A New End-to-End Flow-Control Mechanism for High Performance Computing Clusters

Javier Prades, Federico Silla, José Duato
Departament d'Informàtica de Sistemes i Computadors
Universitat Politècnica de València
Camino de Vera, s/n 46022 Valencia, Spain.
Email: japraga@gap.upv.es, {fsilla,jduato}@disca.upv.es

Holger Fröning, Mondrian Nüssle
Computer Architecture Group
University of Heidelberg
B6, 26, Building B (3rd floor) 68131 Mannheim, Germany.
Email: {froening,nuessle}@uni-hd.de

***Abstract*—High Performance Computing usually leverages messaging libraries such as MPI or GASNet in order to exchange data among processes in large-scale clusters. Furthermore, these libraries make use of specialized low-level networking layers in order to retrieve as much performance as possible from hardware interconnects such as InfiniBand or Myrinet, for example. EXTOLL is another emerging technology targeted for high performance clusters.**

These specialized low-level networking layers require some kind of flow control in order to prevent buffer overflows at the received side. In this paper we present a new flow control mechanism that is able to adapt the buffering resources used by a process according to the parallel application communication pattern and the varying activity among communicating peers. The tests carried out in a 64-node 1024-core EXTOLL cluster show that our new dynamic flow-control mechanism provides extraordinarily high buffer efficiency along with very low overhead, which is reduced between 8 and 10 times.

***Keywords*-EXTOLL; VELO; RMA; MPI; dynamic flow-control; static flow-control;**

I. Introduction

The computing landscape has been traditionally driven by the demand for a never enough computing power. In this regard, even the powerful commodity computers available nowadays, which provide up to 80 computing cores and up to 2TB of RAM memory [1], do not satisfy the computing requirements of most High Performance Computing (HPC) applications. As a result, these demanding applications are split into as many parallel processes as possible that are concurrently executed in the cores available across large clusters, thus aggregating huge amounts of computing resources. In order to exchange data among this myriad of processes, some kind of messaging layer is usually leveraged. One of them is the Message Passing Interface (MPI) library [2].

MPI has proven efficiency and effectivity for large-scale computing deployments, where the use of shared-memory programming is not possible. Because of this, even though MPI suffers from a non-negligible associated overhead due to tag-matching, progress, and copy operations, the near future parallel programming will still for sure 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, of the emerging Berkeley Unified Parallel C (UPC) [3], which provides support for shared-memory programming across distributed systems, usually leveraging the GASNet [4] messaging layer.

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 high performance computing system as far as the appropriate implementation is available. Because so many HPC applications rely on these messaging layers, most interconnects intended for high-performance computing have at least one available implementation.

This is case, for example, of Gigabit Ethernet, which is currently the most widely used interconnect according to the TOP500 supercomputing list [5]. However, due to the simplicity of this interconnect, very few optimizations related with the intrinsics of this technology are possible. On the contrary, InfiniBand [6], which has also gained noticeable popularity during the last years as a high performance network solution, allows for a lot of optimizations given its much more complex nature.

One of the key issues when designing an efficient messaging layer implementation for any interconnect technology is flow control. The flow control mechanism prevents a fast sender from overwhelming a slow receiver and exhausting its buffer space resources. The proper design of a flow control mechanism is an important issue because 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 receiving side are efficiently managed in order to avoid processes at the sending side from stalling due to the lack of buffers to store received messages. Regarding scalability, the amount of receiving 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 much memory resources across the cluster to the internals of the communication scheme, instead of devoting those resources to the actual needs of the executing applications.

In this paper we present an efficient new flow control mechanism for the new EXTOLL [7] networking layer, which recently evolved from an early prototyping stage to

a high-performance commercially available network. This new interconnect, which targets the HPC market segment, provides an innovative communication solution that enables ultra low latencies and avoids costly and power-hungry external switches, thus comparing in performance to the fastest InfiniBand versions but at a much lower economic cost. In order to show the extraordinary scalability and performance features of our new flow control mechanism, we center our presentation around the MPI implementation for the EXTOLL interconnect, based on OpenMPI [8]. However, other messaging layers, such as GASNet, could also be used, as we recently completed its implementation on the EXTOLL interconnect [9]. Notice that all the experiments presented in this paper have been conducted in our 64-node 1024-core EXTOLL cluster.

The rest of the paper is organized as follows. In Section II we introduce the main features of the new EXTOLL network design. Later, in Section III, we briefly present the implementation of OpenMPI for the EXTOLL interconnect. Section IV presents some background on flow-control mechanisms. After that, Section V presents a thorough analysis of a static flow-control mechanism, which will be used as the baseline for our new dynamic flow-control technique, that will be presented in Section VI and analyzed in Section VII. Finally, Section VIII presents the main conclusions from the paper.

II. The EXTOLL Interconnect

EXTOLL is a high performance interconnection network that provides dedicated support for high performance computing. The main goals of EXTOLL are to reduce the messaging 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 are suffering from high overhead compared to bulk transfers.

