CHEMNITZ UNIVERSITY
OF TECHNOLOGY

Faculty of Computer Science
Chair of Computer Architecture

Diploma Thesis

Efficient Broadcast for Multicast-Capable Interconnection Networks

Christian Siebert
christian.siebert@cs.tu-chemnitz.de

30th September 2006

Advisor: Dipl.-Inf. Torsten Hoefler
Supervisor: Prof. Dr.-Ing. Wolfgang Rehm

Task of the Thesis

The subject matter of this diploma thesis is an optimized implementation of the collective operation *MPI_Bcast()* which is part of the *Message Passing Interface* (MPI) standard. This special $1 : n$ communication function sends a given message from one process to all other processes of the same group. A possible implementation of this operation could use the native *multicast* capability of the underlying interconnection network if this is supported (e.g. *Ethernet* and *InfiniBand*). Contrary to the specified reliable data delivery of *MPI_Bcast()*, *multicast* does normally only support unreliable data delivery. Many different algorithms are possible to ensure this reliability.

A theoretical analysis and practical investigations should lead to an efficient strategy to solve this problem. The work will focus its attention on massive-parallel applications for *High Performance Computing* (HPC) *cluster*. A resulting implementation, based on *IP multicast*, for a recent version of the *Open MPI* library will be used to compare this algorithm with existing solutions. Importance should be attached especially towards *stability*, *portability* and *hardware independence*.

Thesis Declaration

I hereby declare that this diploma thesis was composed by myself and all work included has been done by me.

Chemnitz, 30th September 2006

Christian Siebert

Theses

i) It is possible to create a broadcast operation whose running time is (in practice) independent of the number of involved processes.

ii) Multicast is suited to use the intermediate phase of the broadcast operation more efficiently than point-to-point communication.

iii) There cannot exist a non-adaptive general broadcast algorithm which is always superior to all other broadcast algorithms in all imaginable scenarios.

iv) Balanced collective operations are a prerequisite to prevent additional process skew in parallel applications.

v) For each scenario, there exists a message size threshold value so that for all larger messages the *fragmented chain* algorithm is always the fastest broadcast method based on point-to-point communication.

vi) The preceding work does not take into account the theoretical foundations properly.

Abstract

According to long-term studies in *High Performance Computing Centers* (see e.g. [Rab99]), almost all parallel applications are using collective communication operations. The broadcast function *MPI_Bcast*(), which is a part of the *MPI-1.1* standard, is one the most heavily used collective operations for the widely used message passing programming paradigm. Inefficient implementations of this function can therefore cause a disastrous performance loss of the whole application. This thesis will try to make use of a feature called *multicast*, which is supported by several network technologies (like *Ethernet* or *InfiniBand*) and notwithstanding often goes to waste, to create a more efficient *MPI_Bcast*() implementation, especially for large communicators and small to medium sized messages. Several problems in conjunction with this feature (like re-establishment of reliability) needs to be solved to comply with the semantics of the target function. Existing solutions will be analysed, and new solutions will be proposed based on theoretical deductions and conclusions. The analysis of existing real-world applications (the *HPL* benchmark and *Abinit*) as well as a generalization of the broadcast behaviour using statistical assumptions lead to a solution which does not only perform well for synthetical benchmarks but also even better for a wide class of parallel applications. The finally derived broadcast algorithm has been implemented for the open source *MPI* library *Open MPI* using *IP multicast*. Instead of creating just another ”experimental” prototype, special care has been taken to make the implementation portable and stable enough for productive utilization. The achieved microbenchmark results prove that the new broadcast is usually always better than existing point-to-point implementations when the number of *MPI* processes exceeds the 8 node boundary. For as little as 13 nodes, the broadcast of a $4\ KiB$ message needs 49.1% longer when the original implementation of *Open MPI* is used instead of the new broadcast. With 28 *MPI* processes the same message can be transferred twice as fast. Since the the new broadcast scales independently of the number of involved processes, the performance (compared with point-to-point algorithms) differs more and more when the communicator size increases further. For 342 nodes and an $8\ KiB$ message, the difference amounts a factor of 4.896! Real-world applications can benefit even more from this new implementation, because it uses the intermediate phase of the broadcast operation more efficiently and because it achieves a pretty balanced behaviour. These additional improvements will be exemplarily verified with the *HPL* benchmark, which achieves a higher *GFLOPS* rate with the new and general broadcast algorithm, than with the supplied and purpose-built broadcasts.

Contents

List of Figures

Listings

List of Tables

1 Introduction

An optimal implementation of collective communication will take advantage of the specifics of the underlying communication network (such as support for multicast, which can be used for MPI broadcast), and will use different algorithms, according to the number of participating processes and the amount of data communicated.

1.1 Discussion of the Problem

The above citation from ”MPI - The Complete Reference” [MSD98, p. 194] suggests to use the *multicast* feature (of course only when it is supported by the underlying communication network) to create an optimal implementation of the *MPI* broadcast operation. Even eight years after this well-known publication, merely a handful of rather ”experimental” implementations by various people have been created. A couple of problems need to be solved to make the *multicast* feature useful for a broadcast implementation which conforms to the *MPI* standard - and many existing solutions are often so expensive that the performance gain of the final implementation is pretty small.

MPICH, a common *MPI* implementation, gives the following statement in its *FAQ* (http://www-unix.mcs.anl.gov/mpi/mpich1/faq.html):

”Does MPICH use IP Multicast for MPI_Bcast?”
”No. In principle, MPICH could use multicast, but in practice this would be very difficult. [...] There is a fairly easy way to replace any collective routine in MPI, but no-one has offered us a multicast-based MPI_Bcast yet...”

This diploma thesis deals with the problematic aspects of *multicast*, presents existing and new solutions to these problems, and shows how a well-chosen subset of these solutions can be combined to create an efficient implementation for the *MPI_Bcast*() operation. It is mainly targeted at developers and users of (massively) parallel applications and libraries (including but not limited to the *MPI* library), especially in conjunction with high-grade networked cluster systems for *High Performance Computing* (i.e. switch-based interconnection networks).

1.2 Outline of this Work

The rest of this first chapter gives an introduction to the *MPI* standard and its broadcast function, as well as a short overview of existing implementations for that. It gives a general description of the *multicast* feature with its advantages and disadvantages, and also a survey of a very promising *MPI* implementation, which will be used as the target library for the final solution.

The second chapter analyses two parallel applications with respect to their broadcast behaviour, before trying to cover a wide range of different usage scenarios with the help of some statistical assumptions and properties.

After giving those fundamentals, the third chapter refers to this knowledge and suggests a complete broadcast algorithm which uses the *multicast* feature. The pros and cons of alternative solutions (existing and new ones) are evaluated before arriving at a decision. The more general description of the algorithm will be completed with some more implementation details.

Microbenchmark results and the effects on the application show the actual achieved performance of this new broadcast implementation in the fourth chapter. Some theoretical boundary values for certain decision functions will derived from (or strengthened by) those results too.

The last chapter gives a conclusion of this diploma thesis, and proposes some additional work for possible future improvements. It gives some hints to developers which want to port this algorithm to other interconnection network, and it also gives some useful advices to users of this implementation.

1.3 MPI Standard

Message passing is a widely used programming paradigm on parallel computers, especially with distributed memory. At the beginning of this era, many different (mainly proprietary) message passing libraries were available, which limited the portability of written code. Hence, the *Message Passing Interface (MPI) Forum* tried to define the syntax as well as the semantics of a standard core of library routines that would be useful to a wide range of users. Special care has been taken to allow efficient implementations on a wide range of computers.

The official documents of the *MPI Forum*, including the standard, are available from the *MPI Forum Web* page at

http://www.mpi-forum.org

The first version of *MPI* was publicly released in May 1994, and version 1.1 [For95] (released in June 1995) made some clarifications and corrections. All the fundamental functionality, like point-to-point communication or collective operations, are already covered by this first version of the standard. A second version of the *MPI* standard [For97] was completed by the Forum in July 1997, and includes several extensions like dynamic process management or one-sided operations.

1.3.1 The MPI_Bcast() operation

MPI_Bcast() broadcasts a message from a special process called ”root” to all processes of the group, itself included. So initially, just this single origin process contains the data, but after the broadcast all processes contain it. The argument $root$ must have identical values on all processes, and $comm$ must represent the same intragroup communication domain. This collective operation can (but is not required to) return as soon as the content of $root$'s communication buffer has been copied to all processes. The completion of a call indicates that the caller is now free to access the data within the communication buffer. This also means that this operation is blocking - the current official standard does not specify any non-blocking collective operations. The local completion does not indicate that other processes in the group have completed or even started the operation (contrary to the synchronizing collective operation *MPI_Barrier*).

General, derived datatypes are allowed for $datatype$. The only restriction is that the type signature of count and $datatype$ on any process must be equal to the type signature of $count$ and $datatype$ at the root. This implies that the amount of data sent must be equal to the amount received, pairwise between each process and the root. Distinct type maps between sender and receiver are still allowed.

This broadcast operation is ”in place” because there is only a single buffer argument, which indicates that data is not moved at the root node.

The standard itself does not support a *multicast* function, where a broadcast executed by a root can be matched by regular receives at the remaining processes. It justifies this decision with the statement:

> *Such a function is easy to implement if the root directly sends data to each receiving process. However, there is little to be gained, as compared to executing multiple send operations. An implementation where processes are used as intermediate nodes in a broadcast tree is hard, since only the root executes a call that identifies the operation as a multicast. In contrast, in a collective call to MPI_BCAST all processes are aware that they participate in a broadcast.*

1.3.2 An Example Using MPI_Bcast()

Assume the root node gets some new input values from the user and wants to send those values to all other *MPI* processes. The following example in *C* will broadcast 100 integers from the process with rank number 0 to every process in the group.

```
MPI_Comm comm;
int array[100];
int root=0;
/* let the 'root' node fill the 'array' */
MPI_Bcast(array, 100, MPI_INT, root, comm);
/* now all nodes obtained the data from 'root' */
```

Listing 1: example which uses MPI_Bcast()

Figure 1 shows an example of a running *MPI_Bcast*() operation on 8 nodes. To be more precise: It is the improved version of the linear broadcast algorithm, which can be found in chapter 1.5.1, where all participating nodes are delayed by a random amount of time. This diagram serves as a perfect example to show how the duration of the broadcast operation can be subdivided into phases.

Since *MPI_Bcast*() is a blocking collective operation, it starts as soon as the first MPI process enters *MPI_Bcast*() and it ends when the last MPI process finishes *MPI_Bcast*(). But there are two other important moments during this operation: The root node is the only node which contains the message data at the beginning. Therefore ”useful” communications can only be started after the root node calls *MPI_Bcast*(). The second

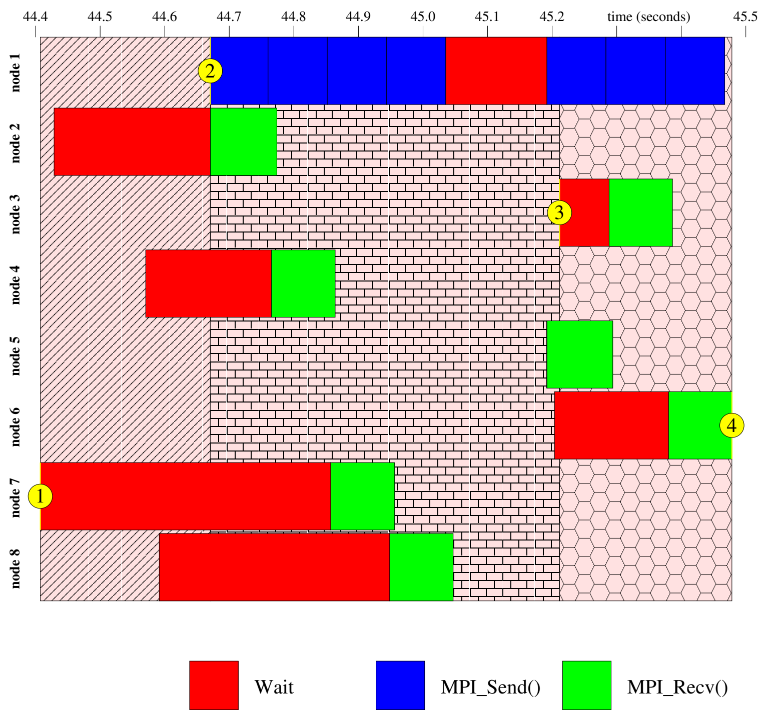


Figure 1: *MPI_Bcast*() operation can be split into 3 phases

important point in time occurs when the last node finally joins this operation. Altogether, the time frame of this collective can be subdivided into three phases: Within the ”startup phase” the first MPI processes enter this operation, but need to wait for the arrival of the root node. The ”intermediate phase” can be used for communication but cannot be completed because not all nodes are already present. The ”final phase” starts as soon as all processes have joined the collective operation and lasts until all processes have completed the operation.

Most synthetical benchmarks let all MPI processes call *MPI_Bcast*() simultaneously, making the first two phases collapsing into a non-existing time frame. Unfortunately, many broadcast algorithms (especially in the past) have been constructed upon this assumption, which is naturally not valid for most real-world applications.

1.4 LogGP Model of Parallel Computation

The performance of each broadcast algorithm depends on many parameters, like the number of nodes, message size, time of node arrival, network topology and parameters, application behaviour, number of network interfaces, communication library, and many more. There are several models for estimating the performance of parallel algorithms, for example the *PRAM* model, the *BSP* model, the *Hockney* model and the *LogP* model. We have decided to give a performance estimation for each algorithm using the more realistic *LogGP* [AISS97] model of parallel computation, which is an extension of the *LogP* model. The parameters for this model can be summarized as follows:

- *L*: the *Latency* of the interconnection network (the time it takes a single bit to travel from the source processor to its target processor)
- *o*: the *overhead*, defined as the time that a processor needs to inject or retrieve a message to or from the network (during this time the processor cannot perform other operations)
- *g*: the *gap* between messages, defined as the minimum time interval between consecutive message transmissions or receptions
- *G*: the *Gap per byte* or time per byte for long messages (the reciprocal of *G* characterizes the available communication bandwidth)
- *P*: the number of participating *Processors* (which equals the size of the communicator in our *MPI_Bcast* scenarios)

Sending an $n\ byte$ message from one processor to another takes $o+(n-1)\cdot G+L+o$ time units under this *LogGP* model. It is possible to measure all *LogGP* parameters for a given platform [TKV00]. We assume a full-duplex network which allows simultaneous message transfers of an initiated send and receive operation. Note that this model does not consider any form of network congestion.

1.5 Existing Techniques

There are several possibilities to implement *MPI_Bcast*(). Most implementations are using simple point-to-point communication because this is the basis of each communication library and therefore always present as well as working properly. This section gives a short overview of the most common techniques. There are many other implementations which are usually based on one of the here presented algorithms. Often they are slightly modified to make use of special additional knowledge or properties (like network topology).

Since there is no single number to express the performance of a collective algorithm, this section will give the estimated minimum, average and maximum completion time of a node according to the *LogGP* model, after presenting the description of each algorithm, some pseudo-code, and the advantages as well as disadvantages compared to the alternatives. For simplicity, we mainly assume that all nodes call *MPI_Bcast*() simultaneously, and that each pair of nodes has the same communication parameters. The parameter n holds the size of the broadcast message and p represents the number of involved MPI processes ($p = 1$ can be ignored because it is a *no-operation*, so p is defined to be larger than 1). f contains the number of fragments and n_f their size.

