Universitat Politècnica de València

Master Thesis

A New Credit-Based End-to-End Flow Control Protocol for High Performance Interconnects

Author:
Javier Prades Gasulla

Supervisor:
Federico Silla Jiménez

A thesis submitted in fulfilment of the requirements
for the Master's Degree in Computer and Network Engineering

in the

Parallel Architectures Group
Department of Computer Engineering

UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

July 2015

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Abstract

Department of Computer Engineering

Master's Degree in Computer and Network Engineering

A New Credit-Based End-to-End Flow Control Protocol for High Performance Interconnects

by Javier Prades Gasulla

High Performance Computing usually leverages messaging libraries such as MPI, GAS-Net, or OpenSHMEM, among others, in order to exchange data among processes in large-scale clusters. Furthermore, these libraries make use of specialized low-level network layers in order to achieve as much performance as possible from hardware interconnects such as InfiniBand or 40Gb Ethernet, for example. EXTOLL is an emerging network targeted at high performance clusters.

Specialized low-level network layers require some kind of flow control in order to prevent buffer overflows at the receiver side. In this work we present a new end-to-end flow control mechanism that is able to dynamically adapt, at execution time, the buffer resources used by a process according to the communication pattern of the parallel application and the varying activity among communicating peers. The tests carried out on a 64-node 1024-core EXTOLL cluster show that our new dynamic flow control mechanism presents very low overhead with an extraordinarily high buffer efficiency, as overall buffer resources are reduced by 4x with respect to the amount of buffers required by a static flow control protocol achieving similar low overhead levels.

Acknowledgements

I would like to express my gratitude to my supervisor, Prof. Federico Silla, whose expertise, understanding, and patience, added considerably to my experience. I appreciate his vast knowledge and skill in high performance interconnection networks, and his assistance in writing reports (i.e., papers, journals and this thesis). I would like to thank the EXTOLL developers, Dr. Holger Fröning and Dr. Mondrian Nüssle for the assistance they provided me during this research project. Finally, I would like to thank Prof. José Duato, main researcher of the Parallel Architectures Group.

I take this opportunity to express gratitude to all of the Department of Computer Engineering members for their help and support.

I thank my fellow labmates and friends in the Parallel Architectures Group.

I would like to thank my family for the unceasing encouragement, support and attention.

Finally, I also place on record, my sense of gratitude to one and all, who directly or indirectly, have lent their hand in this venture.

Contents

List of Figures

List of Tables

Acronyms

RAM   **R**andom-**A**ccess **M**emory

HPC   **H**igh **P**erformance **C**omputing

MPI   **M**essage **P**assing **I**nterface

UPC   **U**nified **P**arallel **C**

GASNet   **G**lobal-**A**ddress **S**pace **Net**working

TOFU   **TO**rus **FU**sion

HT   **H**yper**T**ransport

PCIe   **P**eripheral **C**omponent **I**nterconnect Express

VOQ   **V**irtual **O**utput **Q**ueuing

NIC   **N**etwork **I**nterface **C**ard

ATU   **A**ddress **T**ranslation **U**nit

VELO   **V**irtualized **E**ngine for **L**ow **O**verhead

RMA   **R**emote **M**emory **A**ccess

PIO   **P**rogrammed **I**nput/**O**utput

CPU   **C**entral **P**rocessing **U**nit

API   **A**pplication **P**rogramming **I**nterface

DDR2   **D**ouble **D**ata **R**ate type two

FPGA   **F**ield **P**rogrammable **G**ate **A**rray

LiMIC   **Li**nux kernel module for **M**PI **I**ntra-node **C**ommunication

FIFO   **F**irst **I**n **F**irst **O**ut

IMB   **I**ntel **M**PI **B**enchmarks

Definitions used in this Thesis

- **available_slots counter**: in the dynamic flow control, counter at the receiver side, that tracks the amount of available slots in the data region of the mailbox.

- **compulsory credit return request**: type of control packet used by receivers to force senders with low or null activity, from the receiver point of view, to forward the credits that they are not using.

- **compulsory credit return response**: type of control packet used by senders as a response to a previous compulsory credit return request. This packet will contain the amount of returned credits.

- **compulsory request flag**: in the dynamic flow control, flag that signals the existence of a compulsory credit return request packet that could not be sent due to lack of credits.

- **compulsory response flag**: in the dynamic flow control, flag that signals the existence of a compulsory credit return response packet that could not be sent due to lack of credits.

- **control packet**: type of packet used by the flow control protocol. Control packets are divided into three categories: credit packet, compulsory credit return request, and compulsory credit return response.

- **credit counter**: counter, at the sender side, that keeps track of the credits available to send VELO packets to a given receiver.

- **credit packet**: a packet containing credit information.

- **credit quota**: amount of slots, in the data region of a receiver's mailbox, owned by a sender.

- **credit region**: region of the mailbox intended to store incoming credit packets from all the processes receiving data from the mailbox owner.

- **credit slots**: amount of slots, in the credit region of a sender's mailbox, owned by a receiver.

- **current_credits counter**: in the dynamic flow control, counter at the receiver side, that tracks the amount of credits currently granted to the sender.

- **data packet**: a packet containing application data.

- **data region**: region of the mailbox intended to store incoming data packets from all the processes sending data to the mailbox owner.

- **dynamic region**: in the dynamic flow control, part of the mailbox containing most of the data slots of the mailbox (all except those in the static region).

- **intended credit quota**: in the dynamic flow control, counter at the receiver side, that represents the amount of credits the sender should have in the near future.

- **mailbox**: a contiguous memory region at the receiver configured as a circular buffer with a single write pointer and a single read pointer. A mailbox is statically sliced into slots able to contain a single VELO packet.

- **monitoring point**: in the dynamic flow control, the end of the period of time required by a sender to consume all of its credits.

- **retrieved_packets counter**: counter, at the receiver side, that keeps track of the packets removed from the mailbox.

- **slot**: each of the slice of a mailbox. A slot is able to contain a single VELO packet.

- **static region**: in the dynamic flow control, part of the mailbox containing the minimum amount of data slots, as defined by the flow control protocol.

- **static_slots**: in the dynamic flow control, minimum amount of data slots, as defined by the flow control protocol.

Chapter 1

Introduction

The computing landscape has been traditionally driven by the demand for a never-enough computing power. Thus, even the powerful commodity computers available nowadays, which provide up to 80 processor cores and up to 2TB of RAM [1], do not satisfy the requirements of most High Performance Computing (HPC) applications in areas as diverse as computational algebra [2], quantum mechanics [3], biochemical dynamics [4], and fluid dynamics [5], to name only a few. As a result, these demanding applications are split into as many parallel processes as possible, which are concurrently executed on the cores available across large clusters, thus aggregating huge amounts of computational resources. In order to exchange data among the processes involved in the application execution, some kind of messaging layer is usually leveraged. One of them is the Message Passing Interface (MPI) library [6]. Actually, the MPI library is currently the *de facto* standard for programming large-scale parallel computing applications.

MPI has proven efficiency for large-scale computing deployments, where the use of shared-memory programming is not possible. Thus, in the near future parallel programming will surely rely on MPI. However, as MPI puts lots of burdens on the programmer, other parallel programming approaches closer to the shared-memory programming model have been devised. This is the case, for example, for Berkeley Unified Parallel C (UPC) [7], which provides support for shared-memory programming across distributed systems, usually leveraging the GASNet [8] messaging layer. GASNet is a language-independent, low-level network layer that provides high performance communication primitives for implementing parallel global address spaces in environments like clusters.

The popularity of messaging layers such as MPI or GASNet is mostly due to the performance and portability they offer. Applications written according to these libraries can be run on any underlying HPC system as far as the appropriate library implementation is available. Because so many applications rely on these messaging layers, most high performance interconnects have at least one available implementation. This is the case, for example, of the well-known Ethernet and InfiniBand [9] networks, which are the most widely used interconnects according to the TOP500 supercomputing list [10]. Other less used high performance interconnect technologies that provide an implementation for at least one of these messaging layers are TOFU [11], Quadrics [12], and the interconnect used in the last Cray systems [13].

One of the key issues when designing an efficient messaging layer implementation is flow control. The flow control mechanism prevents a fast sender from overwhelming a slow receiver and exhausting its buffer space resources, what could not only affect performance but also the proper functioning of the communicating peers, depending on the exact network technology used. Furthermore, the efficient design of a flow control mechanism is an important issue as it affects both the performance and the scalability of any messaging layer implementation. In the case of performance, the flow control mechanism has to ensure that buffers at the receiver are efficiently managed in order to avoid sending processes from stalling due to the lack of buffers to store received messages. Regarding scalability, the amount of receive buffers required to ensure an optimum performance level of the communication flow should be kept as low as possible in order to avoid devoting too many memory resources across the cluster to the internals of the communication scheme, therefore maximizing the amount of resources available for applications. Actually, the best scenario would be that per-process resource usage grew sublinearly with the number of processes in order to make a flow control mechanism appropriate for large-scale systems. However, a linear growth is the usual case, which is also acceptable.

In this work we present an efficient new end-to-end flow control mechanism that dynamically assigns buffer resources to active communication flows. In order to show the extraordinary scalability and performance advantages of our new flow control mechanism, we center our presentation around the MPI implementation for the new EXTOLL [14] interconnect, although other messaging layers, such as GASNet, could also be used. Notice that all the experiments presented in this work have been conducted in our 64-node

1024-core EXTOLL cluster.

The rest of the work is organized as follows. Chapters 2 and 3 briefly introduce the main features of the new EXTOLL network and its implementation of OpenMPI. Chapter 4 discusses the rationale behind flow control mechanisms. Then, Chapter 5 introduces a thorough analysis of a static flow control mechanism, which will be used as the baseline for our new dynamic flow control technique, later presented in Chapter 6 and analyzed in Chapter 7. Chapter 8 describes related work. Finally, Chapter 9 presents the main conclusions and contributions from this work.

Chapter 2

The EXTOLL Interconnect

EXTOLL is a high performance interconnection network intended for HPC systems. Its main goals are to reduce message latency to a minimum, to maximize the sustained message rate, and to provide a high scalability. Thus, EXTOLL puts special attention on optimizing communication for small messages, which typically suffer from high overhead compared to bulk transfers.

EXTOLL's architecture is shown in Figure 2.1; comprised of the host interface shown on the left side, which is currently based on HyperTransport (HT)[1], the network interface including multiple communication engines shown in the center, and the network section with switch and six network links shown on the right side.

The complete EXTOLL architecture is implemented on a single chip that is typically located on an add-in card. All the required switching resources are already integrated in that chip, so that besides cabling no further resources are required. With its six available links, direct topologies like 3D meshes or tori are a natural choice. The integrated switch implements a variant of *Virtual Output Queuing* (VOQ) at switch level to reduce *Head-of-line* blocking, and uses cut-through switching that enables very low switching latencies. Deadlocks are avoided by employing two virtual channels. The switch ensures that packets are forwarded in-order, which means that a sequence of packets sent from node A to node B will arrive in the same order as they were initially injected into the network. This feature can be used by software components in order to simplify the design

---

[1]Other EXTOLL incarnations replace the HT interface with a PCIe interface. From an architectural point of view, there is no difference between these two implementations.
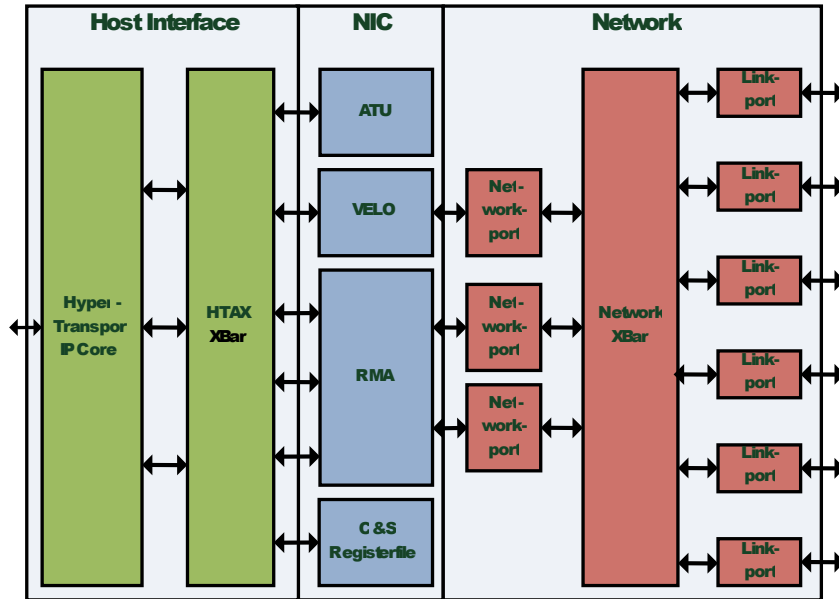
FIGURE 2.1: EXTOLL's top-level block diagram

of upper-level protocols. Nevertheless, future versions of the EXTOLL interconnect may leverage out-of-order delivery to support adaptive routing, although this feature will make upper software layers slightly more complex.

As EXTOLL is specifically designed to efficiently support fine grain communication schemes, it includes a special communication engine named VELO (*Virtualized Engine for Low Overhead*) [15] for an optimized transmission of small messages. Its highly tuned interface between hard- and software not only provides low latency, but also high message rates. On the sending side, basically only one PIO (*Programmed Input/Output*) write to a special address is required for up to 56 bytes[2]. Meta-data like destination and length are encoded in the address, so the complete PIO write payload is available for software use. On the receiving side, packets are directly written into main memory using a single ring buffer per process, called *mailbox*. A mailbox is statically sliced into slots able to contain a VELO packet. In this way, receiving processes can poll on known main memory locations and the coherence protocol ensures that no unnecessary bus traffic is generated.

Beside this support for fine grain communication, we address efficient bulk transfers with the RMA (*Remote Memory Access*) communication engine [16], which offers Put/Get semantics to the user. A hardware-based address translation unit (ATU) assists it, and in conjunction user-level secure data transfer is guaranteed. The RMA unit almost

---

[2]The typical payload of VELO packets is 56 bytes.

completely off-loads the communication task from the CPU, providing high overlap and low overhead.

In combination, both units provide efficient support for all message sizes. Using the EXTOLL API libraries (libvelo and librma), the programmer can decide which method is more suitable for a given data transfer based on its size. Nevertheless, notice that the flow control proposed in this work is intended for the VELO unit, as it uses a constrained amount of memory (the mailbox) for receiving packets. In this way, the flow control code will be incorporated into the libvelo library, thus being transparent to users.

Finally, reliable transmission is guaranteed by a link-level retransmission protocol, another important feature to simplify the design of software components. Notice, however, that although the hardware includes this link-level retransmission protocol, it does not feature an end-to-end flow control mechanism, contrary to what happens in other high-performance interconnects, such as InfiniBand. This is an important difference. In InfiniBand, if a message arrives and there is no receive buffer posted yet, the network interface at the receiver will issue a Receiver-Not-Ready message. The network card at the sender will then wait for a time-out and retry the send operation until the corresponding receive operation is posted. On the contrary, in the EXTOLL technology, when the receiver network interface cannot store the incoming VELO packet because the mailbox is full, back pressure is used to avoid the preceding router in the path between sender and receiver to forward further packets. This back pressure finally arrives at the sender network card, forcing the entire sender node to block. This makes necessary, in EXTOLL, to implement the end-to-end flow control mechanism with the thin software layer on top of the hardware.

In summary, while a software end-to-end flow control in InfiniBand just improves performance, as stated in [17], in the case of the EXTOLL interconnect a software end-to-end flow control is required for the right behavior of the system. Notice that this software end-to-end flow control should be included in the thin libvelo layer and would provide larger flexibility than a hardware flow control (actually, in this work we are proposing two different flow control choices, what would be difficult to do in case flow control is implemented in hardware).

We have built a 64-node EXTOLL cluster that will be used as testbed throughout this work. Each node is based on the Supermicro H8QM8-2+ motherboard containing four

2.1GHz quad-core Opteron processors. Each processor is attached 4GB of 800MHz DDR2 memory. Thus, each node features 16 cores and 16GB of main memory, accounting for a total of 1024 cores and 1TB of RAM memory across the cluster. Furthermore, each nodes owns an add-in card that will implement the EXTOLL architecture. This card includes an FPGA configured with the architecture depicted in Figure 2.1 and six fiber links, which allow to configure the EXTOLL interconnect according to a 3D mesh topology (4x4x4).

