CollGM - A Myrinet/GM optimized collective component for Open MPI

Torsten Hoefler[1,2] Marek Mosch[2] Torsten Mehlan[2]
Wolfgang Rehm[2]

[1]Open Systems Laboratory, Indiana University, Bloomington IN 47405, USA
htor@cs.indiana.edu

[2]Technical University of Chemnitz, Chemnitz, 09107 GERMANY
{htor,mosm,tome,rehm}@cs.tu-chemnitz.de

Abstract

The Open MPI collective framework offers a way to implement hardware-specific collective operations for Open MPI. We used this framework to develop a Myrinet/GM collective component by combining common knowledge of the implementation of collective algorithms with GM protocol optimized techniques to achieve highest performance. Our results show that the good performance of the existing point-to-point based tuned collective implementation in Open MPI can be improved with the use of these techniques.

1 Introduction

Cluster systems dominate, due to their excellent price-performance ratio, today's high performance computing (HPC) market[1]. Especially small and mid-sized cluster systems are built from commodity components. However, commodity interconnection networks like Gigabit Ethernet are often not able to deliver the required communication performance. Thus, special cluster interconnection networks are often used to connect workstations to a cluster system.

One of those specialized networks is Myrinet [1], distributed by the company Myricom. It has been analyzed in detail by Qian et al. in [14]. Myrinet is defined in the ANSI standard document ANSI/VITA 26-1998 [18]. It offers special features, such as low-latency, cut-through switching, communication offload flow control and continuous link monitoring, that are not common in commodity Ethernet networks. Those features are especially beneficial in the context of HPC applications. Myrinet also supports large systems with switches that can connect up to 512 nodes.

[1]cf. Top 500 list 06/2007

Two versions of Myrinet are currently available, Myrinet 2000 and Myrinet 10G. For Myrinet 2000, two main Application Programming Interfaces (APIs) are available, the Glenn's Messages (GM) API [12] and the Myrinet Express (MX) [11] API. Both are fundamentally different. The GM API resembles a Virtual Interface Architecture (VIA) [3] and the MX API is closer to the Message Passing Interface (MPI) standard [8, 9]. Myricom decided to discontinue the support for the GM API, but there are still many systems that run on this well proven and stabilized API (e.g., Europe's currently fastest Supercomputer Mare Nostrum, the 256 CPU "Strider" cluster at the High Performance Computing Center Stuttgart or the 16 CPU "Oscar" cluster at the Technical University of Chemnitz).

While it seems natural to map MPI point-to-point operations to the MX API (which offers non-blocking point-to-point functionality similar to MPI), it is not that obvious for collective communication. The similarity of MX and MPI suggests that the potential of a low-level implementation compared to the existing highly optimized collective component (based on point-to-point messages) is very low. Thus, we explore the possibility to implement collective operations directly on top of the low-level GM API to use the full semantics for collective algorithm design.

Our main theses for optimization potential in GM are:

1. special GM optimized algorithms (e.g., n-ary trees)
2. special handling of memory registration/deregistration
3. optimized small/large message handling
4. avoiding the overhead of the point-to-point messaging layer in MPI (PML/BTL, see Section 3)

5. optimized message forwarding (uses pre-registered buffers to forward messages)

The remaining document is structured as follows. The GM API is described in detail in Section 2. We sketch our implementation using the semantic advantages of the GM API in Section 3 followed by benchmark results in Section 4. Conclusions and Future Work are presented in Section 5.

2 The Myrinet/GM API

The sole vendor of Myrinet hardware – Myricom – provides several software stacks for interfacing the network. As discussed in Section 1, the programmer can choose between two APIs, GM[2] and MX. Each of these alternatives comprises a user space library, an operating system driver and a firmware intended to run on the network adapter's processing unit. The two versions of GM and MX are incompatible since they use different wire–protocols. We are going to present a short overview about the GM API.

