on the destination, the compute processor should have given to the network processor a list of memory addresses for the next message. In theory, this could occur at anytime before the last step. In practice, the only way for the application to guarantee this constraint is to do this step before beginning the send. 1) The compute processor informs the network processor of its sending request. 2) The network processor gets the data from the memory into the board memory with a DMA transfer. 3) The network processor sends the data into the network. The remote network processor gets the data from the network. 4) Finally, the remote network processor puts the data with a DMA transfer into the memory. 5) On both sides, after the last DMA transfer the network processor marks the communication as completed by modifying an appropriate structure in main memory. Note that for a long message, step 2, 3, 4 are repeated in a pipelined iteration.

The communication load on the compute processor is very low. The network processor handles most of the communication management. There is no computational load in the send and receive operations and the compute processor returns almost immediately after the call to the BIP asynchronous send and receive primitives.

4 BIP overlap experiments

Using BIP native primitives, it is theoretically possible to overlap communications and computations. We executed a test program with both blocking and non-blocking communications on two processors. First, processor 0 sends data to processor 1, they wait until the end of the communication and executes a computation and then processor 1 sends back data to the processor 1, both wait until the end of the communication and execute the same computation (see Figure 2).

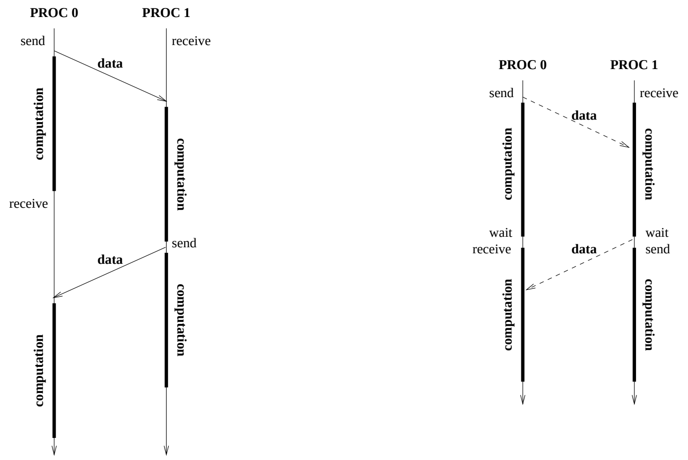


Fig. 2. Running the test program with blocking and non-blocking communications.

The exchanged messages vary from 0K to 1MB. Figure 3 shows the results for a small computation time; it is a matrix vector product where the matrix size is 53 KB (with this size, the computation time equals the time taken for a 4.5 KB message to be exchanged). In the blocking version, the total time is the computation time and the communication time. In the non-blocking version, the total time is the computation time plus an overhead which is not overlapped with the communication time. In the ideal non-blocking case, if the communication time is lower than the computation time then the communication time should be entirely overlapped and the total time should equal the computation time, if the communication time is higher than the computation time then the total time should equal the communication time. Let us remark that this ideal case does not take into account the potential host memory bus contentions that could appear when DMA tranfers and computation are proceeding both using memory at the same time.

5 MPI requirements and the MPI-BIP interface

While the BIP requirements impose that the matching receive should be posted before the corresponding send, MPI imposes that a non-blocking send can be posted whether a matching receive has been posted or not.

In the same way, an MPI implementation should be able to support a large number of pending non-blocking communications whereas BIP specifies that only one send and receive can be posted at a time. Those differences lead to important implementation differences.

5.1 Interaction between the compute and the network processors

First, in the MPI-BIP implementation, three communications are implemented to send data with a non-blocking primitive (see Figure 4): the send request,

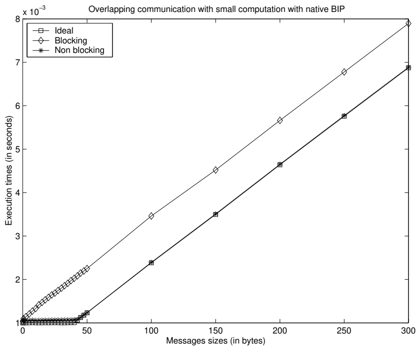


Fig. 3. Overlapping native BIP communications with small computations.