EXTOLLs architecture is shown in Figure 1; compromised of the host interface shown in 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 in the right side.

The complete architecture is implemented in a single chip that is typically located on an add-in card. Note that all required switching resources are already integrated in that chip, so that besides cabling no further resources are required. With the 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) on switch level to reduce *Head-of-line* (HOL) blocking and employs cut-through switching that enables very low switching

[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.

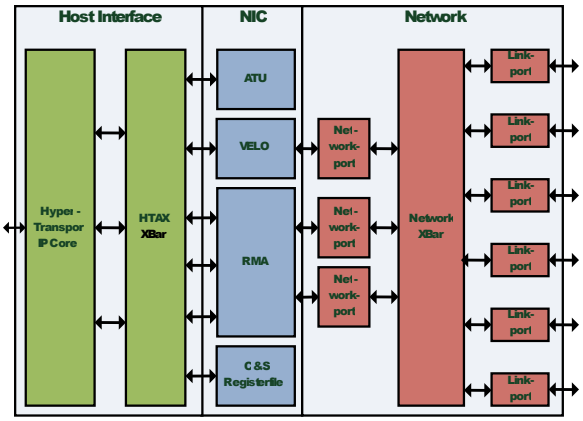


Figure 1. EXTOLL's top-level block diagram

latencies. Deadlocks are avoided by using virtual channels. The switch also supports in-order delivery of packets, which can be leveraged by software components to simplify protocol design. A reliable transmission is guaranteed by a link-level retransmission protocol, another important feature to simplify the design of software components.

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*) that provides optimized transmission for small messages. Its optimized interface between hard- and software minimizes the overhead associated with sending and receiving of packets. This contributes not only to a low latency, but also to achieving high message rates. For sending, basically only one (or two) PIO writes to a special address is required. Meta-data like destination and length are encoded in the address, so the complete PIO write payload (typically 64 bytes) is available for software use. On the receiving side, packets are directly written into main memory using a single ring buffer per thread, called mailbox. A mailbox is statically sliced and its entries include status information, so receiving threads can poll on known main memory locations, and the coherence protocol ensures that no unnecessary bus traffic is generated. Further details on VELO can be found in [10].

Beside this support for fine grain communication, we address efficient bulk transfer with the RMA (*Remote Memory Access*) communication engine, which offers Put/Get semantics to the user. A hardware-based address translation unit (ATU) supports it, and in conjunction user level secure data transfer is guaranteed. The RMA unit almost completely off-loads the communication task from the CPU, providing high overlap and low overhead. Notifications can be generated both on source and destination side in order to signal completion of Put/Get requests. Also, a Put/Get passive target is notified of completed transfers from remote nodes. Further details on RMA and ATU are provided in

[11].

In combination, both units provide efficient support for all message sizes. Using the API libraries (libvelo and librma), the user or library can decide which transfer method is more suitable for a given message size. Furthermore, software components are completely unburdened of any overhead associated with in-order delivery or re-transmission of lost packets.

Finally, notice that the flow-control proposed in this paper is intended for the VELO unit, as it uses a constraint 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.

We have built a 64-node EXTOLL cluster that will be used as test bench throughout this paper. In our cluster, each node includes 4 4-core sockets and 16GB of memory, thus accounting for a total of 1024 cores and 1TB of RAM across the cluster. We have configured the EXTOLL interconnect according to the 3D mesh topology.

III. MPI over EXTOLL

The MPI implementation of EXTOLL is based on the popular OpenMPI implementation. In the level below MPI, low-level API libraries (libvelo and librma) provide direct, user-level access to the functionality of the respective communication engines within EXTOLL.

OpenMPI is a highly modular, component-oriented implementation of MPI. Among the different possibilities to support point-to-point messaging, namely *Byte Transfer Layer* (BTL) and *Matching Transfer Layer* (MTL), MTL has been chosen for EXTOLL. Here, the component handles all the protocol handling including the matching of send and receive requests following the MPI specification.

Small messages up to a certain threshold are sent using an *eager protocol* over VELO. The threshold was chosen to be 2KB. In this eager protocol, a first VELO message carries the necessary MPI header containing information needed for message matching and payload up to the maximum 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, the MPI matching is performed and, if needed, multiple VELO packets are reassembled to complete the operation. Messages larger than the 2KB threshold are sent using a *rendez-vous protocol* leveraging RMA. For this protocol, a small VELO message is first sent carrying information that describes the buffer which is actually to be sent. Upon matching against a posted 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 are heavily used to signal completion of such a large MPI transfer, both on the sender and the receiver side, yielding a very efficient protocol.

A third protocol has been implemented for intra-node communication, taking advantage of the shared memory of modern multi-processor nodes.

