A practically constant-time MPI Broadcast Algorithm for large-scale InfiniBand Clusters with Multicast

Torsten Hoefler,[1,2] Christian Siebert,[1] and Wolfgang Rehm[1]

[1]Dept. of Computer Science
Chemnitz University of Technology
Strasse der Nationen 62
Chemnitz, 09107 GERMANY
{htor,chsi,rehm}@cs.tu-chemnitz.de

[2]Open Systems Laboratory
Indiana University
501 N. Morton Street
Bloomington, IN 47404 USA
htor@cs.indiana.edu

Abstract

An efficient implementation of the MPI_BCAST *operation is crucial for many parallel scientific applications. The hardware multicast operation seems to be applicable to switch-based InfiniBand cluster systems. Several approaches have been implemented so far, however there has been no production-ready code available yet. This makes optimal algorithms to a subject of active research. Some problems still need to be solved in order to bridge the semantic gap between the unreliable multicast and* MPI_BCAST. *The biggest of those problems is to ensure the reliable data transmission in a scalable way. Acknowledgement-based methods that scale logarithmically with the number of participating MPI processes exist, but they do not meet the supernormal demand of high-performance computing. We propose a new algorithm that performs the* MPI_BCAST *operation in a practically constant time, independent of the communicator size. This method is well suited for large communicators and (especially) small messages due to its good scaling and its ability to prevent parallel process skew. We implemented our algorithm as a collective component for the Open MPI framework using native InfiniBand multicast and we show its scalability on a cluster with* 116 *compute nodes, where it saves up to* 41% MPI_BCAST *latency in comparison to the "TUNED" Open MPI collective.*

1-4244-0910-1/07/$20.00

1 Introduction

Cluster systems gain, due to their very good price-performance ratio, more and more importance for scientific applications. More than 72% of all supercomputers in the 28th TOP500 list [22] are cluster systems. The Message Passing Interface (MPI, [11, 12]) emerged in the last years as today's de facto programming model for parallel high-performance applications on such systems. Within this model, the use of collective operations is crucial for the performance, performance portability among different systems, and the parallel scaling of many applications [5]. That means that collective operations deserve special attention to achieve the highest throughput on those cluster architectures. We investigate the widely used collective operation MPI_BCAST in this work (Rabenseifner identified it as one of the most time-consuming collective operations in his usage analysis [18]). This operation provides a reliable data-distribution from one MPI process, called root, to all other processes of a specific communication context that is called communicator in MPI terms. Since only the semantics and the syntax of this function are standardized, we are able to present an alternative implementation. We show a new scheme for InfiniBand™ with the use of special hardware support to achieve a practically[1] constant-time broadcast operation.

A key property of many interconnects used in cluster systems is the ability to perform a hardware-supported multicast operation. Ni [14] discusses the advantages of hardware multicast for cluster systems and concludes that it is very important for cluster networks. This feature is very common for Ethernet-based systems and is supported

[1]practically in the meaning of average case for a wide variety of application patterns; more details in Section 2.1

by the TCP/IP protocol suite. Other widely used high-performance networks like Myrinet or Quadrics use similar approaches to perform multicast operations [4, 23, 24, 25]. The new emerging InfiniBand™ [21] network technology offers such a hardware-supported multicast operation too. Multicast is commonly based on an unreliable datagram transport that broadcasts data to a predefined group of processes in almost constant time, i.e., independent of the number of physical hosts in this group. Multicast groups are typically addressed by network-wide unique multicast addresses in a special address range. The multicast operation can be utilized to implement the MPI_BCAST function, however, there are four main problems:

1. the transport is usually unreliable
2. there is no guarantee for in-order delivery
3. the datagram size is limited to the maximum transmission unit (MTU)
4. each multicast group has to be network-wide unique (i.e., even for different MPI jobs!)

We examine and resolve all those problems in this article and introduce a fast scheme that ensures reliability and makes the implementation of MPI_BCAST over InfiniBand™ viable.