1.5.1 Linear Algorithm

The most simplest algorithm is derived from the definition of broadcast, and sends an individual message from the root node to all participating nodes. Therefore it is sometimes also called ”simple” or ”flat-tree” algorithm.

```
int MPI_Bcast_linear(void *buffer, int count, ...)
{
    ...
    MPI_Comm_size(comm, &nodes);
    MPI_Comm_rank(comm, &myrank);

    if (myrank == root) {
        /* root node sends to all other nodes */
        for (dest = 0; dest < root; dest++) {
            MPI_Send(buffer, count, dtype, dest, ...);
        }
        for (dest = (root+1); dest < nodes; dest++) {
            MPI_Send(buffer, count, dtype, dest, ...);
        }
    }
    else {
        /* non-root nodes receive from root */
        MPI_Recv(buffer, count, dtype, root, ...);
    }
}
```

Listing 2: linear MPI_Bcast() implementation

Although this linear implementation (contrary to e.g. an implementation with a logarithmic worst case running time) usually does not scale well when used with large communicators, it achieves acceptable performance for smaller communicators. Replacing those *MPI_Send*() by non-blocking *MPI_Isend*() calls and adding a corresponding *MPI_Waitall*(), improves this algorithm, especially in the case when the MPI processes enter this collective operation in a deferred but unknown chronological order. Unfortunately, when all processes call this function simultaneously (which is for instance the case when running a synthetical benchmark), the average completion time per MPI process is

$$\varnothing T_{linear}(n,p) = \frac{1}{p} \cdot T_{Send}(n) \cdot \left((p-1) + \sum_{i=1}^{p-1} i \right) = T_{Send}(n) \cdot \left(\frac{p+1}{2} - \frac{1}{p} \right)$$

Using the *LogGP* model, rank i receives the message after time

$$2 \cdot o + L + i \cdot (n-1) \cdot G + (i-1) \cdot g$$

The maximum completion time is therefore

$$T = L + 2 \cdot o + (p-1) \cdot (n-1) \cdot G + (p-2) \cdot g$$

One node receives the message (and therefore completes MPI_Bcast) after the first send operation from root and one node receives the message after $p-1$ rounds, giving the following extreme performance numbers:

$$T_{Send}(n) \leq T_{linear}(n, p) \leq (p-1) \cdot T_{Send}(n)$$

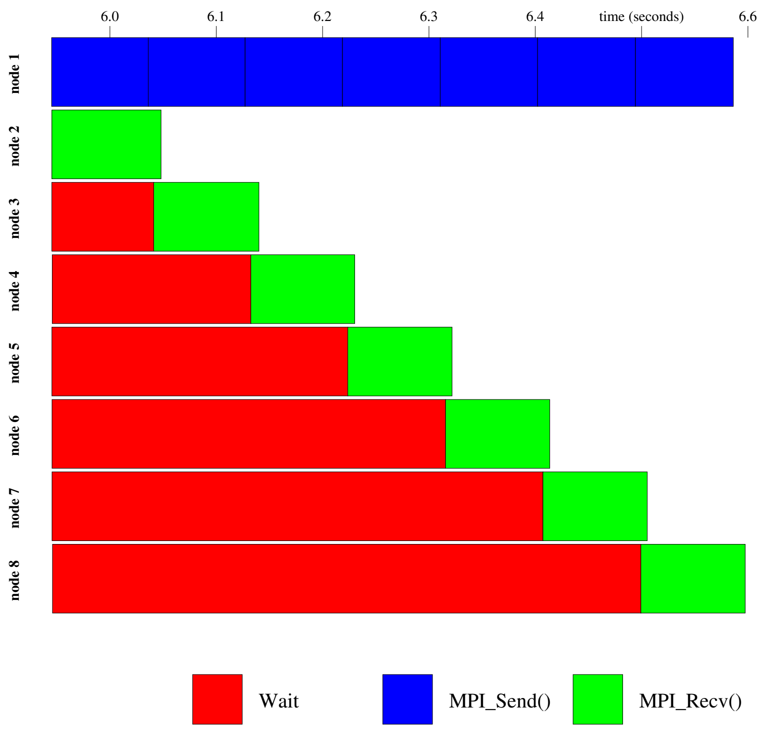


Figure 2: linear broadcast running on 8 nodes

Figure 2 shows how *MPI_Bcast_linear*() broadcasts a $1\ MiB$ message to 8 nodes on a Fast Ethernet network. For a better understanding, the time for *MPI_Recv*() has been divided into the time it waits for the first byte and the actual transmission time. The broadcast duration per node was $\{0.641, 0.103, 0.195, 0.284, 0.376, 0.468, 0.560, 0.651\}$ seconds, giving a real average duration of 0.410 seconds per node.

1.5.2 Chain Algorithm

Another implementation with a similar ”bad” performance lets each node send and receive at most one message. This effectively creates a kind of ring topology where each node has one predecessor from which it receives the message, and one successor to which it sends the message (for that reason it is also sometimes called ”ring” algorithm). Since the root node does not need to receive the message, the ring is reduced to a chain where the last node skips the send part.

```
int MPI_Bcast_chain(void *buffer, int count, ...)
{
   ...
   MPI_Comm_size(comm, &nodes);
   MPI_Comm_rank(comm, &myrank);

   pred = (nodes + myrank - 1) % nodes;
   succ = (myrank + 1) % nodes;
   if (myrank != root) {
      MPI_Recv(buffer, count, dtype, pred, ...);
   }
   if (succ != root) {
      MPI_Send(buffer, count, dtype, succ, ...);
   }
}
```

Listing 3: chain implementation

Listing 3 uses a simple modular addition/subtraction of 1 to determine the successor and predecessor of the own node. A better way to support other network topologies (like meshes) is to use *MPI_Cart_create*() with a single dimension to embed the virtual chain topology into the real underlying topology. The actual neighbours can then be determined with a call to *MPI_Cart_shift*().

Usually this algorithm is even slightly worse than the linear algorithm because the MPI processes are served in a predefined chronological order. A single late node is enough to stall the whole chain algorithm (contrary to the improved version of the linear algorithm). When there are no delays, the average completion time per MPI process is

$$\varnothing T_{chain}(n,p) = \frac{1}{p} \cdot \left(\sum_{i=1}^{p} i - 1 \right) \cdot T_{Send}(n) = T_{Send}(n) \cdot \left(\frac{p+1}{2} - \frac{1}{p} \right)$$

The root node completes the broadcast after a single send, and the last node in the chain needs to wait $p-1$ rounds until it receives the message. This gives the following extreme performance numbers:

$$T_{Send}(n) \leq T_{chain}(n,p) \leq (p-1) \cdot T_{Send}(n)$$

The maximum time of the *chain* algorithm, according to the *LogGP* model is

$$T = (p-1) \cdot (L + 2 \cdot o + (n_f - 1) \cdot G) + (f-1) \cdot (g + (n_f - 1) \cdot G)$$

The ”stairs” in figure 3 show how the nodes get the message from their neighbours. The broadcast duration was $\{0.090, 0.194, 0.297, 0.399, 0.503, 0.607, 0.709, 0.721\}$ seconds, giving a real average duration of 0.440 seconds per node.

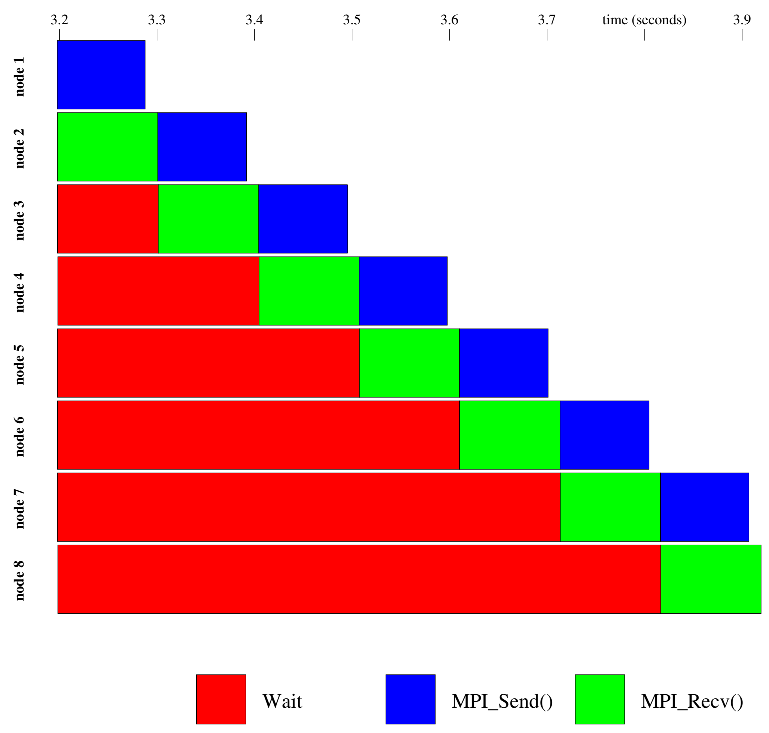


Figure 3: chain broadcast running on 8 nodes

If this chain algorithm has no obvious advantages (except the good support for ”cheaper” network topologies - even without any switches at all [1]), then why should we care about this algorithm? Because there is an optimization possibility, which turns this ”bad” algorithm into the best algorithm for large messages [2]: Normally, each node waits until the message has been received completely before sending it to the next node. When we split this message into several fragments, each node can start sending as soon as it received the first fragment: A trivial implementation could call *MPI_Bcast_chain()* for each fragment. This introduces an overlapping of send and receive requests and leads to the principle of pipelining.

When the $1\ MiB$ message gets split into $64\ KiB$ fragments, the fragmented chain algorithm achieves an overwhelming performance compared to the non-fragmented version. Figure 4 shows this behaviour. Note that the optimal size of the fragments depends on several parameters (number of nodes, message size and network parameters) and might need to be recalculated for every new broadcast operation. The broadcast duration per node was $\{0.097, 0.104, 0.110, 0.116, 0.123, 0.130, 0.136, 0.142\}$ seconds, giving an astounding real average duration of just 0.120 seconds per node. The non-fragmented version is therefore by a factor of 3.67 slower than this fragmented version.

[1]Example: Several modern mainboards are equipped with two Gigabit Ethernet ports on-board. Connect such cluster nodes in a real ring topology and you have created a very cheap cluster without any switches. The (fragmented) chain broadcast is always the optimal algorithm for such a network.

[2]For a discussion of fragmented tree versus fragmented chain algorithm see the appendix.

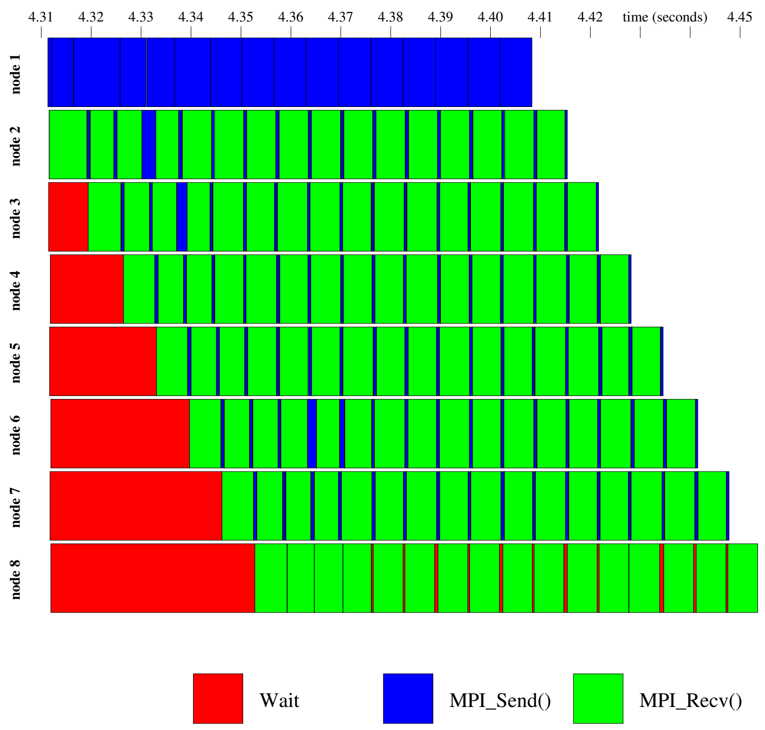


Figure 4: fragmented chain broadcast running on 8 nodes

The next large class of broadcast algorithms use virtual tree topologies to limit the number of rounds to some logarithmic function. This reduces the average and maximum broadcast duration per node and is therefore very useful for medium and large-sized communicators. The broadcast messages traverse the trees starting from the root node, and going towards the leaf nodes through intermediate nodes.

1.5.3 Binary Tree Algorithm

A *binary tree* is a well-known data structure in computer science. Nodes, which represent MPI processes, are connected by directed edges, which indicate the direction of the message transfer. To get a good performance, we require that each parent node has two children - except the leave nodes which are allowed to have only a single or no children (this is often called *complete binary tree*).

```
int MPI_Bcast_binary(void *buffer, int count, ...)
{
    ...
    /* assumption: root == 0 */
    MPI_Comm_size(comm, &nodes);
    MPI_Comm_rank(comm, &myrank);

    lchild = (myrank << 1) + 1;
    rchild = (myrank << 1) + 2;
    parent = (myrank - 1) >> 1;

    if (parent >= 0) {
        MPI_Recv(buffer, count, dtype, parent, ...);
    }
    /* send message to both children */
    if (lchild < nodes) {
        MPI_Send(buffer, count, dtype, lchild, ...);
    }
    if (rchild < nodes) {
        MPI_Send(buffer, count, dtype, rchild, ...);
  }
}
```

Listing 4: binary tree implementation

A trivial binary tree implementation can be found in listing 4. Note that this algorithm assumes that the broadcast root has always rank 0. The usual way to circumvent this restriction is to introduce virtual rank numbers, so that the root node gets the virtual rank 0. A rank rotation, e.g. using the modular arithmetic trick from the chain algorithm, can be used to create such a mapping between real and virtual rank numbers.

Although 8 nodes create a nearly-balanced (and symmetric) binary tree (with the exception of a single node in an additional level), diagram 5 demonstrates that the broadcast duration per node ($\{0.184, 0.287, 0.377, 0.298, 0.298, 0.299, 0.390, 0.320\}$) is not very balanced when a binary tree is used as a broadcast topology. Nevertheless, the average broadcast duration of 0.307 seconds per node is already better than the non-fragmented algorithms of the linear-scaling class.

The maximum time of the *binary tree* broadcast, according to the *LogGP* model is

$$T = (\lceil log_2(p+1)\rceil - 1) \cdot (L + 2 \cdot (o + (n-1) \cdot G + g)) + 2 \cdot ((n-1) \cdot G + g)$$

The reason for this imbalance is that each node usually serves two children but can not send two messages simultaneously over a single network interface. So instead of thinking of an ”usual” balanced binary tree, the real tree structure - when used as a broadcast topology - can be seen in figure 6.

When the communicator size increases, the imbalance will get even worse, because the root node in a binary tree finishes always after two rounds, whereas all leave nodes

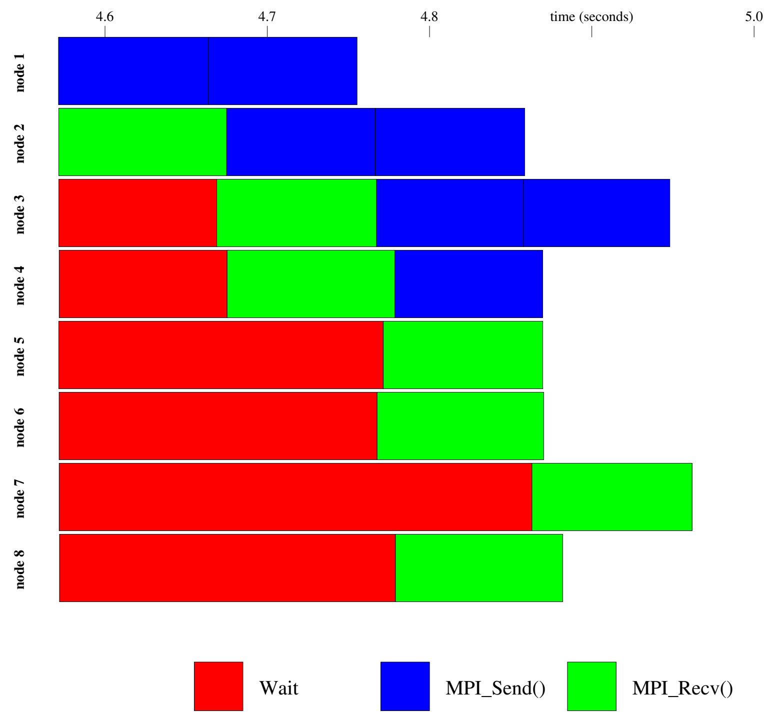


Figure 5: binary tree broadcast running on 8 nodes

(and there are $\lceil 0.5 \cdot p \rceil$ of them) have to wait $\lceil log_2\ p \rceil$ rounds. Fortunately, there is another tree structure which takes care of this issue.

1.5.4 Binomial Tree Algorithm

A *binomial tree* is a more sophisticated tree structure, and can be defined recursively:

- a binomial tree of order 0 is a single node
- a binomial tree of order k has a root of degree k and its children are roots of binomial trees of orders $k-1, k-2, \ldots, 2, 1, 0$

A *binomial tree* of order k has at most 2^k nodes and height k. Figure 7 shows a possible structure of a binomial tree for a broadcast operation on 8 nodes.

The different communication pattern results in a much more balanced broadcast behaviour compared to the simple binary tree structure (see figure 8).

The broadcast duration on this binomial tree used $\{0.274, 0.285, 0.286, 0.285, 0.297, 0.297, 0.297, 0.308\}$ seconds per node, which is all very close to the average value of 0.291 seconds. An application where all (especially 2^k) nodes are calling *MPI_Bcast*() simultaneously, can expect that all nodes complete this collective operation in a similar amount of time. This very useful feature and the slightly better overall performance of this binomial tree algorithm makes it the favourite tree-based broadcast algorithm, despite the slightly more complicated handling. Figure 9 shows a binomial tree for

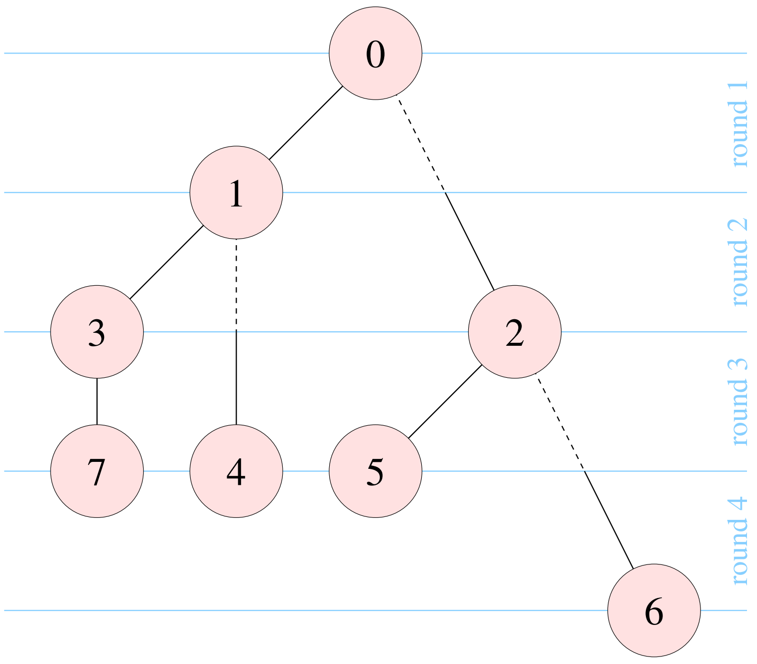


Figure 6: binary tree structure for an 8-node broadcast

16 nodes, where the rank numbers are ordered in a way which makes computation of parent and child nodes easier than the originally suggested ordering, and they are presented in binary notation to make it easier for the reader to follow the bit manipulation description.