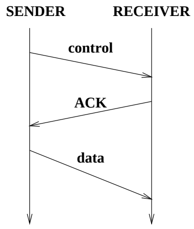


Fig. 4. The necessary communications for a MPI-BIP long message non-blocking send.

the acknowledgment returned, and the communication of the data themselves. Those three communications are necessary for two reasons. The first reason comes from the MPI requirements. As a matter of fact, the implementation of three communications gives a simple way to support a large number of pending non-blocking sends because it provides flow control. The second reason is to avoid memory copies, the intermediate acknowledge in the protocol will allow to delay the transfer until the destination buffer has been provided by the application, then the data can be directly transferred from the network into the application buffer without going into an intermediate storage. When implementing MPI over BIP, the BIP API constraints are fullfilled by doing the asynchronous BIP receive call to receive the data before sending the acknowledgement. With MPI over BIP, receiving and sending the control, acknowledgement, and data messages requires intervention of the host processor using BIP calls. That means that messages will not be handled immediately as they arrive if the compute processor is doing some other computation. For instance, there are three different cases of bad (but common) communication schemes in terms of communication overlap. The first case is the case where the compute processor on the receiver is busy with some computation when the control message containing the send request arrives. The send request acknowledgment will be sent only at the end of the computation. The second case is another case where the compute processor on the sender is busy with some computation when the control message containing the send request acknowledgment arrives. Then, the compute processor will initiate the data transfer only at the end of its computation. The last case is the gathering of the two previous cases. More communication steps are needed to complete a nonblocking send (see Figure 5). A first communication is done to send the message control and a second to return the acknowledgement message (steps 1 to 8). They required intervention of the compute processor which is in charge of all the communication data structure management (for instance those that allow to do the context/tag/peer matching between MPI send and receive calls). The next steps are the communication of the application data.

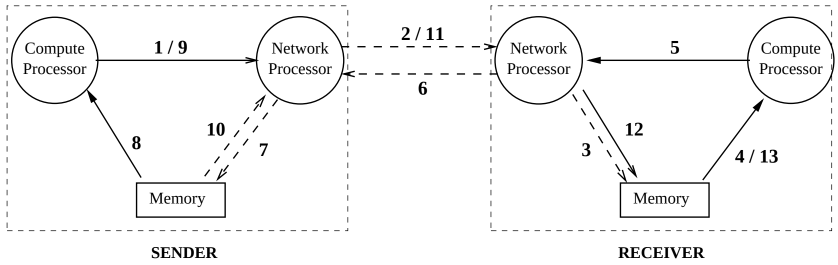


Fig. 5. The different steps for a complete non-blocking send. The solid lines show the steps dedicated to the MPI interface while the broken lines show the induced BIP native steps.

5.2 Communication load on the compute processor

With MPI, most of the communication management is handled by the compute processor (although this is a fixed overhead independent of the message size). For instance, as there is no limitation in the number of pending communications, communication requests queues and search tables are stored on the compute processor, because the memory and capacity of the network processor are too limited to handle all those data structures as efficiently. This leads to the intricate interaction scheme in the Figure 5 and brings us to the conclusion that every communication is mainly handled by the compute processor.

6 MPI-BIP overlap experiments

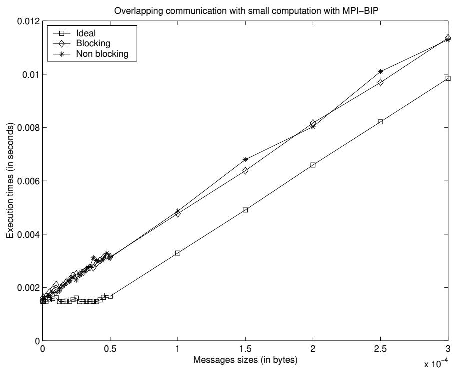


Fig. 6. Overlapping communication with computation using MPI-BIP.

The current MPI-BIP implementation does not provide any overlap in practice. The main restriction comes from the high interaction between the compute and the network processors. We executed the same test program as with BIP native primitives (Section 4). The results are shown in Figure 6. We observe that the total time of the non-blocking version is close to the total time of the blocking one. Results are even worse in a non-blocking version probably because of overhead of splitting the communication in two calls.