The terms (MPI) process and node are used throughout the paper. We consider a "process" similarly to MPI as an activity that may join a multicast group and a "node" as a physical machine. Although multiple processes are allowed on a single physical machine, we used only one process per node for our benchmarks in Section 4. Node-locally, the underlying communication library is responsible to deliver the received hardware multicast datagrams efficiently to all registered processes. Furthermore, this work addresses mainly cluster systems with flat unrouted InfiniBand™ networks. We assume and we show with our benchmarks that the multicast operation finishes in an almost constant time on such networks. However, the ideas are also applicable to huge routed networks, but the hardware multicast might loose its constant-time property on such systems. Anyhow, it is still reasonable to assume that the hardware multicast operation is, even on routed InfiniBand™ networks, faster than equivalent software-initiated point-to-point communication.

The next section describes related work and shows the most critical problems with those existing approaches. We propose new ideas to solve the four main problems (described above) in Section 2. Our implementation for the Open MPI framework is described in the following Section 3. A performance analysis of the new broadcast is presented in Section 4 before we give a conclusion and an outlook to future work in Section 5.

1.1 Related Work

Some of the already mentioned issues have been addressed by other authors. However, all schemes use some kind of acknowledgement (positive or negative) to ensure reliability. Positive acknowledgements (ACK) lead to "ACK implosion" [8] on large systems. Liu et al. proposed a co-root scheme that aims at reducing the ACK traffic at a single process. This scheme lowers the impact of ACK implosion but does not solve the problem in general (the co-roots act as roots for smaller subgroups). The necessary reliable broadcast to the co-roots introduces a logarithmic running time. This scheme could be used for large messages where the ACK latency is not significant. Other schemes, that use a tree-based ACK, do also introduce a logarithmic waiting time at the root process. Negative acknowledgement (NACK) based schemes do usually not have this problem because they contact the root process only in case of an error. However, this means that the root has to wait, or at least store the data, until it is guaranteed that all processes have received the data correctly. This waiting time is not easy to determine and usually introduces unnecessary process skew at the root process. Sliding window schemes can help to mitigate the negative influence of the acknowledgement-based algorithms, but they do not solve the related problems.

Multicast group management schemes have been proposed by Mamidala et al. [10] and Yuan et al. [26]. Both approaches do not consider having multiple MPI jobs running concurrently in the same subnet. Different jobs that use the same multicast group receive mismatched packets from each other. Although errors can be prevented by using additional header fields, a negative performance impact is usually inevitable.

Multicast group management should be done with the standardized MADCAP protocol [6]. However, the lack of available implementations induced us to search for a more convenient scheme.

The multicast operation has also been applied to implement other collective operations like MPI_BARRIER, MPI_ALLREDUCE or MPI_SCATTER [2, 7, 9]. We use the scheme proposed in [19] for MPI_BCAST and adapt it for the use with the InfiniBand™ multicast technology.

2 The Multicast-Based Broadcast Algorithm

Several multicast-based broadcast algorithms have been proposed in the past. The most time-critical problem, especially for smaller broadcast messages, is the re-establishment of the reliability which is needed by MPI_BCAST but usually not supported by hardware multicast. We propose a two-stage broadcast algorithm as illustrated in Figure 1. The unreliable multicast feature of

the underlying network technology is used in a first phase to deliver the message to as many MPI processes as possible. The second phase of the algorithm ensures that all MPI processes finally receive the broadcast message in a reliable way, even if the first stage fails partially or completely.