Chapter 3

MPI over EXTOLL

The MPI implementation of EXTOLL is based on OpenMPI [18]. In the level below MPI, the low-level API libraries libvelo and librma provide direct, user-level access to the functionality of the respective communication engine.

Typically, small messages up to 2KB[1] are sent using an *eager* protocol over VELO. In this protocol, a first VELO packet carries the necessary MPI header with the information for message matching and also some MPI payload up to the typical 56-byte size of a single VELO packet. If the MPI message does not fit into a single VELO packet, additional VELO packets that are tagged accordingly are sent. On the receiver, MPI matching is performed and, if needed, multiple VELO packets are reassembled to complete the operation.

MPI messages larger than the 2KB threshold are sent using a *rendezvous* protocol leveraging RMA. For this protocol, a small VELO packet is first sent carrying information that describes the buffer that is actually to be sent. Upon matching against a receive operation on the receiver side, the receiver completes the transfer by issuing one or more RMA get operations. These RMA operations then complete the data transfer in a zero-copy fashion. The notification features of RMA (notifications can be generated both on source and destination sides in order to signal completion of Put/Get requests) are used

---

[1]The threshold between VELO and RMA is configured to 2KB by default, although it can be set by the user to another packet size.

to signal completion of such a large MPI transfer, both on the sender and the receiver side, yielding a very efficient protocol[2].

A third protocol has been used for intra-node communication, taking advantage of the shared memory architecture of modern multi-processor nodes through a kernel module interface called LiMIC [19].

---

[2]The higher efficiency of notifications is based on the fact that without notifications more network traffic is necessary. Other solutions rely on polling on the last data word transferred, but this requires ensuring that the value of this last word changes and also prohibits out-of-order packet completion (some chipsets might opt to complete memory transactions out of order).

Chapter 4

About Flow Control Mechanisms

Basically, a flow control mechanism prevents that slow receivers, or receivers that are busy performing other tasks, get their receive buffers overflowed because of senders transmitting too fast. Notice that if receive buffers had an unlimited size, then this overflowing concern would not exist and hence a flow control mechanism would not be required. While this is obviously not feasible, the use of very large buffers would also require a lot of memory resources to be devoted to something that is not the application itself but the underlying communication infrastructure, the cost of which should be kept as low as possible. Moreover, notice that these memory resources are dependent on the number of processes involved.

Therefore, flow control mechanisms can be seen as techniques that establish a maximum boundary to the memory resources used by the communication layer. However, this limit may increase execution time for two reasons: first, some processes may stall because of lack of buffers at the receiver side. Second, the flow control protocol itself introduces computational overhead as well as additional network traffic because of the need of communicating the state of receive buffers. Thus, the efficiency of a flow control mechanism can be seen as a trade-off between the resources it requires and the overhead it generates.

In order to make a light-weight flow control implementation we have started with a static flow control that evolved to an efficient dynamic one. Both protocols are based on the use of credits to track the amount of available slots at the receive buffers, although the dynamic version manages credits in a flexible way, thus achieving better performance.

It is important to remark that the proposed credit-based flow control protocols in this work are entirely implemented within the libvelo library, being therefore transparent to upper software layers, such as MPI or GASNet, which would not require any modification to benefit from the new end-to-end flow control protocols implemented. In the following chapters we describe both protocols, along with a thorough analysis of their performance.

Chapter 5

Static Credit-Based Flow Control

In order to implement the static flow control version, we have started our development from the commonly used *credit-based flow control mechanism*, so that our implementation is a generalization of this mechanism in order to adapt it to the internals of our interconnection network.

5.1   Three major adaptations

The original credit-based flow control mechanism ensures that each sender owns certain buffer space at the receiver side. The exact amount of buffer resources is explicitly stated by the number of credits the sender is given at initialization time. In this way, a receiver usually has as many independent buffers (or buffer partitions) as senders exist. However, in our EXTOLL interconnection network a receiver only has one buffer, referred to as mailbox, which is shared among all of its senders. Therefore, the first adaptation to be performed is a trivial change that will allow us to adapt our monolithic mailbox scheme to the credit philosophy. This change is simply to equally distribute the mailbox space among all the potential senders. In this way, all senders will own a portion of the mailbox, all portions having the same size. This size will depend on the mailbox size and the amount of senders. Additionally, it will be constant along the application execution time. Notice that packets from different senders, when stored in the mailbox, will be mixed up given that buffer space in the mailbox is distributed among senders but the exact mailbox slots that each sender should use are not defined in order to make the

storage process of incoming packets simpler and more efficient. Furthermore, as we still have a single write pointer and a single read pointer associated to the entire mailbox (no hardware modification has been done to the EXTOLL network card during this adaptation process), it would not be possible to define separate mailbox partitions for different senders. In this regard, it is worth remarking that the mailbox of each receiver is a contiguous memory region configured as a circular buffer. Figure 5.1 depicts this distribution process. However, before describing this figure we need to address the two other major adaptations to be done to our monolithic mailbox scheme.

The second change to be performed, with respect to the original credit-based flow control protocol, is related to the way that the credit count is updated at the sender. In the original credit-based flow control mechanism, when the receiver frees up a slot of the receive buffer, a credit is sent back to the sender. However, as updating every single credit may generate large amounts of traffic, many implementations [20][21] accumulate multiple credits in order to send back a single packet with the credit information, thus reducing network traffic. In our implementation of the end-to-end credit-based flow control we establish that a receiver process will accumulate credits up to a certain threshold. Once that limit is reached, the process will generate a packet containing credits and will send it back to increment the sender's credit count. From now on, we will refer to this kind of packets as *credit packets* because they carry credit information. In a similar way, we will refer to the regular data packets generated by the parallel application simply as *data packets*[1].

Finally, the third major and more important adaptation to perform has to do with the fact that in our interconnection network the only way of communication between processes running at different cluster nodes is by exchanging messages, which require buffer space at the receiver in order to be stored while waiting to be appropriately processed. This also applies to credit packets. However, in many other implementations of the credit-based flow control, although credit packets are also used, they do not require buffer space because they are decoded and processed by the network adapter as soon as they arrive. This is a small but very important difference, as it means that in implementations of the flow control that do not buffer credit packets, these packets can

---

[1]Notice that credit packets may alternatively be referred to as *control packets* because they are generated by the flow control mechanism. Nevertheless, in Chapter 6 it will be shown that the dynamic flow control comprises two additional types of control packets. Therefore, using a more specific term, such as *credit packet*, will be helpful.
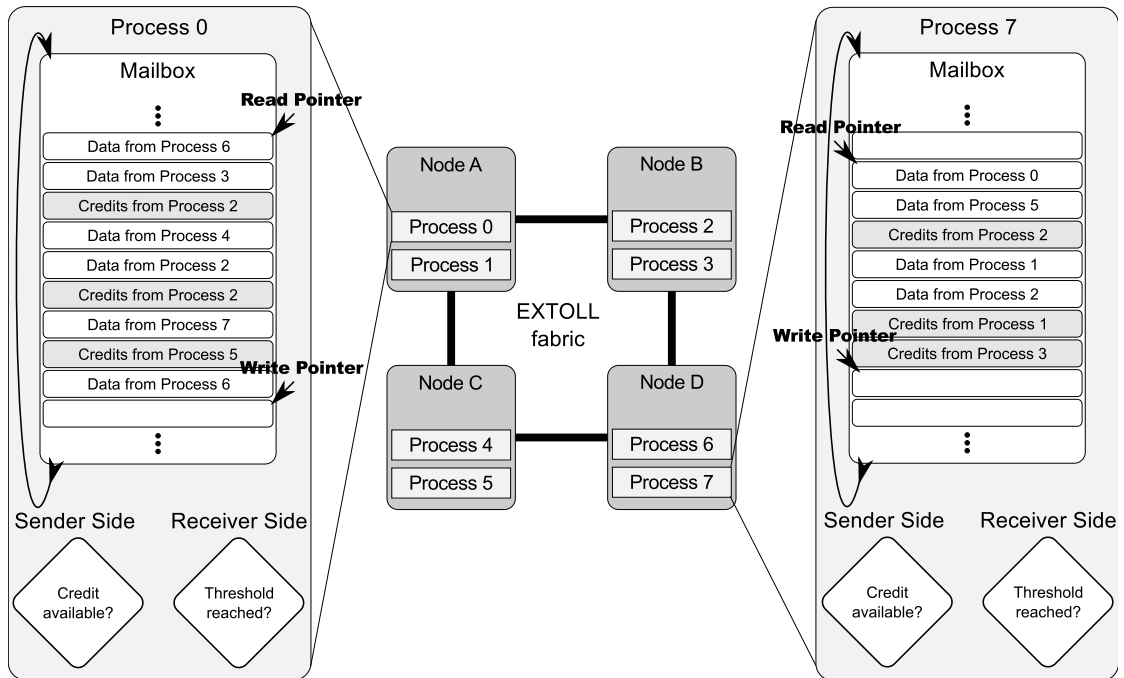
FIGURE 5.1: Example of a small EXTOLL network depicting the mailboxes of two different processes.

be sent at any time, whereas in our EXTOLL implementation one or more credits may be required to send them, what may cause a protocol deadlock if this situation is not properly managed. Therefore, it should be ensured in our implementation of the credit-based flow control that credit packets can be sent whenever they are required without causing a buffer overflow at the receiver. This will be achieved by saving, in a logical way, a small part of the mailbox for storing credit information and by properly defining an efficient threshold for credit return, as it will be deeply explained and analyzed in next Section 5.2.

Figure 5.1 depicts the main concepts related to the adaptation of our monolithic mailbox scheme to the credit-based flow control mechanism. This figure shows four cluster nodes interconnected by the EXTOLL fabric by making use of a 2D-mesh network topology. In this small example, each node is executing two different processes of an MPI application that spans all the four nodes. Processes at each node have been identified with different numbers for the sake of clarity. The figure also shows the mailboxes as well as the sender and receiver parts for process 0 running at node A and also for process 7 running at node D. It can be seen in these mailboxes how credit packets and data packets are stored in the mailbox in the same order as they arrive, thus mixing both types of packets in the mailbox if required. Furthermore, packets from different senders (either data or credit

packets) are also mixed up in the mailbox. Additionally, the figure depicts the read and write pointers for each of the mailboxes displayed. Finally, it also shows the main decisions taken by each process: first, the sender side of a process has to check for credit availability before sending a packet to a given destination. Keeping track of the credits available to send VELO packets to a given receiver will be achieved by making use of an array of counters, with length equal to the amount of receivers. The counter associated with a given receiver will be referred to as *credit* counter. Second, the receiver side checks whether the amount of packets retrieved from the mailbox has reached the threshold in order to send back a credit packet which will update the sender's credit count. The count of packets removed from the mailbox will be stored in the *retrieved_packets* counter[2].

5.2   Static credit-based flow control operation

A process can simultaneously behave according to two roles: sender and receiver, as it has been shown in Figure 5.1. We will refer to a process as *sender* when the process is sending data packets to other processes and it is extracting credit packets from its mailbox in order to retrieve credits to continue sending packets. We will refer to a process as *receiver* when the process is extracting data packets from its mailbox and it is sending back credit packets to update the state of its buffers at the sender side.

Initially, the mailbox at each process is split in a logical way into two regions: one for accommodating data packets, *data region*, and the other one for storing credit packets, *credit region*. The data region will contain packets from all the processes sending data to this receiver whereas the credit region will store incoming credit packets containing credit information which were sent by the other processes receiving data from this process. The data region will be equally[3] distributed among all the processes that may send data to this process. Thus, each sender will own some amount of slots in the mailbox of each receiver. This number of slots will be referred to as *credit quota* because each slot can

---

[2]As in the previous case, the receiver will have an array of *retrieved_packets* counters, each element of the array devoted to a different sender.

[3] Equally distributing the data region may not be possible if the number of slots is not dividable by the number of senders. However, as the user can define the mailbox size prior to program execution, this concern should not appear. Nevertheless, in case the user sets a wrong amount of slots, two options may be used: a) granting the exceeding slots to a subset of processes; b) rounding (upper or lower) the number of slots assigned to each process in order to provide all of them the very same space. Upper rounding would make use of a slightly larger mailbox whereas lower rounding would use a smaller mailbox. The system administrator (or even the user) may decide which option to use. These adjustments will not be required for the dynamic flow control, as shown later.

contain one VELO packet and, therefore, each slot will be equivalent to one credit. In a similar way, the credit region in a sender's buffer will be equally[4] distributed among all the processes that may receive data from it, so that they will own some amount of slots in the mailbox of each sender where to forward credit packets. The portion corresponding to each receiver will be referred to as *credit slots*. Finally we set the *threshold* that sets the amount of freed slots that have to be accumulated before sending back a credit packet. The exact value of this threshold will depend on the size of both data and credit regions. Additionally, the value assigned to the threshold will guarantee that, every time the threshold is reached, a credit packet can be safely sent back to the sender because the sender will have the buffer space required to store the credit packet. In this way credit packets can be sent without requiring the use of credits, given that this definition of the threshold ensures the existence of the required buffer space. The threshold is defined as:

$$threshold = (credit\_quota \ div \ (credit\_slots + 1)) + 1 \tag{5.1}$$

Notice that "div" stands for the integer division, thus discarding the fractional part of the result. Equation 5.1 means that, when the amount of freed slots accumulated by a receiver reaches the threshold, it sends a credit packet to the sender knowing that this credit packet will find at least one free credit slot in the sender's mailbox. We can state this because in order for the receiver to reach the threshold for the $(credit\_slots + 1)th$ time, the sender has to free a credit slot in order to obtain more credits to be able to continue sending data packets. In this way, this condition ensures that new credit packets can always be stored at the credit region of the sender's mailbox. Notice, therefore, that credit packets do not consume credits.

Finally, while computing the exact value of the threshold, we add the restriction shown in Equation 5.2, that sets the minimum size for the credit and data regions as well as the relationship between them so that the credit region is not unnecessarily larger than required.

$$data\_region \geq credit\_region \ \geq 1 \ slot \ per \ process \tag{5.2}$$

---

[4]The same concerns mentioned before for the data region also apply to the credit region when its size is not dividable by the number of senders.

A couple of considerations must be done in order to completely understand the way the threshold works. First, notice that the use of a threshold at the receiver in order to trigger the process of sending back a credit packet with credit information to the sender means that the receiver will *"only and only if"* send such credit packet once the amount of freed slots accumulated reaches the established limit. This may seem to be deadlock-prone, given that a process, when sending an MPI message, may not own all the required credits before starting the transmission. Therefore, one may think that given that the source process would not send the MPI message to the receiver, then the latter would not retrieve any packet from its mailbox and thus would not reach the threshold, therefore not returning to the sender the credits it requires for sending the message and finally a deadlock would occur. However, notice that this is not the way an MPI message is actually sent with the EXTOLL interconnect. When an MPI message is to be sent, it is forwarded at the source node from the MPI layer to the VELO or RMA library, depending on message size. In case the MPI message is shorter or equal than 2KB the VELO library will receive the MPI message, packetize it into VELO packets and forward to the receiver as many VELO packets as credits are available at that time (the entire MPI message will probably not be sent due to lack of credits). Eventually, the receiver will retrieve packets from its mailbox, even if the entire MPI message has not been completely received (it will be reassembled by the upper MPI layer). While retrieving VELO packets, the threshold will be reached at some point and then a credit packet will be sent back, which will allow the sender to resume transmission. As can be seen, no deadlock is possible.