The GM software stack provides the GM Mapper for automatic network discovery. This entity detects network switches and hosts and calculates the routing information. Thus, any user application can assume that the network is properly configured. Applications usually access the network via the functions of the library. These functions serve as interface to the operating system driver as well as to the hardware directly (bypassing the operating system).

Each application has to connect to the hardware by opening a so called "port" to access further services of GM. The port works as software context and logical root of any other resources that may be needed to communicate over the network. The logical network port belongs to a specific application and is not accessible from other applications. GM–1 provides maximum 8 ports while GM–2 can provide maximum 16 ports. Figure 1 shows the organization of GM ports.

The basic concept behind the GM interface is based on queues. Each port provides one send queue, one receive queue and one event queue. The application performs queue operations to send and receive data. The protocol is connectionless and guarantees lossless reliable in order transmission of all data. The working principle of Myrinet, mainly the simple hardware design of the switches, does not permit the native support of multicast and broadcast messages. However, GM allows zero–copy transmission of data. Thus, the virtual addresses of the user space has to be translated into physical addresses before any data transmission can take place. The DMA engine of the network adapter has to issue physical addresses while the user application only has knowledge of the virtual addresses.

[2]two versions of GM exist, GM–1 and GM–2, we concentrate on the more recent GM–2 in this paper

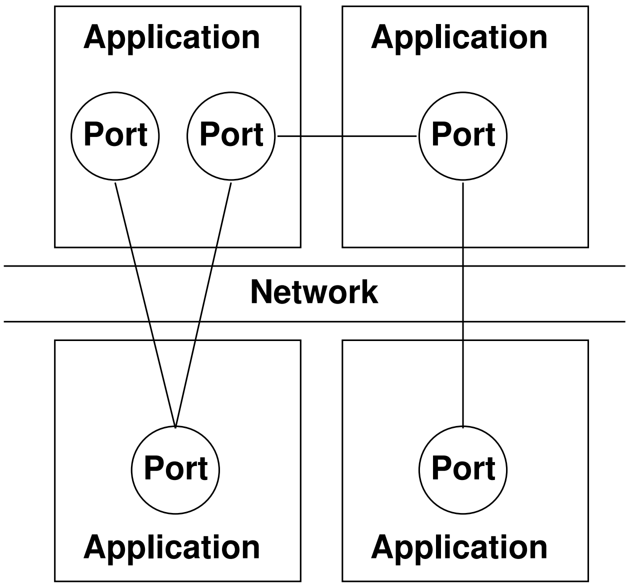

Figure 1. Communication ports of GM

The functions of the GM library reflect the attributes of the GM protocol described so far. The application opens and closes a port by calling `gm_open()` or `gm_close()` respectively. The send and receive buffers have to be registered with the library before use, which effectively pins the pages in memory (disables swapping). Memory registration is performed by the functions `gm_register_memory()` or `gm_dma_alloc()`. The latter function allocates and registers memory in one step. De-registration of memory is done by the functions `gm_deregister_memory()` and `gm_dma_free()`. The function `gm_send_with_callback()` appends one send request to the send queue and returns immediately. The real data transfer is done by the network adapter's DMA engines while the host CPU is free to do other work.

To prepare the receipt of a message the application has to post a receive descriptor to the receive queue that describes a buffer in memory. This is done by a call to `gm_provide_receive_buffer_with_tag()`. On completion of the receive request a receive event becomes available in the event queue. Once the data transfer is complete or some error happened a corresponding event descriptor is put into the event queue of the related port. The application calls either `gm_receive()` or `gm_blocking_receive()` to get the next event from the event queue. Until data transfer completion the application must not change the contents of a send buffer or must not make any assumptions on the content of a receive buffer. In case of successful data transmission the application can access the data buffer. The send buffer may be changed in any way and the receive buffer is guaranteed to contain valid data. Figure 2 shows the message reception process.