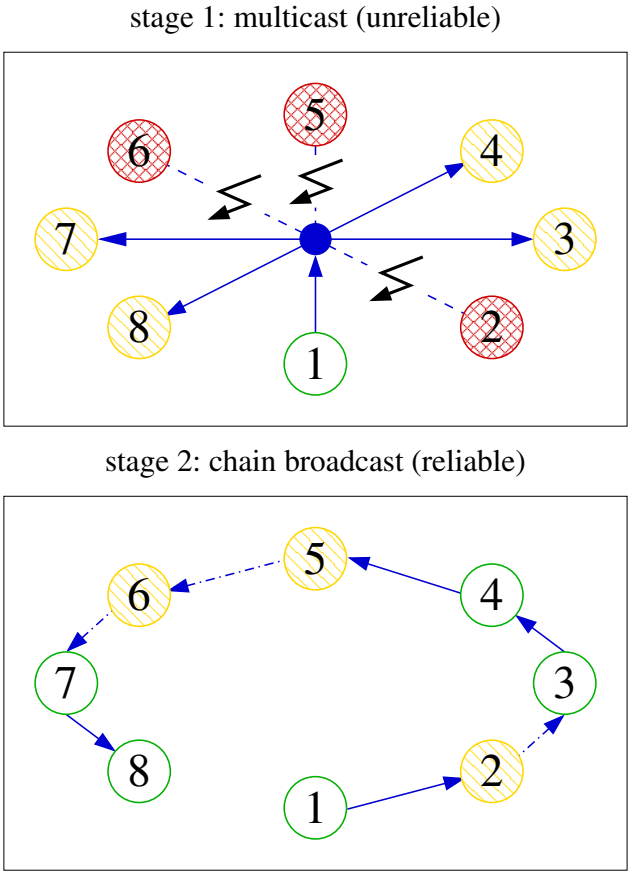

Figure 1. The two-stage broadcast algorithm

2.1 Stage 1: Unreliable Broadcast

Multicast datagrams usually get lost when the corresponding recipient is not ready to receive them or due to network congestion. Therefore, a common approach is to use a synchronizing operation (similar to MPI_BARRIER) that waits until all P processes are prepared to receive the datagrams. If such an operation is build on top of reliable point-to-point communication this synchronization will need $\Omega(log\ P)$ communication rounds to complete. Instead of targeting at a 100% rate of ready-to-receive processes, it is more than sufficient if only a subset of all MPI processes is already prepared, provided that a customized second stage is used for the broadcast algorithm. A further disadvantage of such a complete synchronization operation is the fact that real-world applications are usually subject to the principle of process skew which can lead to a further increment of the operation's time consumption.

It can be shown that a wide variety of applications works perfectly without any synchronization operation during this stage. However, when the root process is the first process that calls MPI_BCAST, all non-root processes would not be ready to receive the multicast message and therefore an immediately executed multicast operation might become totally useless. This remaining fraction of applications, with such a worst-case broadcast usage pattern, can be handled by explicitely delaying the root process. A user-controlled delay variable (e.g., MCA parameter for Open MPI) is not only the simplest solution for implementors of this algorithm, but also very effective because an optimal value for a given application can be determined using a small number of test runs. Adaptive delay parameter adjustments at runtime, e.g., based on heuristic functions, might be feasible too. A randomized single-process synchronization (instead of a complete MPI_BARRIER synchronization) is a third solution to this problem: a seed value is distributed at communicator creation time to all MPI processes. Within each MPI_BCAST operation, a certain non-root process is chosen globally with the help of a pseudorandom number generator and the current seed. The root process than waits until this single non-root process joins this collective operation. On average, such a procedure prevents the worst broadcast scenarios and is thereby independent of the application type. However, the first solution (without any delay) offers naturally the highest performance for applications where the root process rarely arrives too soon.

The first phase of the new broadcast algorithm starts with this optional root-delay and uses multicast to transmit the complete message (fragmenting it if necessary) from the root process to all recipients. A process-local status bitmap can be utilized to keep track of correctly received data fragments.

2.2 Stage 2: Reliable Broadcast Completion

Even without any preceding synchronization, it is not unusual that a large proportion (typically about 50%) of all MPI processes have correctly received the broadcast message during the unreliable broadcast stage. The third synchronization method ensures this 50% proportion in the average case even if the application processes always arrive in the worst-case broadcast pattern. This second stage of our new algorithm guarantees that those MPI processes which have not yet received the data (whether partially or completely) will accomplish this eventually. The common approach is to use some kind of acknowledgement scheme to detect which processes have failed and to retransmit the

message to these recipients. Unfortunately, existing ACK schemes (positive or negative ones) are quite expensive because of the introduced performance bottleneck at the root process and the necessary time-out values.

Instead of using this kind of "feedback" channel, which can be efficient for large messages where those overheads are negligible, it is more efficient for smaller messages to send the message a second time using a fragmented chain broadcast algorithm. This means that every MPI process has a predefined predecessor and successor in a virtual ring topology. The root process does not need to receive the message because it is the original source of this broadcast. Therefore, the connection with its predecessor (e.g., $8 \rightarrow 1$ in Figure 1) is redundant and can be omitted. As soon as a process owns a correct fragment of the broadcast message, it sends it in a reliable way to its direct successor. Whether a fragment has been received via multicast or via reliable send is not important - the second receive request can be cancelled or ignored.