IV. About Flow-control Mechanisms

Basically, a flow-control mechanism allows that slow receivers, or receivers that are busy performing other tasks, do not get their reception buffers overflowed because of senders transmitting too many data packets. Notice that if receiving buffers had an unlimited size, then this overflowing concern would not exist and hence a flow-control mechanism would not be required. However, these unlimited size buffers would require a lot of memory resources to be devoted to something that is not the application itself but to the underlying communication infrastructure, the cost of which should be kept as low as possible. Moreover, notice that these memory resources are extremely expensive and, what is worse, dependent on the number of processes involved.

Therefore, flow-control mechanisms can be seen as techniques, that allow to establish a maximum boundary to the memory resources used by the communication layer. However, this limit comes at the cost of an execution time overhead of the parallel application due to two causes: first, some processes may stall because of lack of buffers at the receiver side. Second, the flow-control protocol itself introduces some computational overhead as well as additional network bandwidth requirements because of the need of communicating to senders the state of the receiving buffers. Thus, the efficiency of a flow-control mechanism can be seen as a balance 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 credit-based flow-control that has been later evolved to an efficient dynamic one. Both protocols are based on the use of credits to track the amount of available slots at the reception buffers, although the dynamic version manages credits in a flexible way, thus achieving better performance.

V. 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 our interconnection network.

A. Major adaptations

The original credit-based flow-control mechanism establishes that each sender owns certain buffer space at the receiver's memory. The exact amount of buffering resources is explicitly stated by the number of credits the sender is given at initialization time. In this way, a receiver has as many independent buffers (or buffer partitions) as senders exist. However, in our interconnection network a receiver

only has one buffer, referred to as mailbox, which is shared among all its senders. Therefore, a trivial change will allow us to adapt our monolithic mailbox scheme to the credit philosophy, as we simply have to divide our mailbox into as many independent buffer partitions as potential senders may exist. Notice however, that the mailbox is split into several portions in a logical way, but not in a physical way. Thus, packets from different senders will not necessarily be stored in different mailbox partitions, but they may be mix as buffer space is being split in the mailbox, but not the physical distribution of that space. In this way, we still have a single write pointer and a single read pointer associated to the mailbox.

Another change 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 receiving buffer, a credit is sent back to the sender. However, as updating every credit may generate large amounts of traffic, many implementations [13],[14] of this flow-control accumulate several credits in order to send back a single packet, thus not wasting network resources. As in our interconnection network the only way of communication between processes is by exchanging messages, on our implementation a receiver process will accumulate credits up to certain threshold. Once reached that limit, the process will generate a data packet containing credits and send it back to increment the sender's credit count. From now on, we will refer to this kind of packets as *control packets* because they are generated by the flow-control mechanism. In a similar way, we will refer to the regular data packets generated by the parallel application simply as *data packets*. Notice that we only use explicit control packets to update the state of the receiving buffers at the sender side, thus not using other techniques such as *piggy-backing*. The reason for this is that VELO supports fine-grain messaging and high message rates and, therefore, the additional complexity of piggy-backing is avoided.

B. Static credit-based flow-control operation

In order to prevent possible misunderstandings about the duality of the sender/receiver roles that can coexist at either process, we will refer to a process as *sender* when the process is sending data packets to other processes and it is extracting control packets from its mailbox in order to retrieve credits to continue sending packets and we will refer to a process as *receiver* when the process is extracting data packets from its mailbox and it is sending back control packets to update the state of its buffers at the sender side.

Initially, the mailbox at each receiver is split into two regions: one for accommodating data packets, *data region*, and the other one for storing control packets, *control region*. The data region will be equally distributed among all the processes. Thus, each sender will own an amount of slots in the mailbox of each receiver. This number of credits will be referred to as *quota of credits*. As each slot can contain one VELO packet, each slot will be equivalent to one credit. In a similar way, the control region will be equally distributed among all the receivers, so that they will have an amount of slots in the mailbox of each sender where to send control packets. The portion corresponding to each receiver will be referred to as *control slots*. Finally we set the *threshold for credit return* as:

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

As can be derived from formula 1, when a receiver reaches the threshold, it sends out a control packet to the sender knowing that, in order for the receiver to reach the threshold, the sender had to free a control slot to obtain more credits to be able to continue sending data packets. This condition ensures that new control packets can be stored at the senders mailbox. Finally we add a restriction, for the minimum size of the quota of credits, which will be:

$$(2)\ credit\_quota\_min=ctrl\_slots+1$$

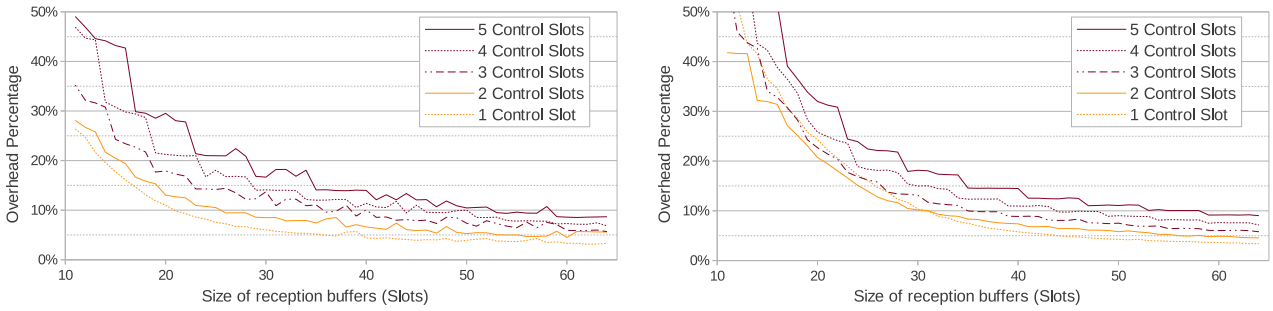
This restriction guarantees, that the threshold for credit return will be greater than or equal to 2.