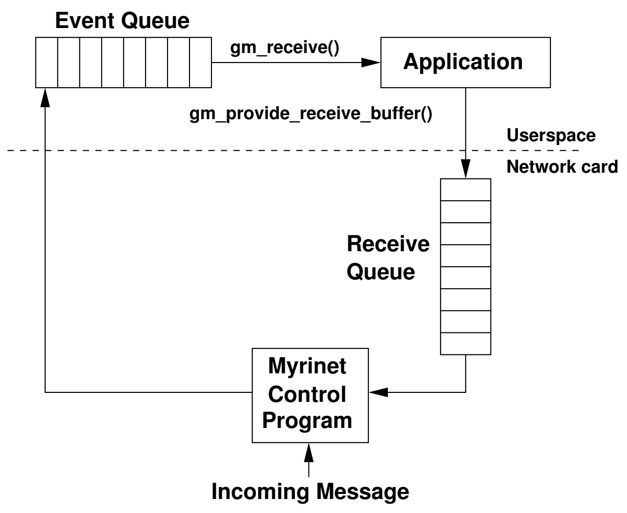

Figure 2. Reception of a message in GM

The Myrinet GM user interface supports Remote Direct Memory Access (RDMA) as well. Using RDMA the application can directly write to remote memory or read from remote memory. The receive queue of the remote application is not involved in any way. Thus, the only way for the remote application to notice an RDMA write operation is reading the memory where it expects the data. Myrinet does not support any access restrictions for RDMA operations. Thus any process is able to read and write the registered memory of all other processes with open Myrinet ports.

A comparison between Myrinet GM and the more recent technology InfiniBand [17] shows, that both network technologies use fairly similar concepts. The user interfaces of both, Myrinet GM and InfiniBand, are based on queues. The application has to post requests to the send queue or receive queue respectively. The completion of requests is signaled via a completion queue. The difference between Myrinet GM and InfiniBand is the addressing scheme. Myrinet GM uses a connectionless mechanism while InfiniBand provides both, end–to–end connections and connectionless datagrams. One should note that the connectionless datagram service of InfiniBand works differently from the mechanism of Myrinet GM. Finally both network technologies require registered memory for data transfers. The process of memory registration and deregistration in InfiniBand [10] and Myrinet GM is very similar.

3 Implementation of CollGM

This section describes the implementation of the collgm component.

3.1 Open MPI Structure

Open MPI [5] founds on the Modular Component Architecture (MCA), that provides a flexible way for defining frameworks [16]. For instance there are frameworks handling point–to–point messages, datatype conversion, collective communication and many other tasks. One single framework defines an interface that may be implemented by several components. In this work, The collective framework serves as target for the implementation of a component that handles the MPI functions MPI_Barrier, MPI_Bcast, MPI_Scatter[v], MPI_Gather[v] and MPI_MPI_Alltoall[v] over Myrinet GM in an optimized way. We refer to this component as the collgm component in the following discussion.

3.2 The collective GM component

The collgm component is divided into two parts which are explained in the following.

The MPI level The MPI level handles the entire semantic of the collective operation. This part is responsible for:

- Copying (packing) the actual data of complex data types into consistent memory areas
- Selecting the appropriate protocol for sending data
- Handling the protocol transactions, such as message segmentation
- Selecting an appropriate algorithm that is considered optimal
- Management of memory registration and deregistration

In contrast the GM level of the collgm module provides basic communication services that encapsulate the underlying GM user interface:

- Provide reliable data transfer functions either blocking or non–blocking
- Encapsulate the underlying protocol
- Save messages that can not be processed at time of reception for later processing

The GM level The GM level hides the upper layer from several implementation details of the GM protocol. There is a hard limitation of the number of ports the applications can use. Thus, we decided to use only one port. Consequently there is only one send queue, one receive queue and one event queue. Any incoming message results in an entry to this event queue. Since the precise ordering of incoming messages can not be controlled by the application, some of these events have to be saved for later processing. An example situation occurs while performing a blocking send operation. The completion of the send operation is reported through the event queue. Thus, the GM level of the collgm module polls this queue. It may happen that an unexpected receive event is reported first. This event has to be saved for further processing during an impending receive operation.