Using this technique, each MPI process that gets the message via multicast serves as a new "root" within the virtual ring topology. After forwarding this message to its single successor, a process can immediately finalize its broadcast participation. Only those processes that have failed to receive the multicast datagram(s) need to wait until they get the message in the second stage. If its predecessor received the message via multicast then only a single further message transfer operation, called "penalty round" in the following, is necessary. But the predecessors might have failed too in the first stage and the number of "penalty rounds" would increase further. For a given failure probability ϵ of a single message transmission, the chance that P processes fail in a row is ϵ^P. Therefore, the average number of "penalty rounds" is reasonably small (given that $\epsilon = 50\%$, the number of penalty rounds is just 1.0). Nevertheless, the worst-case (i.e., all processes failed to receive the multicast message) leads to a number of "penalty rounds" (and therewith time) that scales linearly with the communicator size. However, real-world applications that call MPI_BCAST multiple times are mainly affected by the average case time and only minor by this worst-case time.

A different kind of virtual distribution topology (e.g., a tree-based topology) for the second stage could help to reduce this worst-case running time. However, with the knowledge about the applications broadcast usage-pattern or a proper synchronization method, this worst-case scenario will rarely occur. While a process in the virtual ring topology needs to forward the message only to a single successor, a process in a virtual tree-based topology would need to serve several successors (e.g., two in a binary tree) which usually increases the time for the second stage by this fan-out factor. In addition, the broadcast duration per process would not be as balanced as in the proposed chain broadcast. When a single MPI process enters the collective operation late, it can not delay more than one other process in the ring topology but it will delay all its direct successors in a tree-based topology.

3 Implementation Details

Our prototype is implemented as a collective component within the Open MPI [3] framework. The component uses low-level InfiniBand™ functionality to access the hardware multicast directly for the first stage of our algorithm (cf. Section 2.1). The reliable message transmission in step 2 uses the send/receive functionality of the Point-to-point Management Layer (PML) of the Open MPI framework.

3.1 Multicast Group Management

A single cluster system is often used by several independent MPI jobs. Thus, a proper multicast group management is necessary to prevent multicast group collisions. An ideal solution to provide a global (i.e., cluster-wide) multicast group management would be the implementation of a server-based allocation protocol like MADCAP [6], with a single master server for every cluster system. However, to the best of the authors knowledge, there is no MADCAP implementation available that is able to handle InfiniBand™ multicast global identifiers (MCGIDs).

Our approach is to statically assign a network-wide unique multicast group to every new communicator at creation time (cf. [26]). As alternative to the MADCAP solution, we choose these groups at random using a cryptographically secure pseudorandom number generator ("BBS" [1]). This does not implicate any performance problems because our unoptimized implementation is able to produce $52\ KiB$ of random data per second on an ordinary $2\ GHz$ computer, which is sufficient considering that only 14 bytes are necessary for each new InfiniBand™ MCGID. This generator is seeded at application startup using a pool of collected data from the following sources:

1. dynamic data from a high-resolution time stamp (MPI_Wtime)
2. inter-node data derived from host-specific identifiers (MPI_Get_processor_name; this resolves the concurrency problem)
3. intra-node altering data using temporary file names (tmpname; e.g., needed when some MPI processes reside on the same node)
4. other sources like the */dev/urandom* Linux device (only if available)

This randomized selection of multicast groups with due diligence makes collisions much more unlikely than, e.g., a catastrophic hardware failure because of the large MCGID address space of $m = 2^{112}$. If there are n multicast groups in use, the probability that no collisions occur at all is $Prob(n, m) = \frac{m!}{m^n \cdot (m-n)!}$ according to the "Birthday paradox". This will be almost 1.0 in practical scenarios, e.g., $99.999999999999999999999999\%$ for 1000 groups that are concurrently in use.

