ToCL: A Thread Oriented Communication Library to Interface VIA and GM Protocols

Albano Alves[1], António Pina[2], José Exposto, and José Rufino

[1] Instituto Politécnico de Bragança,
Campus Sta. Apolónia, 5301-857 Bragança-Portugal
albano@ipb.pt
[2] Universidade do Minho,
pina@di.uminho.pt

Abstract. In this paper we present ToCL a thread oriented communication library specially designed to fully exploit multithreading in a multi-networked cluster environment. ToCL provides a basic set of primitives to handle zero-copy message passing between application threads spread among cluster nodes. Large messages are fragmented and sent to remote threads as single messages using multiple low-level communication subsystems. The current implementation supports both Myrinet through GM and Gigabit Ethernet through VIA but we plan to extend it to other communication subsystems.

Keywords: multithreading, message-passing, intermediate-level library.

1 Introduction

With the advent of commodity SMP workstations and high performance SANs it became possible to do parallel computation at low cost. However, to fully exploit such power we need new programming models.

Multithreading and message-passing are two well-known techniques that may be combined to build appropriate platforms to use in a cluster of multi-processor machines.

1.1 Low-Level Communication Libraries

The increasing processors performance stressed the need of higher speed communication subsystems to achieve low-overhead calls to access the network.

User-level communication libraries are used to interface network adapters directly bypassing the operating system. Usually, these libraries provide a low-level interface for data exchange between processes executing in networked machines avoiding memory copies – zero-copy message transfers.

However, user-level API libraries are not intended to be used at the application level. To take advantage of the specificity of network adapters hardware, such as on-board processors, memory and DMA controllers, these libraries offer just a few basic facilities to application programmers.

GM [12] and MVIA [14] are two well-known low-level communication subsystems to exploit Myrinet and Gibabit/Fast Ethernet, respectively. FM [15], BIP [9], LFC [2] and MyVIA [5] also offer user-level interfaces to specific hardware.

1.2 High-Level Abstractions

If we want programmers to easily use libraries we need to improve, enrich and extend the facilities offered by low-level communication libraries. High-level abstractions and much richer interfaces are available from MPI [16], PM^2 [13] and PANDA [3], for example, but the use of these platforms forces programmers to learn specific programming models.

These high-level programming environments, among others, may be used to develop multithreaded applications, and inclusively support remote thread creation. However, application threads are not first class programmer entities involved in communication.

Traditionally the use of parallelism and high-performance computation has been directed to the development of scientific and engineering applications. However, the increasing use of multithreading on commercial and internet applications has shifted the market to new areas, where conventional programming models like MPI don't fit programmers' needs. In this context, the development and use of multithreading programming techniques in high-performance SMP cluster environments would benefit from the existence of a thread oriented communication library and higher level programming models.

2 ToCL Approach

ToCL is an intermediate-layer communication library designed to exploit high-performance networking hardware through available low-level communication subsystems. ToCL introduces the notion of thread oriented communication library.

Presently it provides a unified interface for GM and VIA but the overall design easies the integration of other low-level communication subsystems. The primary choice of GM and VIA [6] (mainly MVIA) is directly related with the research we are pursuing to take advantage of Myrinet and Gigabit high-performance technologies to build a scalable information retrieval environment.

2.1 Entities

ToCL uses three major entities to model applications executing in a cluster: hosts, processes and threads. Global identifiers are assigned to hosts and processes, to uniquely identify them within the context of a cluster. Threads may, also, have a global identifier that is compatible with the existence of a local identification in the context of a process.

ToCL entities have several attributes, as name, parent host (for processes), parent process (for threads), etc, and applications may locate them by using attribute-based queries. A distributed directory service is used to register and query entities present in the system. The current directory service prototype is based on a NFS shared database but more sophisticated systems may be considered, like [7] that uses LDAP fundamentals.

ToCL primitives offer remote spawning of processes and threads that register themselves into the directory through specialized enroll primitives. Processe registration is preceded by initialization of the existent low-level communication subsystems.

The attributes of entities, most of the times, remain unchanged during the overall application execution. To minimize the impact of the directory services, ToCL uses process local caches. The use of local identifiers for threads also contributes to minimize thread creation latency.

2.2 Communication

ToCL programs do not explicitly create communication end-points; processes and thread identifiers are used as message origin and destination, following PVM and TPVM approaches [8].

Threads in the same process share the communication facilities through port multiplexing. This approach introduces latency overheads but it has the advantage of not compromising scalability, given that the number of communicating entities (threads) is not limited by low-level communication end-points.