When the initialization stage has finished, the static flow-control operation is identical to the original credit-based flow-control mechanism: whenever a data packet is forwarded to the receiver buffer, the credit counter at the sender is decremented. If the counter reaches zero, it means that there is no available slot at the receiver buffer and, therefore, no data packet can be forwarded. On the other hand, when the receiver frees up a slot in its buffer, the recovered-credit count is incremented. When the count reaches the update threshold value, the receiver sends back a control packet to update the sender credit count and then the receiver resets its recovered credit-counter. Notice that a control packet does not consume credits when it is sent, neither it increments the recovered-credit counter when it is extracted from the mailbox. This type of packets is automatically controlled by condition 1 and the control region allocated in the mailbox.

C. Setting the static flow-control parameters

In the previous section we have introduced the initialization of the static flow-control as well as its operation. During the initialization phase, one parameter's essentially used: the amount of control slots allocated in the mailbox (control region). In this section we analyze how the value of this parameter influences the flow-control behavior and how its efficiency strongly depends on this value.

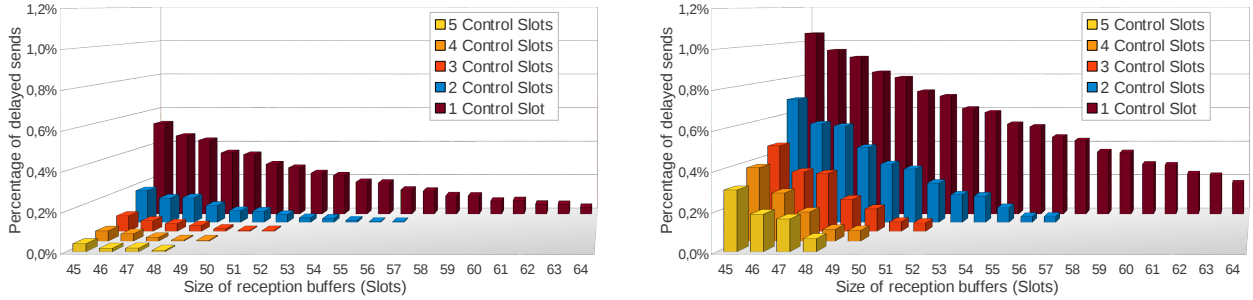
The first test we have performed shows how the distance affects the behavior of flow-control mechanism. For so we leveraged the multi-pingpong test included in the Intel MPI Benchmark Suite [12]. In this test the 16 processes belonging to a node perform 16 simultaneous pingpong operations with the 16 processes belonging to other node. We have used 2KB



(a) 1 hop distance

(b) 14 hops distance

Figure 2. Overhead generated by flow-control in multi pingpong test



(a) 1 hop distance

(b) 14 hops distance

Figure 3. Percentage of sends delayed by flow-control in multi pingpong time test

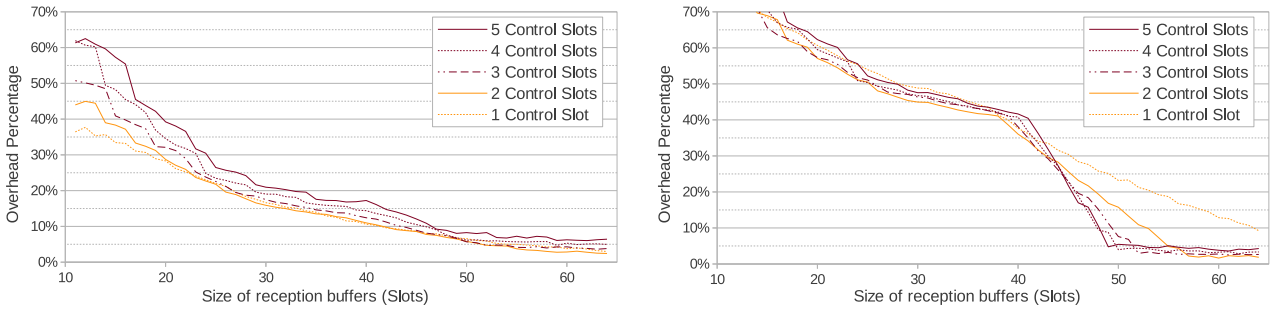
data messages as this size is the one that most stresses the flow-control protocol. Moreover, in addition to the tests that leverage the flow-control for comparison purposes, we have executed the same tests with very large reception buffers and without the flow-control mechanism in order to compute the lower bound of the multi-pingpong time, when will be later used as a reference for the results obtained by our flow-control. Notice that disabling the flow-control is possible because of the extremely large buffer size, that avoids buffer overflow even in the worst conditions within this test.

