An Improved NIC Program for High-Performance MPI

Loïc Pryll†, Bernard Tourancheau and Roland Westelin
RHDAC, CNRS-INRIA ReMap Project, LHPC
Université Claude Bernard Lyon
69622 Villeurbanne, France

†LIP, CNRS-INRIA ReMap Project, LHPC
Ecole Normale Supérieure de Lyon
69634 Lyon, France
e-mail: bip-team@lhpc.univ-lyon1.fr

Abstract

We developed an MPI-BIP implementation, designed for Myrinet networks, and based on MPICH. By using our “Basic Interface for Parallelism: BIP” software layer, this implementation of the MPI protocols squeeze the most out of the high speed Myrinet network, without wasting time in system calls or memory copies, giving all the speed to the applications.

But it was subject to seldom message losses due to hardware timeout on the switches and contentions. The solution was the splitting of very long messages in several messages or packets. We first tried to split messages in MPI-BIP but it leads to a decrease in the throughput performance.

We then, decided to modify the BIP layer to solve this problem. We present in this paper the re-designed of our Myrinet Control Program that was the heart of BIP. The design is based on a state automaton implemented with an event-driven strategy leading to a very efficient and modular NIC program.

1 Introduction

In the last decade, researchers tried to use COWs (Cluster Of Workstations) as parallel computers. These clusters are typically connected by Ethernet networks and are often programmed with communication libraries like PV (Parallel Virtual Machine [6]), or MPI over TCP-UDP/IP (Internet Protocol). The poor performance of these solutions restricts application programmers to coarse grain parallelism.

It is now possible to build a platform with efficient components. The price performance ratio of COWs depends on:

The processing power commodity components (like Intel Pentium or DEC alpha processors) now deliver competitive performance.

The network speed Massively Parallel Processor technologies (SCI and Myrinet) are available to interconnect workstations.

The operating system free software operating systems (like Linux) provide the flexibility and performance needed.

The communication libraries

The communication software in use with such clusters is MPI (the Message Passing Interface). Classical implementations of this standard are based on TCP-UDP/IP. This protocol stack was designed for slow, unreliable and wide area networks. Experiments show that the performance of these implementations are far under the potential of the hardware. When bandwidth increases, the latency should decrease as much in order to keep the system balance. With the current network technology, the main bottleneck is most of the time the software that makes the interface between the hardware and the user.

However, MPI implementations can be easily tuned to uncover the performance of the underlying hardware. We designed a new MPI implementation for a Myrinet network based on MPICH. By using our “Basic Interface for Parallelism: BIP” software layer, this implementation of the MPI protocols squeeze the most out of the high speed Myrinet network, without wasting time in system calls or memory copies.

But it was subject to seldom message losses due to hardware timeouts on the switches. Basically, this was the result of the monopolization of a route by very long messages because BIP was sending any message as a whole network packet. The solution was the splitting of very long messages in several messages or packets, depending of the level chosen, in order to avoid this long contention problem. The realization of the segmentation of the messages at the Network Interface Card (NIC) program level prove to be the solution. Regarding the BIP optimized design, this leads to major changes. We present in this paper the re-designed of our Myrinet Control Program that was the heart of BIP. The design is based on a state automaton implemented with an event-driven strategy leading to a very efficient and modular NIC program.

This paper is organized as follows. Sections 2, 3 and 4 introduce previous works in this domain, BIP and MPI-BIP, our adaptation layer protocol for MPICH. Section 5 explains the problem we faced. Two solutions we investigated are described in sections 6 and 7. We conclude in section 8.
2 Background

Methodologies for the design of protocol stacks changed with the new high performance networks. The bottleneck can often be the interface between the network and the host or the communication system if they are badly designed, especially in regards to latency.

Several research teams proposed new approaches:

Active Messages (AM) [13] use an RPC-like (Remote Procedure Call) mechanism. Each message begins with the address of a handler. It is executed by the receiver to integrate the data in the ongoing computation. A MPI implementation on top of the AM is available [3].

Fast messages (FM) [10] use the same RPC-like system. The designers of FM provide an implementation of MPI [8].

PM [7] is a fast communication layer for the myrinet network. MPI was ported to FM [9].

GM [1] is the native API for the myrinet hardware developed by myricom. MPI is also available on top of GM.

Virtual Interface Architecture (VIA) [4] is a new software architecture for an efficient access to the network hardware from the user applications proposed by computer industry leaders (Microsoft, Intel and Compaq). VIA borrows a lot to U-Net [2]. At the time of writing, no free implementation of MPI is available for VIA.

These projects try to optimize the data path from the user application buffer to the network. All the above MPI implementations are based on MPICH [6].

3 BIP (Basic Interface for Parallelism)

BIP [11] is a low level layer for the Myrinet network developed by our team in Lyon. The goal of this project is to provide an efficient access to the hardware and to allow zero memory copy communications. BIP only implements a very raw link level flow control. It is not meant to be use directly by the parallel application programmer as it supplies very few functionalities. Instead, upper layers are expected to provide a development environment with a good functionality/performance ratio.