3.2 Fragmentation and Packet Format

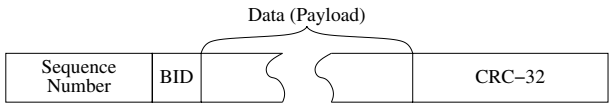

Figure 2. Structure of a packed multicast datagram

All multicast datagrams, that are used by our broadcast implementation, start with a four byte header containing a sequence number and a broadcast identifier ("BID", see Figure 2). Hardware multicast has a limited datagram size and does not guarantee in-order delivery. This makes a fragmentation of large messages necessary and the sequence number indicates the corresponding position of every fragment within the data buffer. The broadcast identifier corresponds to a communicator-specific counter that keeps track of the last issued broadcast operation. Individual broadcast messages can potentially pass each other because our algorithm does not explicitly synchronize; the BID field in the header prevents any possible mismatches. A final cyclic redundancy check value is used to detect defective datagrams. Although this was meant to be optional, we observed during our benchmarks that CRC errors occured with 0.287% of all transmitted datagrams on our test systems.

3.3 Implementation in Open MPI

The framework for the modular architecture and the collective components in Open MPI is described in detail in [20]. A context-wide agreed multicast group is assigned to each communicator within the `module_init` function that is called during the creation of a new communicator. All involved processes join this new group and build one unreliable datagram queue pair (UD QP) per communicator for the transmission of the multicast packets. Because the protocol for UD is connectionless, a single QP is sufficient and any scalability problems are avoided.

To further improve the implementation, we pre-post n receive requests (RRs) in `module_init` to buffer n incoming multicast packets (they are dropped if no pre-posted RR is found). This adds a constant memory overhead of $n \times MTU$ per MPI process. Useful values for n are highly system and even application dependent. We chose a default value of $n = 5$ to achieve a good balance between memory overhead and multicast performance. As long as there are pre-posted receive requests at the non-root processes, they will very likely get the next broadcast message during the first stage of the algorithm (at least if it is small enough to fit in the pre-posted buffers) even if they call MPI_BCAST after the root process had issued the multicast operation. Therefore pre-posting can also help to diminish the effect of the mentioned worst-case scenarios in real-world applications.

The `bcast` function itself decides at runtime upon the current scenario (message size, number of processes and user parameters) if it should execute an existing broadcast from the "TUNED" component, or our new multicast-based broadcast (described in Section 2).

During the destruction of every communicator the `module_finalize` function is called. All resources that were previously allocated by the init function are freed here.

Our InfiniBand™ API-independent macro layer was used to access both the OpenFabrics (formerly known as OpenIB) and the MVAPI interface with the same source code. We proved in another study [13] that the introduced overhead is negligible and the programming effort to support both interfaces is simplified.

4 Performance Evaluation

We evaluated our implementation with the Intel Microbenchmark suite version 3.0 (formerly known as Pallas Microbenchmark [16]) and a second, more realistic, microbenchmark that uses the principles mentioned in [15]. The following results show a comparison of our collective component called "IB" with the existing "TUNED" component in Open MPI 1.2b3. We focus on small messages because their performance is extremely important for the parallel speedup of strong scaling problems (a constant problem size with an increasing number of processes causes small messages) and the new broadcast is especially suited for this use case.

4.1 Classical Implementation

The classical way to implement MPI_BCAST for small messages uses point-to-point communication with the processes arranged in a virtual tree topology. However, for very small communicators a linear scheme, where the root process sends its message to all ranks sequentially, might be faster. Therefore, the used Open MPI "TUNED" component leverages such a linear scheme for communicators

with at most 4 MPI processes and a binomial tree distribution scheme for larger communicators. The binomial tree communication is staged into $\lceil log_2 P \rceil$ rounds and (assuming that the root process has rank #0, which can be acomplished by a rank rotation) each MPI rank sends every round to the MPI rank with the rank $\#round - 1$ greater than its own. The resulting communication pattern for a communicator of size 8 is shown in Figure 3.