Moreover the GM interface maintains a pool of tokens to provide some basic flow control. It is the application's responsibility to acquire a token before every send or receive request. The GM level of the collgm module checks the availability of tokens before the actual send or receive operation starts. In case of a lack of tokens any blocking send or receive operation blocks until a token becomes available. The non–blocking functions report an appropriate error.

3.3 Communication Protocols

The MPI level of the collgm module implements two types of protocols. The eager protocol avoids synchronization between sender and receiver and the rendezvous protocol performs a handshake before the actual data transmission starts. The eager protocol works with preregistered memory and the MPI level of the collgm module is in charge of copying the data from the source location into a buffer of registered memory. This memory consists of segments of 64 kByte size and larger data blocks have to be segmented. This helps to improve performance because the first segment can be processed by the network adapter while the remaining data is copied into subsequent segments. Figure 3 shows the measurements with Netgauge [7] of a transmission of 32 MByte data with different segment sizes. The best performance is reached with a segment size of 64 kByte.

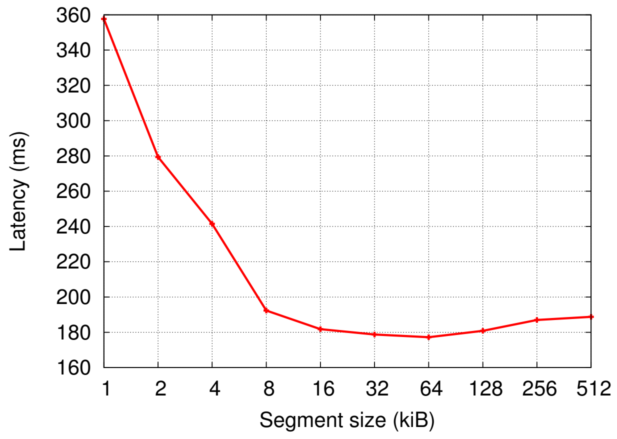

Figure 3. Optimal eager segment size

A limit on the number of segments to be transmitted prevents the sender from flooding another node. If a large message needs more segments to be transmitted, the sender has to send a request and has to wait for a positive response. This request is sent at the beginning of the data transfer increasing the probability that a positive acknowledgment arrives before the data transfer has to be interrupted. The scheme is shown in Figure 4. The rendezvous protocol does not copy the data into local memory. Instead the original memory area is registered on the fly at the sender. Also the receiver has to register the appropriate memory area. Thus, two messages are needed to announce the pending data transfer and to report the receiver's memory address back to the sender. In order to avoid any confusion with entries in the receive queue belonging to eager protocol messages, the rendezvous protocol makes use of the RDMA feature of GM. The initiator of the RDMA transaction directly writes into the remote memory. Moreover the protocol does not register the entire memory area at once. Blocks of 128 kByte size are registered in a pipeline fashion (cf. [15]). The data transfer starts as soon as registration finished. While the data transfer progresses the collgm module is able to start the next memory registration. Due to the time consuming nature of memory registration this helps to partially hide the registration process behind the data transfer. The principle is shown in Figure 5.