Figures 2 and 3 show the main results from this test, which have been carried out for distances between one (minimum network distance) and fourteen hops (maximum network distance in our EXTOLL cluster).

Figure 2 shows the generated flow-control overhead for 1 and 14 hops. The minimum reference times for these cases are 50.04 and 51.17 usecs for 1 and 14 hops, respectively. As we can observe when the buffering resources are low (buffers size less than 30 slots) the overhead for large distances is very high compared to small distances. However, when buffer size is larger, the behavior is very similar independently of the distance between processes. Figure 3 shows the percentage of delayed messages because of using the flow-control (no credits available). We observe that the delayed message percentage increases with the distance but the breakpoints where the application flows without interruptions due to flow-control are identical, independently of the distance. Regarding the number of control slots per process, Figure 3 shows that the stalls due to flow-control decrease when we increment the number of control slots. However, as we can see in Figure 2, the overhead generated by flow-control is minimum when we use one control slot. This behavior is very interesting because a reduction in the number of stalls does not reduce the overhead generated. So the effort for reducing waits (additional traffic) is more expensive that the waits themselves.

The next test carried out to analyze the static flow-control is similar to the previous test but in this case we leverage the alltoall operation for different sizes of groups of processes (256 and 1024 processes). Data messages exchanged have a size of 2KB. Basically, during an alltoall operation, each process will first send a 2KB message to every process and then it will receive a 2KB message from each of 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 communication trees with the goal of reducing the amount of communication between processes [15]. However, we have disabled this optimization because we want to stress as much as possible the use of reception buffers. As in the previous tests, we have obtained a minimum reference time when no flow-control is used in order to better analyze the behavior of the protocol.

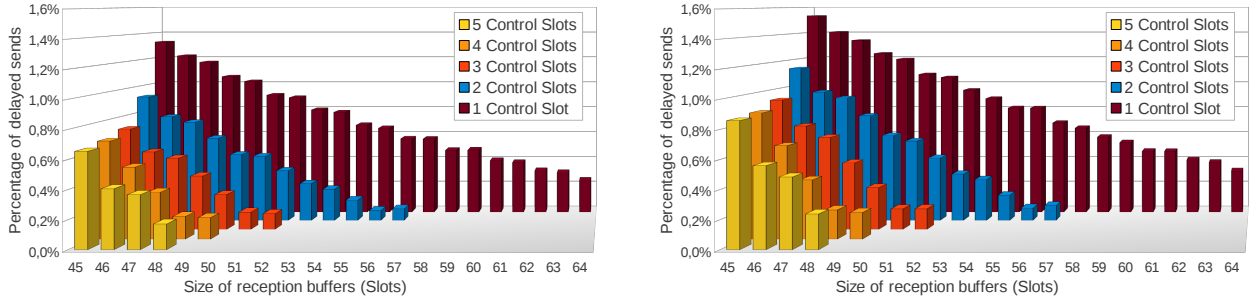
The minimum reference times obtained for the alltoall operation have been 52908.41 usecs for 256 processes and 278515,08 usecs for 1024 processes. Figure 4 shows the overhead generated by the flow-control for alltoall operation with 256 and 1024 processes. As we can observe, the overhead increases with the number of processes involved



(a) 256 processes

(b) 1024 processes

Figure 4. Overhead generated by flow-control in alltoall test



(a) 256 processes

(b) 1024 processes

Figure 5. Percentage of sends delayed by flow-control in alltoall test

but from values close to 50 slots, this overhead is very similar in the two cases. On the other hand in Figure 5 we can see the flow-control retained messages percentage. The percentages are very similar independently of the number of processes and the breakpoints where the application flows without waits are the same than in the multi-pingpong test. The study of the influence of the number of control slots allow us to appreciate a different behavior from multi-pingpong because in this case the reduction of the waits provides a reduction in the overhead time. This is because in this test the use of reception buffers is very high and the time expended in the search for returned credits is large when the mailbox is full.

After the analysis we can configure the flow-control protocol so that it works efficiently. According to the results obtained in the tests, two control slots per process is the best trade-off because it provides the best results in the alltoall test whereas in the multi-pingpong test this value provides results close to the best option, represented by one control slot. Regarding the size of reception buffers, values around 58 slots are the best choice because for this size the overhead obtained approaches to 5%. We have analyzed two extreme communication patterns, one in which the communication is a very localized, multi-pingpong, and another in which the communication is extensively balanced, alltoall. We can see that the behavior is similar and we can establish that the behavior for intermediate communication patterns will be similar too. This behavior shows an important concern about the static flow-control that will be shown in the next section.

