MPICH-CM: a Communication Library Design for a P2P MPI Implementation

Anton Selikhov, George Bosilca, Cecile Germain, Gilles Fedak, Franck Cappello

Laboratoire de Recherche en Informatique - CNRS and Université de Paris-Sud,
Bâtiment 490, F-91405 Orsay Cedex, France
{selikhov|bosilca|cecile|fedak|fci}@lri.fr

Abstract. The paper presents MPICH-CM – a new architecture of communications in message-passing systems, developed for MPICH-V – a MPI implementation for P2P systems. MPICH-CM implies communications between nodes through special Channel Memories introducing fully decoupled communication media. Some new properties of communications based on MPICH-CM are described in comparison with other communication architectures, with emphasis on grid-like and volunteer computing systems. The first implementation of MPICH-CM is performed as a special MPICH device connected with Channel Memory servers. To estimate the overhead of MPICH-CM, the performance of MPICH-CM is presented for basic point-to-point and collective operations in comparison with MPICH p4 implementation.

1 Introduction

Global Computing (GC) and Peer-to-Peer (P2P) [1] systems gather unprecedented processing power from borrowing time of idle computers. From the number of processor criterion, a GC system is thus an ideal infrastructure to run massively parallel applications. Nevertheless, only limited attempts have been done towards parallel programming on such systems.

Parallel execution models have been designed with a traditional machine model in mind, which can be summarized as strongly coupled: machines are reliable, and information flows reliably across computing entities (processes or threads). On the other hand, GC systems are extreme representatives of distributed systems. Failures are very frequent, for instance when a user reclaims his machine or unplugs his laptop. Failures are the worst case of faults, in that they are also perfectly (quite) unexpected: the machine simply disappears, leaving the system in whatever state it can be.

Message-passing in such a highly unreliable environment needs to address transparent virtualization as the basic issue: stable user-level MPI processes and logical communication endpoints must be implemented on top of ephemeral processes running in a GC system; a reliable communication protocol must be run on a fuzzy set of tasks. On top of this virtualization system, fault-tolerance can be achieved through execution rollback: tasks are checkpointed and lost ones are restarted from a process image saved in a stable storage [2].

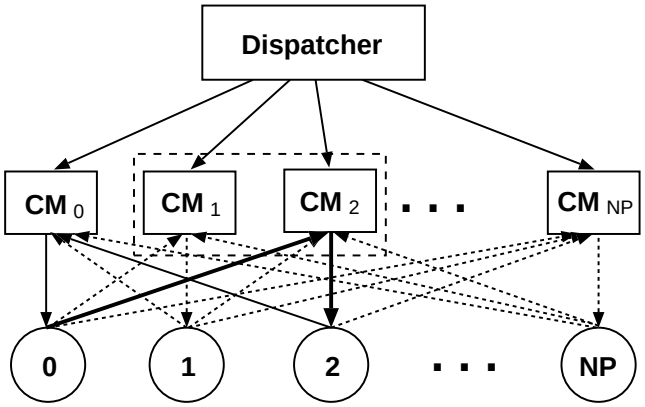

Fig. 1. MPICH-CM architecture components and their communication structure. Bold lines show a virtual channel from node 0 to node 2. The dashed box shows two Channel Memories hold by one CM server

All these issues are reflected in implementation of MPICH-V system [3], which the MPICH-CM communication library has been developed for. This paper presents only the virtualization layer: MPICH-CM, the communication layer of a GC parallel system. MPICH-CM is a full-fledged MPI implementation based on MPICH; it is highly distributed, thus is adapted to a P2P context, and fully asynchronous, being able to cope with non-existent target endpoints. Execution rollback, described in [3] is outside the scope of the paper; however, we will sketch in section 4 how the architecture allows for easy communication recovery.

2 MPICH-CM Architecture

MPICH-CM is based on the concept of communication tunneling, where a persistent communication channel is built on top of a variable architecture [4]. However, the variability tackled here is not at the level of transfer protocol, but at the level of endpoints. MPICH-CM mediates communication through *Channel Memories* (CM). The CMs act as proxies for sent messages, and repositories for the messages not yet delivered.