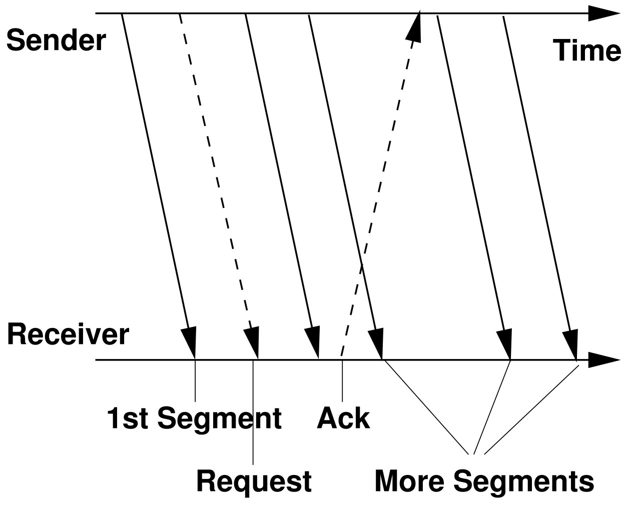

Figure 4. Flow control of large messages

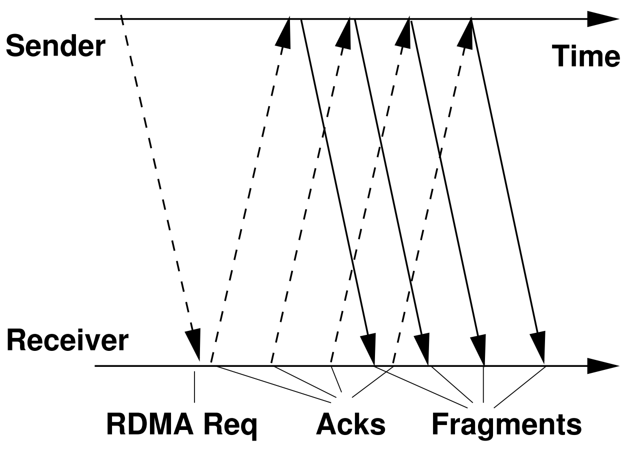

Figure 5. RDMA transfer of large messages in a pipeline fashion

3.4 The Algorithms of the MPI level

The Myrinet GM interface does not provide native broadcast or multicast features. Thus the collgm module relies completely on point–to–point messages to implement the collective communication. Many algorithms are available in this area. Thus, we performed measurements in order to decide which algorithm is beneficial for Myrinet/GM and which algorithms are suitable for different data sizes and communicator sizes. The barrier implementation relies on recursive doubling and the algorithm of Bruck [2]. The benchmark section shows that the results are very close to the [4] module of Open MPI.

The options for implementing the broadcast operation are a flat tree (linear), binomial tree, a binary tree, a splitted binary tree [13] and a pipeline. Measurements show that very small messages benefit from a binomial tree. Small to standard size messages should use the splitted binary tree and large messages show the best performance with the pipeline scheme.

The operations scatter and gather may be accomplished by a flat tree or a binomial tree. In a flat tree scheme the root node of the scatter operation sends the messages directly to each of the receivers. Accordingly during a gather operation the root node receives all messages directly from the source node. The binomial tree algorithm applies some message aggregation at nodes in the middle of the binomial tree. Measurements show that for small messages the binomial tree works best and for large messages (1 kByte or larger) the flat tree performs best.

Message forwarding Collective operations are often implemented on top of MPI point-to-point functions as in the tuned module of Open MPI. Network technologies like Myrinet or InfiniBand require registered memory for data transfers. Thus, each MPI point-to-point function has to copy the user data into a preregistered memory area or register/de-register the user buffer on the fly. This design can lead to performance loss when nodes have to forward messages as in the pipeline broadcast algorithm. The message forwarding works as follows: First, MPI_Recv copies the received message into the specified user buffer. The following MPI_Send function must copy the message again, this time from the user buffer to a preregistered memory area[3]. The collective functions of the collgm module have direct access to the transfer buffers. This has several advantages. A received message can be forwarded immediately by performing a send request using the preregistered receive buffer. Further on the message is copied only once (into the user buffer) while the network adapter already sends the data to other node(s). The broadcast algorithms of the collgm module make extensive use of this technique.

4 Microbenchmark Results

We benchmarked our implementation on the strider system at the High Performance Computing Center Stuttgart (HLRS). This cluster system consists of 125 dual 2Ghz Opteron compute nodes connected by Myrinet 2000 running the GM-2 API. We analyze alltoall, broadcast and scatter/gather operations for small (16) and large node counts (64) running with a single process per node. All benchmarks have been conducted with NBCBench [6].