The second consideration to be made is about the formula used to set the threshold (see Equation 5.1). Other formulae may also be thought to be valid, such as:

(a) $threshold = (credit\_quota \; div \; credit\_slots) + 1$

(b) $threshold = (credit\_quota \; div \; credit\_slots)$

However, these two formulae may not properly work. For example, Formula (a) will not work when the value of $credit\_slots$ is equal to one, as shown in Table 5.1. In the case of Formula (b), it will lead to buffer overflow in case the $credit\_quota$ is equal to three and the value of $credit\_slots$ is equal to two, as shown in Table 5.1. This table shows the value that would be assigned to the threshold when using the three different formulae

TABLE 5.1: Example of values assigned to *threshold* depending on the exact formula used

credit_quota	credit_slots	Equation 5.1	Formula (a)	Formula (b)
100	1	51	**101**	100
100	2	34	51	50
100	3	26	34	33
100	4	21	26	25
100	5	17	21	20
60	2	21	31	30
40	2	14	21	20
20	2	7	11	10
10	2	4	6	5
3	2	2	2	**1**

reviewed, Equation 5.1, Formula (a), and Formula (b). Several example combinations of *credit_quota* and *credit_slots* are depicted. Two main conclusions can be derived from this table. First, it can be clearly seen that setting the threshold according to Equation 5.1 translates into a higher credit update frequency. This higher frequency provides more accurate status data to the sender but also generates additional overhead due to the increased control traffic. Therefore, a trade-off exists, which will be later analyzed in depth. Second, and more important, the table shows that both Formulae (a) and (b) do not properly work. In the case of the former, it can be seen that when *credit_quota* and *credit_slots* are respectively set to 100 and 1 slots, the value assigned to the threshold is 101, which will cause a deadlock because it will never be reached, given that the sender only owns 100 credits and the receiver needs to retrieve 101 data packets in order to return a credit packet that would update the credit information at the sender. In the case of Formula (b), it can be seen that when *credit_quota* and *credit_slots* are respectively set to three and two slots, the value assigned to the threshold is one. This means that the receiver will return a credit packet as soon as it extracts a data packet from its mailbox. Therefore, given that the sender is allowed to send three data packets in a row before stalling, the receiver will eventually return three credit packets before the sender retrieves any credit packet from its mailbox. However, the credit region at the sender's mailbox has been configured with two slots and, therefore, a buffer overflow will occur.

Figure 5.2 shows an example of the mailbox partitioning and the static flow control operation. In this example we have chosen a quota of credits equal to 20 and a total
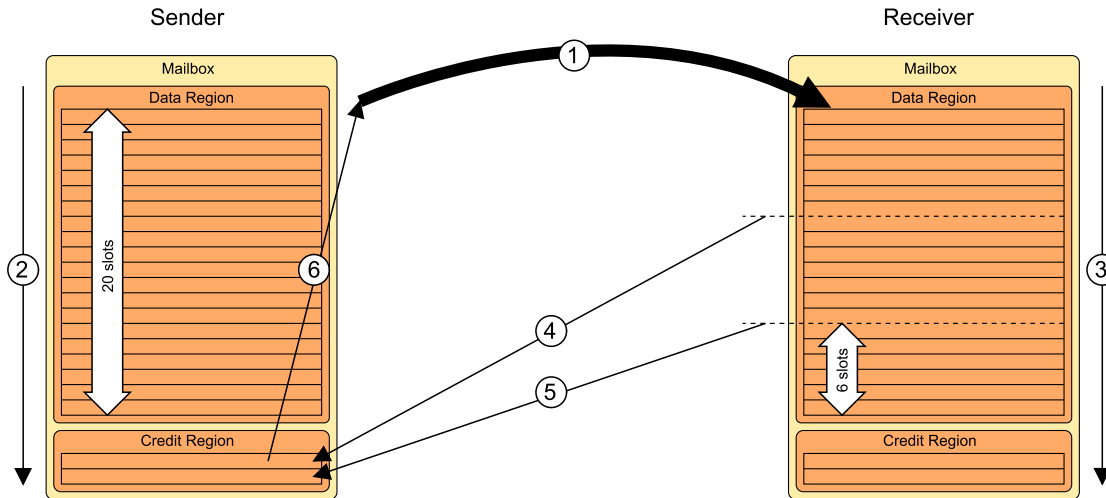
FIGURE 5.2: Mailbox partitioning and flow control operation

amount of credit slots equal to 2. Only the portion of the mailbox owned by the other process depicted in the figure is shown. Both mailboxes would be larger if the data and credit regions allocated for the rest of processes of the application were shown. Applying Equation 5.1 we obtain a threshold for credit return equal to 7 (notice that for the receiver to reach the threshold for the third time ($credit\_slots + 1$), the sender has to free at least one control slot to obtain more credits). Once the initialization stage has finished, the operation of our static flow control is identical to that of the original credit-based one: let's assume that at point 1 the sender starts a data transmission (MPI message) composed of an amount of data packets larger than its credit quota (that is, the number of data packets is larger than 20). Then, the sender process fills the receiver's data region and consumes all its credits. At this moment the sender process must stop sending data and has to wait for new credits: it searches its mailbox (point 2) for an incoming credit packet with credit information (remember that mailboxes are split into data and credit regions, but actually data packets and credit packets will be mixed within the mailbox). On the receiver side, at point 3 the receiver starts retrieving data packets from the mailbox. Every time the receiver frees up a slot in its buffer, the *retrieved_packets* counter is incremented. Every time that the count reaches the threshold value (points 4 and 5), the receiver sends back a credit packet to update the sender *credit* counter and resets its *retrieved_packets* counter. After returning the credit packet at point 5, only six more slots containing VELO packets can be freed at the receiver at that time, as depicted in the figure. This clearly shows how the threshold, as defined by Equation 5.1, is not reached again unless the sender frees at least one credit

slot in its buffer and resumes transmission (remember that the threshold has been set to 7), thus ensuring that a buffer overflow at the credit region will never happen. Once the sender retrieves a credit packet from its mailbox, it owns new credits to resume data transmission (point 6) and communication makes progress.

Finally, in order to fully understand the proposed flow control protocol, it is important to remark that all received VELO packets consume a slot in the mailbox. However, when packets are removed from the mailbox, there is an important difference in the way our flow control manages VELO packets containing MPI data (*data packets*) and VELO packets containing credit information (*credit packets*). In this regard, every time a data packet is retrieved, the *retrieved_packets* counter is incremented. On the contrary, this counter is not updated when a credit packet is extracted from the mailbox. This is because Equation 5.1 defines a threshold for credit return that only applies to data packets. Nevertheless, the way Equation 5.1 defines such a threshold, taking into account the amount of *credit_slots*, indirectly ensures the proper utilization of the credit region, avoiding buffer overflow at both regions, data and credit, as seen in the example depicted in Figure 5.2.

5.3   Setting the static flow control parameters

During the initialization phase of the static flow control the value of one parameter is key: the amount of credit slots allocated in the mailbox (size of credit region). Notice that the larger this parameter is, the more frequently the receiver updates its buffer status (this is good) but also the less space is left at the sender's mailbox to store incoming data packets (this is not good). Therefore, the exact value of this parameter is a trade-off. In this section we analyze how the value of this parameter influences the flow control behavior and how the flow control efficiency strongly depends on this value.

The first test we have performed shows how the distance between sender and receiver affects the behavior of the end-to-end flow control mechanism. For doing so, we leveraged the multi-pingpong test included in the Intel MPI Benchmark Suite [22]. In this test, the 16 processes belonging to one node perform 16 simultaneous pingpong operations with the 16 processes being executed in another node. We have used 2KB data messages as this size is the one that most stresses the flow control protocol (remember that packets
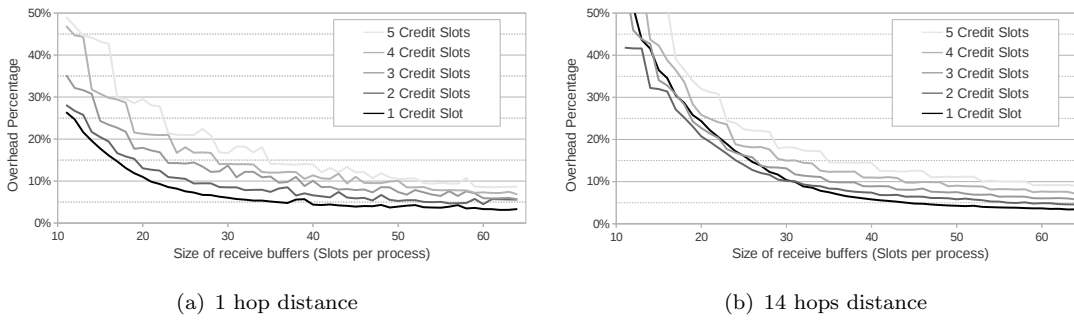
(a) 1 hop distance

(b) 14 hops distance

FIGURE 5.3: Overhead generated by the static flow control in the multi pingpong test

(a) 1 hop distance
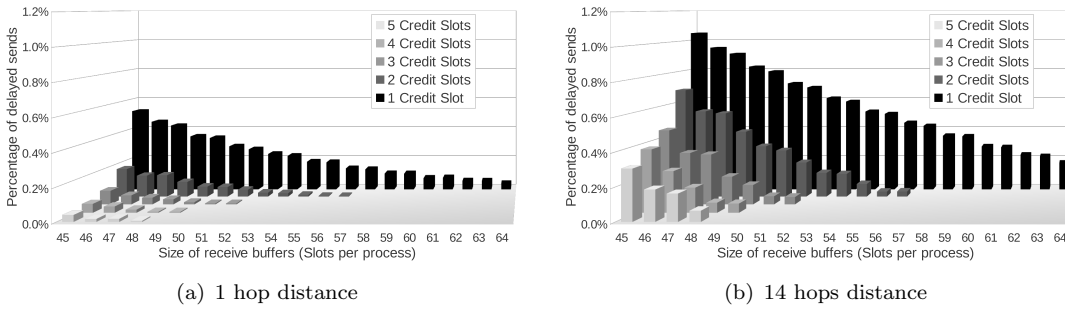
(b) 14 hops distance

FIGURE 5.4: Percentage of delayed sends by the static flow control during the execution of the multi pingpong test

larger than 2KB do not use VELO but leverage the RMA unit, thus not making heavy use of the mailbox).

Moreover, in addition to the tests that leverage the flow control, for comparison purposes we have executed the same tests with very large receive buffers and no flow control, in order to compute the lower bound of the execution time, which will be later used as a reference for the results obtained by our flow control. Notice that disabling the flow control is feasible because of the extremely large buffer size, avoiding buffer overflow even in the worst conditions within this test. This type of executions that do not make use of any flow control will also be performed for most of the later experiments in this thesis in order to provide a reference execution time which will be used to be compared against the execution times when a flow control is used, thus allowing to put all the results into the right perspective.

Figures 5.3 and 5.4 show the main results from this test, which have been carried out for distances between one and fourteen hops (maximum network distance in our EXTOLL cluster configured for this test as an 8x8 2D mesh in order to achieve longer distances than with a 3D mesh).

Figure 5.3 shows the overhead of the execution time due to the flow control for 1 and 14 hops. The reference execution times (without flow control) for these cases are 50.04 and 51.17 μs for 1 and 14 hops, respectively. As we can observe, when the buffer resources are low (buffer size less than 30 slots per process) the overhead for large distances is higher when compared to small distances. However, when buffer size is larger, the behavior is very similar independently of the distance between processes. Figure 5.4 shows the percentage of delayed messages due to the flow control (no credits available). We observe that the percentage of delayed messages increases with distance. However, the break-even points where the application executes without interruptions due to flow control are identical, independent of the distance. Regarding the number of credit slots per process, Figure 5.4 shows that stalls due to flow control decrease when we increment the number of credit slots. However, as it can be seen in Figure 5.3, the overhead generated by flow control is minimal when we use one credit slot. This behavior is very interesting because a reduction in the number of stalls does not reduce the overhead generated. Therefore, the effort for reducing waits (additional traffic) is more expensive than the waits themselves.

The next test is similar to the previous one but in this case we use an alltoall operation for different sizes of groups of processes (256 and 1024 processes[5]). Exchanged data messages have a size of 2KB. Basically, during an alltoall operation, each process will first send a 2KB message to every other process and then it will receive a 2KB message from all the other processes. Notice that the MPI library provides a collective alltoall operation that is optimized for large number of processes in such a way that it creates specific communication patterns among nodes tailored to the exact amount of processes involved in the collective operation as well as optimized for the actual message length to be exchanged. The purpose of this optimized alltoall operation is to reduce the amount of traffic among processes [23]. However, we have disabled this optimization because we want to stress as much as possible the use of receive buffers. As in the previous tests, we have obtained a minimum reference execution time when no flow control is used. These reference execution times have been 52,908.41 μs for 256 processes and 278,515.08 μs for 1024 processes.

---

[5]The experiment with 256 processes is performed using 16 nodes of the cluster, configured in a 4x4 2D mesh, whereas the experiment with 1024 nodes is performed in the 4x4x4 3D mesh configuration involving all the nodes of the cluster.
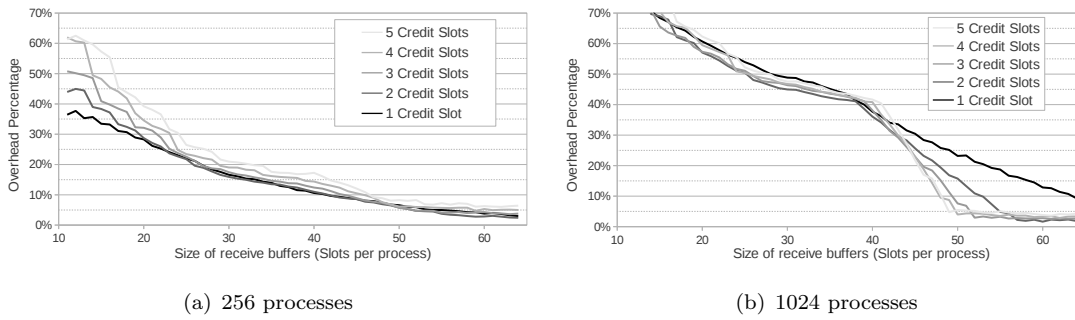
(a) 256 processes   (b) 1024 processes

FIGURE 5.5: Overhead generated by the static flow control in the alltoall test

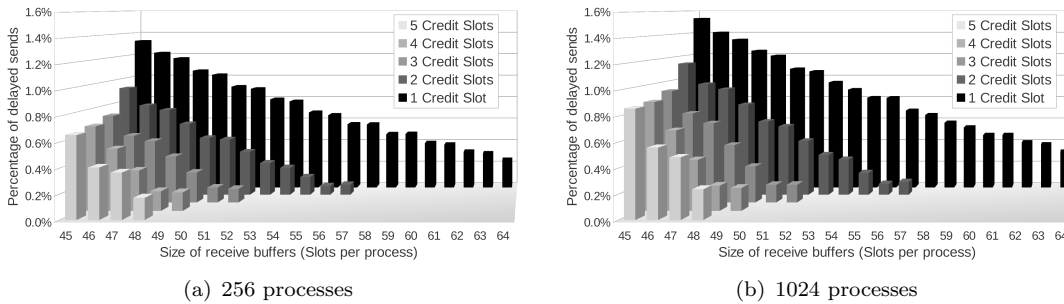(a) 256 processes   (b) 1024 processes

FIGURE 5.6: Percentage of delayed sends by the static flow control during the execution of the alltoall test

Figure 5.5 shows the overhead in the execution time for the alltoall operation (256 and 1024 processes). As can be seen, the overhead increases with the number of processes involved. However, for mailbox sizes close to 50 slots per process, this overhead is very similar in both cases. Furthermore, Figure 5.6 shows the percentage of messages delayed because of flow control. The percentages are very similar independent of the number of processes. Additionally, the break-even points where the alltoall application executes without waits are the same that in the multi-pingpong test.

Notice that figure 5.5(b) shows a huge drop in the overhead between 40 and 50 slots. In order to understand this drop, we first must recall that we are making use of 2KB-long MPI messages in this experiment. Furthermore, notice that the MPI header, which is prepended to the 2KB payload, is 16 bytes long. Therefore, each MPI message will require 37 VELO packets to be sent (remember from Chapter 2 that the length of VELO payload is 56 bytes). With this data in mind, it can be seen in Figure 5.5(b) that the overhead begins decreasing at different points depending on the exact size of the credit region (amount of credit slots used). When using 1 credit slot, the overheard starts decreasing at 38 slots per receive buffer. For credit regions having 2, 3, 4, and 5 slots, the decrement of the overhead starts at 39, 40, 41, and 42 slots per receive buffer,

TABLE 5.2: Minimum size of receive buffer that makes the overhead stable

Credit slots used	Size of receive buffer that stabilizes overhead (slots/process)	Size of data region (slots/process)	Size of credit region (slots/process)
1	Unknown; $\ggg$ 64 slots	?	?
2	57 slots	55 slots	2 slots
3	52 slots	49 slots	3 slots
4	50 slots	46 slots	4 slots
5	49 slots	44 slots	5 slots

respectively. Notice that the receive buffer includes the data region and the credit region. Therefore, it can be derived that, for all the 5 cases, the overhead starts decreasing when a data region with size equal to 37 slots is used. In summary, the huge drop is due to the fact that the sender begins owning enough credits to send the entire MPI message before stalling.