Fig. 1 describes the overall MPICH-CM architecture. The Dispatcher is responsible for mapping 1) the global MPI objects (parallel applications and Communicators), to sets of tasks running on Nodes, 2) Channel Memories servers to Nodes. The last mapping is *destination oriented*: each Node is associated with one CM, called its *owner CM*, which stores the messages en route to this Node.

Channel Memory servers handle communication requests, actually decoupling the communicating nodes. Besides, whatever may be the firewall in front of the nodes, they only need to communicate with a CM Server. While a parallel node may fail unexpectedly, CM servers are expected to be stable, therefore al-

ways reachable (meaning that a failure is fatal). Currently, this is implemented through dedicated servers; whether the CM service should be provided by volatile resource, or kept for stable ones, it is an instance of the general open question of flat vs hierarchical internal organization for P2P systems [5].

The Dispatcher distributes CMs to CM servers at the initialization time through an Application Channel Memory Map (ACMMap). The ACMMap actually translates MPI Communicators into CM ownership: one CM Server needs only to know about its owned nodes, and the nodes which may send messages to one of its owned nodes; thus, the CM Server in-degree is determined by its out-degree (number of owned nodes), which is a decision of the Dispatcher, and by the communication topology of the application. For instance, a 5-points stencil application (2D grid topology) gives a in-degree bounded above by 4 times the number of owned nodes (in-degree). On the contrary, fig 1 shows the CM communication structure associated to a fully connected application topology: each node connects to all CMs, as it has to communicate with all the other nodes; as stated before, only one CM is input for any node. Knowing the communication requirements of an application, for instance from its Communicators, the Dispatcher can tune the resources (storage and server nodes) allocation accordingly.

3 The Node Library

MPICH-CM is nearly transparent to the user: the only requirement is to link with the MPICH-CM library, which is in charge of interfacing with the Channel Memory system. The MPICH-CM library is built in the regular MPICH manner, through a device implementing the Chameleon Interface functions *PIbsend*, *PIbrecv* (for blocking communications with Channel Memory), *PInprobe*, *PIfrom*, *PIiInit* and *PIiFinish*. This *ch_CM* device is thus at the same level as ch_p4.

3.1 Initialization and Finalization

When a parallel application is dispatched, each MPI process calls the *ch_CM* implementation of the *PIiInit* function. *PIiInit* connects to the owner CM Server, and receives the ACMMap. The next step is synchronization: before starting to communicate, the region of the CM set that the node will have to deal with must be ready. Thus, the node registers to all its destination CMs, resulting in resources allocation at the CM level, and creation of connections. *PIiInit* returns only when the node has received acks from all CMs it has registered to.

The connections established with CM servers during this phase are TCP sockets open until the application finishes communication by calling *MPI_Finalize*.

At the user level, *MPI_Finalize* notifies all processes about the end of the MPI communication structure. The device-dependent *PIiFinish*, which task is to shut down the device, sends a system message (see next subsection) to all CM servers to notify them and closes all corresponding sockets.

3.2 Communication

The implementation of all communication functions follows the same scheme. First, a *system message* notifies a CM about the type of communication and the properties of the message to be sent or received (*source*, *destination*, *size*, *tag* and *kind*). Next, the message body (PIx buffer) is sent or received. System messages are used by CM Servers 1) to select the appropriate handler for the next request 2) to know about its parameters. We named them System and not Control messages, to be not confused with MPICH high level Control messages.

All the communications with the CM server are passed through blocking *write* and *read* on UNIX sockets.

PIbsend copies its *tag*, *length* and *to* parameters to the corresponding fields of a system message, fills *source* information with its MPI rank and sends the message to a CM server, which is selected according to the destination rank using the ACMMap. The message body follows using the *buffer* parameter of the *PIbsend*.