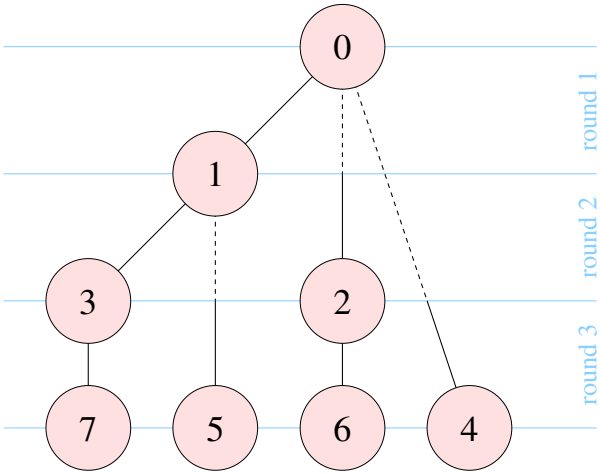

Figure 3. A classical binomial tree broadcast scheme for 8 MPI processes, as used by the "TUNED" implementation in Open MPI

A broadcast along a binomial tree has two main disadvantages. First, the communication time is clearly unbalanced when the communicator size is not a proper power of two. And even if it is, the root process might return immediately from a call to MPI_BCAST when the outgoing messages are cached by the underlying communication system (e.g., in eager buffers), while the last process (e.g., rank #7 in Figure 3) needs $\lceil log_2 P \rceil$ communication rounds to complete the operation (cf. [17]). This introduces an artificial process skew regardless of the initial process skew (the MPI processes might have been completely synchronized). Second, the overall duration of the broadcast operation increases logarithmically with the number of participating MPI processes. Contrary, our new broadcast algorithm is able to overcome both disadvantages.

4.2 Benchmark Environment

Our measurements have been executed on the "Odin" cluster that is located at Indiana University. This system consists of 128 compute nodes, each equipped with dual 2.0 GHz Opterons 246 and 4 GB RAM. The single Mellanox MT23108 HCA on each node connects to a central switch fabric and is accessed through the MVAPI interface. We used one MPI process per node for all presented runs. Our implementation has also been tested successfully on different smaller systems using the MVAPI and OpenFabrics interface.

4.3 Results

The results gathered with the IMB and a 2 $byte$ message are shown in Figure 4. The small-message latency is, as expected, independent of the communicator size[2]. Our implementation outperforms the "TUNED" Open MPI broadcast for communicators larger than 20 processes with the IMB microbenchmark.

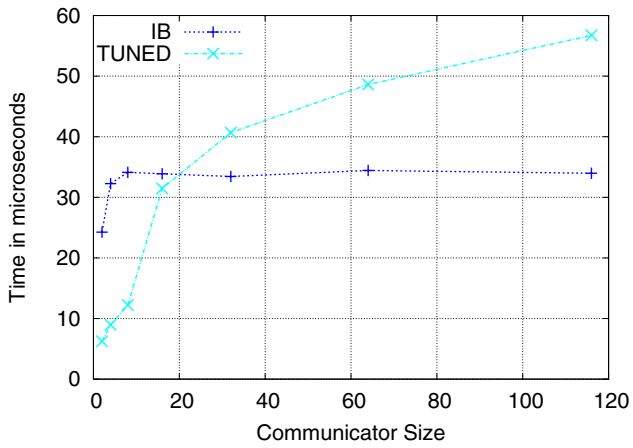

Figure 4. (IMB) MPI_BCAST latency in relation to the communicator size

For this reason, our collective module calls the "TUNED" component if the communicator contains less than 20 MPI processes (this value is system-dependent and therefore adjustable by the user with an Open MPI MCA parameter to tune for the maximum performance).

Our own (more comprehensive) broadcast benchmark gives a detailed insight into the performance of the new implementation. We measured the time that every single process needs to perform the MPI_BCAST operation with a 2 $byte$ message. The result for a fixed communicator size of 116 is shown in Figure 5. It can be seen that the "TUNED" broadcast introduces a significant process skew (rank #1 finishes 79.4% earlier than rank #98), which can have a disastrous impact on applications that rely on synchronity or make use of different collective operations (that cause dif-

[2] the IB outlier with two processes exists because the virtual ring topology is split up before the root process (this optimization saves a single send operation at the last process in the chain)