A different issue is the reason why the overhead stabilizes after the huge drop in the way it does in Figure 5.5(b). First, notice from the figure that the size of the receive buffer at which the overhead stabilizes depends, again, on the size of the credit region. In this way, Figure 5.5(b) provides the information summarized in Table 5.2.

With the information in Table 5.2 in mind we can begin our analysis of the reason why the overhead stabilizes at these values. Notice that the main idea behind the way the threshold is defined by Equation 5.1 is that the receiver will return back to the sender an amount of credits equal to, or slightly higher than, the *credit_quota* every *credit_slots*+1 credit packets (this can be directly derived from Equation 5.1). That is, we may see the *credit_quota* as split into (*credit_slots*+1) parts. In this way, the credits associated with each of these parts will be returned by one of the credit packets. Therefore, in a steady communication flow where the receiver is retrieving VELO packets from its mailbox while the sender keeps forwarding data to it, the sender will approximately own at every moment an amount of credits equivalent to *credit_slots* parts of the *credit_quota*, whereas the credits associated with the remaining part will be returned to the sender as soon as the threshold is reached in the receiver. In this situation, the overhead will be stable as far as the sum of the credits associated with the *credit_slots* parts is at least 37 (the amount of VELO packets required to transmit a 2KB MPI message). That is, as far as the sender owns at least 37 credits at every moment, the overhead will stabilize

because the sender will not stall. This can be observed in the break-even points in Figures 5.4 and 5.6. This reasoning can be summarized with the following equation:

$$credit\_quota - (threshold - 1) \geq 37 \tag{5.3}$$

which translates into the following equation when replacing *threshold* with its definition provided in Equation 5.1:

$$credit\_quota - ((credit\_quota \ div \ (credit\_slots + 1)) + 1 - 1) \geq 37 \tag{5.4}$$

Operating and simplifying, and taking also into account the credit slots, we get that the amount of slots per process that stabilizes the overhead is:

$$receive\_buf \geq ((37 * (credit\_slots + 1)) \ div \ credit\_slots) + credit\_slots \tag{5.5}$$

It can be checked that the points mentioned above at which the overhead gets stable follow the previous equation.

Once analyzed in detail the reasons for the huge drop in Figure 5.5(b) we can resume the coarse-grain analysis of the results presented in this section. In this regard, the study of the influence of the number of credit slots with the alltoall operation allows us to see a different behavior from multi-pingpong, as in the alltoall case the reduction of the waits provides a reduction in the overhead time. This is because in this test the use of receive buffers is very high and the time spent in the search for returned credits is large when the mailbox is full.

Finally, after the analyses in this section we can set the value of the *credit_slots* parameter so that the flow control works efficiently. Notice that we have analyzed two extreme communication patterns: one with a very localized communication (multi-pingpong), and another with a very balanced communication (alltoall). Although more cases should be investigated, such as real applications, given that the behavior of the static flow control is very similar in the two extreme communication patters, it will be, presumably, also similar in the intermediate traffic patterns, which are expected to present a network load and buffering pressure in some intermediate point between the two ends analyzed in this section. Therefore, according to the obtained results, two credit slots per process is
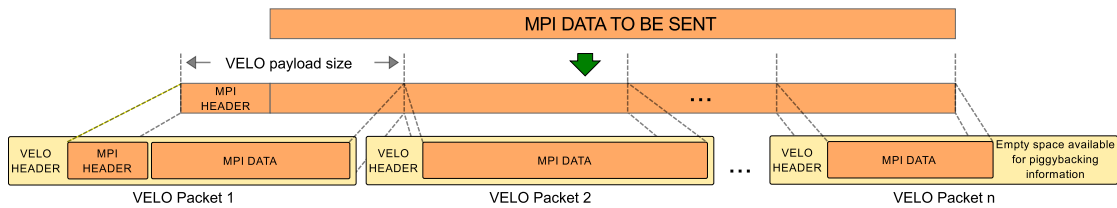
FIGURE 5.7: MPI transmission with piggybacking information

the best trade-off as it provides the best results for alltoall whereas in the multi-pingpong test this value provides results close to the best option (one credit slot). Regarding the size of receive buffers, 57 slots is the best choice, as this size provides very low overhead in all cases.

5.4 Using piggybacking

In the previous sections we have only considered the use of explicit credit packets to update the status of receive buffers. However, there are also other approaches, such as the use of piggybacking. This technique uses data messages to convey credit information, thus saving explicit control messages and therefore reducing the extra traffic generated by the flow control mechanism.

We may use this technique simply by adding a small field in the header of VELO packets, which would contain the credit information. Unfortunately, when VELO was designed, this was not taken into account, and there is no space left in the VELO header to store this information. Another approach would be to add the piggybacking information in the message header of the upper level software layer (MPI, GASNet, etc). However, with this approach the flow control mechanism would not be completely enclosed within VELO, thus not being independent of the upper layers. To solve this problem, we have decided to integrate the piggybacking information into VELO packets that are not completely full, so that the VELO payload is not reduced.

Figure 5.7 shows this idea. The upper part of Figure 5.7 displays the data generated by an application to be sent by using an MPI message. This data is then encapsulated into an MPI message by prepending the required MPI header (center part of Figure 5.7). Finally, the MPI message is packetized into multiple VELO packets. Thus, the last VELO packet is not usually completely filled and can hold piggybacking information[6].

---

[6]We have tried higher credit-status updating frequencies leveraging other policies, but the higher update frequency did not report additional benefits.
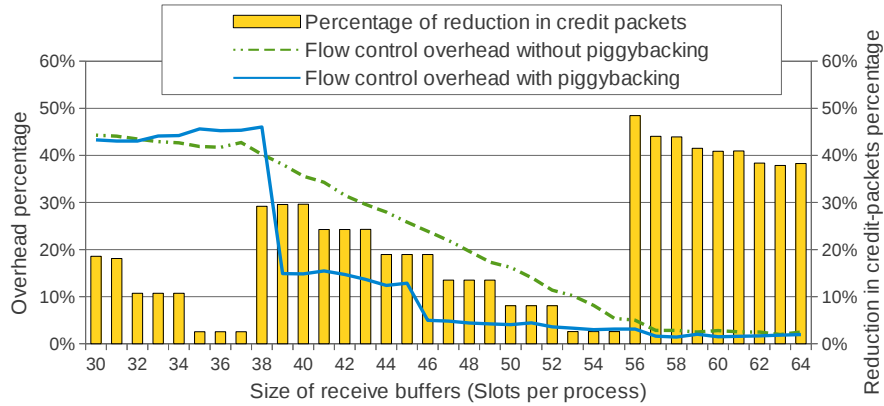
FIGURE 5.8: Alltoall operation with piggybacking

An important issue is how to efficiently signal whether a VELO packet carries piggy-backed credit information or not. This is accomplished by making use of a spare bit that was still available in the VELO header. If this bit is set, then the packet contains credit information, which will be located just after the payload, at the last two bytes of the packet. Thus, the amount of credits that may be returned with a single piggybacked packet is large enough for any application and traffic pattern.

The use of piggybacking offers substantial improvements when the communication pattern is symmetrical and the size of messages allows the inclusion of piggybacking information. In Figure 5.8 we can see the benefits obtained by using piggybacking for an alltoall operation with 1024 processes and a message size of 2044 bytes. In this figure we see a clear reduction of both the overhead and the number of explicit credit packets required by the flow control mechanism. Notice the ramp pattern in the percentage of reduction in credit packets, which is due to a correlation among the receive buffer size, the amount of credit slots (credit-return threshold) and message size. Also, note that we have chosen a message size slightly smaller than 2KB. This is because for the 2KB messages used in the previous tests, the VELO packetization does not provide any space to include the piggybacking information. It is expected that MPI messages from real applications, when packetized, allow for some empty space for the credit information in most cases. Finally, notice that the use of piggybacking has not been taken into account when setting the static flow control parameters because this technique cannot be used always.

5.5   Concerns about the static flow control

The static flow control presented in this chapter has a serious concern. A static partitioning of the mailbox is well suited when the communication pattern of the parallel application is balanced, i.e. when all processes exchange data messages directly with all other processes, as in the previous alltoall test. In this case, all the memory resources devoted to buffering are uniformly used. Unfortunately, parallel applications do not always follow this communication pattern. Many times a given process only exchanges messages with a small percentage of the rest of processes. In this situation, a static partitioning of the mailbox is not efficient because the buffer resources not used by a sender cannot be granted to other senders requiring them, due to the static nature of the partitioning. In other words, all processes are allocated the same portion of the mailbox, independently of their activity, causing that processes with a lot of activity cannot use the resources that other processes are wasting.

This situation is shown in Figure 5.9, where the overhead of the flow control protocol is depicted for four different communication patterns corresponding to the execution of four similar MPI applications, all of them involving 1024 processes[7]:

1. **Application 1**: 1024 processes execute an alltoall operation

2. **Application 2**: 512 processes execute an alltoall operation whereas the other 512 processes are waiting in a barrier

3. **Application 3**: 256 processes execute an alltoall operation whereas the other 768 processes are waiting in a barrier

4. **Application 4**: 128 processes execute an alltoall operation whereas the other 896 processes are waiting in a barrier

In all cases the overhead is not acceptable until the size of receive buffers is larger than 55 slots. However, for the three last communication patterns the overhead could be much lower for smaller receive buffers if buffers at the receivers not used by inactive senders could be allocated to the active ones. For example, in the case of application

---

[7]The alltoall operation within each application has been iterated several times in order to get stable average execution times. Furthermore the collective optimizations have been disabled.
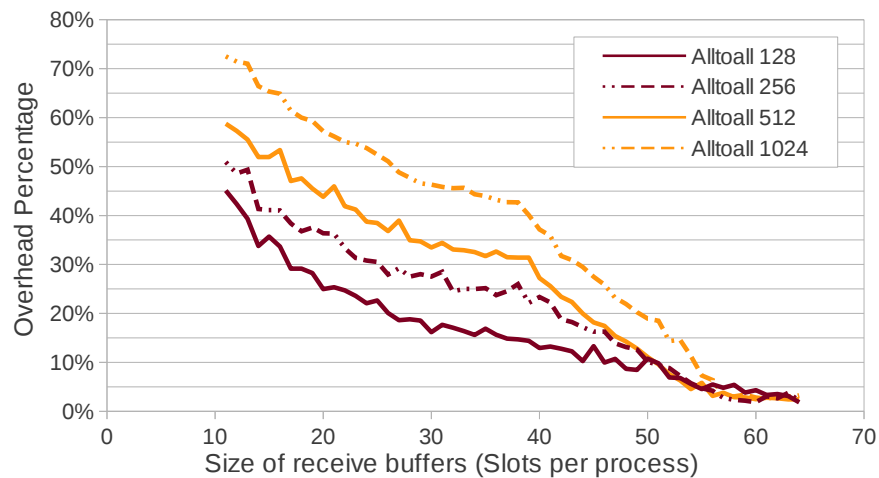
FIGURE 5.9: Overhead due to static flow control for four different communication patterns

4 involving 128 processes in the alltoall collective operation, a given process inside the alltoall operation only communicates with 127 processes, leaving the credit quota of 896 processes unused. This means that 87.5% of the buffer resources are wasted whereas the active processes would potentially benefit from these resources.

In the next chapter we introduce a dynamic flow control mechanism aimed to solve this concern.

Chapter 6

Dynamic Credit-Based Flow Control

Results shown in Figure 5.9 clearly point out the lack of flexibility of the static flow control, which causes inefficiency thus reducing performance.

In this chapter we present a new dynamic flow control protocol able to adapt the buffer resource allocation according to the actual communication pattern, thus assigning more buffer slots to those processes requiring them. Our dynamic flow control continuously monitors the activity of each sender and updates its quota of credits according to its needs. In the following we present the new flow control, highlighting its most important aspects.

6.1 New types of control packets in the dynamic flow control

The new dynamic flow control is an extension of the static version presented in the previous chapter and therefore it includes new types of control packets. Thus, in addition to the credit packets already revisited in the previous chapter, two new control packets are introduced:

1. **Compulsory credit return request**. This type of control packet will be used by receivers to force senders with low or null activity, from the receiver point of

view, to forward the credits that they are not using. Notice that receivers will not ask senders for returning a specific amount of credits but for returning the excess of credits, whatever is that number.

2. **Compulsory credit return response**. This type of packet is used by senders as a response to the previous packet type and will contain the amount of returned credits. Notice that every compulsory request packet must have its associated compulsory return response. Therefore, in case the sender does not actually own any excess of credits, it must still answer the request by including a zero value in its response.

An important concern about these two new control packets is that, contrary to credit packets, they require the availability of credits to be sent. Remember from the previous chapter that Equation 5.1 ensured that, as soon as the threshold is reached, a credit packet could be sent back to the sender without causing a buffer overflow. Unfortunately, given that the decision of sending any of the two new control packets may happen at any time, applying them the same policy used for credit packets may lead to overflowing the region devoted to packets carrying credit information. This overflowing may be avoided if this region was enlarged with two additional slots to host the compulsory request or response[1]. However, these additional slots would be seldom used, thus wasting buffer resources. Therefore, an easy way to ensure that no overflow nor waste of resources happens is to manage these new packets in the same way as data packets by including them into the credit rules. Managing these new packets in this way does not represent any constraint in practice because in case the source of the packet (either request or response) does not own the required credit to send it, then the packet will be enqueued (see Section 6.2). Enqueued control packets are given a higher priority over regular data packets. Therefore, as soon as there is an available credit, the control packet will be flushed.

6.2 Data structures used by the dynamic flow control

Given that the dynamic flow control algorithm is noticeably more complex than the static one, it requires to be supported by a larger amount of data structures, mostly

---

[1]Given the nature of these new control packets and the use that the dynamic flow control protocol makes of them (explained later), just two additional slots would be enough.

located in the receiver side, although the sender also needs additional support.

At the sender side, there will be an array of *credit* counters containing the available credits for all the possible receivers. Notice that this list was already present in the static flow control. Furthermore, the sender will own for each receiver a couple of flags signaling the existence of control packets that could not be sent due to lack of credits. One of these flags will be used for the compulsory credit return requests (*compulsory request flag*) whereas the other will be devoted to the compulsory credit return responses (*compulsory response flag*).

At the receiver side, there will be a single counter which will track the amount of available slots in the data region of the mailbox (*available_slots* counter). Additionally, for each sender, the receiver will have the following:

- A FIFO list of thresholds composed of *credit_slots+1* elements. Every time one element of the list is extracted, another new one will be inserted. We will refer to the element at the head of the list as *current_threshold*. The purpose of this list is to support the continuous ongoing adjustment of the credits granted to the sender while ensuring that this credit adjustment does not cause a buffer overflow at the receiver.

- A counter with the *intended* credit quota. This counter represents the amount of credits the sender should have in the near future, but may differ from the real amount of credits it currently has. This counter is dynamically adjusted as it will be shown in the following sections. Notice that the sum of the *intended* credit quota counters for all senders is equal to the size of the data region.

- A counter tracking the amount of packets from this sender that have been retrieved from the mailbox (*retrieved_packets* counter). This counter was already present in the static flow control and was used there to trigger the creation of a credit packet. In the dynamic flow control the value of this counter will be compared against the *current_threshold* (the head of the FIFO list of thresholds) in order to create a credit packet. However, the exact amount of credits returned will be dynamically computed (increased or decreased) according to the sender's activity, as it will be shown later.

- In case the piggybacking mechanism is being used, the receiver will own a counter accumulating the amount of piggybacked credits. This counter will be referred to as *piggybacked_credits* counter.

- A counter tracking the actual amount of credits currently granted to the sender (*current_credits* counter). This counter will be equal to the amount of credits the sender owns plus the packets in flight to the receiver.

- Blocked flag. This flag will be set during the time interval required to readjust the credits at the sender. This interval will start when a compulsory credit return request is sent to the sender and will end with the reception of the compulsory credit return response. The purposes of this flag are to avoid sending a second request to senders that have already been asked for returning their excess of credits and also to keep those senders with the minimum amount of credits.

Finally, the receiver will have four different lists that will allow to classify senders according to their activity. Three lists will contain the identifiers of senders presenting high, medium, or low activities whereas the fourth list will hold the identifiers of senders with null activity. That is, those processes whose *intended* credit quota has reached the minimum after suffering several reductions. More details about this classification will be provided later. A given sender can only appear in one of these lists. Therefore, the sum of the lengths of the four lists will be equal to the amount of senders.