To find the parent of a node [3] in a such a binomial tree structure, clear the least significant set bit of the rank number. For all (valid) least significant clear bits, there is one children whose rank number can be figured out by setting this corresponding bit within the node's rank number. Note that is is important to send the broadcast message in the correct order to the children - start with the highest such clear bit and proceed up to the lowest such clear bit.

The maximum duration of the *binomial tree* broadcast, according to the *LogGP* model is

$$T = \lceil log_2 p \rceil \cdot (L + 2 \cdot o + (n-1) \cdot G)$$

The performance chart of all four basic algorithms (from 2 to 32 MPI processes) in figure 10 shows that the two algorithms of the first class (linear and chain algorithm) scale linearly with the number of involved MPI nodes (but the linear algorithm usually has a better gradient). The tree algorithms of the second class scale logarithmically with the number of the involved MPI processes. As expected, the binomial tree algorithm performs somewhat better than the simple binary tree algorithm.

[3]Note that rank 0 is always root and therefore has no parent node. See also the previous discussion about virtual ranks.

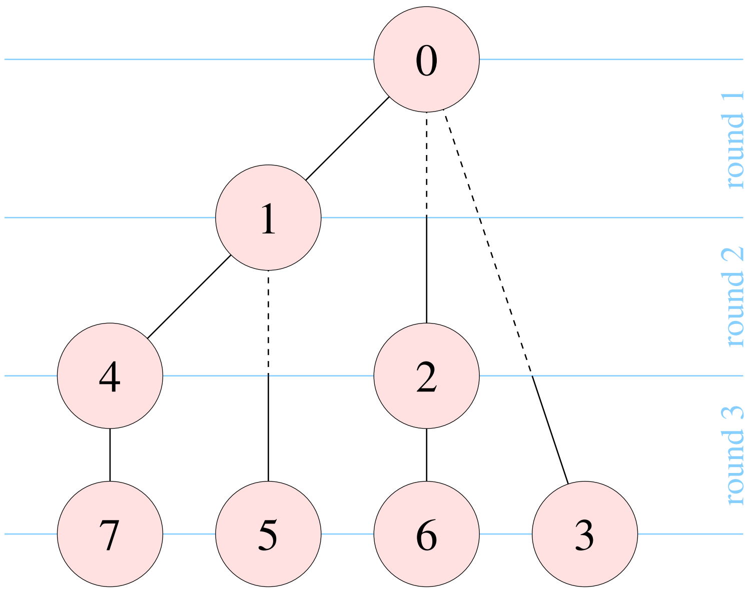


Figure 7: binomial tree structure for an 8-node broadcast

1.5.5 Other Algorithms

Many additional broadcast algorithms have been proposed in the literature.

The *Splitted-binary tree* algorithm [PGAB$^+$05] splits the original message into two parts, and then sends the ”left” half of the message down the left half of the binary tree, and the ”right” half of the message down the right half of the tree. In the final phase of the algorithm, every node exchanges messages with its ”pair” node from the opposite side of the binary tree.

It is also possible to build a broadcast algorithm out of other collective operations: *MPI_Scatter*() followed by an *MPI_Allgather*() [PMG95] distributes the message in parts over all nodes, and subsequently collects all parts using for example the *recursive doubling* algorithm (see [GDBC03] or [RTG05]).

This ”splitting” of messages can be generalized for any arbitrary broadcast algorithm: A larger message can be seen as a collection of several fragments, and each fragment can be delivered independently of the others. If a node sends a message to several destinations, then the communication can be done interleaved, which involves other nodes much earlier. Succeeding communications can be started as soon as the first fragment has been received and therefore before the complete message has been received. This transmission scheme leads to the well-known pipelining effect. The best usage example for this property is the *fragmented chain* algorithm for large messages.

1.5.6 Limits Of Those Algorithms

The most limiting parameter for all presented algorithms so far, comes from the usage of point-to-point communication and the fact that a single MPI process can not inject or

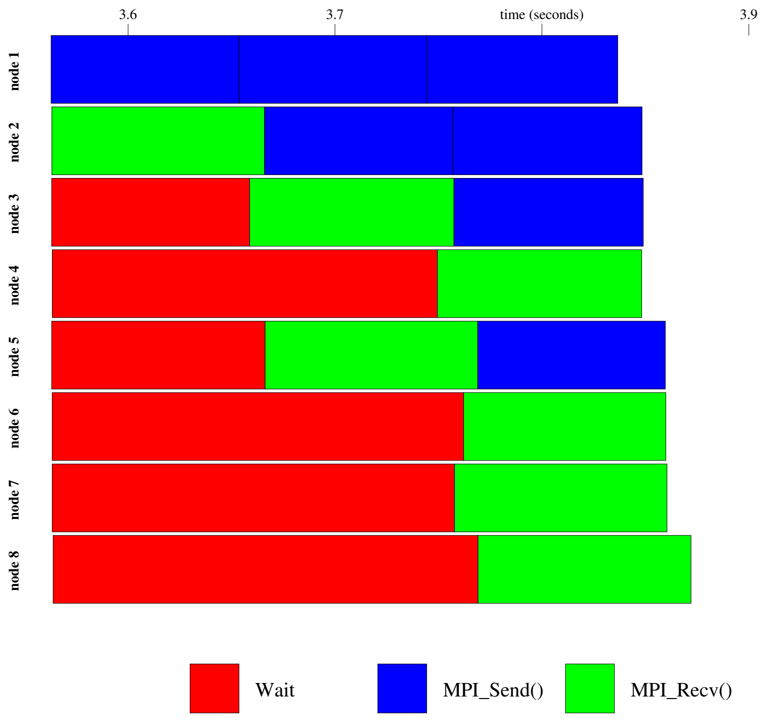


Figure 8: binomial tree broadcast running on 8 nodes

retrieve several messages simultaneously into or from the network (”unicast”). For any broadcast operation to p nodes (which is implemented on top of this communication scheme) there are at least $\lceil log_c\ p \rceil$ [4] transmission rounds necessary, otherwise at least one node will never receive anything.

Many network technologies (like Ethernet and InfiniBand) are equipped with special support for other communication schemes besides simple point-to-point. The following sections will describe features which are known as *hardware broadcast* and *hardware multicast*, and show how this can be used to implement *MPI_Bcast*().

1.6 Hardware Broadcast

Although some network technologies support a direct *broadcast* feature which could be used to implement *MPI_Bcast*(), clusters are often used simultaneously by more than one parallel job and therefore subdivided logically into several parts. A broadcast packet will be send to all nodes in the specified domain, and can therefore influence the performance of other jobs (by consuming processing time within the network stack were those packets will be rejected, and by directly reducing the network bandwidth too). In addition, hardware *broadcast* has usually the same drawbacks like *multicast* (e.g. the unreliable data delivery). On the other hand, there are network technologies

[4]The constant parameter c is usually 2 but might be increased when there are several network interfaces available (”fan-out”).

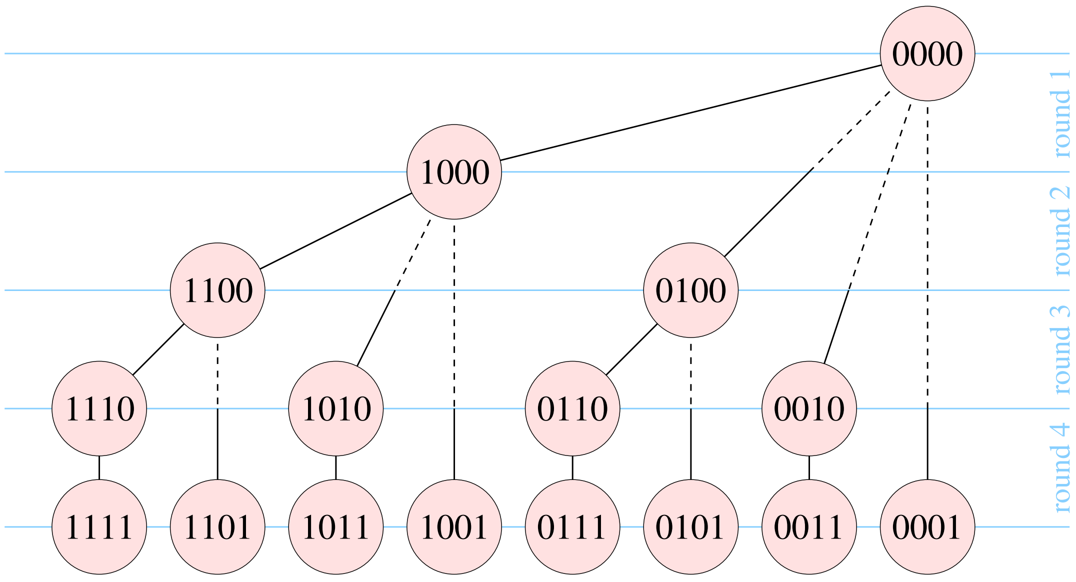


Figure 9: reordered binomial tree structure for 16 nodes

which do not support multicast or have other interesting features (*Quadrics* for example supports a hardware-based acknowledgment scheme for its special ”range” broadcast [WYG05]).

1.7 Hardware Multicast

Multicast is similar to a *broadcast*, because it can be used to send a message to more than one recipient. Contrary to the one-to-all *broadcast* feature, *multicast* is a one-to-many operation which sends a message selectively to nodes that have agreed prior to receive those packets. This advantage makes it the better candidate for an *MPI_Bcast*() implementation upon *IP*-based interconnects.

Ethernet for example, can support *IP multicast* if the underlying hardware is multicast-capable (e.g. at least layer 2 switching). *Multicast* traffic is handled at the transport layer with UDP, and multicast-capable hosts need necessarily an *Internet Group Management Protocol* (IGMP) implementation in their TCP/IP stack. In 1993, the first multicast implementation saw the light in the 4.4 BSD release. Today, *IP multicast* is a pretty mature feature, and is supported by many hardware components as well as nearly all recent operating systems.

1.7.1 How does Multicast Work?

Before an application can receive any multicast datagrams, it must tell the operating system (”kernel”) which multicast groups it is interested in. Multicast groups can be for instance class D IP addresses for *Ethernet* or a so-called *global identifier* (GID) for *InfiniBand*. This explicit ”group joining” is necessary because multicast datagrams are

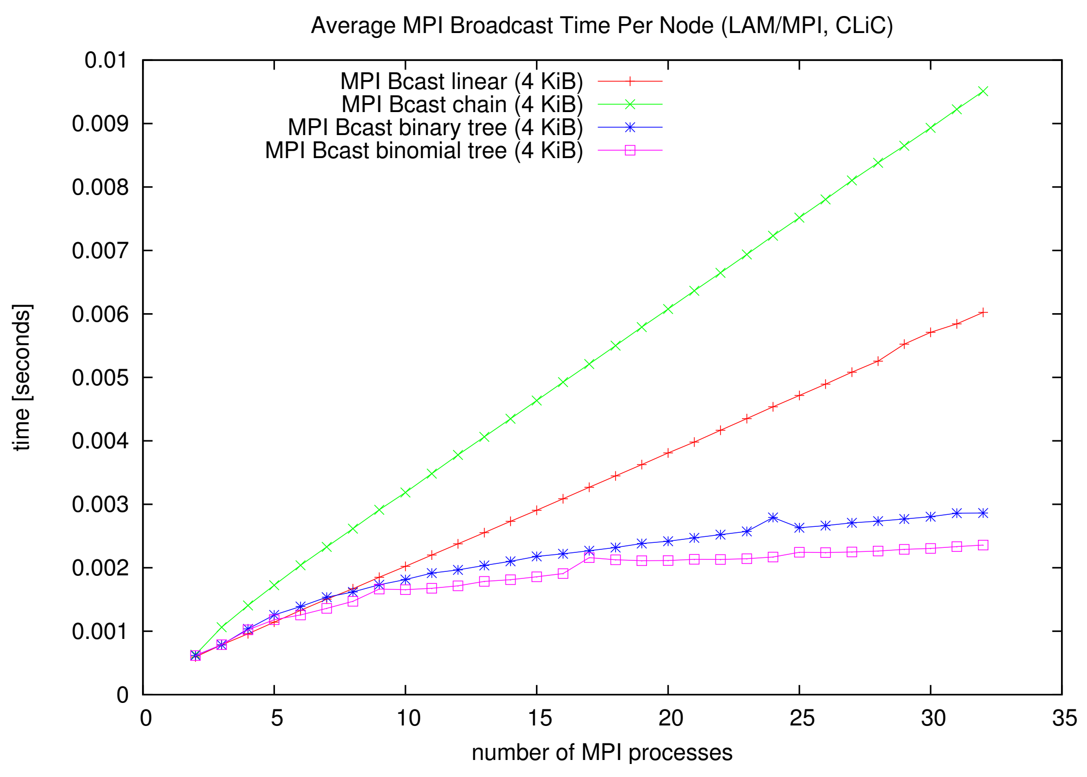


Figure 10: performance of the four basic broadcast algorithms

filtered by the hardware or by the network protocol stack (and, in some cases, by both). Only those packets with a destination group which has been previously registered, are accepted and delivered to the corresponding application.

Once an application has successfully joined a multicast group on a particular network interface, it can receive multicast datagrams which are simply sent to this group. Sending of multicast datagrams usually does not need any special preparation (except e.g. opening an UDP socket for IP multicast). Finally, the application can leave a multicast group by informing the kernel that it is no longer interested in this group. [5]

When a communicator is created, a new multicast group should be assigned to it and all participating MPI processes should join this group. The central switch will be informed about any joins or leaves, and stores this information for any port. The example scenario in figure 11 shows a small cluster consisting of 8 nodes, which are connected through a central switch. In the first step, the nodes $1, 2$ and 3 join a multicast group A by sending a join request to the switch. Afterwards, nodes $3, 5, 7$ and 8 join another multicast group B, and nodes 4 and 6 remain unused in this scenario. Both groups (or their corresponding communicators) can be part of a single *MPI* instance or they can belong to totally different jobs. When a node sends a multicast datagram to one of the registered groups, the switch will forward this packet to all ports, which are associated with the destination group. So logically spoken: a single message arrives at the switch,

[5]For a detailed programming guide of IP multicast, I recommend the book [BWRS03].

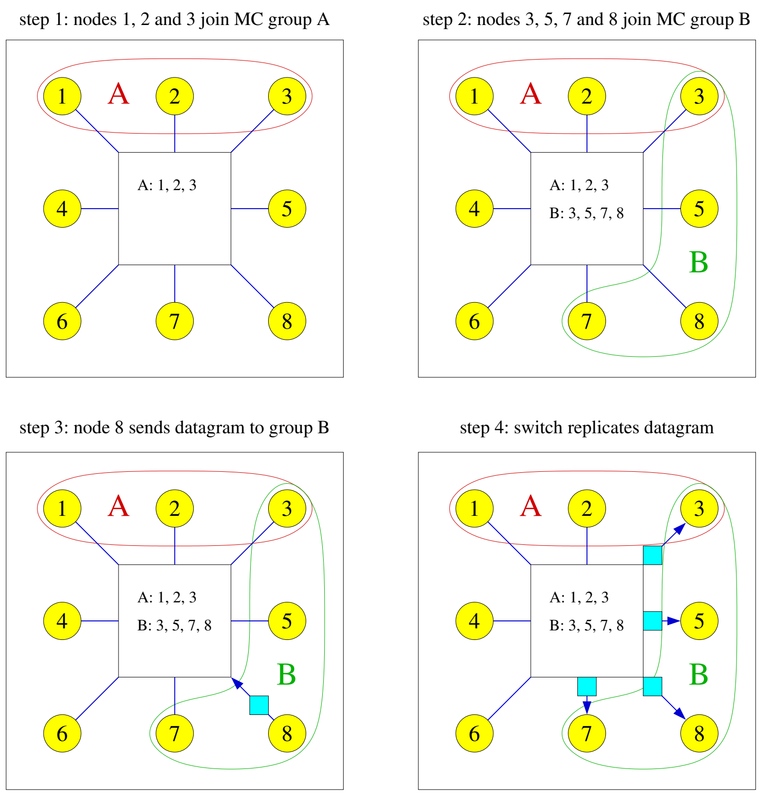

step 1: nodes 1, 2 and 3 join MC group A
step 2: nodes 3, 5, 7 and 8 join MC group B
step 3: node 8 sends datagram to group B
step 4: switch replicates datagram
A
B
A: 1, 2, 3
B: 3, 5, 7, 8
1
2
3
4
5
6
7
8

Figure 11: exemplary multicast scenario with 8 nodes

gets ”duplicated” there, and finally arrives at several receivers ”simultaneously”. In our example scenario, node 8 sends a multicast datagram to the destination group B in step 3. The switch recognizes this group and sends the packet to the associated nodes $3, 5, 7$ and 8. Note: it is possible to suppress the ”boomerang” packet of node 8 (the final implementation does this to reduce this unnecessary overhead which gives a small performance improvement).