PIbrecv uses the *tag* parameter to determine the source of the message to be received (when the source is important and for non-rendezvous protocol of communications), sets the *tag* and *length* fields and sends it to the owner CM server. Next, it waits to receive a system message to know about the source and the size of the body, which is received immediately after this.

PIfrom returns the value of the *source* field of the last system message received by *PIbrecv*.

PInprobe uses the *tag* parameter and the rank number to fill the corresponding fields of a system message. It sends the message to the owner CM and receives information about existence of messages in the CM.

4 The Channel Memory Architecture

A CM is basically a data structure devoted to message storage. A CM Server interfaces one or multiple CMs to other components (Nodes and Dispatcher). The main functions of a CM Server include accepting requests to handle new applications from a Dispatcher, accepting connections from the nodes involved in the application and managing all communications (handling multiple CMs). To make the best use of the available bandwidth, a CM Server is multi-threaded. A pool of thread is allocated at init time, and at each instant, only one thread is waiting on *select* for any activity on all known sockets. Due to lack of space, we focus on communication handling, skipping all init phases.

An essential property of the Channel Memory protocol is existence of intermediate message queues. This differs from the p4 device, where all messages are retrieved from a channel and queued locally. Therefore, the protocol involves two overheads: queue management and doubling the number of communications. While the second one cannot be avoided, the first one is decreased by special data structures, and tuned message storing and retrieving mechanisms.

4.1 Data storage structures

According to MPICH specifications [6], passing any user data breaks down into Control Message(s) and, maybe, a Data message, depending on the actual size of the user data (very short messages are included in the Control message body). A CM Server stores these two types of MPICH messages, for each owned node, in a separate Control queue, and an array of data queues, which has n queues, where n is the whole number of nodes in the communication group. An index of a data queue in the array corresponds to the rank of source of these data messages. Both Control queue and Data queue are organized as FIFO. This organization allows to keep the time ordering of messages issued from a particular node and to retrieve Data messages by source rank. The first property means that all the messaging events are fully logged (data and temporal ordering), allowing communication replay [3]. The combination of the two properties provides constant time data message retrieval.

4.2 Queues Management

Handling of request to communication depends on the *kind* of request received in a system message.

The *PInprobe* handler tests the Control queue corresponding to this node and sends the result.

Upon a request to *receive* a Control message, the first message (if exists) is retrieved from the Control queue and sent to the node; otherwise the node is marked as "waiting for a Control". Receiving Data message includes an attempt to retrieve the first data message in the queue according to the requested source and, like in the case of Control messages, leads to either sending the data message found to the node, or marking the node as "waiting for a Data".

Upon a request to *send* a message, whether Data or Control, to the owned destination node, the message is first buffered. Next, if the destination client is "waiting for a Control (Data)", the message is directly sent to the destination node; otherwise it is stored in the Control (Data) queue.

5 Performance Evaluation

Performance tests results were obtained for both p4 and CM- based MPICH on the LRI cluster of PCs (500 to 700MHz), connected through a switch with 100 Mbit/s maximal throughput for each node. The aim of the tests was to explore the overhead involved by the CM architecture and the architecture scalability. All measurements, except the last one, are averages over one hundred runs.

5.1 RTT performance

First, the round-trip time (RTT) for blocking communication was measured for two nodes, communicating through two CM servers (Fig.2, left graph), which is

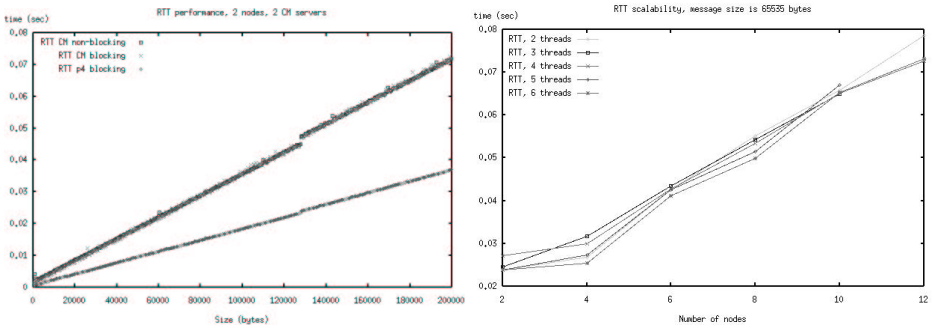

Fig. 2. RTT vs message size (left graph) and number of nodes on one CM (right graph)