6.3 Flow control initialization

At the initialization stage, a credit region in the mailbox is reserved exactly in the same way as in the static flow control. The rest of the mailbox slots, which were the data region in the static approach, are split into two parts: the *static region*, with the minimum amount of slots established by Equation 5.2, and the *dynamic region*, that will contain the rest of the data region, typically a quite large amount of slots. The static region will be distributed among the senders during initialization, representing the minimum credit quota a given sender can own at any time during the execution of the parallel application. From now on we will refer to this minimum amount as *static_slots*. This minimum credit quota will be, according to Equation 5.2, equal to the

size of the credit region. The dynamic region will be assigned to senders later during application execution and will be used to increase the credit count of senders presenting higher activity. Notice that if the dynamic flow control protocol was configured with no dynamic region, then it would provide the same behavior than the static flow control configured with identical buffer resources.

Once the dynamic flow control algorithm has been initialized, we have the following configuration:

- Sender side: all credit counters of each receiver are initialized with the same amount of credits, which is equal to the amount of *credit_slots* (*static_slots*). Furthermore, the compulsory request and response flags will be unset.

- Receiver side: the *available_slots* counter will be initialized to the size of the dynamic region. Additionally, all the senders will be stored in the low activity list, ordered according to their identifier in increasing order. Finally, the individual data structures for each sender will be initialized as follows:

  - The FIFO list of thresholds will have all of its components initialized to one.

  - The *intended* credit quota counter will be initialized with the amount of credits obtained by leveraging a static partition of the entire data region[2] as explained in Chapter 5.

  - The *retrieved_packets* counter will be initialized to zero.

  - The *piggybacked_credits* counter will be equal to zero.

  - The *current_credits* counter will be initialized to the amount of credit slots (*static_slots*).

  - The blocked flag will be unset.

6.4 Overview of the dynamic flow control

The basic behavior of the dynamic flow control is similar to the operation of the static one except that the former saves a large portion of the mailbox (dynamic region) which

---

[2]Notice that although the *intended* credit quota comprises the dynamic and static regions, the actual amount of credits granted to senders at this point (*current_credits* counter) is only the static part. This is an example of *current_credits* and *intended* quota counters being different.

can be assigned, during the execution of the application, to those senders presenting higher activity. Furthermore, once the dynamic region has been exhausted, credits from senders with lower activity will revert to senders with higher activity levels. Notice that moving credits from some senders to others must be carried out in a way that the former do not overflow their new credit count. This is achieved by reducing the rate of credits returned and by sending compulsory credit return requests. Therefore, the dynamic flow control follows the same basic principles of the static one, being the main differences among them:

- A monitoring function incorporated into the receivers that tracks the activity of their respective senders.

- Management of the new control packets (compulsory credit return requests and responses).

- A more complex management of the piggybacking mechanism.

In the dynamic flow control, when a receiver is to return back credits to a given sender, the relative activity level of that sender, with respect to the receiver, is taken into account in order to increase or decrease its quota of credits. The main idea is to reduce the quota of credits for those senders presenting lower activity while increasing the amount of credits granted to more active senders.

The activity of a given sender is defined in terms of the time it requires to consume all its credits. This event (*monitoring point*) is easy to be detected by a receiver because, as it can be derived from Equation 5.1, a sender will entirely consume its credit quota approximately after crossing *n+1* times the threshold for returning credits[3], being n the number of credit slots in use.

At each monitoring point for a given sender, the receiver detects which of its senders have not spent their credit quota and it will choose the one with the oldest activity (*victim*) to steal credits from. These credits will be used to increment the credit quota

---

[3]In the case of the static flow control it can be ensured that the sender will consume its credit quota after crossing *n+1* times the threshold. However, in the dynamic flow control the threshold (FIFO list) varies in order to adapt the buffer space granted to the sender according to its activity. Furthermore, its quota of credits also varies according to that activity. Therefore, in the dynamic flow control it cannot be ensured that the sender will exactly consume its credit quota at the monitoring point. However, it can be ensured that it has consumed all the credits it received since the last monitoring point.

of the monitored sender in the midterm and also its *intended* credit quota at this point. This also applies to the credit and *intended* credit quotas of the victim process. The amount of credits exchanged between both senders will be:

$$cred = max(credit\_slots + 1, abs(cq\_mon\_sdr - cq\_vic\_sdr) \; div \; 2) \qquad (6.1)$$

being *credit_slots* the amount of credit slots set during the initialization stage and *cq_mon_sdr* and *cq_vic_sdr* the *intended* credit quotas of the monitored and victim senders. Remember that "div" stands for the integer division. Equation 6.1 shows that when the monitored and victim senders have a similar credit quota, the amount of stolen credits is low but when this difference is larger, the amount of credits exchanged increases too. It should be remarked that the victim process will always keep at least a credit quota equal to the *static_slots*, which cannot be stolen. Therefore, if the victim process, after being stolen the amount of credits defined by Equation 6.1, would keep less credits than *static_slots*, then the amount of credits to steal should be reduced to preserve that region.

In summary, the goal of this flow control is reaching a balanced point in the system so that all the processes consume their respective credit quotas in the same amount of time, independent of the exact size of that quota. This helps keeping all the application processes synchronized, making progress at the same pace and therefore avoiding unnecessary application stalls.

6.5   Detailed operation of the dynamic flow control

The dynamic flow control operation is similar to that of the static version, although in this case the activity of senders is monitored and their credit quota is dynamically updated. In this way, credit control and threshold computation are performed at execution time, given that the credit quota evolves during application execution, too.

Figure 6.1 shows the flow diagram of the dynamic flow control. The starting point is the retrieval of a VELO packet from the mailbox. After the retrieval, the *slot management* and *activity control* modules are called in order to adjust counters and thresholds. Next, it must be checked whether a control or a data packet has been retrieved.
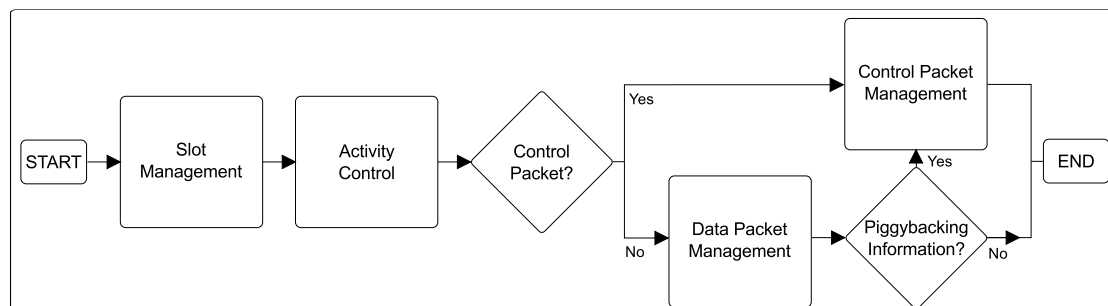
FIGURE 6.1: Flow diagram showing how the dynamic flow control works

In case a data packet has been retrieved, it is forwarded to the *data packet management* module. On the contrary, if the packet retrieved from the mailbox is a control packet, then it is forwarded to the *control packet management* module. This module is also called in case the data packet carries piggybacked credit information.

In the following sections we will describe the different modules depicted in Figure 6.1.

6.5.1 Slot management module

This small module is in charge of adjusting the *available_slots* counter of the receiver and the *retrieved_packets* counter associated with the sender of the packet. The *current_credits* counter is also updated. Notice that these actions should only be performed in case the retrieved packet is a data packet or a compulsory request or response for credit return. In case a credit packet was retrieved from the mailbox no counter update is required.

6.5.2 Activity control module

This is probably the most important module in the new dynamic flow control protocol. This module computes the amount of credits to return, which will mainly depend on the sender's activity and the amount of available slots in the mailbox (*available_slots* counter).

The first action to perform in this module is to check whether the threshold has been reached. If not, the module ends. On the contrary, two different workflows may exist: the *blocked* and the *unblocked* flows.
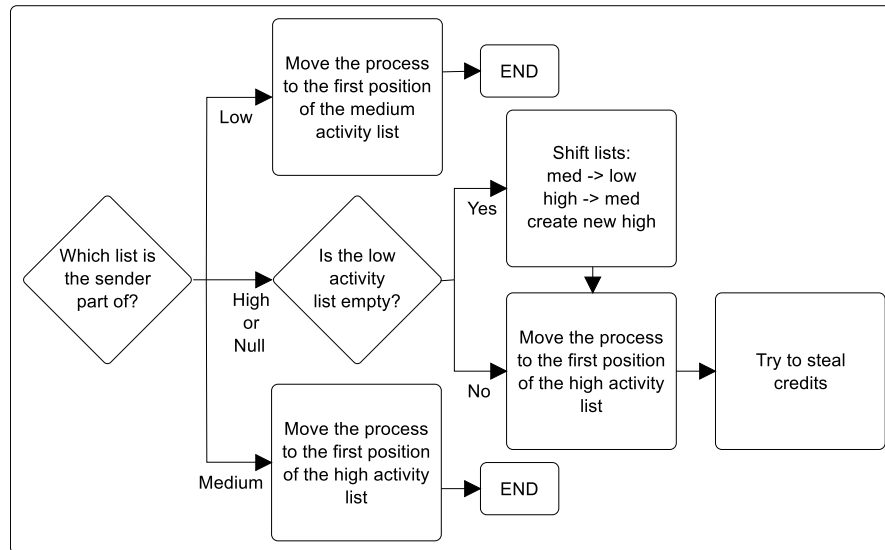
FIGURE 6.2: Flow diagram of the whitelisted flow of the activity control module

The blocked workflow takes place when the sender of the VELO packet retrieved from the mailbox has its blocked flag set. Recall from Section 6.2 that this flag is set for those processes that are in the process of having their credit quota adjusted and, therefore, this workflow will happen with low probability. In this flow, the only action to be performed is to check the actual amount of credits owned by this sender (*current_credits*). If the sender owns an amount of credits equal to or larger than *static_slots*, then no credit will be returned to the sender. On the contrary, one credit will be returned.

Contrary to the blocked workflow, the unblocked flow will happen when the sender of the retrieved packet has its blocked flag unset, thus making this the common case. In this flow two different cases may be faced: the sender has reached a monitoring point or not.

Whenever a sender reaches a monitoring point (see Figure 6.2), it is moved to the first position of the list with the next higher activity level, unless that sender is already in the highest activity level list. In this case, the low activity level list is examined by the receiver. If this list is empty, then the three activity lists are shifted: the high activity list becomes the medium activity list, which in turn becomes the low activity list. A new high activity list is created[4]. Next, the monitored sender identifier is moved to the first position of the new high level activity list and finally we try to steal credits from a low

---

[4]An efficient implementation of this shifting process would not destroy and create a new list, but would just make use of pointers in order to save time. Nevertheless, in this discussion we will avoid implementation issues for the sake of simplicity.
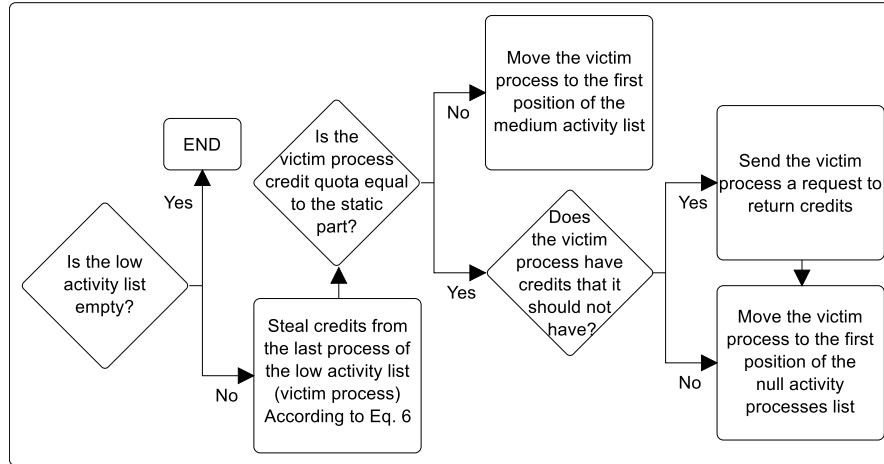
FIGURE 6.3: Flow diagram of the operation for stealing credits

activity process. Notice that stealing credits is always performed between a process in the high activity list and a process in the low activity list. This will prevent processes to continuously steal credits from each other in a balanced communication pattern, given that in such scenario all the processes will keep grouped in two consecutive activity lists.

Figure 6.3 shows the flow diagram of the credit steal operation. If the low level activity list is not empty, the process identifier in the last position of this list is selected as victim and then we apply Equation 6.1 to steal credits, with the restriction of not being allowed to steal any of its static credits. Once the credit exchange has been made, we check the *intended* credit quota of the victim sender. Two possibilities may happen at this point. First, if this is equal to its static part, the victim sender will not be able to provide more credits. For this reason it is moved to the list of null activity processes. Before moving the process we check the current amount of (real) credits that the victim sender has (*current_credits*). If this amount is greater than its *intended* credit quota, we send it a compulsory request to return the excess of credits and will set its blocked flag. On the contrary, the second possibility that may happen is that its *intended* credit quota is still higher than the static part (still has dynamic credits). In this case it is moved to the first position of the medium activity level list. This change to the medium activity list is intended to avoid stealing credits again from this process the next time credits are required. This ensures that all processes in the low activity list contribute with their unused credits. Another possibility would be to move the victim process to the first position of the low activity list. However, moving it to the medium activity list makes the victim process to mix with the other processes, thus providing better results.

Finally, notice that in the flowchart of Figure 6.2, the null activity list also appears, and a sender which is in this list is managed in the same way as if it was in the high activity list. This is because when a process that only has static credits reverts to activity, it needs to quickly recover credits.

Independently of whether the sender did reach a monitoring point or not, the *current_threshold* is removed from the FIFO list of thresholds and a new one is computed according to Equation 5.1. For this computation, the *intended* credit quota is used as *credit_quota* in the equation. If the *available_slots* counter contains enough free slots, this amount of slots is used to send a credit packet back to the sender. Otherwise, the credit packet will contain the value of the *available_slots* counter (at least 1). Finally, the new threshold just computed is inserted into the list of thresholds (if piggybacking is being used, the new threshold will be adjusted according to the discussion in Section 6.6), appropriately updating all the involved counters.

6.5.3   Data packet management module

This is a very small module, which basically takes care of forwarding the packet payload to the upper software layer (the bottom layer of the MPI software). In case the packet carries piggybacked credit information, it is extracted from the end of the packet payload prior to forwarding the data to the upper MPI software and the credit information is appropriately processed by the control packet management module as if a credit packet was received.

6.5.4   Control packet management module

This modules takes care of the three types of control packets used in the dynamic flow control. It is also in charge of handling the credit information extracted from data packets carrying piggybacked credits. The actions to be performed for these possibilities are the following:

- **Credit packet**. In case this type of control packet was retrieved from the mailbox, then the credit information contained in it is used to update the counter holding the amount of credits this process owns to send data to the source of the credit

packet just retrieved. Once credit information has been updated, the state of the compulsory request and response flags must be checked just in case there is any packet waiting. Notice that first the request flag is analyzed, and then the response flag. In case only one credit has been received, and both flags are set, then flushing the request has higher priority.

- **Credit information got from data packets**. The same actions as in the previous case are performed.

- **Compulsory credit return request**. This control packet forces the receiver to return the credits that is not using. Therefore, a compulsory credit return response will be sent with an amount of credits equal to $max(n - static\_slots, 0)$ being n the *credit* counter value (remember that the credit quota granted to a given process will always be at least equal to the *static_slots*). Additionally, remember that compulsory responses require one credit to be sent and, therefore, the final credit count of this process will be $static\_slots - 1$.

- **Compulsory credit return response**. After retrieval of this packet, the *available_slots* counter and the *current_credits* counter are appropriately updated according to the amount of credits returned. Also, the blocked flag is unset.

6.6   Considerations with piggybacking

The use of piggybacking has not been deeply analyzed when presenting the dynamic flow control in the previous sections as its use presents several concerns that must be separately addressed.

The immediate concern to address is that the use of piggybacking means that the return of credits will be likely performed before the threshold is reached. Thus, the amount of credits returned by piggybacking should be considered so that these credits are not returned again. Nevertheless, controlling piggybacked credits is easy as it can be achieved by adding a new counter (*piggybacked_credits* counter, already listed in Section 6.2).

Another more important concern is related with the update of the thresholds for credit return. Addressing this issue is noticeably more complex than the previous one because of the meaning of the list of thresholds. In the following we analyze this issue in detail.