D. Concerns about the static flow-control

The flow-control presented in this section presents a serious concern. A static partitioning of the process 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 buffering 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 showed in Figure 6, where the overhead of the flow-control protocol is represented for four different communication patterns:

1) 1024 processes execute a alltoall operation involving all processes
2) 1024 processes execute 2 simultaneous alltoall operations involving 512 processes each
3) 1024 processes execute 4 simultaneous alltoall operations involving 256 processes each

4) 1024 processes execute 8 simultaneous alltoall operations involving 128 processes each

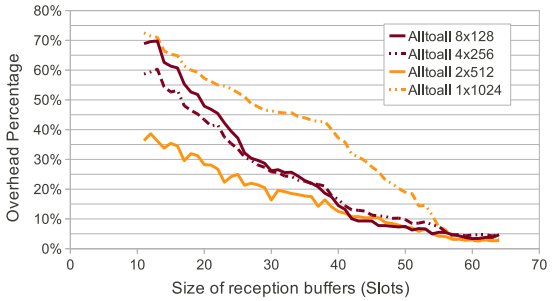


Figure 6. Overhead due to static flow-control for four different communication patterns

In all cases the overhead is not acceptable (less than 5%) until the size of the reception buffers is not larger than 55 slots. However, for the three last communication patterns the overhead could be much lower for smaller reception buffers if the buffers not used by inactive senders could be allocated to the active ones. For example, in the case of the 8 simultaneous alltoall operations, a given process only communicates with 127 processes, leaving the credit quota of 896 processes unused. This means that 87.5 % of the buffering resources are wasted whereas the active processes may probably benefit from these resources.

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

VI. Dynamic Credit-Based Flow-Control

Results shown in Figure 6 clearly point out the lack of flexibility of the static flow-control, which causes inefficiency and lower performance. Figure 7 presents the results for the same experiments when our dynamic flow-control mechanism is leveraged. The dynamic flow-control is able to adapt the buffering resource allocation according to the actual communication pattern, thus assigning more buffer slots to those processes requiring them. As can be seen in the figure, the better use of the buffering resources is evident for the cases 8x128, 4x256, and 2x512, where the overhead has been reduced below 5% even for a small number of slots. These good results are achieved because this dynamic flow-control continuously monitors the activity of each sender and updates their quota of credits according to their needs. In the following we present the main characteristics of the new flow control. Although the proposed flow control is quite simple from a conceptual point of view, its implementation is very complex. Therefore, and given the space constraints, most of the exact implementation details have been omitted in order to make the presentation easier to understand in a reasonable amount of space.

A. Dynamic Flow-Control Operation

The dynamic flow-control follows the same basic principles presented for the static one, being the main difference

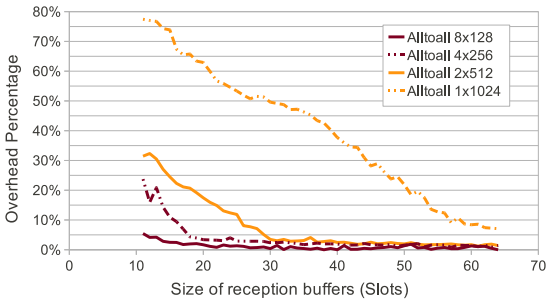


Figure 7. Overhead due to dynamic flow-control for different communication patterns

among both versions a monitoring function incorporated to the receivers that tracks the activity of their respective senders. In this way, when a receiver is to return back credits to a given sender, the relative activity level of that sender 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.

At the initialization stage, a control region in the mailbox is reserved exactly in the same way as in the static flow control. The rest of the mailbox slots, which was the data region in the static approach, is split into two data regions: the *static region*, that will contain the minimum size required by the static flow control, according to Equation 2, and the *dynamic region*, that will contain the rest of the mailbox, typically a quite large number of slots. The static region will be distributed among the senders during the initialization, representing the minimum credit quota a given sender can own during the execution of the parallel application. The dynamic region will be assigned to senders after the initialization stage and will be used to increase the credit count of senders presenting higher activity. Once the dynamic region has been exhausted, credits from senders presenting lower activity will be diverted to senders presenting higher activity levels. Notice that diverting credits from some senders to others must be carried out in such a way that the former do not overflow their new credit count. The way to achieve so includes reducing the rate of credits returned to them and also sending them an explicit request to return unused credits.

Once the dynamic flow control algorithm has been initialized, its operation is the same as for the static version, although in this case the activity of senders is monitored and their credit quota updated.

B. Activity Monitoring

The activity of a given sender is defined in terms of the time it requires to consume all its credit quota. This event is easy to be detected by a receiver because a sender will entirely consume its quota after traversing $n+1$ times the threshold for returning credits, being n the number of control slots used, as it can be deduced from Equation 1. We have tried higher activity monitoring frequencies but they do not

filter instantaneous high activity levels, thus confusing the updating algorithm.