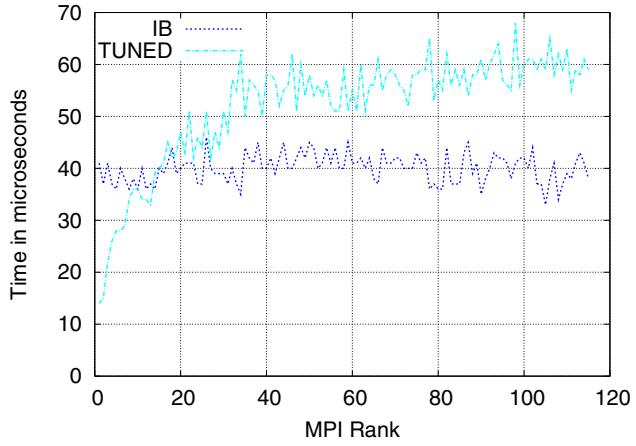

Figure 5. MPI_BCAST **latency for each MPI rank with a communicator size of 116**

ferent skew patterns). Contrary, our multicast-based implementation delivers the data to all processes in almost the same time (only a 14% deviation from the median), minimizing the skew between parallel processes. Several (e.g., round-based) applications derive benefit from this characteristic that reduces waiting time in consecutive communication operations.

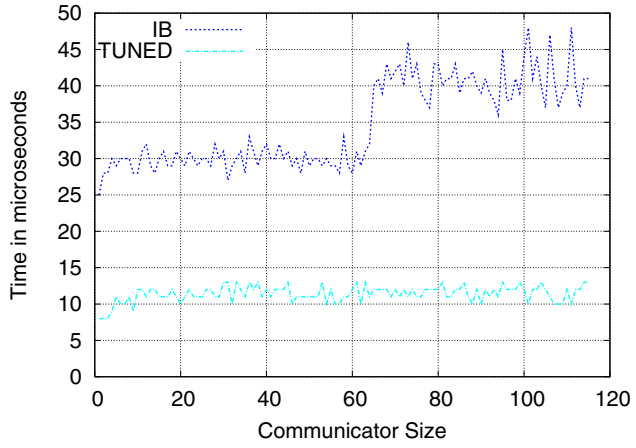

Figure 6. Broadcast latency of rank #1 of a 2 byte message broadcasted from rank #0 for varying communicator sizes

Figure 6 shows the MPI_BCAST latency of rank 1 for different communicator sizes (the sudden change at 64 nodes has to be attributed to the fact that we had to take the measurements for $1-64$ processes and $64-116$ processes separately due to technical problems). Figure 7 shows the

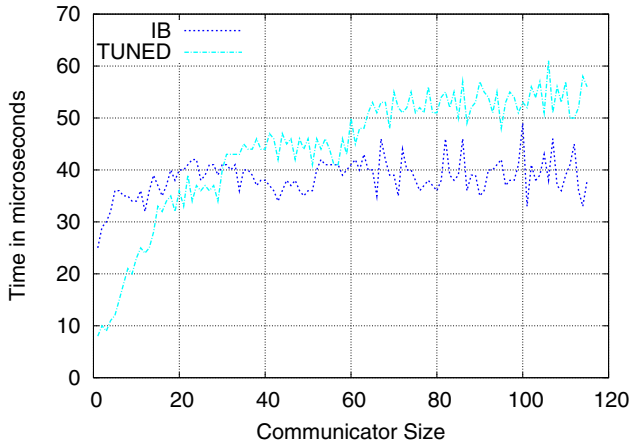

Figure 7. Broadcast latency of rank #N-1 of a 2 byte message broadcasted from rank #0 for varying communicator sizes N

latency of the MPI_BCAST operation at the last node in the communicator. The increasing running time can be easily seen. With the "TUNED" component, rank #1 leaves the operation after receiving the message from the root process - much earlier than it finishes in our implementation. However, process 1 is the only exception for this component that achieves a constant behaviour like in our implementation. Apart from that, the latency to the last rank (like to all other processes) steadily increases with the size of the communicator. Whereas our "IB" component reveals a similar latency for each process, without any noticeable influence of the communicator size.

5 Conclusions and Future Work

We have shown that our new algorithm is able to deliver high performance using InfiniBand™ multicast. Contrary to all other known approaches, we are able to avoid all scalability/hot-spot problems that occur with currently known schemes. The new multicast-based broadcast implementation accomplishes a practically constant-time behaviour in a double meaning: it scales independently of the communicator size and all MPI processes within a given communicator need the same time to complete the broadcast operation. Well-known microbenchmarks substantiate these theoretical conclusions with practical results. We proved it for up to 116 cluster nodes, but there is no reason to assume scalability problems with our approach.