ToCL is based on POSIX threads (actually Linux Threads); it does not implement thread facilities. Asynchronous events from low-level communication subsystems are managed by using a few threads of control. More complex strategies to integrate communication management and thread scheduling are presented in [10, 11]. However, these strategies depend on the developing of specialized thread libraries, thus compromising applications portability and reutilization.

3 GM and VIA Basics

GM was developed by Myricom to take advantage of Myrinet hardware. VIA standard defines an architecture for the interface between general high performance network hardware and computer systems. MVIA is a well-known VIA implementation that makes possible to experiment a few Gigabit and FastEthernet network adapters.

Although distinct on the interface and protocols they use, both GM and VIA are low-level approaches to high-performance message passing.

3.1 Zero-Copy Messaging

Both VIA and GM use regions of registered memory to achieve zero-copy messaging. Registering a memory region comprises pining the associated memory pages and informing networking hardware (or communication library middleware) about buffer addresses. This means that applications must register data buffers before calling send primitives.

Message reception only succeeds if the communication library was previously notified to use pre-registered buffers. These buffers must be large enough to hold the incoming data, otherwise the message will be discarded. A single buffer may be used to store several incoming messages if applications explicitly inform the library, after each message reception, to reuse that buffer. To manage variable size messages, applications usually allocate huge buffers wasting memory.

3.2 Message Addressing

GM applications use ports to send/receive data. The number of ports is usually limited to eight per network adapter, so we just have a few valid destinations per host (assuming a network interface per host). Messages are addressed to remote ports by means of a pair ⟨node number, port number⟩. Node numbers correspond to network interfaces and are assigned by GM network mapping tools while a port number is returned when the application opens a port using a specific network interface.

VIA is a connection oriented communication protocol that uses pairs of VIs (Virtual Interfaces) to connect remote entities (processes or threads). To send messages to a particular destination, applications must create a VI and request a connection to the remote destination using the VIA network address. Messages are addressed to local VIs, however each local VI is related to one only remote VI. In MVIA, VIs are limited to 1024 per network interface. A full-connected multi-threaded application using h hosts and t threads per host would require at least $t^2 \times (h - 1)$ VIs per host.

3.3 Sending and Reception

VIA uses descriptors for send and receive operations. Descriptors are registered memory regions containing information about the send/receive operations: buffer addresses, data length and other control information. On reception descriptors are processed according to a FIFO policy, thus making it hard to receive variable size messages. For example, notifying VIA to prepare reception of two different size messages requires knowledge about the order the messages will be sent.

GM library only needs to know the address and length of data. On reception, when multiple buffers are available they are used according to message needs, what makes it easy to deal with variable size messages.

Multithreaded applications may freely use VIA primitives – VIA is thread safe – but for GM it is necessary to use mutexes to handle concurrency.

4 ToCL Operation

ToCL offers a unique interface for the development of multi-networked and multithreaded cluster applications. It uses the basic functionality from the low-level communication libraries to not compromise the performance of the specific hardware and communication protocols.

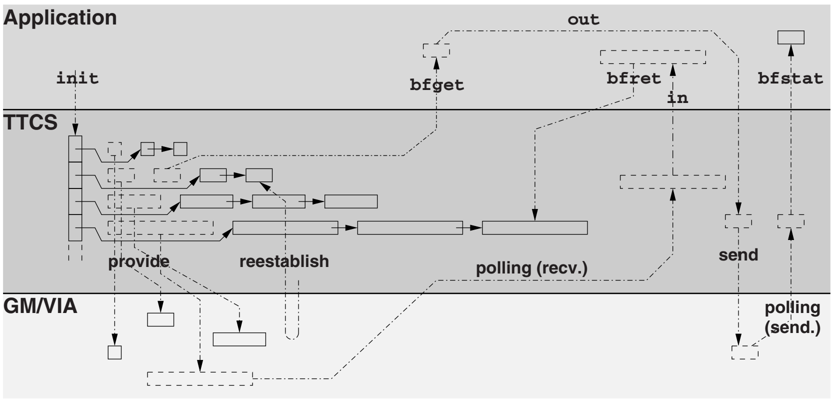

Fig. 1. Buffer management.

4.1 Buffer Management

ToCL manages a pool of pre-allocated registered buffers to ensure zero-copy communication. Because memory registering is a time consuming operation, buffers are created when ToCL initializes the underlying communication subsystems.