Such a multicast feature can lead to an *MPI_Bcast*() implementation with a performance which scales independently of the number of involved processes! To create such a solution, there are a several problem which need to be solved: IP multicast sockets are UDP-based and therefore multicast is unreliable! This means that nothing is guaranteed and the user is responsible for any necessary reliability, privacy and control messages, as well as scheduling an event. Especially if a node is not ready to receive a multicast datagram, then an incoming datagram might not need to be stored and therefore it gets usually lost. Furthermore, larger messages need to be fragmented to fit into IP packets, and a proper multicast group assignment is also not trivial.

1.7.2 Multicast Group Assignment

Class D addresses, in the range 224.0.0.0 through 239.255.255.255, are the multicast addresses in IPv4. The low-order 28 bits of a class D address form the multicast group

ID and the 32-bit address is called the group address. Unfortunately, in Ethernet or IEEE 802 networks only the low-order 23 bits of the IP multicast address are copied to the Ethernet multicast address. There are a few special multicast addresses and several reserved multicast addresses, which can be found in a regular updated list by the *Internet Assigned Numbers Authority* [IAN06]. For some general multicast assignment guidelines see [ZAS01]. When we have a set of useful multicast addresses, we need a proper way to assign a new address to each new communicator. This would be simple if we had only one MPI instance which could keep track of all currently used addresses. Since there can be several MPI jobs running in parallel on a single cluster, maybe even using different MPI implementations, there is no globally visible state anymore which could fulfill this task. The best solution for this problem is to reintroduce a global state by adding a special server which distributes new multicast addresses on request. *RFC 2730* describes the *Multicast Address Dynamic Client Allocation Protocol* [SHS99] (”MADCAP”) which could be used for this purpose. Every time a new communicator is created, this MADCAP server could be asked for a free multicast group. As long as all used multicast addresses are known to this server, there will be no clashes at all and many different MPI jobs can work safely in parallel.

Another solution to this address assignment problem, is to choose the multicast address and port at random, and hope that there are no collisions. The final implementation uses this approach, but is prepared to use a MADCAP server or similar and fall back to this solution if there is none available. Using the reserved multicast address ranges 225.0.1.0 to 231.255.255.255 and 234.0.1.0 to 238.255.255.255 gives 200,540,160 possible addresses. Adding the port number (range 5000 to 32768 [6]) to this pool as well, gives another 27769 possibilities. Altogether, we can select an (address, port) pair out of about $5.5 \cdot 10^{12}$ different possibilities. When the number contemporaneously used communicators increases, the collision probability increases even more according to the *birthday paradox*. If there are currently n multicast groups in use (e.g. for n different communicators), then there are $n \cdot (n-1)/2$ pairs, each of which with potentially identical values. It is easier to first calculate the probability that all groups are different. Let m be the total number of available <multicast group, port> pairs and assume that each pair is selected with the same probability. Note that it is pretty important to choose the pairs at random, otherwise collisions can be very likely!

$$\overline{p}(n,m) = 1 \cdot \left(1 - \frac{1}{m}\right) \cdot \left(1 - \frac{2}{m}\right) \cdot \dots \cdot \left(1 - \frac{n-1}{m}\right) = \frac{m!}{m^n \cdot (m-n)!}$$

The probability that there is at least one collision is then the complementary of $\overline{p}(n,m)$. Through the additional port number, the number of possibilities is large enough, so that this probability keeps tolerable small, as can be seen in table 1.

Although only the MADCAP solution (or similar) gives a 100% certainty of never producing any collisions, the probability that the second solution fails is, in many cases, acceptable small.

[6]This range restriction was introduced because of portability issues (”ephermal ports”).

number of communicators	probability of at least one collision
1	0.0 %
2	0.0000000000179 %
3	0.0000000000538 %
4	0.0000000001077 %
5	0.0000000001795 %
6	0.0000000002693 %
7	0.0000000003771 %
8	0.0000000005028 %
9	0.0000000006464 %
10	0.0000000008080 %
11	0.0000000009876 %
...	...
20	0.0000000034118 %
30	0.0000000078113 %
40	0.0000000140066 %
50	0.0000000219975 %
100	0.0000000888881 %
200	0.0000003573481 %

Table 1: collision probability when using several communicators

1.8 Open MPI

Open MPI (http://www.open-mpi.org) is a very promising project with the demand to build the best *MPI* library available. Since it combines the knowledge of many predecessor projects (*FT-MPI*, *LA-MPI*, *LAM/MPI* and *PACX-MPI*), it uses well-established technologies as well as new ideas to build a completely new framework which supports (or will support in the near future) many features (like complete *MPI-2* compliance, thread safety and fault tolerance) and still achieves high performance and portability. *Open MPI* offers several advantages for computer science researchers which makes it the perfect platform for new developments.

1.8.1 Architecture of Open MPI

The primary software design motif of *Open MPI* is a lightweight component architecture called the *Modular Component Architecture* (MCA). This backbone architecture provides management services for all other layers and contains component frameworks for each major functional area in *Open MPI*. Each of this component frameworks (currently the *Open MPI* components, the *Open Run Time Environment* components and the *Open Portable Access Layer* components) is a collection of self-contained software units that export well-defined interfaces and can be deployed and composed with other components. The MPI component framework contains for example (see e.g. [GWS05] for more details):

- Point-to-Point Management Layer: this component manages message delivery and implements the semantics of a given point-to-point communications protocol
- Byte-Transfer-Layer: this component handles point-to-point data delivery over the networks
- Collective Communication: the back-end of MPI collective operations, supporting both intra- and intercommunicator functionality
- Process Topology: Cartesian and graph mapping functionality for intracommunicators (this allows MPI to optimize communications based on locality)
- Parallel I/O: modules for parallel file and device access
- ...

This theses makes use of this component-based approach, and - because an implementation of the collective operation *MPI_Bcast*() is one of its objectives - it is especially interested in the *COLL* framework. Since components are free to implement the standardized MPI semantics in any way that they choose, we will later use a combined approach which is layered over point-to-point functions as well as an alternate communication channel for *IP multicast* [7].

1.8.2 COLL Component

A *COLL* component is essentially a list of top-level function pointers that will be selectively invoked upon demand. A component becomes a *module* when it is paired with a communicator. Top-level MPI collective functions, like *MPI_Bcast*(), are thin wrappers that perform error checking and afterwards call the provided functions in the appropriate module (depending on the communicator). There are effectively five phases in a *COLL* component's life cycle: selection, initialization, checkpoint/restart, normal operation, and finalization. Since at the time of writing the *checkpoint/restart* feature is currently not really existent in *Open MPI*, and [SL04, p. 11] states

> *It is not an error if a module does not include the functionality required for checkpointing and restarting itself; support for checkpointing/restart in a* COLL *module is optional.*

we can simply mark our implementation to not support this, and get the simplified life state diagram in figure 12 with only four phases for our ipmc broadcast component.

Every time a new communicator should be created (e.g. by directly calling the function *MPI_Comm_create*(); but also including the one-time setup of *MPI_COMM_SELF* and *MPI_COMM_WORLD* at startup), *Open MPI* queries each available *COLL* component to determine if it can be used with this newly-created communicator. A priority

[7] For a good description of the component architecture of *Open MPI* especially with regard to the collective framework, see [SL04]

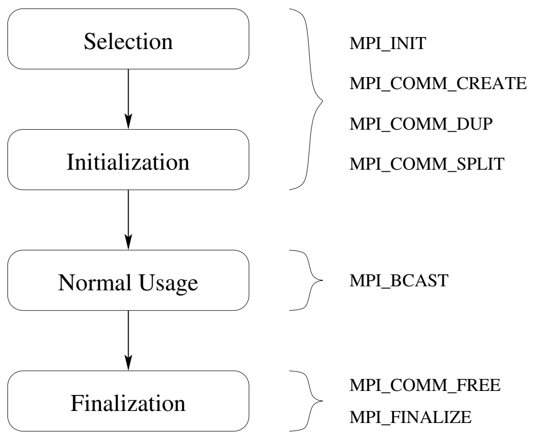


Figure 12: four phases in the life of the ipmc component

value (from 0 to 100) will be returned by each component, and the component with the highest priority will be *selected* by the framework. Once a *COLL* module is selected for a given communicator, the component's initialization function will be called which performs any one-time setup required by the module (since the binding to the communicator remains static after this step, pre-computations might be done here to achieve some run-time optimizations). The initialization function returns a module, which includes a list of function pointers for its algorithms. After a *COLL* module has been initialized, those routines will be called whenever an MPI collective function is invoked on the communicator. When a communicator should be destroyed (e.g. by *MPI_Comm_free*) the modules finalization method will be called, which is responsible for cleaning up all resources associated with this communicator.

1.9 Summary

Today, many parallel applications are implemented using the *Message Passing Interface*, and their performance depends on the underlying *MPI* library. *MPI_Bcast*(), one of the most used collective operations, can be implemented in many ways. The usual point-to-point communication scheme is too limiting, whereas *multicast* - besides its many problems - has promising advantages for a broadcast implementation. *Open MPI* provides an ideal framework for new developments, and we will try to create a multicast-based *MPI_Bcast*() implementation for this relatively new open source *MPI* library.

2 Existing Applications which use MPI_Bcast()

Regrettably, many collective operations has been optimized especially for synthetical benchmarks (where there is no ”process skew”), and later score badly when used with real-world applications. Fortunately, more papers regarding the optimization of collective operations in view of application behaviour appeared in the last few years (see e.g. [AMP04]). In order to prevent the same mistake, this section shortly introduces two applications which makes quite heavy use of *MPI_Bcast*(). Each application has its own typical broadcast pattern, which will be show in a graphical form, similar to a *Gantt chart*. Such charts are produce by profiling a running application: Events such as calling *MPI_Send*() will be logged together with a global time stamp and afterwards visualized with postprocessing tools (see e.g. [ZLGS99]).

2.1 High-Performance Linpack Benchmark

The famous *High-Performance Linpack Benchmark for Distributed-Memory Computers* [8] is the parallel benchmark that is used to measure the performance of the most powerful computer systems. Twice a year, the TOP500 project [9] assembles and releases the 500 most powerful systems according the performance measures of the Linpack benchmark.

2.1.1 Algorithm

This benchmark solves a dense system of linear equations in double precision arithmetic. The used algorithm does an *LU factorization* of a random matrix with partial pivoting. The operation count for the algorithm must be

$$\frac{2}{3}n^3 + O(n^2)$$

floating point operations.

This portable implementation requires an MPI 1.1 compliant Message Passing Interface library as well as a *Basic Linear Algebra Subprograms* (*BLAS*) library.

2.1.2 Results

For 16 nodes on the *CLiC* testbed, the HPL benchmark achieves a performance of 7.538 GFLOPS (total running time of 1941 seconds) for a problem size N of 28000 and a blocking factor NB of 40. The freely available *ATLAS* (Automatically Tuned Linear Algebra Software) BLAS implementation [10] was used because it outperforms many other implementations (including some of the well-known commercial libraries).

Each of the 16 processing nodes called approximately 7350 times the level-3 BLAS routine *dgemm()* which consumes a time around 1435 seconds (73.9% of the total running time), and called 700 times the data broadcast function to transfer around 755 MiB of data in 280 seconds (14.4% of the total running time).

Figure 13 shows a snapshot (seconds 35 to 84) of the *HPL benchmark* running on 12 *Intel Celeron* CPUs (2.0 GHz each), connected by a *Fast Ethernet* network. The

[8] see http://www.netlib.org/benchmark/hpl/

[9] see http://www.top500.org/

[10] see http://math-atlas.sourceforge.net/

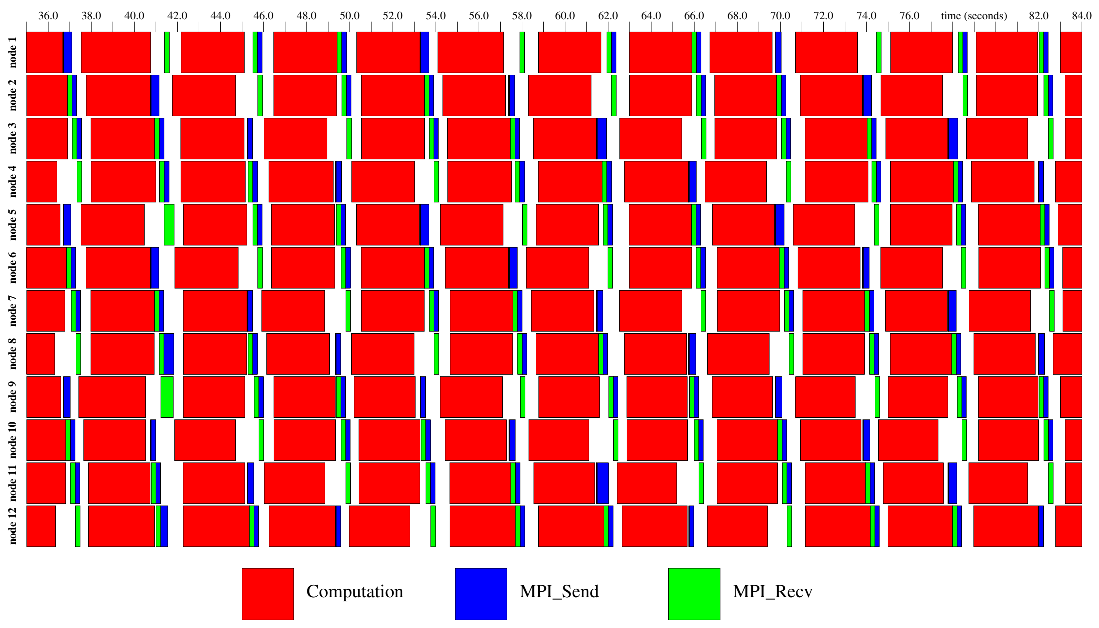


Figure 13: HPL benchmark running on 12 nodes

executable was linked against *Open MPI*-1.1 and *ATLAS*-3.7.11. A problem of size $N = 24576$ with a blocking factor of $NB = 40$ was solved using a processor grid of $P*Q = 3*4$ and the ”*1ring*” broadcast algorithm. The red boxes show the computation slices (i.e. the calls to *cblas_dgemm*), and the blue and green boxes show the data transmission operations within the *HPL_bcast*() function.

2.1.3 Conclusion

The *HPL* benchmark is a typical round-based application where the main computational parts are periodically interrupted by shorter communication parts. The data broadcast function (delivered with HPL) is written to allow an explicit overlapping of communication and computation by using non-blocking point-to-point communication functions. However, most open source MPI implementations today do not really benefit from using those function. So it was not astounding that replacing the non-blocking with blocking functions in the data broadcast algorithm let the benchmark report nearly identical performance numbers. Exchanging the different broadcast algorithms gives slightly different running times. Quite large messages (usually more than $1\ MiB$ in size; starting with larger ones and decreasing in size over the time) are broadcasted to all ranks within the MPI job. Although concurrent computation parts need nearly the same amount of time, the available broadcast implementations introduce additional gaps between consecutive rounds.

The new broadcast algorithm, based on *IP multicast*, should be able to achieve a good throughput and can hopefully diminish those gaps between the rounds because of its balanced manner.

2.2 Abinit

Abinit [11] is an application package to find the total energy, charge density and electronic structure of systems made of electrons and nuclei. The main code exists as sequential version (abinis) as well as parallel version (abinip).

2.2.1 Algorithm

The *Abinit* application takes the description of the unit cell and atomic positions and assembles a crystal potential from the input atomic pseudopotentials. It uses either an input wavefunction or simple gaussians to generate the initial charge density and screening potential, then uses a self-consistent algorithm to iteratively adjust the planewave coefficients until a sufficient convergence is reached in the energy. The code can be adjusted to perform molecular dynamics or to find responses to atomic displacements and homogeneous electric field, so that the full phonon band structure can be constructed. There are several approaches to parallelize this task [HR05]. We will devote our attention to the version which uses parallelism over the bands.

2.2.2 Results

Figure 14 shows a snapshot (seconds 116 to 124) of the parallel version of *ABINIT* running on 8 nodes of the *FRIZ* cluster (see 4.1.1). After each band computation (the red blocks), the root node collects the intermediate results and decides upon the next ”best” wavefunction. Subsequently the root node broadcasts the new block to all other processors using *MPI_Bcast()* (the yellow parts).

2.2.3 Conclusion

The parallel *Abinit* application is round-based too. Yet this form of parallelization is not as balanced as the *HPL* benchmark. Due to the additional gather part of the root node, it is nearly always the last node which enters the collective broadcast operation. For that reason all other nodes are waiting quite long before the actual transmission phase begins. The broadcast messages in this setup contained always $370,560$ double precision values (the size is therefore a few megabytes) which are broadcasted to all ranks in the MPI job.