Figure 5.8 showed that the use of the piggybacking mechanism reduces the amount of credit packets generated during the execution of an application. This is due to the fact that the information hold at the *retrieved_packets* counter is appended to data packets if possible, thus resetting this counter afterwards. This process of resetting the *retrieved_packets* counter delays, or even avoids, reaching the current threshold for credit return.

Additionally, in the previous sections it has been discussed how the control of the activity of senders is based on the relative time that they require to entirely consume all their credit quota. It has also been explained that this time is measured by tracking the amount of times that the threshold is reached. Furthermore, every time the current threshold is reached, it is removed from the list of thresholds and a new one is computed. The value of this latter threshold is actually the amount of credits to be returned to the sender.

As can be derived from this discussion, the use of the piggybacking technique noticeably influences the time when thresholds are reached, what directly conflicts with the way that the activity of senders is measured. Therefore, the use of piggybacking along with our dynamic flow control creates two different important problems that need to be solved:

- Properly measuring when the threshold is reached, so that sender activity is correctly monitored. Or, in other words, properly estimating when the threshold would have been reached, considering that the counters that support this event are adjusted every time that some credit information is piggybacked.

- Properly computing the amount of credits to be returned taking into account those that were already returned in piggybacked data packets.

In the following sections we address these two concerns.

6.6.1 Estimating when the threshold would have been reached

When the piggybacking technique is in use along with the dynamic flow control, we should take into account that the current threshold may be reached in two different ways. The first one, that will be referred to as the *regular* way, consists of the procedure

described in the previous sections: every time a slot is freed, the *retrieved_packets* counter is incremented by one and the value of this counter is then compared against the current threshold. As soon as both values are the same, the threshold is reached. In the second way, referred to as *piggybacking* way, every time a data packet is to be piggybacked with credit information, the sum of *retrieved_packets* and *piggybacked_credits* is computed. If this sum is greater than or equal to the current threshold, then the threshold has been reached. At that point, all the actions described in the activity control module are performed. Notice that in the previous sections the activity control was performed when a slot was freed whereas here it is performed when a packet is to be sent. This could be seen as a way of enforcing the monitoring activity.

Finally, notice that in the regular way the threshold will be matched with a precise count (the *retrieved_packets* counter was increased by one at a time) whereas in the piggybacked way the threshold may be overtaken instead of being precisely matched because the count may be increased with a value larger than one each time. This should later be taken into consideration.

6.6.2 Computing the credits to be returned

Once the threshold is reached (or exceeded), the sender should be updated with new credit information. This update will be different depending on the way the threshold was reached.

If the threshold was reached in the regular way, then the value of the *piggybacked_credits* counter will not be considered because no further adjustment is required. In this case the new threshold will be computed and inserted into the FIFO list as already explained in the previous sections. Additionally, the value of the new threshold will be used for the credit packet to be returned as far as this value is lower than or equal to the value of the *available_slots* counter. Next, the *retrieved_packets* counter for this sender as well as its *piggybacked_credits* counter will be reset and the *current_credits* counter updated.

On the contrary, if the threshold was reached in the piggybacking way, two different situations may occur:

- Adjustment by deficit: this situation happens when the computed threshold is equal to or greater than the amount of credits already returned with piggybacking.

In this case, the new computed threshold will be inserted into the threshold list, but only the remaining credits not returned yet are sent back to the sender.

- Adjustment by excess: this situation happens when the new computed threshold is smaller than the amount of credits already returned with piggybacking. In this case, the new computed threshold will be inserted into the threshold list and the new *current_threshold* is increased with the excess and no credits are explicitly returned.

6.7   Partial trace of the dynamic flow control

Figure 6.4 shows a trace of the dynamic flow control mechanism configured with two control slots. On the sender side we can see (left to right) the position of the returned credits in the mailbox (credit packets are shown with shaded background whereas piggybacked packets carrying credits are shown with white background) and the credit counter. On the receiver side we can see (left to right) the threshold list (composed of three elements given that we are using two credit slots per process), the *intended* credit quota of the sender being served, the *retrieved_packets* counter and the *piggybacked_credits* counter.

This trace is split into 8 blocks:

- Block 1: Initial configuration. In this example the intended credit quota has been initialized to 10. The sender sends a packet and when the receiver removes it from the mailbox, the update threshold is reached in a regular way and an explicit credit packet is sent back with 4 credits, according to the computations explained above. The threshold list is also updated.

- Block 2: Again, after receiving a packet, the threshold is reached in a regular way and an explicit credit packet is sent back. However, notice that in this case only 2 credits are returned, instead of 4 credits according to Equation 5.1, because not all 4 credits are available in the *available_slots* counter (not included in the figure).

- Block 3: Similar to block 1. The threshold is reached in a regular way. This time all the required credits are available in the *available_slots* counter. Notice that the sender has removed a control packet with credits from its mailbox.
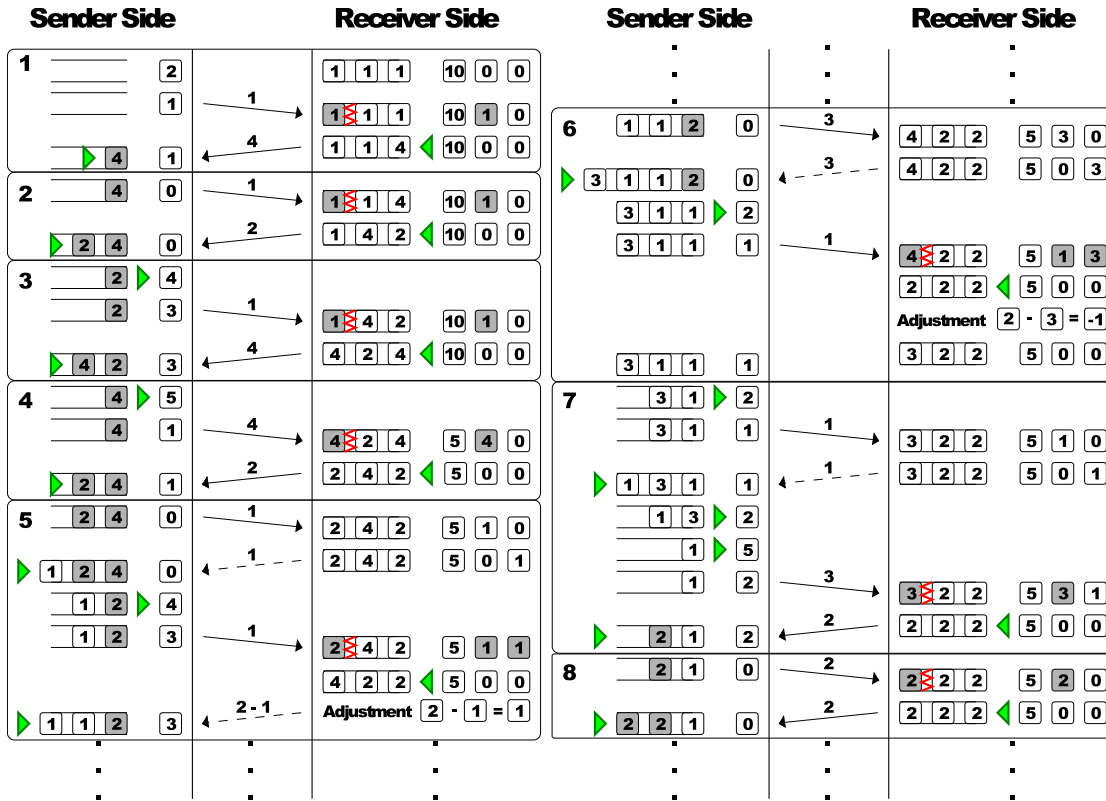
FIGURE 6.4: Trace of the dynamic flow control operation

- Block 4: Similar to block 1. The threshold is reached in a regular way, although in this case the credit quota of the sender has been (dynamically) modified in order to provide more credits to a more active sender. This can be observed in the reduction of the *intended* credit quota counter.

- Block 5: This block shows how the threshold is reached in the piggybacking way. It also shows the use of piggybacking (dashed arrow) to update the credit information at the sender. Notice that in the second piggybacked message, an adjustment by deficit was necessary (two credits should be returned although one credit was already returned and therefore one credit is piggybacked in the data packet).

- Block 6: Threshold update in the piggybacking way where an adjustment by excess is necessary because of previous piggybacked credits. Notice that no credits are returned (two credits should be returned although three credits were already piggybacked and therefore the new *current_threshold* is increased by the difference).

- Block 7: Threshold update in the regular way after an explicit control packet with previous piggybacked credits. An adjustment is not necessary.

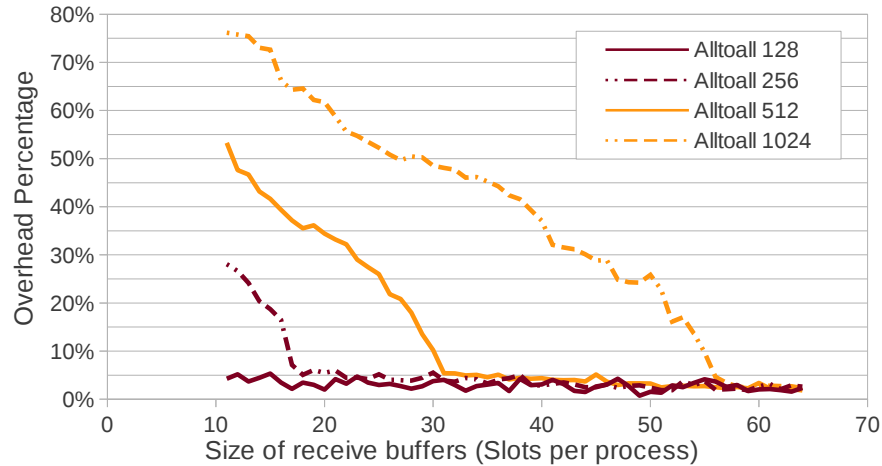- Block 8: Similar to block 1. The threshold is achieved in the regular way.

FIGURE 6.5: Overhead due to dynamic flow control for different communication patterns

6.8 Initial performance evaluation of the dynamic flow control

In this section we present a quick performance evaluation of the new dynamic flow control algorithm. This performance evaluation is the counterpart of the one presented in Figure 5.9 for the static flow control, which presented the overhead of the flow control protocol for four different alltoall communication patterns. Figure 6.5 presents the results for the same experiments as in Figure 5.9 when our dynamic flow control mechanism is leveraged.

As can be seen in the figure (comparing to Figure 5.9), the better use of the buffer resources is evident for the alltoall operations involving 128, 256, and 512 processes, where the overhead is very low even for a small amount of slots per process. This shows that our dynamic flow control is able to adapt the buffer resource allocation according to the actual communication pattern, thus assigning more buffer slots to the processes requiring them. These good results are achieved because our dynamic flow control continuously monitors the activity of each sender and updates their quota of credits according to their needs.

Chapter 7

Experimental Results and Analyses

In this chapter we evaluate the performance of our new end-to-end flow control mechanism. Note that in the following tests the reference execution time has been obtained as in Chapter 5. Moreover, two credit slots per process are used. Therefore, in the following experiments with the dynamic flow control, the size of the static region will be equal to *2*number of processes*, whereas the rest of the mailbox space will be assigned to the dynamic region. In the case of the experiments using the static flow control, the mailbox used will be equal to the sum of the static+dynamic regions of the dynamic flow control protocol. Furthermore, the amount of credit slots will also be the same for both protocols. This will allow to perform a fair comparison between them, given that both protocols will use exactly the same amount of buffer resources.

7.1   The main components of the overhead

Two different components are the sources of overhead: first, the stall time when a process has consumed all its credits as well as the additional traffic generated by the flow control mechanism. Second, the computational overhead due to credit management.

Figures 7.1 and 7.2 show these two components of the overhead, comparing the static and dynamic versions of the flow control for the alltoall and multi-pingpong scenarios. The alltoall test is run for 1024 processes whereas the multi-pingpong is performed
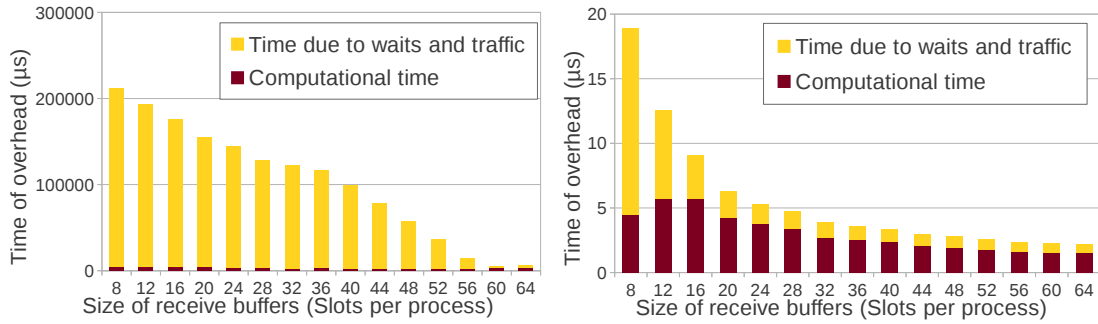
FIGURE 7.1: Overhead in alltoall (left) and multi-pingpong (right) operations, static version
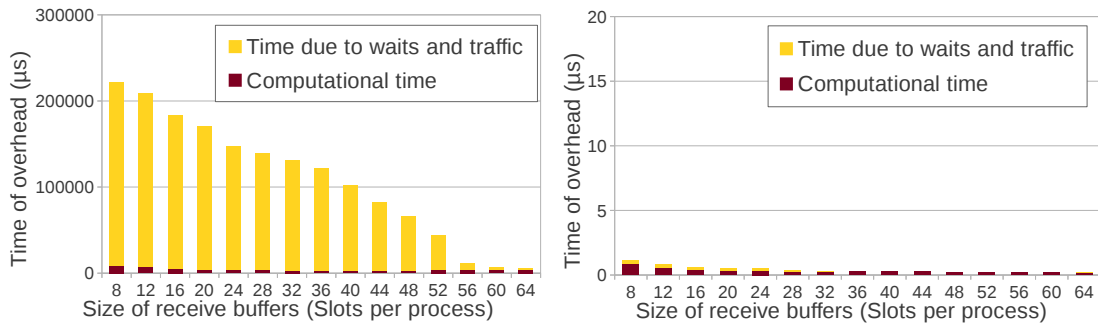

FIGURE 7.2: Overhead in alltoall (left) and multi-pingpong (right) operations, dynamic version

among 16+16 processes on two nodes with a 1-hop distance. Results for alltoall are the average from 150 repetitions of the benchmark while results for the multi-pingpong are the average from 20480 repetitions.

If we compare the results for the alltoall test, the generated overhead is very similar in both cases (static and dynamic) for both the computational component and for the component due to stalls and extra traffic. The main difference can be observed when the size of receive buffers is extremely small (size below 16 slots), as the overhead generated by the dynamic flow control is slightly larger than the static one. This case corresponds to the 1x1024 plot shown in Figures 5.9 and 6.5. Notice that this traffic pattern is perfectly balanced and the distribution of buffer resources provided by the static partitioning of the mailbox is already optimal. But even in this worst scenario for the dynamic flow control it can be seen that the additional overhead is marginal.

On the other hand, the results obtained by the multi-pingpong test show that the total overhead is much lower in the dynamic case than in the static one, confirming the benefits of our approach. It is interesting to notice the larger computational overhead of the static flow control. This larger overhead may seem counter-intuitive, given that the

static protocol only needs a tiny bit of threshold computation at the beginning whereas the dynamic flow control continuously has to update thresholds, manage threshold lists, compute *intended* and *current_credits* counters, etc. However, notice that in the multi-pingpong test a given process will only communicate with another single process. Therefore, when making use of the dynamic flow control, receivers will end granting their entire mailbox to a single sender, causing that the threshold for credit return is reached with a noticeably lower frequency than in the static case. The consequence is that credit packets are created much less often in the dynamic protocol than in the static one, also reducing accordingly the frequency of sending such packets (remember that credit packets are sent making use of VELO packets and therefore a PIO is required). The net consequence is a reduction of the computational part of the overhead.

7.2 Flow control behavior using LiMIC

Chapter 3 showed the three transfer methods used by MPI over EXTOLL: an eager protocol leveraging VELO for message transfers below 2KB, a rendezvous protocol via RMA for bulk transfers, and finally the use of shared memory leveraging LiMIC for intra-node transfers. Notice that processes in the same node communicating by using this shared-memory mechanism will not consume credits. This will provide further performance improvements because the unused slots may be diverted to active communication flows.