Applications must request buffers to store the data involved in a sending operation. To prepare itself for receptions, ToCL notifies the underlying communication subsystems to use pre-allocated buffers.

Figure 1 presents the buffer management mechanism used in ToCL.

Application Driven Operation. Applications must request specific buffers to store data before sending messages (fig.1:`bfget`). Because multiple low-level network libraries may be available, applications must select the target network (currently GM or VIA) when requesting buffers (by default the faster medium is used). The send operation selects the network that matches the buffer type.

To find out if a specific message was successfully transmitted, the status of the corresponding buffer must be checked (fig.1:`bfstat`). A specific buffer may then be reused for sending as long as the pending operation is terminated.

The receive operation returns to the application a library buffer (fig.1:`in`) that may also be reused to send data.

When a specific buffer (requested by the application or returned by a receive operation) is not needed anymore, the application should explicitly return it back to the ToCL library to be integrated in the buffer pool (fig.1:`bfret`). If a send buffer is intended to be used just once, the application may notify the library (before calling the send primitive) to automatically integrate it in the buffer pool after the sending completion.

Library Driven Operation. The ToCL library notifies GM and VIA of the availability of buffers for message reception (fig.1:provide).

When a message arrives (fig.1:polling (send.)), a buffer is used and the ToCL library must provide the communication sub-layer with a fresh buffer. If the buffer pool runs out of buffers, the ToCL library allocates and registers a new buffer to prevent message loss.

To avoid message discarding, whenever a message arrives before ToCL is able to provide a buffer, several buffers are provided in advance to the communication sub-layers. ToCL also exploits idle time to automatically reestablish buffer pool availability (fig.1:reestablish).

Buffer Sizes. For message sending the application requests appropriate size buffers according to data length. The ToCL library just adds a small extra space to the buffer to include message control data.

On message reception the library has to deal with variable length messages and buffer sizes are difficult to estimate.

A first possibility is to use buffers large enough to hold any possible message. This option requires large amounts of memory, unless data is copied from buffers to application memory and the same buffers are used to receive all messages.

ToCL choice is to define a maximum buffer size and to use buffers of 2^n bytes, to ensure that only a few different buffer sizes are handled. This way, for message reception, the library will provide the low-level communication subsystems, particularly VIA and GM, with at least one buffer for each allowed buffer size.

As an example, for a particular x bytes buffer request, a 2^y bytes buffer will be returned according to the formula $2^{y-1} \leq (x + msg.control) \leq 2^y$.

4.2 Message Dispatching

To ensure fair access to the multiple network facilities needed by application threads, ToCL uses a message dispatching mechanism that multiplexes the low-level communication subsystems.

Send and Receive Queues. Send and receive primitives executed by applications interact with the ToCL library through queues.

Messages are dequeued from the send queue and submitted to the low-level communication subsystems according to the buffer type. A FIFO queue is used for message sending, but other scheduling policies may be used to support priority messages.

ToCL uses a polling mechanism to monitor the low-level communication subsystems to detect the arriving of messages. New messages are enqueued into the receive queue and are indexed by source and tag. At application level the message interface may use the source and the tag of a message as a selection mechanism.

Low-level Transmission. ToCL communication model assumes that a thread can send a message to any other thread in a multithreaded multi-networked cluster environment, thus supporting thread oriented fine-grained communication and computation at application level. However, to deal with the expected

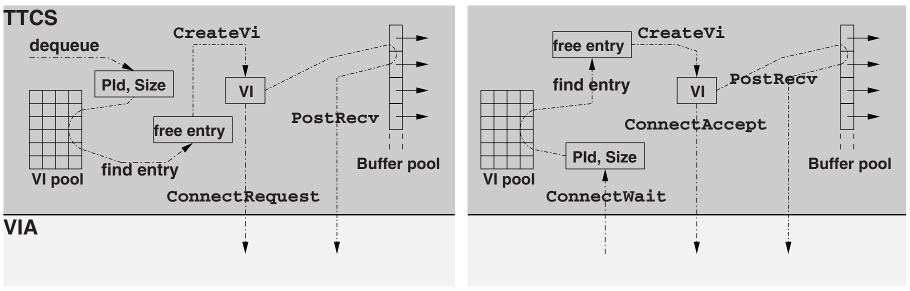

Fig. 2. Connection management for VIA.

large number of threads that applications may use, it creates one only low-level communication end-point for each process and low-level communication subsystem.

For each message dequeued from the receive queue it is then necessary to find the matching low-level subsystem end-point.