4.1 Small node counts

Figure 6 shows the MPI_ALLTOALL performance on 16 nodes. Both the Open MPI tuned implementation used a hard-coded hand-tuned map of algorithms to use for every combination of communicator and data size. The map of the fastest algorithms (also comparing to OMPI/tuned and MPICH-GM) for alltoall is displayed in Figure 7. This map was used to hard-code the algorithm selection in the collgm collective component.

Figure 8 shows MPI_Broadcast performance measurement results. A similar map as for alltoall has been benchmarked for broadcast and is shown in Figure 9.

MPI_SCATTER results are shown in Figure 10. Results for MPI_Gather are due to the similar implementation completely identical and omitted here. The algorithm selection map in Figure 11 shows the optimal algorithm for every node-count/data size combination.

Results for 64 nodes of the strider system for alltoall and Scatter/Gather are shown in Figure 12 and 13 respectively.

[3]assuming no zero-copy implementation

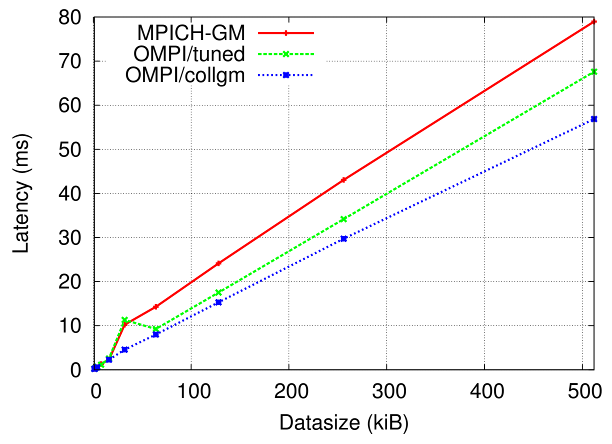