Figure 7.3 shows the overhead generated by both flow control mechanisms, static and dynamic, for the alltoall operation with different percentages of destinations accessible via shared memory: 50%, 25%, 12.5%, and 6.25% of the total possible destinations (all-toall among 32, 64, 128, and 256 processes distributed among 2, 4, 8, and 16 nodes, respectively). In the case of 25% or higher, we clearly see how the dynamic mechanism offers a lower overhead than the static one. For the other two cases, the static and dynamic mechanisms produce similar overheads, given that the percentage of the communications using shared memory is very low with respect to the total amount of communications.
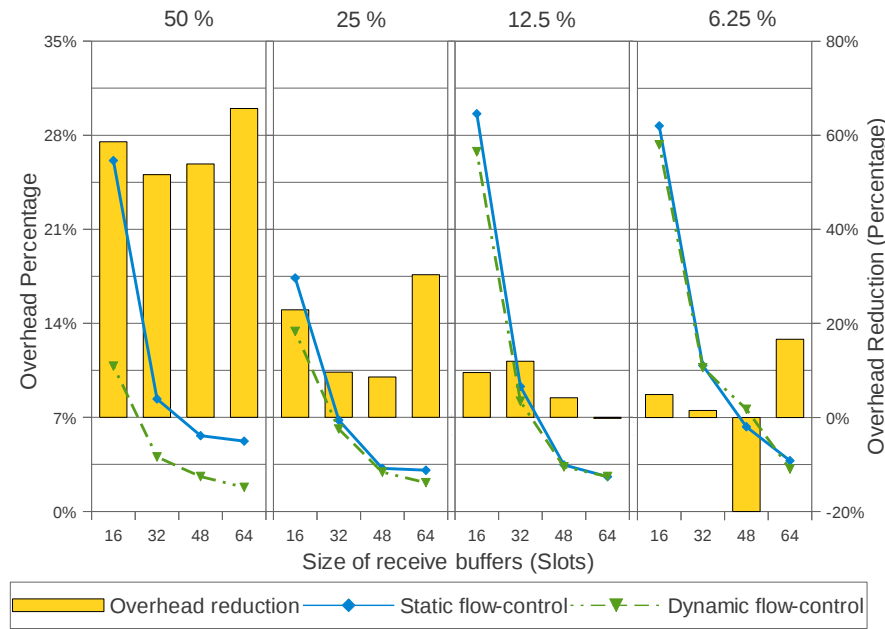
FIGURE 7.3: Overhead of the alltoall operation when leveraging LiMIC

7.3   Credit reallocation after communication changes

In this test we have executed a multi-phase MPI parallel application, that is, an application where the communication pattern changes with time. We analyze how the dynamic flow control adapts to the changes.

In this test the parallel application performs many subsequent alltoall operations involving different number of processes: 100 consecutive alltoall (all 1024 processes), 100 alltoall (ranks 0-255), 100 alltoall (ranks 0-511), 100 alltoall (all 1024 processes), 100 alltoall (ranks 0-511), 100 alltoall (ranks 0-255), and 100 alltoall (all 1024 processes).

This test has been performed with a receive buffer size of 30 slots per sender (30K slots in total per receiver), exchanging 2KB messages in the alltoall operation, and without collective optimizations in the MPI layer. Notice that the receive buffer size of 30 slots per sender has been selected according to the results presented in Figure 6.5. A size equal to 30 slots per sender allows performing alltoall operations involving 512 processes almost without any overhead whereas this size will force receivers to adjust their dynamic partitions when moving from alltoall operations with 256 processes to 1024 processes, which is the purpose of this experiment.
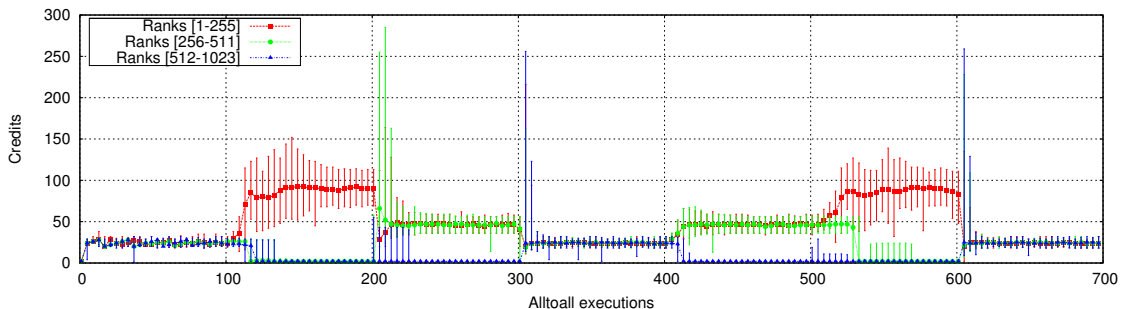
FIGURE 7.4: Credit evolution as communication pattern changes with time

Figure 7.4 shows the average amount of credits that the process with rank 0 owns to send data packets to processes with ranks (0-255], [256-511], and [512-1023]. Each point has bars that show the minimum and maximum values. The point is located at the median value. It can be seen how our dynamic flow control mechanism is able to adapt to the new communication pattern. It is interesting to notice that processes in ranks 512-1023 suddenly gain credits after 200 executions, even so they are not involved before and after the 200 mark in any alltoall. The reason for this behavior is that at the end of each alltoall stage, there is a barrier in the MPI program. This barrier involves all the processes, even if they were not involved in the alltoall operation. As a consequence of the data exchanged during the barrier, those processes that did not present activity during the alltoall operations will, however, send and receive data packets, thus becoming active and therefore seeing some changes in their credit counts.

7.4 Evaluation with the Intel MPI benchmark suite

We have additionally analyzed how the availability of buffer resources affects our flow control mechanism in the context of the Intel MPI benchmarks (IMB). We have executed all benchmarks with 1024 processes and a message size of 2KB. Receive buffer size has been set to 8, 16, 32, and 64 slots per sender, what represent a total of 0.5 GB, 1 GB, 2 GB, and 4 GB, respectively, of memory resources across our 64-node cluster. As in the previous sections, we have obtained a minimum reference execution time for each benchmark without any memory resource limit and without any flow control mechanism, so that our proposal can be put into context. This time the collective optimizations have not been disabled and piggybacking is active again.
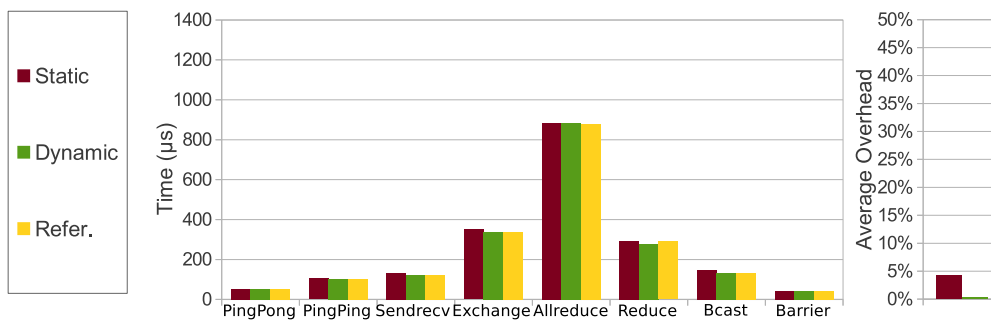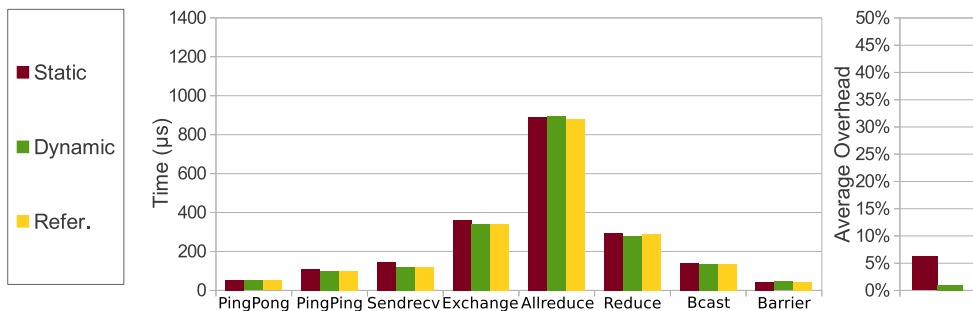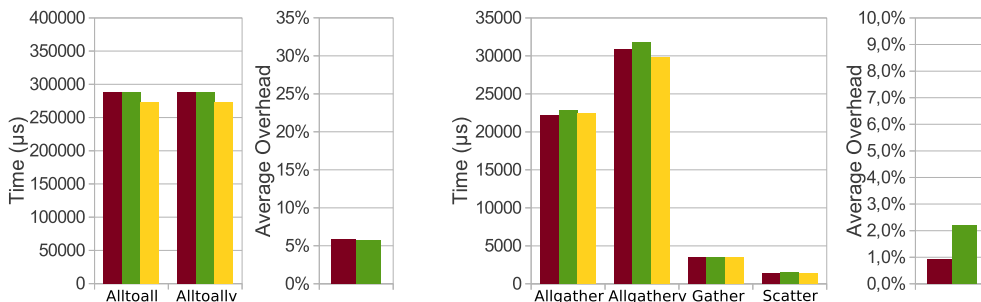
(a) Localized communication pattern
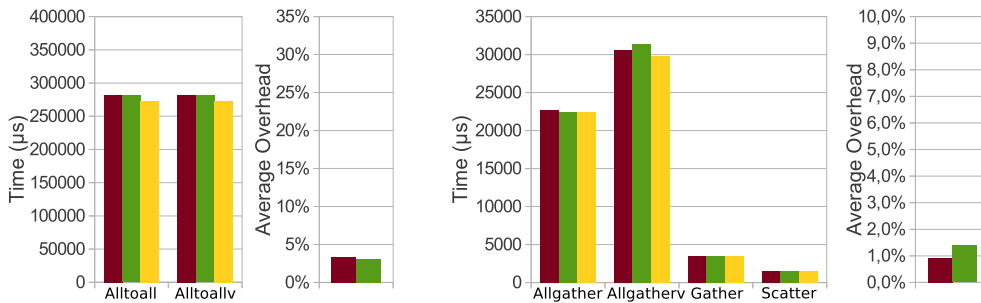
(b) Balanced communication pattern (left) and Communication by RMA (right)

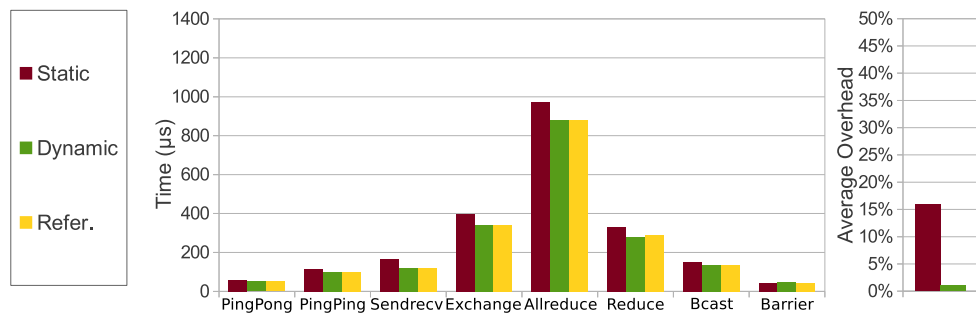FIGURE 7.5: Results for the IMB with 4 GB of buffer resources across the cluster

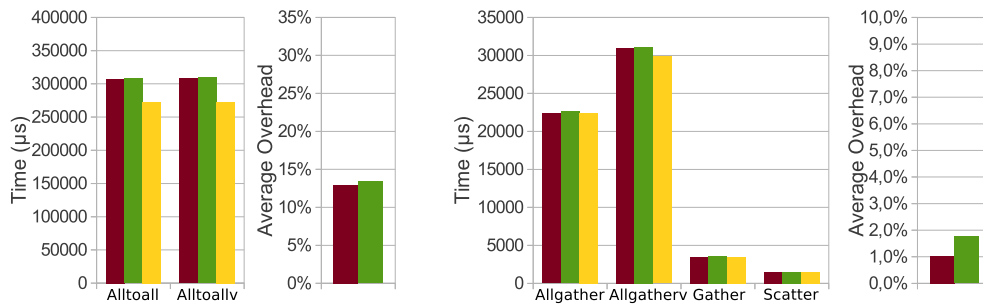(a) Localized communication pattern

(b) Balanced communication pattern (left) and Communication by RMA (right)

FIGURE 7.6: Results for the IMB with 2 GB of buffer resources across the cluster

(a) Localized communication pattern

(b) Balanced communication pattern (left) and Communication by RMA (right)

FIGURE 7.7: Results for the IMB with 1 GB of buffer resources across the cluster
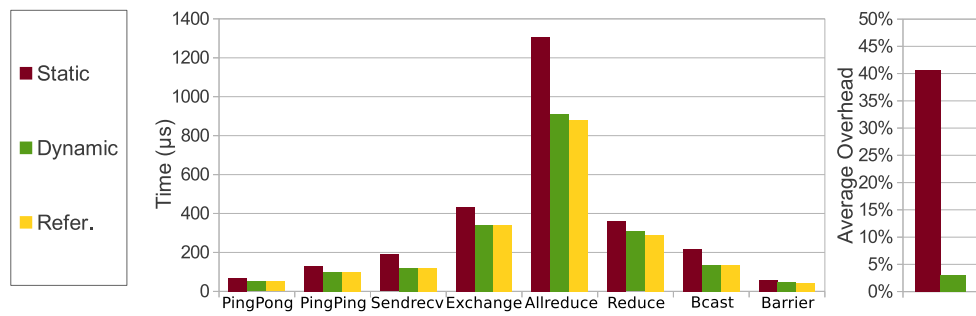

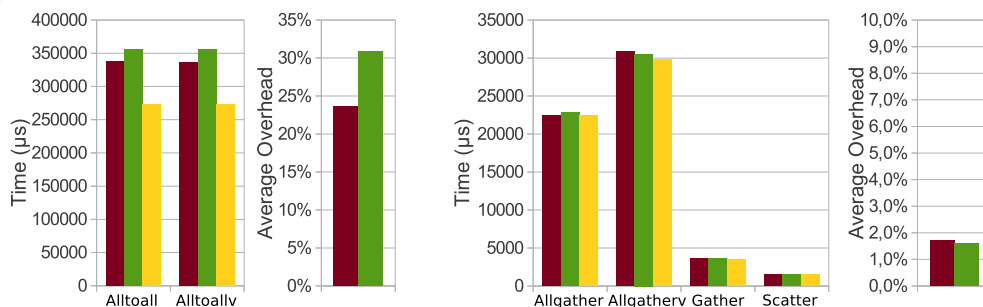(a) Localized communication pattern

(b) Balanced communication pattern (left) and Communication by RMA (right)

FIGURE 7.8: Results for the IMB with 0.5 GB of buffer resources across the cluster

Figures 7.5 to 7.8 show the execution time for selected benchmarks. The tests are grouped according to the type of their communication pattern. There mainly exist three types of communication patterns: localized communication with VELO, balanced communication with VELO, and communication with RMA. The last kind of communication is due to the use of optimizations for collective operations, which often group data into larger messages in order to minimize the number of send operations. For each kind of communication the average overhead is shown too.

7.4.1 Unexpected results in some tests

We have obtained surprising results for some of the IMB tests. In this section we separately evaluate them and analyze the causes for these unexpected results.

Figure 7.9 shows the execution time for the collective operations Scatterv, Reduce_scatter, and Gatherv. As it can be seen, these results are very different from those shown in Figures 7.5 to 7.8.