the best situation (one CM server per Node). MPICH-CM reaches nearly half of the performance of the p4 based MPICH. Moreover, the linear behavior and the slope show that the performance is strictly determined by the underlying TCP bandwidth. Thus, the overhead incurred in the CM server itself is negligible.

Second, the scalability of a CM Server was investigated on the base of RTT measurements (Fig.2, right graph). In this test, only one CM Server was used to manage all CMs. The nodes were divided in pairs, each pair communicating independently. The quasi-linear behavior above 4 nodes follows the bandwidth limitation. However, the threaded architecture helps for low in-degree: from 2 to 4 nodes, and 4 threads, the latency is nearly constant.

5.2 MPI_Alltoall performance

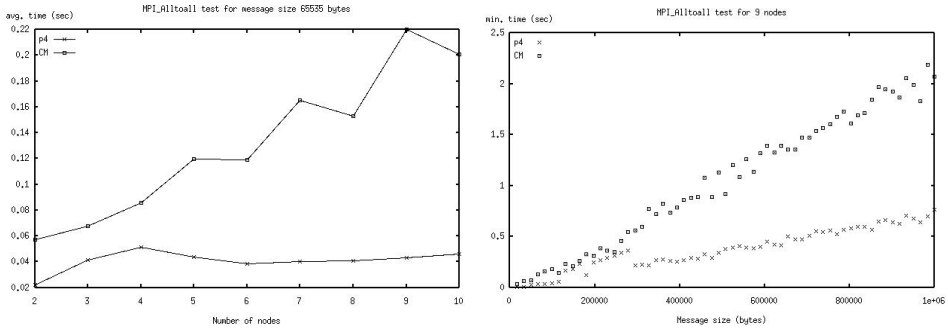

Fig. 3. MPI_Alltoall vs number of nodes managed by one CM server (left graph), and vs message size (right graph)

Estimation of *MPI_Alltoall* time is used often for performance analysis of multiprocessor systems and clusters. It involves very intensive communications and allows to estimate the scalability of the system being tested. For MPICH-CM based system, the effect of message size and number of nodes was measured. In both cases, only one CM Server was used to manage all CMs. The reported times are measured on node 0, with no special synchronization between consecutive calls to the Alltoall routine, besides the one involved by the operation itself.

The increasing times for *MPI_Alltoall* operation presented on Fig.3, left graph, has two reasons: first, the overhead of message transition through a CM, illustrated in the first experiment on Fig. 2 and second, the overhead of managing all the nodes by only one CM server, which is illustrated in the Fig.2.

Fig.3 (right graph) presents the results of measurement of *MPI_Alltoall* minimal time for 9 nodes with increasing size of message being sent. For the same reason of having only one CM server for managing all CMs, the increase of time for this operation is faster than in the case of RTT measurements.

6 Related Work

The Channel Memory approach has two motivations: communication decoupling and communication events logging. The first one has been pioneered by Linda [7]. Besides that, most of Linda features are related to a high-level parallel programming language, based on a shared-memory abstraction for interprocess communications, implemented through associative search, while MPICH-CM is a pure message-passing system with named communicating entities. The project closest to MPICH-V is MPI-FT [8]. It uses a monitoring process, the observer, for providing MPI level process failure detection and recovery mechanism based on message logging. Our work is a step more in the same direction, as it merges communication decoupling and logging. Many other works [9, 10], tackles the issue of fault-tolerance by exposing the faults and the computation state at the application level. Yet another way has been explored by the Condor team [11], by global distributed checkpointing following the Lamport algorithm.