Although the percentage of the consumed broadcast time regarding the total execution time of *Abinit* is very high, most of the time is wasted in the waiting phase of *MPI_Bcast*() which cannot be shortened significantly by using another blocking broadcast algorithm. A nice advantage for the multicast implementation is the fact that the root node is always the last node which joins the collective operation. Therefore an immediately executed *multicast* can be expected to be most effective because no packets need to be discarded at the receiver sides. On the other hand, this is also a drawback because the intermediate phase of the broadcast has an extent of zero. The relatively

[11] see http://www.abinit.org

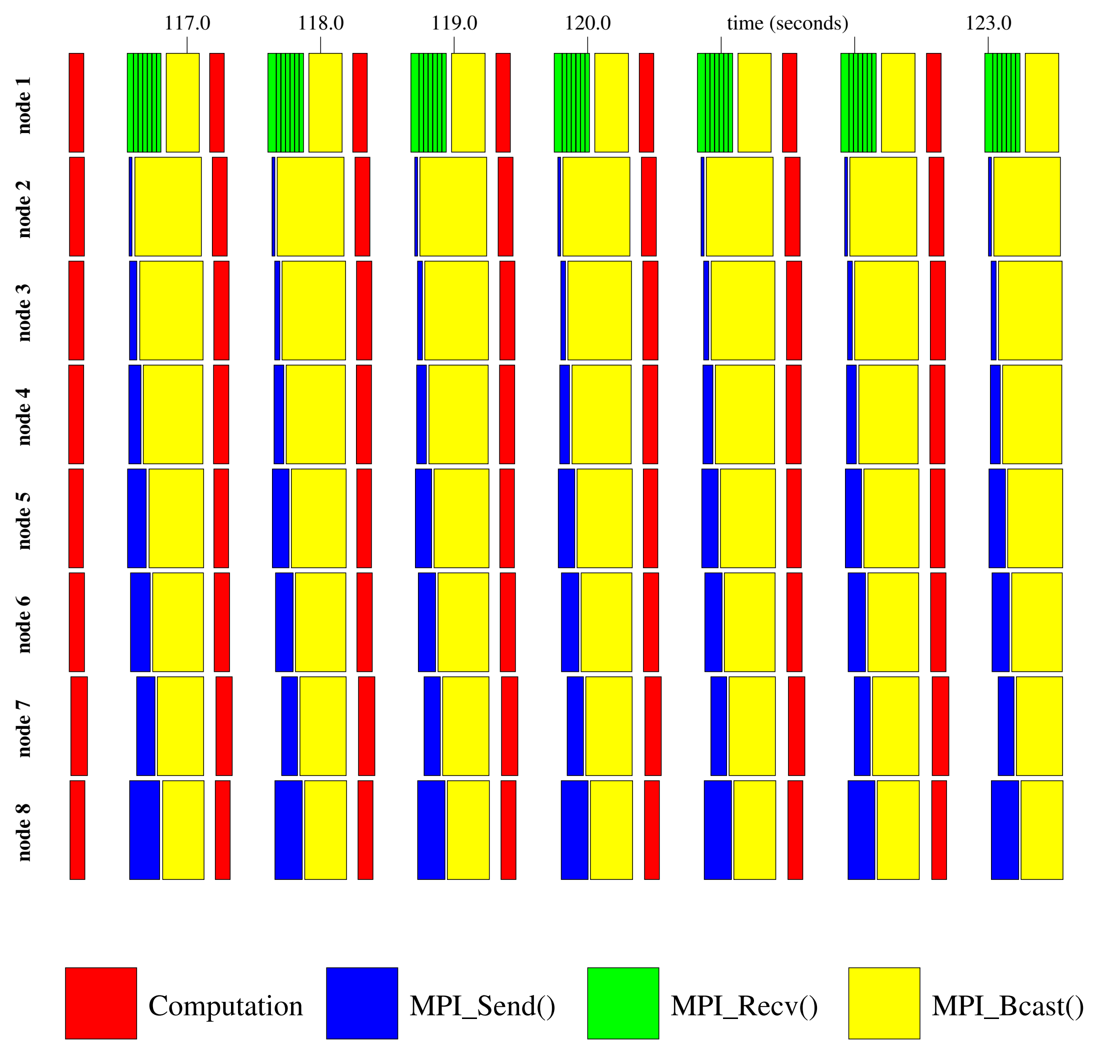


Figure 14: ABINIT running on 8 nodes

large message size and the small number of involved processes in the test scenario, should make the *fragmented chain* algorithm the best choice.

2.3 Statistical Properties and Assumptions

Most applications with different algorithms (and therefore behaviour) will have their own unique ”broadcast fingerprint”. There cannot exist a non-adaptive general broadcast algorithm (*NAGBA*) which is always superior to all other broadcast algorithms in all imaginable scenarios. Proof: take a fixed scenario with any regular application, where regular means that when the application is running several times under the same conditions, it will always behave exactly the same. Theoretically, all (maybe an infinite quantity of) parameters (and options) can be investigated and their values can be determined exactly (after an infinite amount of time). Once all parameter values are known, a deterministic schedule can be created which results in the optimal broadcast algorithm for exactly this application and scenario. The *NAGBA* can never be better than the so constructed broadcast algorithm. Once we change only a single parameter which is not known to the *NAGBA*, a better performing broadcast algorithm can be constructed, which therefore beats the *NAGBA*.

Either we construct a special broadcast algorithm for a chosen application and scenario, or we build a general-purpose broadcast algorithm which should hopefully perform well for a large class of applications and scenarios.

Figure 1 from the introductory chapter shows how every *MPI_Bcast*() operation can be subdivided into three phases. As long as this operation is blocking, the first phase can never be used to make any progress. The only solutions to this problem is to adapt the application or to make the broadcast operation non-blocking, which helps to postpone this task and execute succeeding operations earlier. A common approach to achieve this is to introduce new collective operations with another semantic, which work in a non-blocking way (see [HSB+06]). Another approach is to use the concept of *Memory-Mapped Messages* to maintain the semantic of a blocking behaviour and implicitly achieve the advantages as if the operation would be non-blocking (see [SR06]). Interestingly, even with a blocking broadcast it is nevertheless often possible to reduce the time consumption of this first phase by choosing a well-balanced broadcast algorithm (examples are round-based application schemes where unbalanced broadcast algorithms can lead to undesirable deferrals, which usually widen the first phase in the consecutive round).

The intermediate broadcast phase should be used effectively by a broadcast algorithm so that most of the work has been done already before the final phase even starts. *Multicast* can be leveraged to achieve this effectivity. Because of its unreliable datagram transport, a message can be send to all nodes without knowing if they are ready or not, whereas reliable communication channels need time consuming handshaking operations (or additional buffering) during this phase. The *multicast* approach has two extreme cases: If the root node is the first node which calls *MPI_Bcast*(), then an immediately executed *multicast* operation would have no positive effect since the other nodes are not waiting for the datagrams causing them (in most cases) to get lost. If the root node is the last node joining the broadcast operation, then the *multicast* operation is most effective and will very likely reach all participating processes. The first ”bad” case can be turned into a much better case by simply delaying the *multicast* operation by a certain amount of time. For regular applications we can always find a good delay parameter so that the new broadcast algorithm achieves its optimal performance.

When the application behaviour is in a way unpredictable, all we can do is to assume some random order in which the nodes call *MPI_Bcast*(). Assuming a uniform distribution of the arrival time means that every node has the same probability to be the first node calling the collective operation. On average we can expect that when the root arrives, there are already $(p-1)/2$ nodes waiting and $(p-1)/2$ nodes will follow. This yield is not too bad, because this implies that about halve of all nodes are ready to receive the *multicast* datagrams, even if the *multicast* operation is started immediately. These insights lead to the final algorithm which can be found in the next chapter.

3 Solution

Now that we know the advantages and disadvantages of the *multicast* feature, as well as common application usage patterns of *MPI_Bcast*(), this section presents a reliable algorithm to implement this collective operation which benefits from this knowledge.

3.1 Objective Target

The main goal of this thesis is to construct a broadcast algorithm, which performs especially well for larger communicators in conjunction with small and medium sized messages. For large enough messages (with respect to the communicator size), we can always fall back to the *fragmented chain* algorithm, which can broadcast such messages very efficiently (see chapter 1.5.2). If it is possible, the new broadcast operation should scale independently of the number of involved processes. *Multicast* in combination with a clever way to restore the reliability should be the key to achieve this goal. Moreover, the new broadcast algorithm should still perform decently when used with real-world applications and not just with synthetical benchmarks.

3.2 Multicast-based Broadcast Algorithm

We propose a two-stage broadcast algorithm, where the first part uses the unreliable *multicast* feature to deliver the message to as many nodes as possible. The second part of the algorithm is necessary to ensure that all nodes receive the broadcast message, even in case when the first stage fails partly or completely. No node should be stalled unnecessarily long, and instead finish the operation as fast as possible after correctly receiving the message.

3.2.1 Stage 1: Unreliable Broadcast

The most common approach when using *multicast* is to wait until all processes are ready to receive the datagrams. This can be achieved by using the synchronizing *MPI_Barrier*() operation or something similar ([HACA00] suggests a binary tree gather or a linear algorithm for synchronization before starting the multicast operation). The big advantage is that no packets need to be discarded because of non-ready receivers. On the other hand, there are two drawbacks: It can be proven that any barrier operation needs at least $log_2\ p$ rounds. This proof of optimality for the barrier operation can be found in [HTM05]. So if we would use this operation in our algorithm, then we could never achieve a broadcast performance which scales independently of the number of nodes. The second disadvantage (when using an upstream synchronization operation) would be the complete dissipation of the first two phases of the broadcast algorithm, which does not make it ideal for real-world applications where those phases can consume a significant amount of time (usually all parallel applications are - to some extent - subject to the principle of *process skew*, because of e.g. process scheduling or unforeseeable interrupts). Those reasons suggest that the new algorithm should not use a preceding synchronization operation. Nevertheless, this introduces a problem: When an application causes every time the bad case, where the root node is the first node which calls *MPI_Bcast*(), this first stage could never be useful for the broadcast operation. Therefore an additional *delay* parameter is introduced, which tells the root node how long it should wait before initiating the multicast operation. The value of this parameter is zero per default which disables the sleep, or it can be positive to indicate the number of microseconds the root node should wait at the beginning. This value should be customizable by the user, and it might also be adjustable at runtime. However, the second alternative comes with several problems. An approach could let each broadcast operation keep track of the relative number of lost datagrams and adjust the *delay* parameter as necessary for succeeding broadcasts. If there is too much datagram loss, then the *delay* parameter should be increased. Unfortunately, since these statistics are collected locally for each process, a separate communication channel would be necessary to transmit this information to the root node. Even if this problem is solved (the next section will try to eliminate all ”backward” channels!), this information needs to be processed by a clever (e.g. heuristic) function which returns a promising parameter change value. This does not sound very hard, but imagine a round-based application which uses two broadcasts per round. These two broadcasts might be completely independent and behave oppositional. It is for this reason, wherefore the current implementation does not try to adjust this *delay* parameter at runtime, and instead gives the user the full control. Note that this parameter is usually only necessary for regular applications with the worst case behaviour. We have already shown that for a large class of applications, a zero-value is acceptable.

The broadcast message might be split into several fragments to fit into the multicast datagrams. A sequence number within each datagram helps to re-assemble the fragments in the correct order. Since the broadcast operation does not synchronize, it is possible that some nodes are still processing a certain broadcast, while some faster nodes are already processing the following broadcast. To prevent any overtakings, a broadcast identifier (*BID*) is assigned to each communicator and increases for every broadcast operation. This identifier is also transmitted with the multicast datagram to allow an receiver to detect any such overtakings. An optional data integrity check over the whole multicast datagram (without the *CRC* field itself) can be used to identify defective datagrams at the receiver side. This data integrity check is optional because *Ethernet* frames usually have already their own *Frame Check Sequence* field. Therefore this additional checking can be disabled by the user.

The data fields of a final multicast datagram used by the *ipmc* implementation can be seen in figure 15: It always starts with a $3\ byte$ *Sequence Number* which indicates the position of this fragment in the packed data buffer. Since the minimum payload size of a multicast fragment (except the last one, which can carry a smaller payload - up to a single byte) has been limited to $256\ byte$, message up to $4\ GiB$ can be handled correctly. Far before reaching this limit, the *fragmented chain* algorithm should take over the work. The next field (*BID*) is an $8\ bit$ identifier for the broadcast. It is followed by the actual fragment data which can have any size up to the specified *payload* boundary. If the optional data integrity check is enabled, an additional $4\ byte$ trailer is appended

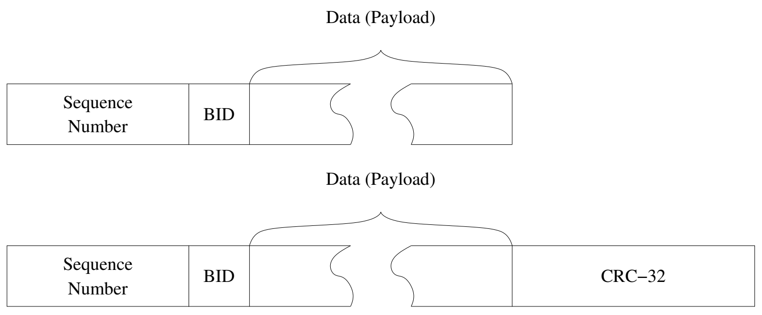


Figure 15: structure of a final multicast datagram

to the datagram, which contains the *CRC-32* value of the whole multicast datagram (including the header, but excluding the *CRC* field itself). Note that no separate length field is necessary because the length of the multicast datagram is returned by the corresponding receive function. In conjunction with the sequence number, the message size and the fragment size, it is possible to check each datagram for its correct length.

At the root node, the complete message is conveyed using *multicast* before starting the second stage. This is important since it is very likely that we are still in the intermediate phase of the broadcast operation, and several nodes might not yet be available for any reliable communication. A status bitmap can be used for each node, to keep track of received and sent fragments. All non-root nodes initiate an asynchronous multicast receive and update their local status bitmap for each correctly received fragment.

3.2.2 Stage 2: Reliable Broadcast Completion

It is always possible that several or all nodes have not received parts or the complete broadcast message correctly during the first unreliable broadcast stage. On the other hand, it is very likely that a large proportion (typically more than 50% [12]) of all nodes are getting the data correctly.

This second stage ensures that those nodes which have not yet received the data correctly, will accomplish this now. Many effort has been spent in the last years to construct reliable multicast transport schemes. There are even several working groups and research groups (e.g. in the *IETF* or *IRTF*). Nevertheless, they are usually designed for wide area communication (i.e. Internet) and not for high-performance cluster components. The common approach is to use some kind of acknowledgement scheme to detect which nodes have failed: This can be a positive acknowledgement where correct delivery is confirmed with an *ACK* and message loss is handled by time-out and retransmission. A negative acknowledgement based scheme is also imaginable where all nodes try to receive the fragments using time-outs and re-request the message from the root when the reception fails. This *NACK* scheme would incur no penalty when all nodes are receiving the message correctly. Except for the root node which has to ensure

[12] See the discussion about the statistical application behaviour, and the purpose of the introduced *delay* parameter.

that all nodes have received the message. However, this scenario can only be guaranteed, if the root node is known to call *MPI_Bcast*() as the last node, or when a separate synchronization operation is used. This reason and the always necessary time-out value dissuades to use this scheme for a general purpose broadcast.

The positive ACK scheme performs bad because the root node needs to wait for all ACKs and becomes a performance bottleneck (also referred to as *ACK implosion*). The authors of [JLP04] try to reduce these disadvantages slightly by introducing a *co-root* scheme as well as several other workarounds like *lazy ACKs*.

Another big problem is the time-out value, which is necessary for all ACK schemes and needs to be determined very carefully: A too small value can lead to *false retransmissions* and a too large value gives bad performance anyways.

An ACK scheme might be efficient when the broadcast message is large enough to cover all of the small message latencies as well as time-outs. Especially for small sized messages, this overhead is too much.

The solution to all those problems is to simply avoid any kind of acknowledgement scheme at all! To eliminate this expensive ”feedback”, we simply send the broadcast message a second time using the *fragmented chain* algorithm (see 1.5.2). This means that every node has a predefined predecessor and successor in a virtual ring topology. As soon as a node owns a correct fragment of the broadcast message, it sends it in a reliable way to its direct successor in the virtual ring topology (the root node does this too after completing the first stage). Whether the fragment has been received by multicast or by reliable send does not matter. Of course, the last node does not need to send the message to the root node. Each node posts a reliable (but asynchronous) receive for each fragment where the source node is the direct predecessor in the virtual ring topology. The root node does not need to receive anything because it already holds the complete message. Therefore the virtual distribution topology becomes effectively a chain and the algorithm becomes the already known chain broadcast. If a node obtains a fragment by the multicast receive request, then the reliable receive request can be cancelled or ignored.

At first sight, it might seem to be wasteful to use the chain broadcast for the second stage. When many consecutive nodes fail to get the message via multicast, it will take many ”penalty” rounds until they finally get the message via reliable send. A tree topology would drastically shorten the number of rounds in this case. There are at least two reasons for preferring the chain algorithm: An intermediate node in the chain algorithm only sends a fragment once, whereas in a tree algorithm it would send a fragment several times. This would lead to an undesirable increase of the broadcast duration per node. The second reason is the usually small chance that a node does not get the message via multicast. For a given failure probability p, the chance that n nodes fail in a row is p^n and therefore this result converges exponentially towards the zero value. Usually this probability can be assumed to be at most 50% (see statistical discussion in 2.3), therefore the expected number of reliable communication rounds is (if X represents all possible numbers of necessary rounds from 0 to $commsize - 1$, then p_i is the probability that $X = x_i$):

$$E(X) = \sum_{i=1}^{n} x_i \cdot p_i = 0 \cdot \frac{1}{2} \,+\, 1 \cdot \frac{1}{2^2} \,+\, 2 \cdot \frac{1}{2^3} \,+\, \ldots \,+\, (n-1) \cdot \frac{1}{2^n}$$

$$E(X) = \sum_{i=1}^{n} (i-1) \cdot \frac{1}{2^i} \qquad \lim_{n \to \infty} \sum_{i=1}^{n} (i-1) \cdot \frac{1}{2^i} = 1.0$$

This means that even for extreme large communicators ($n \rightarrow \infty$), the expected number of reliable communication rounds per node is less than or equal to 1.0, when the failure probability is at most 50%. In simple words: yes, it is possible that a node needs to wait many rounds until it gets the data. However, the probability that such a bad case occurs is negligible. In practice, almost all nodes will have to wait at most a few communication rounds before getting their data.