At each monitoring point for a given sender, the receiver compares the activity of that sender, which is going to be returned credits, against all of the other senders. If this sender presents higher activity than others, then credits from the sender with the lowest activity level will be diverted to the sender being monitored. The amount of credits being assigned to that sender is equal to the threshold value to return credits. Once the credits are moved among senders, a control packet with the updated credit count will be sent to the sender being monitored.

The goal of this flow control is reaching a balance point in the system so that all the processes can consume their respective credit quotas in the same amount of time, independently of the exact size of that quota. This helps to keep all the application processes synchronized, making progress at the same pace and therefore avoiding unnecessary stalls.

VII. Experimental Results and Analyses

In this section we evaluate the performance of our flow-control mechanism. Notice that in all the following tests the reference time has been obtained as in the previous sections.

In Figure 7 we have seen that the dynamic flow-control solves the concerns of the static one. However, the overhead generated in the case 1x1024 is slightly higher than for the static flow-control. The reason is that for this case it does not exist a distribution of buffering resources better than the provided by the static partitioning of the mailbox, as in the alltoall operation the communication pattern is perfectly balanced. However, notice that the collective optimizations are not enabled and this communication pattern is unlikely to be present in a real parallel application. Nevertheless, it is worth to analyze which are the sources for the overhead present in the dynamic flow control.

A. The Main Components of the Overhead

Basically the overhead generated is due to two main causes: 1) the stall time produced when a process consumes all the credits and the additional traffic generated by flow-control, 2) the computational overhead due to credit management.

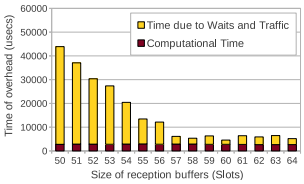


Figure 8. Overhead in alltoall operation, static case

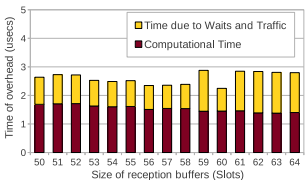


Figure 9. Overhead in multi-pingpong operation, static case

Figures 8 to 11 show these two components of the overhead, comparing the static and dynamic versions of the flow control for the alltoall and multi-pingpong scenarios.

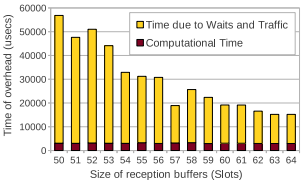


Figure 10. Overhead in alltoall operation, dynamic case

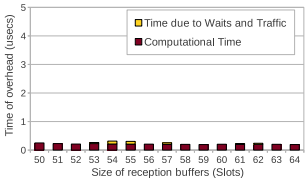


Figure 11. Overhead in multi-pingpong operation, dynamic case

The alltoall test is run for 1024 processes whereas the multi-pingpong leverages 1-hop distances.

If we compare the results for the alltoall test, a larger overall overhead is present in the dynamic case, although the computational overhead is very similar in both cases. This case corresponds to the 1x1024 plot shown in Figures 6 and 7. Notice that this traffic pattern is not expected to be present in real applications where collective operations are optimized. The reason for the larger overhead is the higher amount of messages generated by the dynamic flow control along with the heavily network load, which negatively impacts the delivery of the higher number of control packets. 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 the static one, confirming the benefits of our approach.

B. Credit Reallocation for Communication Pattern Changes

In this test we have executed a parallel application where the communication pattern is changing and we have analyzed how the dynamic flow control adapts to those changes. For so, the parallel application performs many sequential 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), 100 alltoall (all 1024 processes). Additionally, this test has been performed with a reception buffer size of 30 slots per sender (30K slots in total per receiver), exchanging 2KB messages in the alltoall operation, and without the collective optimizations.

Figure 12 shows the average amount of credits that the process with rank 0 owns to send data messages to processes with ranks (0-255], [256-511], and [512-1023]. Each point has error bars that show the standard deviation of the data.

C. Evaluation with the Intel MPI Benchmark Suite

We have analyzed how the availability of buffering resources affects our flow-control mechanism in the context of the Intel MPI Benchmarks. We have executed all these benchmarks with 1024 processes and a message size of 2KB. Reception buffer size has been set to 12, 16, 32, and 64 slots per sender, what represent a total of 768 MB, 1 GB, 2 GB, and 4 GB, respectively, of memory resources across the entire the cluster. As in the previous sections we have obtained a minimum reference time for each benchmarks