7 Conclusion and Future Work

This paper has presented MPICH-CM, a communication library designed for MPICH-V, a MPI implementation for GC and P2P systems. The experimental results have shown 1) predictable and explainable limitations in its performance; 2) scalability if one is willing to provide enough resource for communication, that is enough CM Servers. This may appear costly; however, a network of volatile or unreliable nodes may become a more common substrate when considering very large clusters, or clusters or clusters, running during quite long computing time. In this case, it may be more efficient to allocate resources for ordered logging of communication events, which is the core of the CM architecture, than to restart execution of all nodes many times from the very beginning and to hope successfulfinalizing. Other salient features of MPICH-CM are:

- Firewall bypass. All communications between nodes are initiated by a node and addressed to a specialized port of a CM Server, not directly to a peer. Thus, both communicating peers may stay behind firewalls
- Effective pipelining. Channel Memories may be used for the creation of an effective pipelined system by forwarding on the fly messages on their way from one node to another. This may be useful for some specialized video-processing systems.
- Fully asynchronous communication. Neither the sending nor the receiving node is required to wait until the end of the communication. All communication requests are absorbed by the Channel Memories, allowing nodes with different communication throughput to work with maximal performance.

We are currently integrating MPICH-CM inside the XtremWeb Global computing platform [12, 13] developed at LRI, together with a checkpoint and recovery facility. The complete system will provide a framework for transparent parallel execution of MPI programs on volunteer-based resources.

References

1. Oram, A. (ed.): P2P: Harnessing the Power of Disruptive Technologies. O'Reilly (2001)
2. Elnozahy, E., Jonhnson, D., Wang, Y.: A Survey of Rollback Recovery Protocols in Message-Passing Systems. CMU TR-96-181. Carnegie Mellon University (1996)
3. Bosilca, G. et al.: MPICH-V: Parallel Computing on P2P systems. To appear in IEEE-ACM SC2002: High Performance Networking and Computing (2002)
4. Al-Khayatt, S. et al.: A Study of Encrypted, Tunneling Models in Virtual Private Networks. 4th IEEE Conf. in IT and computing & coding (2002)
5. Kan, G.: Gnutella. In: Oram, A. (ed.): P2P: Harnessing the Power of Disruptive Technologies. O'Reilly (2001)
6. Gropp, W., Lusk, E., Doss, N., Skjellum. A.: A High-performance, Portable Implementation of the MPI Message Passing Interface Standard. Parallel Computing, Vol. 22, (1996) 789-828
7. Bakken, D. E., Schlichting, R. D.: Supporting Fault-Tolerant Parallel Programming in Linda. IEEE Trans. on Paral. and Distrib. Systems, Vol. 6(3), (1995) 287-302
8. Louca, S. et al.: MPI-FT: a Portable Faut Tolerant Scheme for MPI. Parallel Processing Letters, Vol.10(4). World Scientific, New Jersey London Singapore Hong Kong. (2000) 371-382
9. Fagg, G., Bukovsky, A., Dongarra, J.: Harness and Fault Tolerant MPI. Parallel Computing. North-Holland. Vol. 27(11), (2001) 1479-1479
10. Agbaria, A., Friedman, R.: Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. 8th IEEE Int. Symp. on High Perf. Dist. Comp. (1999)
11. Pruyne, J., Livny, M.: Managing Checkpoints for Parallel Programs. Workshop on Job Scheduling Strategies for Parallel Processing, IPPS'96. IEEE Press. (1996)
12. Fedak, G., Germain, C., Neri, V., Cappello, F.: Xtremweb: A Generic Global Computing Platform. IEEE/ACM CCGRID'2001. IEEE Press. (2001) 582-587
13. Germain, C., Fedak, G., Neri, V., Cappello, F.: Global Computing Systems. 3rd Int. Conf. on Scale Scientific Computations, Lecture Notes in Computer Science, Vol.2179. Springer-Verlag, Berlin Heidelberg New York. (2001) 218-227