GM: Each process opens a port and announces its node and port identifiers by using a directory service. The directory service maps process global identifiers into GM ports. Local caches are used to minimize overhead.

GM is a connection-less protocol that allows any process to directly send a message to a target GM end-point. Messages to several destinations may be sent using the same port and messages from several origins may be received using one only port.

VIA: Each process announces itself using its network address and waits for connection requests. A network address is formed by the MAC address and a discriminator[3]. ToCL uses the global process identifier, provided by the directory service, as discriminator. The VIA network address allows the ToCL library to establish a connection to the right end-point and send the data (see below).

VIA Connections Details. Because connection establishment is a heavy operation it is mandatory to maintain persistent connections if we want to lower latency. The basic idea is to create a VI (virtual interface) and establish a connection to a specific remote process only once: the first time a message is sent to that process.

When using one only connection per remote process we have to deal with the following problem: how to receive variable size messages? In fact, notifying VIA to use a set of different size buffers for a particular VI is useless (see 3.3). To overcome this difficulty, ToCL uses a set of connections per remote process - one for each possible buffer size.

For each message to be sent, after dequeuing it from the send queue, the destination process identifier and the message size are used to find the matching

[3] Discriminators are similar to UDP or TCP ports but they may be any byte sequence.

connection (a VI entry) to the target process in the VI pool (fig.2[left]: find entry). In the absence of a connection to the target process, a new VI is created (fig.2[left]: `CreateVi`) and a connection request is issued to the remote process.

A connection request is addressed to the VIA network address registered in the directory service by the remote process. Apart from the request, the library also notifies VIA of the availability of buffers for future incoming messages, i.e. messages that will arrive after the connection is established.

For each process, ToCL uses a single thread to handle connection requests. Every time a request is detected (fig.2[right]: `ConnectWait`), a VI is created and stored in the VI pool. Note that connections are bi-directional, so that a VI may be used to send messages to the process that issued the connection request.

After notifying VIA of available buffers for message reception, ToCL also informs the requesting process that the connection was accepted (fig.2[right]: `ConnectAccept`).

Send and Receive. VIA uses descriptors to describe send and receive operations. Because both operations involve a buffer, ToCL creates a descriptor for each buffer and uses it as a buffer attribute.

After dequeuing a message, if the buffer type matches the VIA communication subsystem, ToCL uses the buffer descriptor to call the `VipPostSend` primitive. For GM, only the buffer address and the data length are needed to call the `gm_send` primitive. To handle sending success acknowledgments and set appropriately the buffer status flag, ToCL uses the GM callback mechanism or the VIA completion queue notifications.

To receive messages, ToCL polls periodically both communication systems; if a GM receive event is detected or if a descriptor is returned by the VIA completion queue then an enqueue operation is started.

4.3 Multi-Protocol Messages

ToCL has been designed to take advantage from the existence of multiple network adapters in cluster nodes. By using ToCL, applications may transparently exploit multiple low-level communication subsystems increasing communication bandwidth. These features are supported by a mechanism of message fragmentation that allows message fragments to travel independently through different communication media, reaching a destination end-point where they are reassembled in a solely message.

Fragment Dispatching. ToCL library distinguishes between two types of messages: short messages and long messages.

Short messages correspond to data stored in registered buffers and are transmitted at once using a single communication subsystem. The maximum size of short messages defaults to $\approx$32kbytes (VIA) and $\approx$64kbytes (GM).

Long messages are transmitted from regular user memory and require data copying.

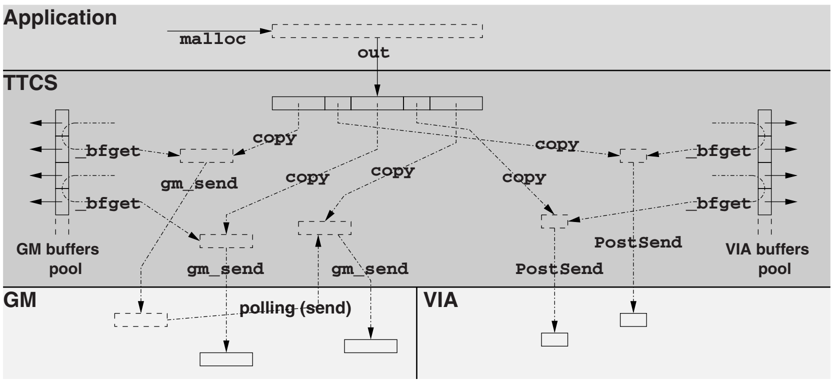

Fig. 3. Fragment dispatching.