A nice side effect of using the *chain* algorithm in the second broadcast stage is, that for larger messages we could simply drop the multicast stage (this means 100% datagram loss!) and reach the highest performance of the *fragmented chain* broadcast.

3.3 A collector to create a nearly-true random seed

In the next section, a good pseudo-random number generator will be presented. Since it is still a generator, it needs some kind of initialization. A proper initialization, especially with true random data, is necessary to minimize the chance of generating the same output twice. Unfortunately, computers are deterministic machines which can't really produce true random data. A good workaround for this problem is the usage of statistic and timing data which is often influenced by other causes like human interaction or small timing derivations in the hardware level. For our purpose, it is enough to create a collector that gathers only a few bits of good random data. This is enough to get different generator seeds for each new initialization with a high probability. The portable implementation uses a complex data structure and gathers the results from several different functions:

- *MPI_Get_processor_name()* is not really random, but at least distinguishes between different MPI processes.
- *MPI_Wtime()* is a high-resolution timer and therefore a much better source of randomness if it is called rarely.
- */dev/urandom* is a non-blocking device in Linux which outputs quite good random data (it is only used if it is available). [13]
- *tmpnam()* should return different strings each time it is called (up to *TMP_MAX*). So even in the unlikely case that two MPI processes on a single node are calling

[13]The */dev/random* device usually blocks when the entropy pool is empty. Cluster nodes (like servers) are often short of entropy sources because there is no human interaction. Therefore the use of the */dev/random* device - contrary to the use of */dev/urandom* - is not recommended.

this collector at the same time (e.g. on a dual core machine), this source should lead to a different seed.

It is easy to add more sources to this collector. This subset however should be enough to get a decent seed value for our purpose.

Finally, after collecting all those bytes together, they will be compressed down to 64 bit using a hash function. The result will be used to seed the pseudo-random number generator which will be explained in the next section.

3.4 Blum-Blum-Shub pseudorandom number generator

It would be best to collect enough true random data to select a proper multicast group and a corresponding port number. This would minimize the chance that several communicators (even in different and independent MPI entities) choose colliding identifiers. Unfortunately, the total amount of entropy that will be collected by our portable and non-blocking implementation can drop to just a few bits of true random data in the worst case. For a 32 bit IPv4 address and a 16 bit port number, we need around 48 bit (a bit less because we do not accept the full range) of good random data (for an IPv6 address even more). A special kind of stretching function should be used to fill this gap (sometimes called *amplifiers* of randomness). It takes the collected data and produces a large enough stream of pseudorandom data. If there are two different sets of collected data which differ only by at least one bit (e.g. influenced by the true random bit), then an optimal function should return two pseudorandom data streams where around halve of all bits are different.

On way to achieve this objective, is to use a so-called hash function which takes an arbitrary amount of data (i.e. the collected data in our case) and produces a fixed-length output. Possible candidates, with the desired property that a single bit changes approximately halve of the output bits, are cryptographic hash functions like *MD5* or *SHA-1*.

Another solution is a pseudorandom number generator, which will be seeded with the collected data. Such a generator would be able to produce any amount of pseudorandom data instead of a fixed amount, and in addition it often requires a much less complex implementation. Many low quality pseudorandom number generators exist (e.g. *Linear congruential generators* implemented with low precision integers) which on the other hand have a high amount of throughput. Since we need only a relatively small amount of random data (e.g. the 48 bits for IPv4), I suggest to use a slightly softened version of a cryptographically strong pseudorandom number generator which gives some kind of guarantee for the high quality of the generated output.

Blum-Blum-Shub (proposed in 1986 by Lenore Blum, Manuel Blum and Michael Shub [LBS86]) is such a pseudorandom number generator. The ingredients for this generator are two large prime numbers p and q which should be congruent to $3\ (mod\ 4)$. A small value for $gcd(\varphi(p-1), \varphi(q-1))$ ensures that the cycle length is large. It is initialized with a seed x_0 which can be any quadratic residue where $gcd(x_0, M) = 1$.

To generate a single output bit, this generator updates an internal state according to

$$x_{n+1} = x_n^2 \; mod \; M$$

where M is the product of the two prime numbers p and q, and returns the bit parity of the new state. The resulting sequence repeats after a period of $\lambda(\lambda(N))$.

In Annex E of ISO/IEC 9899:1990 (often called ANSI C standard), an **unsigned long** datatype is guaranteed to hold at least 32 bits (in other words it needs to be able to represent numbers ranging from 0 to $4,294,967,295$). Since this might be too small for our purpose, I suggest to use at least two such words and implement a minimal big integer package. To remain able to handle possible overflows, we could simply use 31 bit of each word, allowing us to calculate with integers of 62 bit precision.

A valid BBS modulus of this size is

$$M = 2^{62} - 63 = 4,611,686,018,427,387,841 = 64,129,007 \cdot 71,912,637,263 = p \cdot q$$

because $p \equiv 3 \; (mod \; 4)$ and $q \equiv 3 \; (mod \; 4)$. It is also a good modulus with a large cycle length of nearly 2^{61}, because $p - 1 = 2 * 32064503$, $q - 1 = 2 * 223 * 1223 * 131839$ and therefore $gcd(\varphi(p-1), \varphi(q-1)) = 2$. So even if the generator produces 100 million bits per second (a reasonable assumption for a modern CPU), then the first repetition can be expected after 731 years of continuous processing time.

A native implementation on a 64-bit architecture (using the "diminished radix" technique instead of real "div's" to reduce x^2 modulo M [14]) achieves an output rate of more than $8 \; MiB$ per second (tested on an *AMD Opteron 244 with 1.8 GHz*).

The following algorithm to calculate $x^2 \; mod \; M$ will be used to ensure portability:

```
r ← 0
a ← x
b ← x
*while* (b ≠ 0) *do*
    *if* is_even(b)            ♯ a · (2 · b') $mod\ M$ = (2 · a) · b' $mod\ M$
        a ← 2 · a $mod\ M$
        b ← b ≫ 1
    *else*                     ♯ a · (2 · b' + 1) $mod\ M$ = ((2 · a) · b' + a) $mod\ M$
        r ← (r + a) $mod\ M$
        a ← 2 · a $mod\ M$
        b ← (b - 1) ≫ 1
*return* r
```

Figure 16: algorithm to calculate $x^2 \; mod \; M$

Finally, instead of looping through all 62 bits to get the parity, 6 XOR and 5 SHIFT operations suffice to accomplish this task.

[14] I'd like to thank *Tom St Denis* for this useful tip (hint: $(h \cdot 2^{62} + l) \; mod \; M = (h \cdot 63 + l) \; mod \; M$).

3.5 Implementation for Open MPI

All ingredients for the new broadcast implementation have been prepared in the previous chapters. This section will describe in more detail how these components are glued together to form a suitable implementation for *Open MPI*. A first prototype on top of *MPI* has been created first, to prove that the new algorithm is working in practice. This prototypical implementation is therefore usable with every *MPI* library, and not restricted solely to *Open MPI*. Subsequently, the *basic* component of *Open MPI* had been used as a starting framework, to integrate the functionality of the prototype into the component framework of *Open MPI*. The name of the new component is ”ipmc” which stands for *IP MultiCast*.

The final source code package of the new implementation (which is located under *openmpi-x.y.z/ompi/mca/coll/*.) contains the following files:

- *ipmc/coll_ipmc_bcast.c*
- *ipmc/coll_ipmc_component.c*
- *ipmc/coll_ipmc.h*
- *ipmc/coll_ipmc_module.c*
- *ipmc/coll_ipmc_util_crc.c*
- *ipmc/coll_ipmc_util.h*
- *ipmc/coll_ipmc_util_ipv4.c*
- *ipmc/coll_ipmc_util_oob.c*
- *ipmc/coll_ipmc_util_random.c*
- *ipmc/configure.params*
- *ipmc/Makefile.am*
- *ipmc/README*

The ”README” contains a textual description of the package, some installation instructions, and an explanation of all parameters which can be changed by the user to influence the behaviour (and performance) of the *ipmc* implementation. Currently there are the following seven parameters:

- *coll_ipmc_priority* The collective component with the highest priority will be used in *Open MPI*. This parameter describes the priority of the *ipmc* component. The default value is 40, which makes it a bit higher than the priority of the *tuned* component, so that it will immediately get active after an installation.

- *coll_ipmc_crossover_nodes* For small communicators, the *ipmc* broadcast can be slower than an ”usual” broadcast algorithm on top of point-to-point communication. Exactly which broadcast algorithm is the best for small communicators depends on the scenario (see also chapter 1.5). The current implementation falls back to the improved linear broadcast because it is usually more suited to use the intermediate phase of the broadcast operation than the other algorithms. This crossover value determines the minimum communicator size at which the multicast-based algorithm will be used. Although the theoretical value for the optimal crossover value is approximately 8 nodes for synthetical benchmarks, the default value is 4 because applications are usually subject to the principle of *process skew*.

- *coll_ipmc_crossover_size* For very large messages, the *fragmented chain* broadcast is the best choice. This crossover value determines the maximum size of a message at which the multicast-based broadcast will be used. The default value is 1048576, which means that all message above $1\ MiB$ will be broadcasted using the *fragmented chain* algorithm. Note: These two boundary values are in reality both dependent on the message size and the communicator size. One possibility would be to estimate the running time of all three broadcast algorithms with e.g. the *LogGP* model. On the other hand, this would need an exact determination of all *LogGP* parameters, and would still ignore the *process skew* which is hard to determine in advance. These two crossover values are easy to understand for all users and therefore allow a much better user control over the choice of the right algorithm for the user’s application.

- *coll_ipmc_fragment_size* Since *IP* packets as well as datagrams itself have a limited maximum size (usually $65,535$ and *MTU*=$1,500\ byte$), this parameter prescribes the maximum payload size of an *IP multicast* fragment for this implementation. Measurements on the *CLiC* cluster resulted in an optimal fragment size of $4096\ byte$, which is therefore chosen to be the default value. The minimum value of this parameter is limited to $256\ byte$.

- *coll_ipmc_root_wait_time* If the root node of a broadcast is often the first node entering *MPI_Bcast*(), then it is possible that most (or even all) multicast datagrams will get lost. With this parameter you can advice the root node to wait a certain number of microseconds ($1\ \mu s = 10^{-6}\ seconds$) before issuing the multicast operation. The default value is 0, which means that the root node never waits (i.e. it starts the multicast as soon as possible).

- *coll_ipmc_use_crc_checking* Although it is normally not necessary, this switch can be used to force an additional *CRC-32* data integrity check for each multicast datagram. Corrupt data packets can be identified with high probability and will be dismissed. A value of 0 deactivates this check, and any other value enables this additional checking. It is activated per default. If you are sure that no corrupt datagrams will be delivered, you can turn this checking off and get a small additional performance gain (because multicast datagrams are $4\ byte$ shorter and some processor cycles are saved). On the other hand, we have noticed no significant performance penalty on *Fast Ethernet* (only a 0.83% degradion for $16\ KiB$ messages).

- *coll_ipmc_print_statistics* This switch can be used to print some useful statistics every time a communicator is destroyed, like the number of executed *MPI_Bcast*() operations and multicast datagram information. Here you can find out how many datagrams were sent or received, and how many of them were useful for the broadcast or rejected. Note: These statistics are generated for each involved process. A non-zero value activates this output, which is disabled per default.

The files ”configure.params” and ”Makefile.am” are used by the script ”autogen.sh” to produce the ”configure” script in the top level directory and the template file ”Makefile.in”, which are later used for the usual build procedure (”configure” and ”make”).

The new *component* uses a whole bunch of utility functions which are specified in the header file ”coll_ipmc_util.h”.

”coll_ipmc_util_crc.c” contains the function *coll_ipmc_util_calc_crc*() which is used to calculate a cyclic-redundancy-check value (*CRC-32*, see also [Deu96]) for a given buffer. *Open MPI* comes with an own CRC calculation function. Unfortunately, this function is currently buggy and does not work properly. Once those bugs (e.g. the non-existing support for heterogeneous systems) are removed, this function could be replaced. The typical way to speed up CRC calculations is to use lookup tables. A second function is used to create this lookup table (e.g. during the module initialization).

”coll_ipmc_util_random.c” is a bit more extensive and contains the entropy gather function *coll_ipmc_util_random_gather*(), which can be used to seed the pseudorandom number generator state with *coll_ipmc_util_random_seed*(). Once this has been done, it is possible to extract an arbitrary amount of pseudorandom data in form of bits (*coll_ipmc_util_random_get_bit*) or in form of integers between a specified range (*coll_ipmc_util_random_get_ulong*). How the gatherer and the generator work, has been described already in chapter 3.3 and 3.4.

The next larger collection of utility functions covers all the network related functionality and can be found in ”coll_ipmc_util_ipv4.c”. Note that the *ipv4* suffix as well as the chosen form of all function has been introduced with caution, to make it easier to switch to *IPv6*, or even a completely different network interface. This file exports a function to find an unused multicast group and port number, which can be assigned to a new communicator. A two-layered approach is intended here: this function should first try to contact a *MADCAP* server (this is not yet implemented), and alternatively choose the values at random (the already mentioned utility functions from the random package are used here). Another (currently stub) function can be used to free such allocated values when they are not necessary anymore (e.g. when a communicator is destroyed). Then there are functions to create and close a socket which is suitable for *IP multicast*. Two separate preparation functions exist to make a receive socket listen to a specific multicast group and to set a bunch of options for a given send socket. Finally there are two functions to send and receive messages using these sockets. It should be noted that the receive function is non-blocking and can therefore be used in a polling way.

File ”coll_ipmc_util_oob.c” has been added lately to solve a problem with communication during the communicator initialization phase. Originally it was intended to use collective operations from the *basic* module for setup communication. In current versions of *Open MPI*, this is not possible anymore (it fails for larger communicators). As long as this possibility is not re-established, this workaround uses the slower but more stable *OOB* communication. The original (much smaller) code fragments are still contained in the code and can be reactivated easily. Two point-to-point functions are contained within this file: *mca_coll_ipmc_oob_sendto*() sends a small message to the specified destination rank, and *mca_coll_ipmc_oob_recvfrom*() receives a small message from a specified source rank. On top of these function two collective functions have been implemented: a simple broadcast function and a simple reduce function, both using a binary tree distribution topology. Those functions are only used during the initialization phase of a communicator and never for the final broadcast!

”coll_ipmc_component.c” contains the functionality to open the new *ipmc* component at startup. All user-visible parameters are initialized and registered here.

The query, initialization and finalization functions for our component are located in the file ”coll_ipmc_module.c”. The *query* function checks if a given communicator is an intra-node communicator, and if it contains at least two *MPI* processes. Furthermore it checks whether or not *IP multicast* is potentially working. If at least one of those requirements is not fulfilled, then this function returns -1, indicating that it wants to be rejected (another collective component will than be used). An elaborately *multicast* test would be quite expensive. Therefore only a quick test has been implemented. The *initialization* function prepares several (sometimes time-consuming) things that are used instantly in later *MPI_Bcast*() calls. The special rank $\sharp 0$ calls the function to get a free multicast group and port number, and broadcasts the results to all other nodes. Afterwards all nodes create two multicast sockets (one for sending and a second one for receiving), and they try to join the given group as well as bind the receiving socket to the unique port number. Finally, an *MPI_Allreduce*() similar function is used to find out if all nodes could be initialized correctly or if one or more nodes failed to do this. Note that the two provisional collective functions are uses exclusively for this purpose. If all nodes are initialized successfully, a proper collective module is returned to the *Open MPI* instance. If something on any node went wrong, the complete initialization procedure is tried some more times or finally given up. The *finalize* function releases the allocated multicast group and port number, frees all allocated resources and prints some useful statistics if the user has enabled this feature.

Last but not least, the file ”coll_ipmc_bcast.c” contains three different broadcast implementations:

1. the fragmented chain algorithm,
2. the linear broadcast algorithm, and
3. the multicast-based algorithm.

The first two algorithms are derived from already existing implementations in the *tuned* and *basic* component. Therefore they need not to be explained in detail, contrary to the new multicast-based broadcast algorithm. The chain algorithm tries to split the message into several fragments (using the *count* argument and the size of the specified datatype) before sending them in a virtual chain topology. The linear algorithm lets the root node initialize $p - 1$ non-blocking send operations which are matched by a single receive operation on each non-root node. The root node than waits for the completion of all operations with a call to *ompi_request_wait_all*().