The API of BIP is a classical message passing interface. BIP provides both blocking and non-blocking communications. Communications are as reliable as the network, errors are detected and in-order delivery is guaranteed.

It is composed of a user library, a kernel module and a NIC program.

The key points of the implementation are:

A user level access to the network which avoids system calls and memory copies implied by the classical design becomes a key issue: the bandwidth of the network (160 MB/s in our case and 133 MB/s for the I/O bus) is the same order of magnitude than the speed of a memory copy.

The long messages follow a rendez-vous semantic: the send is guaranteed to complete only if a corresponding receive has been posted because of the limited capacity of the network. Messages are split into DMA fragments and the steps of the communication are pipelined. In order to reach the optimal performance, a theoretical model was studied and a variable packet size strategy is implemented to guarantee the pipeline efficiency [12].

The small messages are directly written in the network board memory on the sending side and, copied in a queue in main memory on the receiving side, because initialisations and handshakes between the host and NIC program are more expensive than a memory copy for small messages. The size of this queue is static it is up to the application to guarantee that no overflow occurs. On the receiving side, a memory copy puts the message in the user buffer platform (for this range of messages size, the copy is small compared to the overall transmission).

The communication latency of BIP is 6.5 μs (in the version used by MPI), the maximal bandwidth is 126 MB/s (65% of the theoretical hardware maximum, actually represented by the PCI bottleneck in our case). Half the maximum bandwidth is reached with a message size of 4KB (= N/2).

4 The MPI-BIP design

Our MPI implementation is based on MPICH [6]. This implementation of the standard is organized in layers and designed to ease the porting to a new target. However, MPICH expects the underlying communication software to supply reliable communications.

BIP is not designed to be used directly by the parallel application programmers. BIP communications are not reliable both for small and large messages:

Small messages are stored in a queue of fixed length on the receiving side. If the sender produces messages faster than the receiver consumes them, an overflow occurs and messages can be lost. Some flow control must be provided.

To use BIP small messages in our MPI implementation, we use a credit based mechanism with piggy backing.

Large messages are copied to main memory only when the receiver buffer is known (when the receive is posted). As long as no receive is posted, the bytes of a large messages fill the network buffers and the send can't complete. But, to ensure that the network is deadlock-free, myrinet switches and boards use timeouts. A packet can be dropped if no data movements happen in the network buffers. Thus, a way to ensure that receives are already posted on the receiver when the send starts must be provided.

A three way protocol (request/acknowledgement) is used with a possible buffer allocation on the receiving side.

Our first implementation used BIP directly with these protocols. It leads to very good performance: a latency of 9 μs, a maximal bandwidth of 125 MB/s. Half the maximum bandwidth could be reached with a message size of 8KB (= N/2). However, as we will see in the next section, seldom message losses still occur.
5 Why fragmentation is needed

Myrinet switches are full duplex crossbar. However, if two messages need to use the same output port from two different input ports, the switch arbitrates this conflict using a round robin scheme: for instance, in figure 1, 5 packets enter a switch from 3 different ports (labeled 1, 2 and 3) and conflict for the output port 4. Let's say that the switch begins by transferring the data available on port 1. It proceeds byte after byte until it reads the end of packet character. Then it checks if some data are ready to be copied to output port 4 from port 2, proceeds if it is the case (as in our example) then checks port 3 and goes back to port 1. Arbitration between conflicting ports occurs only when the transfer of a packet is completed. So, for instance, if packet 1 is very large, it will monopolize the switch for a long time and data of packets 4 and 5 will fill the network buffers waiting for their turn.

BIP sends every messages from the upper layer as a single network packet. So in our first MPI implementation, one MPI message was one BIP message and thus, was one packet on the network "wire". As told above, myrinet switches use a timeout to ensure that the network is deadlock-free. In our previous example, if packet 4 remains blocked for a long time at port 2 (if, for instance, packet 1 is very large), it is dropped. Our experiments showed that this kind of message losses occurred with real world applications. Furthermore, this problem is increased in configurations with multiple switches when contention occurs more often.

The solution is to make sure that every packet on the network "wire" has a size inferior to an upper bound. By choosing a clever value for this upper bound, we will be sure that every packet conflicting for an output port will have its chance to cross the switch before the timeout triggers a message loss. In practice, large MPI messages must be split into several network packets.

Splitting in fragments is needed to ensure reliability at the MPI level but if done efficiently, it can also improve the performance of applications. If, for instance, all the communications between two clusters of PCs are done via a single myrinet link, fragmentation obviously improves the fairness of the access to the link and can maximize its use by reducing the "bubbles" (idle flits).