After dequeuing a long message (whose buffer type is regular memory) the library first selects a communication subsystem, then searches for a free buffer in the buffer pool (fig.3: `_bfget`) and finally copies a message fragment to that buffer. Fragments are alternately sent using the available subsystems.

During fragment transmission, sending success acknowledgments may arrive (fig.3: polling send) and buffers used to send initial fragments may be reused to send other message fragments.

To make reassembling possible at destination, all fragments are tagged with the same message identifier along with fragment offsets and the originator process identifier.

5 Discussion

ToCL is an undergoing research being developed as a component of a major project aimed to design and to implement SIRe (a scalable information retrieval environment). SIRe basically will build a cluster architecture based on networked commodity workstations and high-performance networks to provide a large number of users with concurrent and efficient access to multiple text documents spread on cluster nodes.

Economics constraints and performance requirements lead us to develop ToCL as an intermediate-layer communication library, able to support a parallel and distributed multi-threading programming environment that can either scale up to provide higher performance or scale down to allow affordability or better cost effectiveness.

ToCL includes important features present in some of the most relevant communication libraries like Madeleine [4], an intermediate-level communication library used by PM^2 that supports multiple communication sub-systems, and PANDA which includes a specific layer to manage hardware heterogeneity. However, ToCL is unique in the use of existing POSIX threads libraries to built a fine-grained communication library where threads play the principal role.

A preliminary version of ToCL that exclusively supported GM, along with an application example and performance measurements, was presented in [1]. Performance tests were conduced using a small cluster environment consisting of four dual Pentium III (733MHz) machines, interconnected by Myrinet technology (LANai 9, 64bits/66MHz interfaces) .

More recently, a new communication library was devised, in order to support multiple communication subsystems. Presently a new suite of tests is being conduced to evaluate the overall design and performance. We plan to evaluate ToCL VIA support over Gigabit, and compare its performance to GM, by using another small cluster, consisting of four dual Athlon (1.8GHz) machines interconnected by Gigabit technology (SysKonnect SK-9821 interfaces).

References

1. A. Alves, A. Pina, J. Exposto, and J. Rufino. Scalable Multithreading in a Low Latency Myrinet Cluster. In *VECPAR'02*, 2002.
2. R. Bhoedjang, T. Rühl, and H. E. Bal. LFC: A Communication Substrate for Myrinet. In *4th Conf. of the Advanced School for Computing and Imaging*, 1998.
3. R. Bhoedjang, T. Rühl, R. Hofman, K. Langendoen, and H. Bal. Panda: A Portable Platform to Support Parallel Programming Languages. In *SEDMS IV*, 1993.
4. L. Bougé, J. Méhaut, and R. Namyst. Madeleine: Efficient and Portable Communication Interface for RPC-based Multithread Environments. In *PACT'98*, 1998.
5. Y. Chen, X. Wang, Z. Jiao, J. Xie, Z. Du, and S. Li. MyVIA: A Design and Implementation of the High Performance Virtual Interface Architecture. In *IEEE Int. Conference on Cluster Computing*, 2002.
6. Compaq Computer Corp., Intel Corporationnand & Microsoft Corporation. Virtual Interface Architecture Specification, 1997.
7. S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. A Directory Service for Configuring High-Performance Distributed Computations. In *6th Int. Symposium on High Performance Distributed Computing*, 1997.
8. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* Scientific and Engineering Computation. MIT Pres, 1994.
9. P. Geoffray, L. Prylli, and B. Tourancheau. BIP-SMP: High Performance Message Passing over a Cluster of Commodity SMPs. In *SC'99*, 1999.
10. Hansen, J. & Jul, E. Latency Reduction using a Polling Scheduler. In *Second Workshop on Cluster-Based Computing*, pages 27–31. ACM, 2000.
11. Langendoen, K., Romein, J., Bhoedjang, R. & Bal, H. Integrating Polling, Interrupts, and Thread Management. In *6th Symp. on the Frontiers of Massively Parallel Computing*, 1996.
12. Myricom. *The GM Message Passing System*, 2000.
13. R. Namyst and J. Méhaut. PM^2: Parallel Multithreaded Machine. A computing environment for distributed architectures. In *ParCo'95*, 1995.
14. National Energy Research Scientific Comp. Center. M-VIA: A High Performance Modular VIA for Linux. http://www.nersc.gov/research/FTG/via/index, 2002.
15. S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing 95*, 1995.
16. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - The Complete Reference.* Scientific and Engineering Computation. MIT Pres, 1998.