Figure 6. Alltoall results on 16 nodes

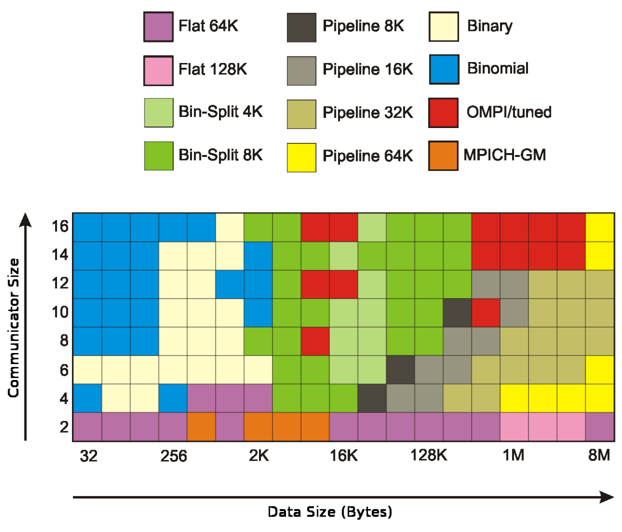

Figure 9. Broadcast algorithm selection map

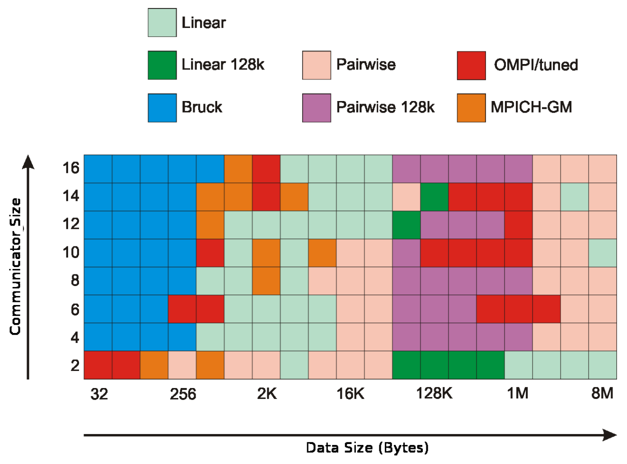

Figure 7. Alltoall algorithm selection map

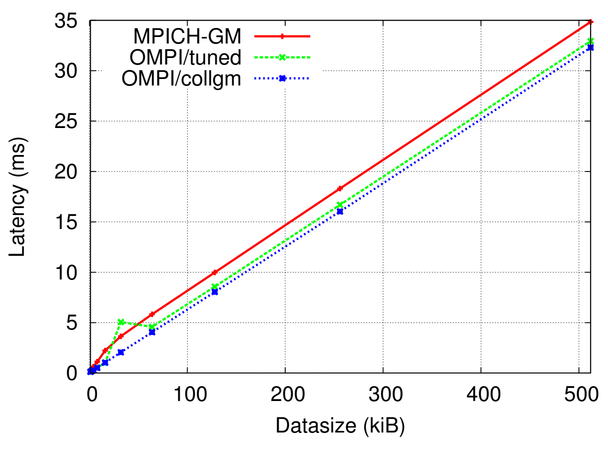

Figure 10. Scatter results on 16 nodes

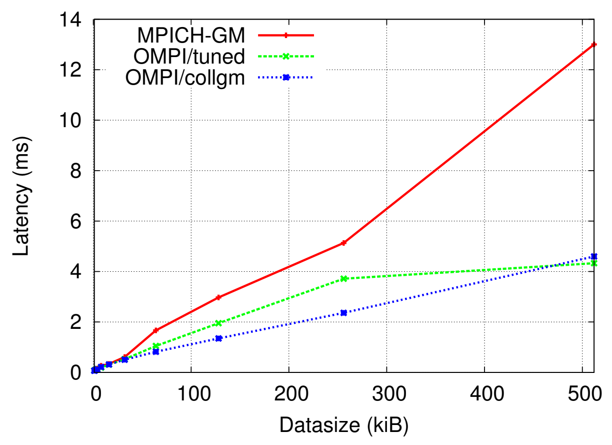

Figure 8. Bcast results on 16 nodes

The alltoall Benchmark aborted with GM errors with all three implementations when run on 64 nodes.

5 Conclusions and Future Work

Our work is the first extensive collective implementation that uses the advantages of the Open MPI MCA structure to optimize collective communication for a specific networking hardware. We showed with the Myrinet/GM interface that a performance benefit can be achieved with this approach. We combine common knowledge of the implementation of collective communication operations with GM protocol specific techniques to achieve the best performance on Myrinet/GM cluster systems. However, we are not sure