7 The overlap trick

To get some overlap with the MPI-BIP interface, we need to interrupt the user's program to switch to communication handling. The cost of the interruption must be low enough so that the improvement due to overlap is not lost. Such a design could be done by using hardware interrupts, and the use of signal handles inside the MPI-BIP implementation. But an evaluation of this strategy can be done by modifiying the application to periodically check the network status, to eventually launch the next step of the communication protocol. To get this effect, we tried to periodically call a "neutral" MPI primitive in the computation program who has no semantic side-effect, but which will potentially allow the background communication to progress by handling the intermediate events. Figure 7 gives the results for two different experiments. In the first experiment, the primitive MPI_Iprobe is not called in the main loop. As we can see, there is no communication overlap. In contrast, in the second experiment, the primitive MPI_Iprobe is called in the internal loop of the computation. In this way, the communication overlap is total (note that the overhead of calling MPI_Iprobe is insignificant).

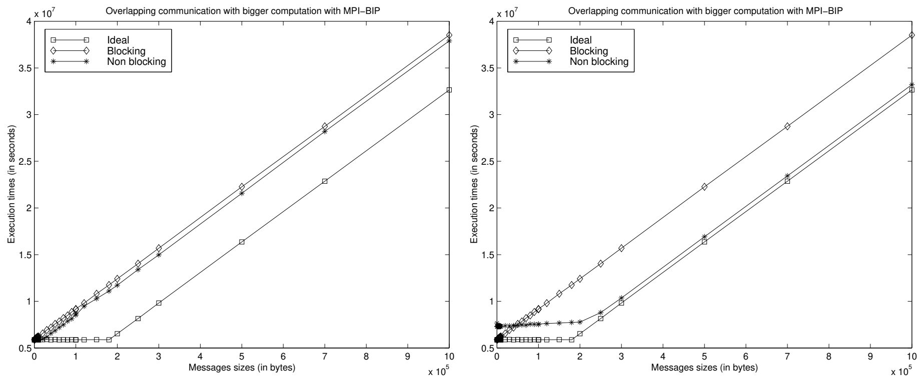


Fig. 7. Calling MPI_Iprobe in the external and internal loop of the computation.

8 Conclusion

In this paper, we tried to test the possibility of *asynchronous* communications with the BIP based implementation of MPI on a cluster of PCs connected by a Myrinet network. Even if MPI has been designed to allow such an overlap, it may look surprising that in practice MPI implementations will not always allow to exploit it. We tried to explore in details what where the reasons for this defficiency in MPI-BIP by analyzing its design. The same problems occur for most other implementations of MPI on clusters. From our study, we can actually derive a set of conditions that must fullfill MPI implementations to actually provide communication overlap. Either it must internally rely on an interrupt-driven mechanism which allow to interrupt the main computation to handle the protocol processing of the communications in the background, or there should be a second processor independent of the main compute processor, which should be able to deal with this processing. Through a simple experiment, we have shown that the interrupt approach would benefit some programs: we simulated its expected behaviour by inserting a periodic check of the network status in the compute code. This looks promising although the exact evaluation of the overhead of interrupt-driven processing for non-overlapping programs remains to be done. A perspective is to have a real implementation of this design to experiment with. An other solution to allow *asynchronous* comunications in MPI applications is to make the network processor handle all the communication protocol issues up to the MPI level. In practice it is not clear if this approach is reasonable with the kind of processor available on high-speed network cards nowadays.

References

1. P.M. Behr, W.K. Giloi, and W. Schröder. Synchronous versus Asynchronous Communication in High-Performance Multicomputer Systems. In *Aspects of Computation on Asynchronous Parallel Processors*, pages 239–249. IFIP, Elsevier Science Publishers, 1989.
2. Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet – A Gigabit-per-Second Local-Area Network. *IEEE MICRO*, pages 29–36, February 1995. `http://www.myri.com/research/publications/index.html`.
3. M.J. Clement and M.J. Quinn. Overlapping Computations, Communications and I/O in Parallel Sorting. *Journal of Parallel and Distributed Computing*, 28:162–172, 1995.
4. D.E. Culler, J. Pal Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, 1998. ISBN 1-55860-343-3.
5. F. Desprez, J.J. Dongarra, and B. Tourancheau. Performance Study of LU Factorization with Low Communication Overhead on Multiprocessors. *Parallel Processing Letters*, 5(2):157–169, 1995.
6. T. Von Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser. Active Message: a Mecanism for Integrated Communication and Computation. In ACM IEEE Computer Society, editor, *The 19th Annual Symposium on Computer Architecture*, pages 256–266. ACM Press, May 1992.
7. Loïc Prylli. *BIP Messages User Manual for BIP 0.94*, June 1998. `http://www-bip.univ-lyon1.fr/bip.html`.
8. Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: the complete reference*. MIT Press, Cambridge, MA, USA, 1996.