The new multicast-based broadcast algorithm first determines the size (or at least an upper bound) of the raw message using *MPI_Pack_size*(), and allocates a temporary buffer for this packed message as well as a bitmap holding the current status of each fragment. This is acceptable because we are usually only sending small or medium sized messages with this broadcast algorithm (see *coll_ipmc_crossover_size* parameter). If the *coll_ipmc_root_wait_time* parameter is larger than zero, then the root node waits this number of microseconds. After this optional delay, it packs the original message into the packed buffer. Now it owns each fragment which is indicated by updating the status bitmap. This message is now conveyed fragment-wise by sending it in repackaged datagrams to the multicast group and port number which are assigned to this communicator. All nodes enter the second stage of the broadcast algorithm where they receive fragments (either reliable or unreliable) and forward them in a reliable way within a virtual chain topology. Two reliable requests are used to send and receive the fragments as necessary. The status bitmap helps to remember which fragments are

1. already owned by this rank,
2. already received using the reliable channel, and
3. already sent using the reliable channel.

A fragment needs to be owned by a rank before it can be forwarded to its direct successor. One reliable receive requests is posted for each fragment which is not yet received using the reliable communication channel. If a multicast datagram is received, then its sequence number, broadcast identifier and optional checksum value will be extracted and checked for validness. The result will be noted using the existing counters for received, useful and rejected datagrams. A correct fragment will be copied into the raw message buffer and the status of this fragment is updated. When all fragments have been received and sent using the reliable channel, this broadcast stage completes. Finally, each non-root node unpacks this raw message into the user-supplied data buffer. After deallocating the temporary buffer, the multicast-based broadcast functions returns to the caller.

When the application invokes *MPI_Bcast*(), the function *mca_coll_ipmc_bcast*() will be called. This function decides at runtime upon the current scenario (message size, number of processes and user parameters) which of the three broadcast algorithms should be used. If the message is larger than the specified *coll_ipmc_crossover_size* parameter, then the *fragmented chain* algorithm will be called. If this is not the case and the communicator contains less than *coll_ipmc_crossover_nodes* nodes, then the *linear* broadcast implementation will be called. Finally, if the message is not too large and the communicator is not too small, the multicast-based broadcast algorithm will be used.

4 Practical Results

This chapter will compare the new broadcast implementation with existing implementations. At the beginning, the environment (hardware and software) on which the numerous tests and measurements have been executed, will be presented. A method for measuring the broadcast duration separately for each node will be explained, before showing microbenchmark results for large (up to 342 nodes) and smaller communicators. Finally, the effect on the already introduced parallel applications will be analysed.

4.1 Benchmark Environment

Mainly, two different clusters have been used during the development of the new *ipmc* broadcast implementation. Both clusters belong to the equipment of the *Chemnitz University of Technology*. All presented measurement results in this thesis have been obtained using those environments. To ensure repeatability of those results, all tests have been executed at least two times, to prove that they reproduced similar results at least once.

4.1.1 FRIZ

The smaller test system is a computer pool of the faculty computing center (in German: *Fakultätsrechen- und Informationszentrum - FRIZ*). A subset of all available nodes has been grouped to form a cluster of 16 nodes, each equipped with

- Intel Celeron 2.0 GHz processor
- 512 MiB main memory
- *SUSE* Linux 9.3 (kernel $2.6.11$)
- *GCC* $3.3.5$ and *G95* $0.90!$

All nodes are connected with a single *Fast Ethernet* switch.

4.1.2 CLiC

The larger system, called *CLiC* (which stands for *Chemnitzer Linux Cluster*), is a cluster of the university computing center. Since the year 2000, it is the largest *Beowulf*-style cluster at *Chemnitz University of Technology*. It will be soon replaced with a modern cluster of approximately the same number of nodes. Each of the 528 nodes is equipped with

- Intel Pentium III 800 MHz processor
- 512 MiB main memory
- *Red Hat* Linux 7.3 (kernel $2.4.18$)
- *GCC* 2.96 and *GNU Fortran* $0.5.26$

The communication network contains a single large *Fast Ethernet* switch (*Extreme Black Diamond*, with $6 \star 96$-port modules), which is directly connected with all nodes. A second service network exists, but has been explicitly disabled for each tests using the

```
--mca btl_tcp_if_include eth1
```

parameter (this instructs *Open MPI* to use only the communication network).

4.2 Microbenchmark Results

This section presents a synthetical microbenchmark for measuring the duration of the *MPI_Bcast*() operation. It investigates the performance of the used broadcast implementation for different message sizes as well as a various number of processes per communicator. The results show the scaling behaviour and can be used to estimate the performance in other scenarios (e.g. with real-world applications). However, this microbenchmark assumes that all nodes call nearly at the same time *MPI_Bcast()*.

Especially for all measurements with a very small timing (i.e. small message size in our case), it is good to repeat the measurement several times. There are often some ”runaways” which need much longer to complete (caused e.g. by additional or unexpected events like interrupts). Therefore many benchmarks output the minimum measured time. Though if you analyse an aggregation of measurements by plotting them in the sorted order, then you will notice that the minimum time is in most cases a ”runaway” as well, whereas typically more than 90% of all measurements are very similar. One could use the average value over all measurements, but a single extreme ”runaway” is sufficient to nullify the result. All in all, I suggest to use the median value (the value which is located in the middle of all sorted values), which represents the duration that can be expected.

4.2.1 Measuring Broadcast/Multicast Performance

Many collective benchmarks measure only the maximum duration of a given operation. For example [FK99] (section 5.2.1) suggests to call *MPI_Bcast*() several times after an initial synchronization point (*MPI_Barrier*) and finally extract the maximum measured time duration [15]:

[15] The original suggested algorithm has been slightly modified (For example: The MPI standard does not guarantee that *MPI_Wtime*() is globally synchronized.)

```
...
MPI_Barrier(comm);
totTime = -MPI_Wtime();
for (i = 0; i < NUMREPEATS; i++) {
    MPI_Bcast(data, len, MPI_BYTE, root, comm);
}
totTime += MPI_Wtime();
MPI_Reduce(&totTime, &maxTotTime, 1,
    MPI_DOUBLE, MPI_MAX, root, comm);
if (myrank == root) {
    maxTime = maxTotTime / NUMREPEATS;
}
...
```

Listing 5: algorithm to measure the maximum MPI_Bcast() duration

Unfortunately, this method for measuring the maximum broadcast duration can report misleading results: Imagine the normal *chain algorithm* which has a pretty bad worst case running time (scales linearly with the communicator size). If you try to measure this algorithm with the above suggested method, then you will implicitly introduce a pipelining effect and get a ”perfect running time” when the number of loops is large enough. [PPY06] suggests a similar technique to measure the broadcast performance, and adds an *MPI_Barrier*() operation after each *MPI_Bcast*() to prevent this pipelined communication between iterations. The drawback of this workaround is that this newly introduced operation can increase the measured durations dramatically, especially for smaller message sizes (for this reason the authors measured only with message sizes above $8\ KiB$ and ignored the barrier overhead).

Therefore I suggest another and more comprehensive broadcast benchmark, which measures the broadcast completion time for each node separately. Slightly modified, this benchmark can also be used to measure multicast performance. This benchmark has the advantage that all performance numbers (like minimum, average or maximum) can be easily derived from the results. It is even sometimes possible to reconstruct the exact distribution topology (e.g. binary tree) using those results. For a given scenario (predefined communicator and message size), a memory block is copied in a *ping-pong* fashion from one buffer at the root node over the network, back into a second buffer at the root node. The forward transfer (*ping*) will be accomplished by the *MPI_Bcast()* operation, and the backward transfer (*pong*) will be accomplished by a simple point-to-point operation, which is only initiated by a predefined target node. It is not possible to measure all target times at once using a *ping-pong* scheme, because the root node would be a bottleneck for all incoming *pongs*. A special synchronization procedure (*MPI_Barrier*() often fulfills the needs - but this depends on the underlying barrier implementation) should take care that the root node is the last node entering the succeeding *MPI_Bcast*() operation and all other nodes are already waiting therein. The accuracy can be further improved by using *MPI*'s ready-mode send for the *pong* operation, which - especially for larger message sizes - prevents the usage of the more expensive and often slightly less predictable *rendezvous protocol*, and uses always the *eager protocol* instead. Finally, a separate (ready-mode) *ping-pong* to each node is necessary to obtain the duration of a the *pong* operation, which is then subtracted from the *MPI_Bcast()-pong* time to get the raw *MPI_Bcast*() duration per node.

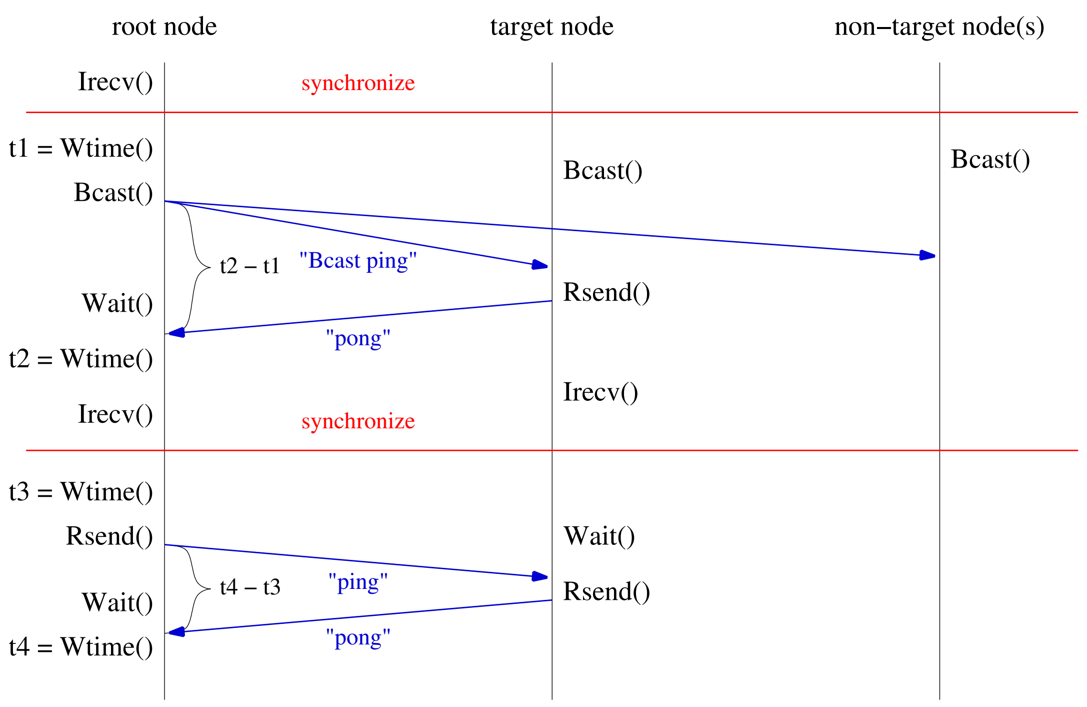


Figure 17: a single round to measure the broadcast duration per node

The complete broadcast benchmark runs over different communicator sizes (using *MPI_Comm_split*) and different message sizes. For each target node, a single measure round - according to figure 17 - is performed. Since the two *MPI_Rsend*() operations in the second part are very similar (they are just working in the opposite direction), it can be assumed that they need the same amount of time (i.e. $t(\mathit{ping}) = t(\mathit{pong})$). Therefore, halving the *ping-pong* duration ($t4-t3$) reveals the *pong* duration, which can then be subtracted from the *Bcast ping-pong* duration to get the raw broadcast duration to this target node. The separation into two buffers at the root node allows checking for transmission errors (e.g. in case of *multicast* or a new *MPI* implementation) and permits fine control over the wanted cache behaviour.

The target node can be predefined in case of a reliable broadcast (e.g. *MPI_Bcast*), or it can be specified within the message (e.g. when unreliable multicast is used). The root node should use *time-outs* in the latter case to avoid stagnation when datagrams are lost.

4.2.2 Results on Large Communicators

The performance chart in figure 18 shows the average broadcast time per node, when the number of nodes increases up to 342 nodes. The measurements have been taken on the *CLiC* cluster, and the results compare the new *ipmc* broadcast with the original broadcast implementation.

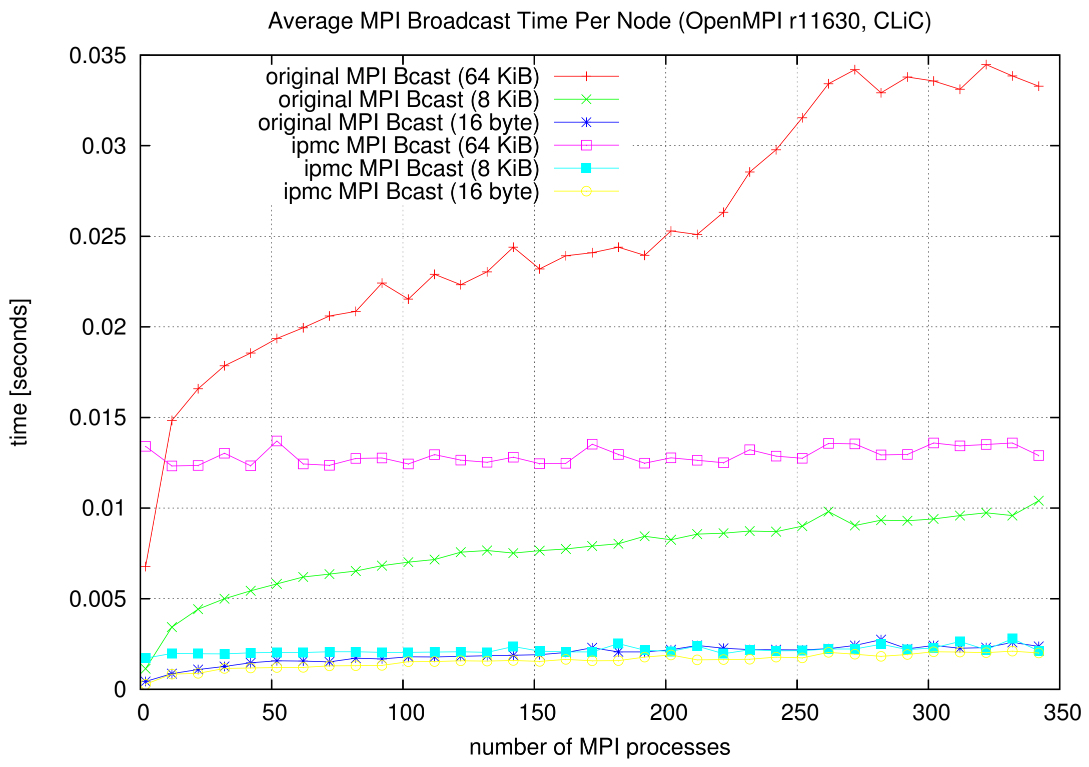


Figure 18: comparison of original and ipmc broadcast up to 342 nodes

Whereas point-to-point implementations get slower when the number of *MPI* processes increases, the almost horizontal curves of the new implementation show that - in practice - the new broadcast implementation scales independently of the number of involved processes. For example: a broadcast of a $64\ KiB$ message using the original *MPI_Bcast*() implementation needs only 0.0068 seconds per node when only 2 nodes are involved, but takes 0.0339 seconds per node when 332 nodes are involved (this is a performance loss by a factor of 4.985!). The new implementation needs 0.0134 seconds per node when only 2 nodes are involved, and needs 0.0136 seconds per node when 332 nodes are involved! A broadcast of an $8\ KiB$ message to 342 nodes is by a factor of 4.896 slower (0.002125 seconds versus 0.010405 seconds) when the original implementation [16] is used.

[16]The original broadcast implementation is selected upon several criteria within the *tuned* component of *Open MPI*. In the presented scenario (small to medium sized messages and large communicators), it uses the *binomial tree* algorithm.

Another advantageous aspect of the new broadcast is the fact that it is very well balanced. Figure 19 shows the broadcast duration for each of the 342 nodes. Whereas the original *MPI_Bcast*() implementation with a $64\ KiB$ message needs a time which varies between 0.0126 seconds and more than 0.04 seconds, the multicast-based implementation needs a very similar time on each node (between 0.0123 seconds and 0.0145 seconds). This means a process skew of up to 3.18 times the broadcast duration when the original implementation is used, whereas the process skew of the new implementation is only up to 1.17 times the broadcast duration.