In the Scatterv test we can see that both the static and the dynamic flow control mechanisms are very inefficient when the overall size of receive buffers is smaller than 4 GB. This behavior is due to two reasons: the absence of optimizations for this collective and the constant change of the communication pattern because the root process of the collective is changed in a round-robin fashion. The first reason is evident if we compare the reference execution times of this test with the Scatter in Figures 7.5 to 7.8 (same amount of data transfered by sender but with optimizations). This performance difference justifies the need for any optimization that most probably will improve the results with flow control mechanisms. Regarding the second reason, we have performed a small modification of the original test with the purpose of observing the results when
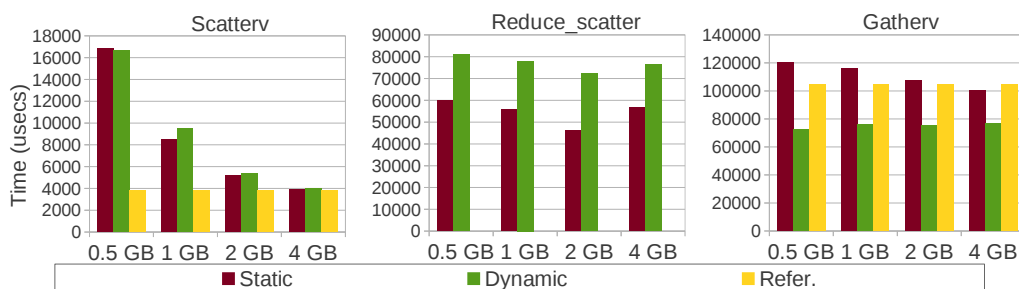

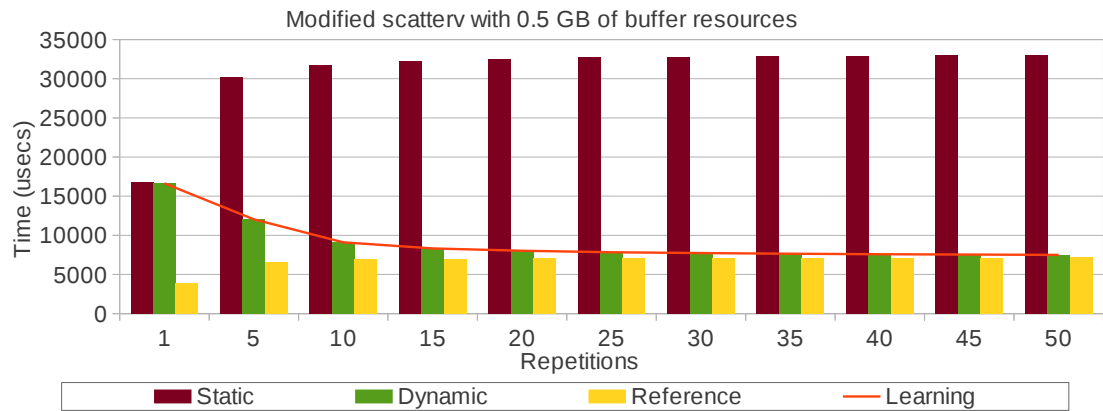FIGURE 7.9: Unexpected results for some IMB benchmarks

FIGURE 7.10: Results for the scatterv modified test

the communication pattern changes more slowly. In this modified test we still change the root process, but only after several repetitions.

Figure 7.10 shows the results from 1 to 50 repetitions. Notice that when the number of repetitions is equal to one, the modified test and the original test are exactly the same. When the number of repetitions increases the dynamic flow control is able to adapt to the communication pattern, so that the results are close to the reference execution time. On the other hand, we can observe an important increment in the reference and the static flow control times. The reason for this is that the overlap among Scatterv operations is reduced as the number of repetitions increases.

Regarding the Gatherv and Reduce_scatter tests, we also obtain surprising results. First, in the Reduce_scatter test we cannot include a reference time because we have not been able to provide the required amount of memory to ensure that receive buffers are not overflowed. Second, the results provided by the dynamic flow control for the Gatherv test are abnormally low (below the reference execution time).

In the case of Reduce_scatter, although we do not have a reference execution time to compare with, we can see that the static flow control presents better results. However, its behavior is not the usual one as the best results are obtained with 2 GB instead of 4 GB.

In order to gather more information about this surprising behavior, we have modified the dynamic flow control mechanism so that the maximum amount of slots that can be assigned to a given sender will be 62, in order to avoid a large increase of the mailbox
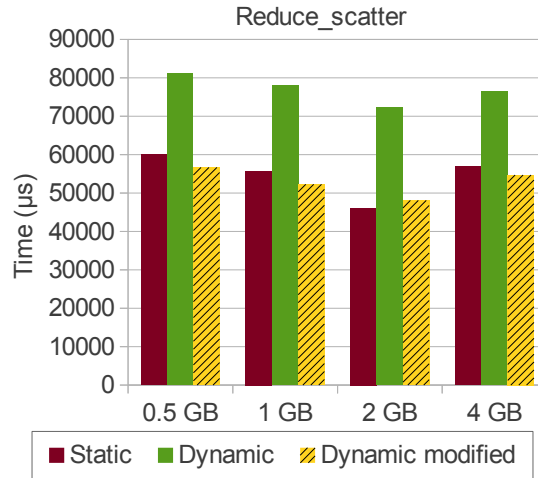
FIGURE 7.11: Reduce_scatter

size[1]. The new results can be seen in Figure 7.11, showing that when we enforce a shorter distance between the read and write pointers of the VELO engine, performance is improved for this particular test. However, we have not been able to extract a clear conclusion.

In the case of Gatherv we did not find any concluding reason for the surprising behavior and more research is required.

7.5 Memory consumption

In addition to the performance improvements achieved by our new dynamic flow control mechanism, it is also important to compare the memory footprints of both static and dynamic approaches. The memory consumption at each receiver (in bytes) of the structures necessary to manage the flow control mechanisms are:

$$Static = 4n + 2 \qquad Dynamic = 150n \tag{7.1}$$

with n the number of processes. As we can see, the dynamic structures are approximately 37 times more weighty than the static ones. However, these larger management structures must be put into the context of performance. In this regard, Figure 7.12 shows the average overhead for all tests in the Intel MPI Benchmark suite, excluding

---

[1]This maximum amount has been selected because in the 4 GB case each sender will own 64 slots in the receiver side, 2 of those being used for control purposes. Thus, in this case the buffer resources of the dynamic flow control will be identical to those of the static one.
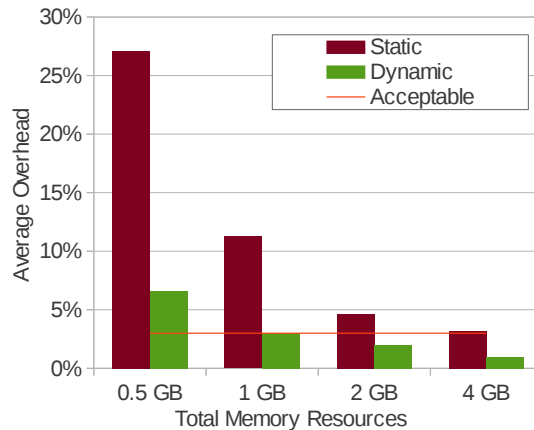
FIGURE 7.12: Average overhead

those with abnormal results. It can be seen in this figure that the dynamic flow control makes better use of buffer resources. Taking 3% as an acceptable value for the overhead, we observe that the overall buffer resources required by the static mechanism are approximately 4 GB, whereas the dynamic flow control requires only 1 GB (64 and 16 slots per process, respectively). Each slot equals 64 bytes, resulting in a total memory consumption (structures + buffer resources) per receiver of:

$$Static = 4n + 2 + 4096n \qquad Dynamic = 150n + 1024n \qquad (7.2)$$

Table 7.1 shows the results obtained with Equation 7.2 for the dynamic and static mechanisms for different number of processes across the cluster. These results clearly show that the scalability properties of the dynamic flow control are much better than those of the static one.

In addition, the dynamic flow control presents another interesting property, as it is able to leverage unused buffer resources for active communication flows. Notice that although the buffer resources used by the dynamic flow control depend on the number of processes of the parallel application, according to Equation 7.2, in practice the buffer requirements for providing a reasonably low overhead will be lower. The reason is twofold. First, it is well known that most MPI applications present communication patterns where a given process only exchanges messages with a subset of processes. Second, most optimizations of collective operations are based on several kinds of trees, whose cost is usually logarithmic. Thus they also provide a communication pattern where a given process exchanges messages only with a subset of processes. As it can be

TABLE 7.1: Total memory consumption, in MB, per receiver

# processes	1K	4K	16K	64K	256K
Static (MB)	4	16	64	256	1024
Dynamic (MB)	1.14	4.6	18.3	73.3	293.5

seen in both cases, the key issue is that a process only communicates with a subset of the other processes of the application. That is, from the point of view of a given process, only a subset of the total amount of processes are active, whereas the rest of processes will be considered as inactive in terms of communication. Therefore, an increment in the total number of processes of the application will usually produce a much smaller increment in the actual number of active processes involved in a given communication. Thus, the amount of senders for a given receiver increases, theoretically, with the total number of processes but, in practice, the percentage of active processes will probably be noticeably lower. This allows the dynamic flow control to take better advantage of unused buffer resources, which cannot be reused when making use of the static flow control because the buffer requirements of the static version do depend only on the total number of process without taking into consideration whether they are active or not. In this way, the buffer resources calculated with Equation 7.2 will show a realistic approach for the static flow control but an upper bound in the dynamic case. In reality they should be lower because as the amount of total processes increases, more unused buffers will be available, thus allowing to further reduce the mailbox size actually allocated with respect to the static flow control.

Chapter 8

Related Work

There are many implementations of the well-known credit-based flow control mechanism, both in on-chip (or between chips) [21][20] and in off-chip interconnects [17][24]. In [17] a user-level static scheme implemented in the MPI level of InfiniBand is presented. This scheme, based on the one previously proposed in [24], is very similar to our static flow control. The main difference is that our mechanism is entirely implemented inside VELO, thus being fully functional with EXTOLL and independent of upper communication layers (MPI, GASNet, etc.).

Unfortunately, there are very few references to credit-based flow control mechanisms able of redistributing credits according to the underlying communication pattern. In [17] a user-level dynamic flow control for MPI over InfiniBand is introduced. However, this scheme has an important difference with respect to our dynamic flow control, as it only increments the amount of pre-posted buffers in the receiver side for the senders that at some time show a high activity level, without being able to decrement the assigned buffers when their activity decreases. Thus, it is inefficient in multi-phase MPI applications whose communication pattern changes, as it is not able to adapt to those changes. On the contrary, our proposed flow control mechanism not only increments the amount of buffers at the receiver side during high activity periods, but it also dynamically decreases that number when activity ceases. Furthermore, our flow control mechanism is able to reassign buffer resources from some processes to others according to their activity level, thus making a very efficient use of the overall buffering in the system. As can be seen, the flow control in [17] is a subset of the one described in this work.

Chapter 9

Conclusions and Publications

In this work we have presented a new flow control mechanism that is able to dynamically adjust the buffer resources according to the communication pattern the parallel application and the varying activity among communicating peers. In order to show the benefits of this new proposal, we have compared its performance against a static credit-based flow control mechanism as well as against a communication layer that has unlimited buffer resources, thus being a theoretical upper bound that does not require a flow control protocol.

The evaluation of this proposal using our 64-node EXTOLL cluster shows that a static partitioning of the process mailbox is only appropriate when the communication pattern is noticeably balanced, that is, when all processes communicate with all other processes at the same rate. However, parallel applications rarely present this kind of communication pattern and therefore the need for dynamically adjusting buffer resources arises. Our new dynamic flow control mechanism provides extraordinarily high buffer efficiency under these circumstances, along with very low overhead.

Although we have focused the performance results on the EXTOLL interconnect, the proposed flow control mechanism can be easily applied to other network technologies such as InfiniBand.

9.1   Publications

The following papers related to this work have been published in different international peer-reviewed conferences and journals:

- J. Prades, F. Silla, H. Fröning, M. Nüssle, and J. Duato, **On the Design of a New Dynamic Credit-Based End-to-End Flow-Control Mechanism for HPC Clusters**. Parallel Computing, vol. 46, July 2015 (JCR 1st Quartile)

- J. Prades, F. Silla, H. Fröning, M. Nüssle, and J. Duato, **A New End-to-End Flow-Control Mechanism for High Performance Computing Clusters**. IEEE International Conference on Cluster Computing (CLUSTER). Beijing, China, September 24-28, 2012 (Core A)

In addition, some related papers have been also published in conferences without review:

- J. Prades, F. Silla, H. Fröning, M. Nüssle, and J. Duato, **Efficient Buffer Usage Through a New Flow-Control Mechanism**. XXIII Jornadas de Paralelismo. Elche, Spain, 2012

Bibliography

[1] SuperServer 5086B-TRF, last accessed 2015. http://www.supermicro.nl/products/system/5U/5086/SYS-5086B-TRF.cfm.

[2] J. Gonzalez-Dominguez, O.A. Marques, M.J. Martin, G.L. Taboada, and J. Tourino. Design and performance issues of cholesky and lu solvers using upcblas. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 40 –47, july 2012. doi: 10.1109/ISPA.2012.14.

[3] Quantum mechanics, dalton site, last accessed 2015. http://www.scalalife.eu/content/dalton-1.

[4] Biochemical dynamics, gromacs site, last accessed 2015. http://www.gromacs.org.

[5] Everett H. Phillips, Yao Zhang, Roger L. Davis, and John D. Owens. Rapid aerodynamic performance prediction on a cluster of graphics processing units. In *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, number AIAA 2009-565, jan 2009.

[6] Message Passing Interface, last accessed 2015. http://www.mcs.anl.gov/research/projects/mpi.

[7] Berkeley Unified Parallel C, last accessed 2015. http://upc.lbl.gov.

[8] D. Bonachea and J. Jeong. GASNet: A portable high-performance communication layer for global address-space languages. In *Parallel Computer Architecture Project*, 2002.

[9] Mellanox, last accessed 2015. http://www.mellanox.com.

[10] TOP500 supercomputer, last accessed 2015. http://www.top500.org.

[11] Y. Ajima, Y. Takagi, T. Inoue, S. Hiramoto, and T. Shimizu. The tofu interconnect. In *High Performance Interconnects (HOTI), 2011 IEEE 19th Annual Symposium on*, pages 87 –94, aug. 2011. doi: 10.1109/HOTI.2011.21.

[12] Ying Qian and A. Afsahi. Efficient rdma-based multi-port collectives on multi-rail qsnet/sup ii/ clusters. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, page 8 pp., april 2006. doi: 10.1109/IPDPS.2006. 1639563.

[13] S.K. Gutierrez, N.T. Hjelm, M.G. Venkata, and R.L. Graham. Performance evaluation of open mpi on cray xe/xk systems. In *High-Performance Interconnects (HOTI), 2012 IEEE 20th Annual Symposium on*, pages 40 –47, aug. 2012. doi: 10.1109/HOTI.2012.11.

[14] H. Fröning, M. Nüssle, H. Litz, and U. Brüning. A case for FPGA based accelerated communication. In *Ninth International Conference on Networks (ICN)*, April 2010. doi: 10.1109/ICN.2010.13.

[15] H. Litz, H. Fröning, M. Nüssle, and U. Brüning. VELO: A novel communication engine for ultra-low latency message transfers. In *37th International Conference on Parallel Processing*, Sept. 2008. doi: 10.1109/ICPP.2008.85.

[16] M. Nüssle, M. Scherer, and U. Brüning. A resource optimized remote-memory-access architecture for low-latency communication. In *38th International Conference on Parallel Processing*, Sept. 2009. doi: 10.1109/ICPP.2009.62.

[17] J. Liu and D.K. Panda. Implementing efficient and scalable flow control schemes in MPI over infiniband. In *18th International Parallel and Distributed Processing Symposium*, April 2004. doi: 10.1109/IPDPS.2004.1303193.

[18] OpenMPI, last accessed 2015. http://www.open-mpi.org.

[19] H.-W. Jin, S. Sur, L. Chai, and D.K. Panda. LiMIC: support for high-performance MPI intra-node communication on linux clusters. In *34th International Conference on Parallel Processing*, June 2005. doi: 10.1109/ICPP.2005.48.

[20] HT specification 3.1, last accessed 2015. http://www.hypertransport.org.

[21] S.S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb. The Alpha 21364 network architecture. In *Hot Interconnects 9*, 2001. doi: 10.1109/HIS.2001.946702.

[22] Intel MPI Benchmarks, last accessed 2015. `http://software.intel.com/en-us/articles/intel-mpi-benchmarks`.

[23] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G.E. Fagg, E. Gabriel, and J.J. Dongarra. Performance analysis of MPI collective operations. In *19th IEEE International Parallel and Distributed Processing Symposium*, April 2005. doi: 10.1109/IPDPS.2005.335.

[24] D. Bonachea and J. Jeong. MVICH: MPI for virtual interface architecture. In *Lawrence Berkeley National Laboratory*, 2001.