Future work includes the detailed modelling and error-probability analysis of our new algorithm. Furthermore,

the influence of the better balance (all processes leave the broadcast nearly at the same time) in comparison to traditional algorithms has to be analyzed in the context of real-world applications. Since different broadcast usage patterns are imaginable (even within a single application), the three presented synchronization methods need to be analysed more carefully and might be mixed into an adaptive function that decides at run-time for an optimal strategy or root-delay parameter. Small communicators or large messages should be handled with different algorithms. However, the exact threshold values depend on several parameters. We will analyze this behavior with a well-known network model and try to find a better way to predict those cross-over points.

References

[1] M. Blum, L. Blum, and M. Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing*, 15(2):364–383, May 1986.

[2] H. A. Chen, Y. O. Carrasco, and A. W. Apon. MPI Collective Operations over IP Multicast. In *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 51–60, London, UK. Springer-Verlag.

[3] 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.

[4] M. Gerla, P. Palnati, and S. Walton. Multicasting protocols for high-speed, wormhole-routing local area networks. In *SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 184–193, New York, 1996. ACM Press.

[5] S. Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Trans. Program. Lang. Syst.*, 26(1):47–56, 2004.

[6] S. Hanna, B. Patel, and M. Shah. Multicast Address Dynamic Client Allocation Protocol (MADCAP). RFC 2730 (Proposed Standard), Dec. 1999.

[7] S. P. Kini, J. Liu, J. Wu, P. Wyckoff, and D. K. Panda. Fast and Scalable Barrier Using RDMA and Multicast Mechanisms for InfiniBand-Based Clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface,10th European PVM/MPI Users' Group Meeting, Venice, Italy, September 29 - October 2, 2003, Proceedings*, pages 369–378, 2003.

[8] J. Liu, A. Mamidala, and D. Panda. Fast and Scalable MPI-Level Broadcast using InfiniBand's Hardware Multicast Support, 2003.

[9] A. Mamidala, J. Liu, and D. Panda. Efficient Barrier and Allreduce on IBA clusters using hardware multicast and adaptive algorithms, 2004.

[10] A. R. Mamidala, H. Jin, and D. K. Panda. Efficient Hardware Multicast Group Management for Multiple MPI Communicators over InfiniBand. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 12th European PVM/MPI Users' Group Meeting, Sorrento, Italy, September 18-21, 2005, Proceedings*, volume 3666 of *Lecture Notes in Computer Science*, pages 388–398. Springer.

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

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

[13] M. Mosch. Integration einer neuen InfiniBand-Schnittstelle in die vorhandene InfiniBand MPICH2 Software. Technical report, Chemnitz University of Technology, 2006.

[14] L. M. Ni. Should Scalable Parallel Computers Support Efficient Hardware Multicasting? In *International Conference on Parallel Processing, Workshops*, pages 2–7, 1995.

[15] N. Nupairoj and L. M. Ni. Benchmarking of Multicast Communication Services. Technical Report MSU-CPS-ACS-103, Department of Computer Science, Michigan State University, 1995.

[16] Pallas GmbH. Pallas MPI Benchmarks - PMB, Part MPI-1. Technical report, 2000.

[17] 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.

[18] R. Rabenseifner. Automatic MPI Counter Profiling. In *42nd CUG Conference*, 2000.

[19] C. Siebert. Efficient Broadcast for Multicast-Capable Interconnection Networks. Master's thesis, Chemnitz University of Technology, 2006.

[20] 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.

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

[22] TOP500 List. http://www.top500.org/, November 2006.

[23] K. Verstoep, K. Langendoen, and H. Bal. Efficient Reliable Multicast on Myrinet. In *Proceedings of the 1996 International Conference on Parallel Processing*, pages 156–165, Washington, DC, USA.

[24] W. Yu, D. Buntinas, and D. K. Panda. High Performance and Reliable NIC-Based Multicast over Myrinet/GM-2, 2003.

[25] W. Yu, S. Sur, and D. K. Panda. High Performance Broadcast Support in La-Mpi Over Quadrics. *International Journal of High Performance Computing Applications*, 19(4):453–463, 2005.

[26] X. Yuan, S. Daniels, A. Faraj, and A. Karwande. Group Management Schemes for Implementing MPI Collective Communication over IP-Multicast, 2002.