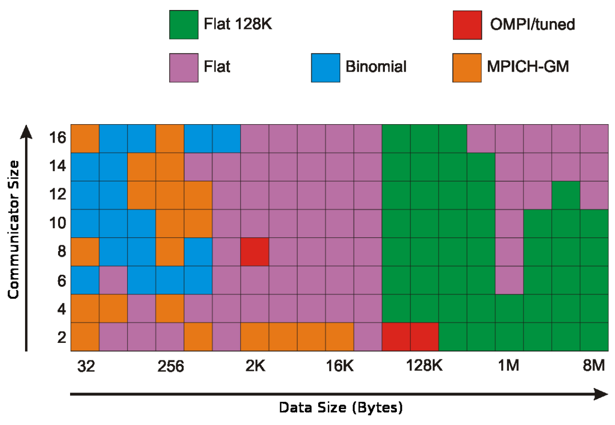

Figure 11. Scatter algorithm selection map

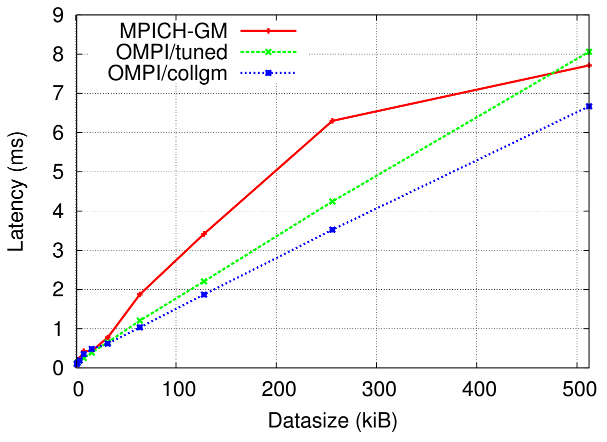

Figure 12. Bcast results on 64 nodes

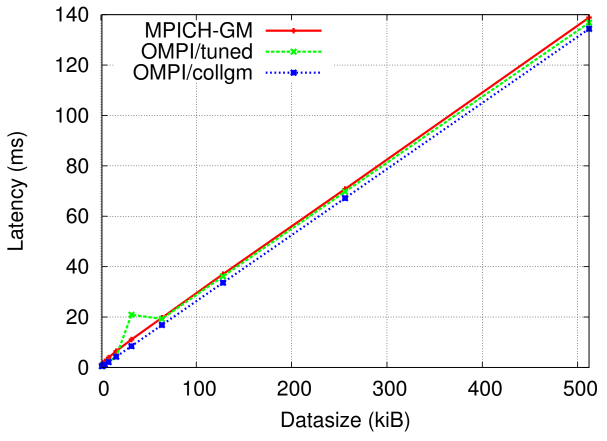

Figure 13. Scatter results on 64 nodes

if the software-technological effort and the implementation costs outweigh the relatively high effort of designing, implementing and maintaining the collgm component.

References

[1] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):29–36, 1995.

[2] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. In *Transactions on Parallel and Distributed Systems*. IEEE Computer Society, 1997.

[3] D. Cameron and G. Regnier. *The Virtual Interface Architecture*, 2002.

[4] G. E. Fagg, J. Pjesivac-Grbovic, G. Bosilca, T. Angskun, J. J. Dongarra, and E. Jeannot. Tuned: An open mpi collective communications component. In *Distributed and Parallel Systems*, pages 65–72. Springer US, 2007.

[5] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004.

[6] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. 11 2007. Accepted for publication at the Supercomputing 2007 (SC07).

[7] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm. Netgauge: A Network Performance Measurement Framework. 9 2007. Accepted for publication at the High Performance Computing Conference 2007.

[8] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. 1995.

[9] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface. Technical Report, University of Tennessee, Knoxville, 1997.

[10] F. Mietke, R. Baumgartl, R. Rex, T. Mehlan, T. Hoefler, and W. Rehm. Analysis of the Memory Registration Process in the Mellanox InfiniBand Software Stack. In *Euro-Par 2006 Parallel Processing*, pages 124–133. Springer-Verlag Berlin, 8 2006.

[11] Myricom. *A High Performance, Low-Level, Message-Passing Interface for Myrinet.* Myricom, 2006.

[12] Myricom. *GM: A message-passing system for Myrinet networks*. Myricom, 2006.

[13] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance Analysis of MPI Collective Operations. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium, 4th International Workshop on Performance Modeling, Evaluation, and Optimization of Parallel and Distributed Systems (PMEO-PDS 05)*, Denver, CO, April 2005.

[14] Y. Qian, A. Afsahi, and R. Zamani. Myrinet networks: A performance study. In *NCA '04: Proceedings of the Network Computing and Applications, Third IEEE International Symposium on (NCA'04)*, pages 323–328, Washington, DC, USA, 2004. IEEE Computer Society.

[15] G. M. Shipman, T. S. Woodall, G. B. andRich L. Graham, and A. B. Maccabe. High performance RDMA protocols in HPC. In *Proceedings, 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, Bonn, Germany, September 2006. Springer-Verlag.

[16] J. M. Squyres and A. Lumsdaine. The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms. In *Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, St. Malo, France, 2004.

[17] The InfiniBand Trade Association. *Infiniband Architecture Specification Volume 1, Release 1.2*. InfiniBand Trade Association, 2004.

[18] VITA Standards organization. *Myrinet-on-VME, Protocol Specification*. VITA Standards organization, 1998.