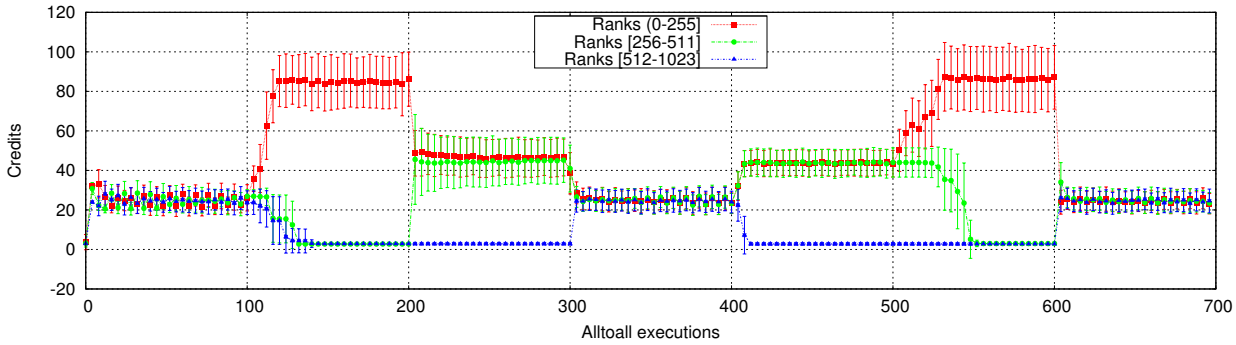


Figure 12. Evolution of credits

without any memory resource limit and without any flow-control mechanism, so that our proposal can be better put into context. Furthermore, notice that in this section the collective optimizations have not been disabled.

Figure 13 shows the execution time obtained for some of the benchmarks with 4 GB of overall memory resources. Figure 14 shows the average overhead for these tests using 768 MB, 1 GB, 2 GB, and 4GB of memory resources across the cluster with the static and dynamic flow-control mechanisms.

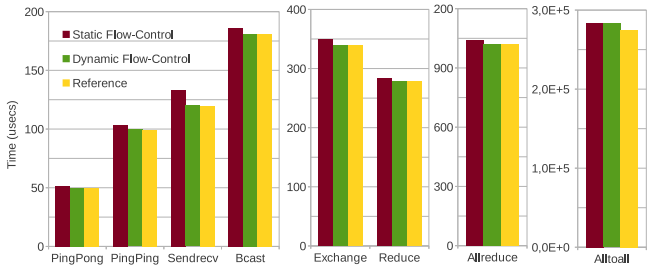


Figure 13. IMB-Benchmarks Suite results

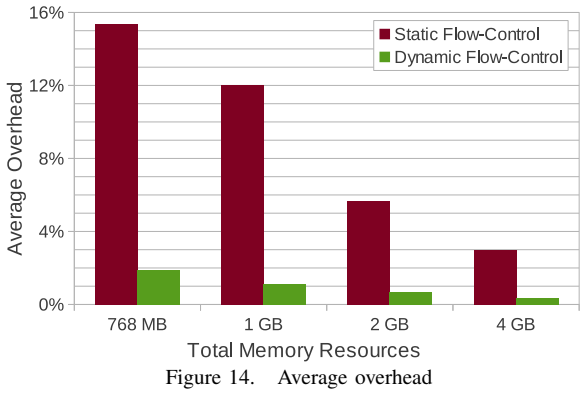


Figure 14. Average overhead

VIII. Conclusions

In this paper we have presented the development of a new flow-control mechanism that is able to adjust the buffering resources according to the parallel application communication pattern 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 buffering resources, thus not requiring a flow-control protocol.

The evaluation of this proposal in 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 with the same intensity. However, parallel applications rarely present this kind of communication pattern and therefore the need for dynamically adjusting buffering resources arises. Our new dynamic flow-control mechanism provides extraordinarily high buffer efficiency in these circumstances, along with very low overhead.

In this paper we have shown the main features of the new flow control. We plan to perform a thorough evaluation of its behavior with real parallel applications using both MPI and GASNet implementations over the EXTOLL interconnection.

References

[1] Supermicro homepage, *http://www.supermicro.com/*
[2] Message Passing Interface hompage, *http://www.mcs.anl.gov/research/projects/mpi/*
[3] Berkeley Unified Parallel C homepage, *http://upc.lbl.gov/*
[4] D. Bonachea, J. Jeong, *GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages*, Parallel Computer Architecture Project, Spring 2002
[5] Top 500 homepage, *http://www.top500.org*
[6] Mellanox homepage, *http://www.mellanox.com/*
[7] H. Fröning, et al., *A Case for FPGA based Accelerated Communication*, ICN 2010
[8] OpenMPI homepage, *http://www.open-mpi.org/*
[9] K. Kujat, et al., *A GASNet Conduit for the New EXTOLL Interconnection Network Architecture*, Submitted to CLUSTER 2012
[10] H. Litz, et al. *A novel communication engine for ultra-low latency message transfers*, ICPP 2008
[11] M. Nüssle, et al., *A resource optimized remote-memory-access architecture for low-latency communication*, ICPP 2009
[12] Intel MPI Benchmarks homepage, *http://software.intel.com/en-us/articles/intel-mpi-benchmarks/*
[13] HT Specification 3.1
[14] S.S. Mukherjee, et al., *The Alpha 21364 Network Architecture*, IEEE Micro, Volume 22 Issue 1, January 2002
[15] J. Pjesivac-Grbović, et al., *Performance Analysis of MPI Collective Operations*, IPDPS 2005