6 The first solution

In order to split a MPI message into several network packets, we first tried not to modify BIP. So a simple MPI message was split into several BIP messages (via several calls to the BIP API). But, only one send can be posted with BIP and our MPI implementation can't be notified asynchronously of an event (like the completion of a send). So, when two applications exchange messages that must be split into several BIP messages, they must synchronize after each BIP messages (with some small message communications). This leads to a loss of performance. When using 100KB fragments, the bandwidth decreases to 110 MB/s ($N_{1/2} = 8000$ bytes) and when using 10 KB fragments (what we think is a reasonable value), the maximal bandwidth is only 66 MB/s ($N_{1/2} = 3072$ bytes). Even, if the performance of MPI with fragmentation in this layer remains in the range of current Massively Parallel Processors, a lot of the potential of the network is lost. Moreover, this splitting strategy will use the host-processor preventing any potential overlap between computation and communication on the network. Furthermore, this type of contention may happen in MPI-BIP and any native BIP applications.

7 The second solution: our state automaton based design

The realization of the segmentation of the messages at the NIC program level proved to be the solution. A fair splitting strategy in the NIC program automatically leads to a good balance on the contention port because the switch itself uses a fair algorithm to arbitrate conflicts.

Regarding the BIP optimized design, this leads to major changes. The new design is based on a state automaton implemented with an event-driven strategy leading to a very efficient and modular NIC program.

Such a design has several advantages:

- As demonstrated in section 7.3, it can be implemented efficiently and leads to good performance.

- It is flexible and can be extended easily. Then added new functionalities, will not impact the ones that are already available.

- It is easy to program, to understand and to debug.

7.1 The state diagrams

The behavior expected from the NIC program can be easily describe with two state diagrams: one for the reception (figure 3) and one for the emission (figure 2).

The four steps of the BIP communication process (copy from the host memory to the NIC memory, sending on the wire, reception from the wire and copy from the NIC memory to the host memory) are pipelined. Every BIP message is split into smaller DMA fragments. We designed an
adaptive strategy based on a theoretical model [12]. This strategy is one key implementation decision that leads to the high performance of BIP for the throughput. With our new NIC program, we wanted to keep this strategy even if fragmentation imposes some constraints on DMA fragment size.

This auto adaptive pipeline strategy is visible in the state diagrams. Let’s, for instance, describe what happens in figure 3 when a big message needs to be send. First the NIC program is in the idle state. The host machine requests a big message send. The state of the NIC program changes to the big message state. It sends the header of the current network packet and switch to the packet sending in progress state. Then it sends each of the DMA fragments that make the packet one by one by going to the send DMA fragment state. When this is done, the packet is sent and the NIC program goes back to the state big message. If the packet just sent was the last one, it signals to the host that the message was sent and becomes idle again. Otherwise, it sends the header of the next packet and goes to the packet sending in progress state.

It is easier to describe the NIC program with two state diagrams even if, in the implementation, they are mixed, because BIP is full duplex. It means that every combination of states between the two diagrams are possible.

7.2 Implementation

As shown by figure 4, the current state of the NIC program is computed from three state variables:

- The hardware state is a register of the NIC processor. It contains informations on the current DMA both on the PCI bus and to the wire.
- The NIC software state is a global variable describing the current state of the NIC program. Only the NIC program modifies it.
- The host software state is a global variable describing the request from the host.

Remark that two variables are used to describe the software states: avoiding any problem with concurrent writes to the memory variables from the host and NIC processor is then easier.

These three variables are combined to make an index used in a first table to obtain the corresponding action. This action is used in a second table to get the address of a handler to execute. The first table is large (2 KB in the current implementation). Several entries can have the same value and some of these entries are useless at the moment (no action needs to be executed). The second table, which is small (36 B in the current implementation), can be modified by the NIC program. The use this indirect level eases the programming of the NIC because at two different times, the same state can lead to two different handlers.

This design was thought to be both flexible, simple and effective. The heart of the event loop is currently coded in a few assembly instructions.

7.3 Performance measurements

Our bandwidth measurements between two pentium pros are shown on figure 5.
The size of the packets in these measurements is 8 KB. For small messages (of size inferior to 256 Bytes), using an event-driven NIC program has nearly no impact. The point to point latency decreases of less than 0.3 µs. For messages of size inferior to the packet size, the bandwidth decreases slightly due to the use of the event-driven strategy. For messages greater than the packet size, the difference is explained both by the new design and by the fragmentation which adds data and processing for each packets.

8 Future works

Our work proves that the performance gap between workstation clusters and parallel machines is nearly filled. However, even if parallel machines are still more powerful, COW based on inexpensive hardware has an unbeatable performance price ratio.

Our future work on MPI-BIP will investigate the increase of the performance by implementing some of the flow control at the BIP level in the Network Interface Card program. We would especially like to improve the throughput of MPI and enable a better communication/computation overlap. We will experiment with new auto adaptive pipeline strategy like those described in [14] to try to hide the overhead of the fragmentation.

We are also working on a new MPICH architecture to ease the porting of MPI on new high speed network.

Furthermore, we are studying the modifications needed at the BIP level to support efficiently the MPI 2 functionalities.

The project home pages are available on the net http://www-bip.univ-lyon1.fr.

Our software has been downloaded by more than 110 different research teams worldwide and computer scientists are using it for large applications.

References