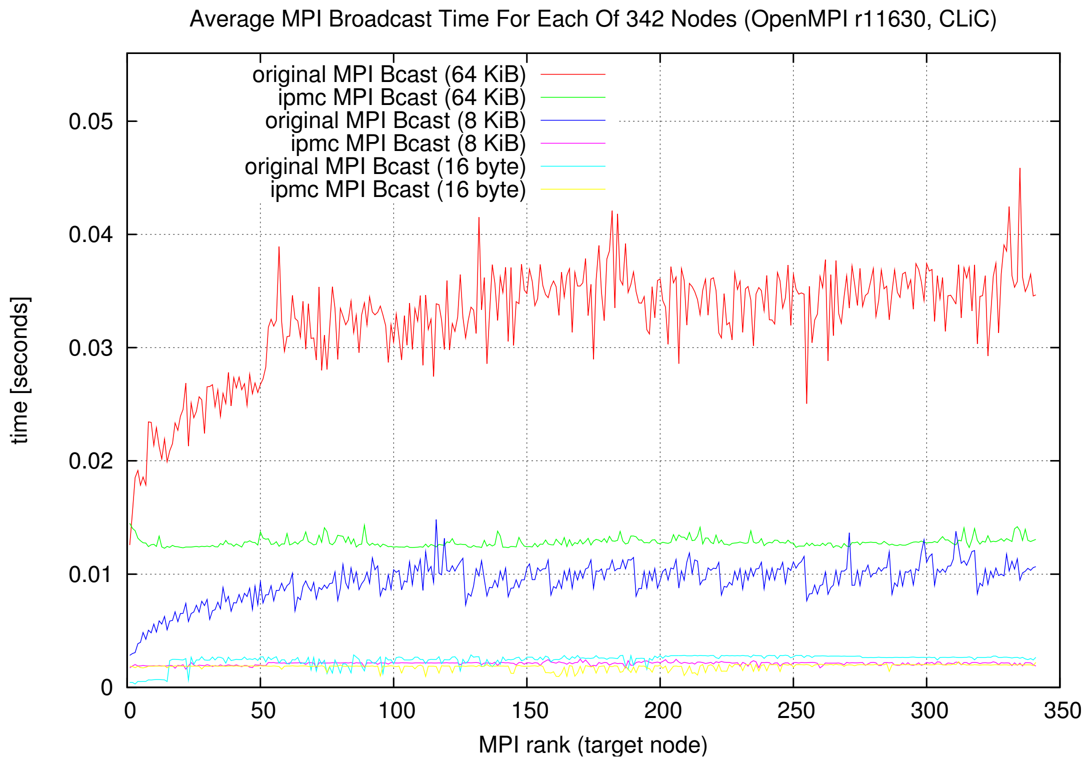


Figure 19: comparison of original and ipmc broadcast with 342 nodes

Conclusion: The new multicast-based implementation of *MPI_Bcast*() achieves a nearly constant running time for any given message size [17]. The term "constant" has a double meaning here: On the one hand it scales nearly independently of the communicator size, and on the other hand for any given communicator size, all nodes need the same amount of time to complete the broadcast operation.

[17]The broadcast of larger messages is even more "constant" through to the positive effect of the fragmented chain algorithm in the second stage of the algorithm (even in the theoretical worst case where the multicast stage fails completely!).

4.2.3 Results on Smaller Communicators

In the last section it was shown that the new broadcast implementation emerges victorious, as soon as the number of involved processes crosses a certain boundary. Figure 20 compares the different *MPI_Bcast*() performance scalings again (with smaller communicator sizes) to find this decision boundary.

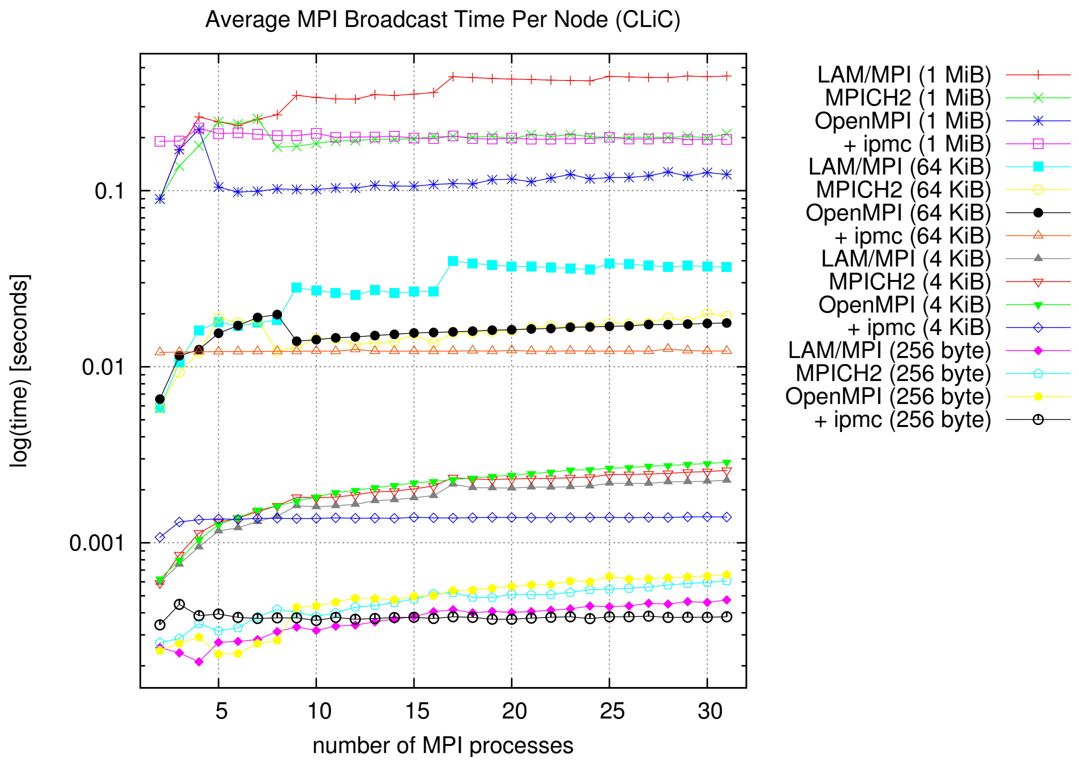


Figure 20: different *MPI_Bcast*() with respect to smaller node numbers

Note that this performance chart has a logarithmic time scale, so that the broadcast duration can be better distinguished for different message sizes ($1\ MiB$, $64\ KiB$, $4\ KiB$ and $256\ byte$). The most recent versions of three well-known open-source *MPI* implementations have been tested:

- *LAM/MPI* - one of the ”predecessors” of *Open MPI* - version 7.1.2
- *MPICH2* - from the *Argonne National Laboratory Group* - version 1.0.4p1
- *Open MPI* - with and without the new *ipmc* component - SVN r11682

The top-most curves show the broadcast duration of a $1\ MiB$ message. Because *LAM/MPI* never uses any fragmentation, the duration grows rapidly when the number of nodes increases (according to the *binomial tree* algorithm). Both other libraries, *Open MPI* as well as *MPICH2*, use fragmentation when the message size is large enough. Unfortunately, some people seem to believe that fragmentation over tree topologies or meshes can be efficient for large message sizes (see e.g. [WG95] or [Trä04]). However, the *fragmented chain* algorithm is by far the most efficient broadcast algorithm for large enough messages (section "Fragmented Tree vs. Fragmented Chain" in the appendix should clarify this if there are still any doubts). *Open MPI* is the only library of those three which uses the *fragmented chain* algorithm for large messages, and is therefore up to a factor of two faster than *MPICH2*, which uses implicit *fragmentation*, using the *scatter-allgather* broadcast. The current *ipmc* implementation is also not suited for such large messages, because it effectively sends the message twice: one time with unreliable multicast and a second time with reliable point-to-point communication. There are possibilities to make the *impc* implementation faster for larger messages (e.g. falling back to an *ACK* scheme), but it is better to find the crossover point and fall back to the *fragmented chain* algorithm.

Although both libraries are still using fragmentation for $64\ KiB$ messages, the *ipmc* implementation outperforms their broadcasts starting with 4 nodes. At $4\ KiB$ there is no fragmentation anymore, and all three *MPI* libraries fall back to a tree distribution scheme, which performs always worse than the *ipmc* broadcast, when the number of processes crosses the 8 node boundary. *LAM/MPI*, which initially achieves only poor broadcast performance for large messages (because it does not fragment), is always better than its opponents for smaller message sizes. In this example scenario, a 6 node boundary would be sufficient for *Open MPI* and this message size.

For very small messages, the current *ipmc* implementation might need a bit more tuning to achieve the theoretical crossover point of 8 nodes [18] and is therefore slightly outperformed at this point by *LAM/MPI*, which gets slower not before 15 nodes. However, for *Open MPI* and *MPICH2* this theoretical boundary holds, even for smaller message sizes (tested with e.g. $16\ byte$).

Conclusion: This section has verified that the new *MPI_Bcast*() implementation is almost always the fastest of the available broadcast algorithms for all small and medium sized messages, when the communicator size is larger than 8 nodes. For as little as 20 nodes, the *ipmc* broadcast is usually at least 31.92% faster than the best available point-to-point broadcast (*LAM/MPI* $4\ KiB$ needs 0.002046 seconds and *ipmc* $4\ KiB$ needs 0.001393 seconds). These results also strengthen the usage of two threshold values: One threshold value to switch to one of the simple broadcast implementations when the communicator size is very small, and another threshold value to switch to the *fragmented chain* algorithm once the message size is large enough. Both threshold values mainly depend on the message size and the communicator size.

4.3 Application Results

Whereas the microbenchmark results have already attested the outstanding performance of the new implementation for a large range of scenarios, this section should analyse

[18] A simple *ping-pong* benchmark reveals that *Open MPI* is slower for small messages than *LAM/MPI*. This is another reason for the ”bad” performance of the *ipmc* component when compared with the *MPI_Bcast*() implementation of *LAM/MPI*.

how this improves the running time of the applications from chapter 2.

4.3.1 High-Performance Linpack Benchmark

Figures 21 and 22 show snapshots of the running *HPL benchmark*. Whereas the first run uses the original *MPI_Bcast*() implementation of *Open MPI*, the second run uses the new *ipmc* component with the multicast-based *MPI_Bcast*() implementation.

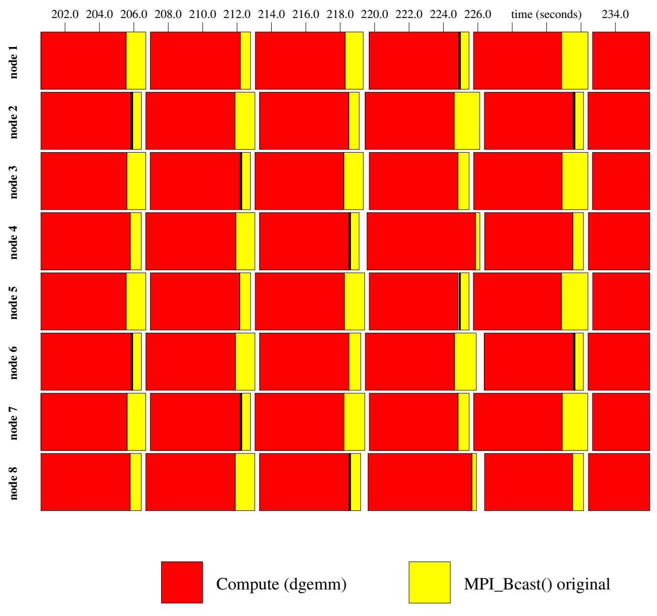


Figure 21: HPL with original MPI_Bcast()

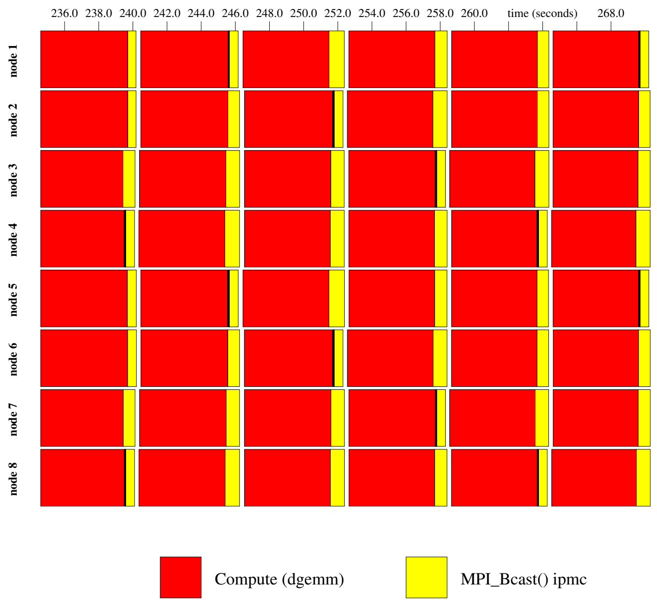


Figure 22: HPL with ipmc MPI_Bcast()

Although the benchmark is running only on 8 nodes [19], the first test run needs 1277 seconds (222.5 + 221.4 + 225.0 + 231.4 + 224.9 + 219.4 + 224.3 + 229.3 = 1798.2 seconds within the *MPI_Bcast*() call) whereas the second test run is 39 seconds faster ($187.4 + 187.0 + 187.1 + 194.4 + 188.3 + 187.3 + 192.0 + 185.0 = 1508.5$ seconds within the *MPI_Bcast*() call). This is a 16.11% *MPI_Bcast*() improvement, simply due to the positive impact of the much more balanced behaviour of the new broadcast operation. Therefore all nodes are calling and leaving this collective operation nearly simultaneously, which minimizes the gaps between consecutive computation blocks. When the number of nodes increases, the performance improvement will be much higher.

Table 2 shows the *HPL* benchmark results for different broadcast algorithms. It has been measured on 64 CLiC nodes with *Open MPI*, a problem size $N = 56320$, a blocking factor $NB = 40$ and a grid $P \star Q = 8 \star 8$.

broadcast algorithm	total duration	achieved performance
(0) 1 ring	4137.43 seconds	28.79 GFLOPS
(1) 1 ringM	4150.74 seconds	28.69 GFLOPS
(2) 2 ring	4188.44 seconds	28.44 GFLOPS
(3) 2 ringM	4098.30 seconds	29.06 GFLOPS
(4) Blong	4092.20 seconds	29.10 GFLOPS
(5) BlongM	4130.56 seconds	28.83 GFLOPS
(6a) original	4197.13 seconds	28.38 GFLOPS
(6b) ipmc	4057.23 seconds	29.36 GFLOPS

Table 2: HPL benchmark results using different broadcasts

Broadcast algorithms (0) to (5) are the special implementations which are developed for and shipped with the *HPL* benchmark. I have written a patch that adds a sixth algorithm to this list, which simply calls *MPI_Bcast*(). Test run $(6b)$ with the original broadcast implementation of *Open MPI* achieved the following results for the consumed time within the *MPI_Bcast*() operation per node:

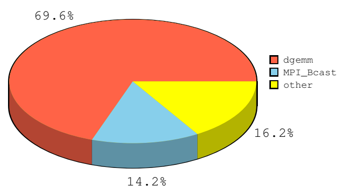
- minimum = 569.22 seconds (let this be 100%)
- maximum = 636.60 seconds (111.84%)
- average = 593.37 seconds (104.24%)

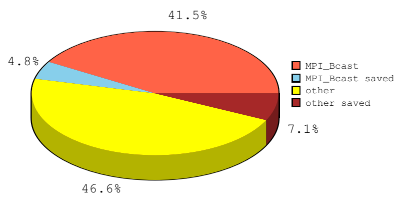
The new *ipmc* broadcast implementation achieved the following results for the consumed time within the *MPI_Bcast*() operation per node:

- minimum = 501.51 seconds (88.10%)
- maximum = 573.30 seconds (100.72%)
- average = 532.89 seconds (93.61%)

[19] 8 nodes is, according to the microbenchmark results, not enough to give any real benefits.

Each node called around 8090 times the *dgemm*() function, which consumed a total time of roughly 2925 seconds. In addition, each node called 1408 times the *MPI_Bcast*() function to transfer a total amount of approximately $1530\ MiB$ of data ($\Rightarrow\approx 1.09\ MiB$ per operation). This is identical for test runs $(6a)$ and $(6b)$ - what differs is the time which is spent within the broadcast operation. All (minimum, average and maximum) broadcast times could be improved significantly.





On average, the broadcast time of the original algorithm is 11.35% slower than the broadcast time of the new *ipmc* implementation. Even the minimum value of the original implementation is larger than the new average value. The residual time without the *dgemm*() and *MPI_Bcast*() phases is called ”others” and needs 15.25% more time with the original broadcast compared with the new implementation. Altogether, when we ignore the equal time for the *dgemm*() operation, the remaining parts are 12.5% slower when the *ipmc* broadcast is not used.

The total amount of sent multicast datagrams was $3,136,948$, the number of received datagrams on all nodes was $18,515,378$, and $4,299$ of them need to be rejected.

Although the first five broadcast algorithms are purpose-built for the *HPL* benchmark, they are all outperformed by the new implementation. Therefore, the *GFLOPS* value can be increased by 0.9% compared with the best available *HPL* broadcast algorithm.

4.3.2 Abinit

The heavy broadcast usage of *Abinit* (at least in the already introduced scenario) has a very large influence of the total running time of this application. Again, 8 MPI nodes are not (according to the microbenchmark results) enough to expect any improvements. Whereas the *process skew* within the running *HPL benchmark* allowed our new multicast implementation to make use of the intermediate phase of the broadcast, this is regrettably not the case for *Abinit* (see chapter 2.2, especially the reason why the root node is always the last MPI process calling *MPI_Bcast*).

broadcast algorithm	total duration	percentage
binomial tree	278.8 seconds	165.26%
ipmc (with MC)	197.2 seconds	116.89%
fragmented chain	168.7 seconds	100.00%

Table 3: Abinit results using different broadcasts

Table 3 shows the total running time of *Abinit* on 8 nodes, using different broadcast algorithms. All runs produced identical results, but the time needed to accomplish this varies heavily. With the *binomial tree* broadcast (which is the default for *LAM/MPI*) the application needs 41.38% longer than with the new *ipmc* broadcast. But the winner of all broadcast algorithms for this application example is the *fragmented chain* algorithm. The large message size ($\approx 3\ MiB$), the small communicator size (8 nodes) and the non-existing intermediate phase of the broadcast operation makes is the best choice here.

4.3.3 Conclusion

Although both applications (the *HPL* benchmark as well as *Abinit* make heavy use of *MPI_Bcast*(), they are broadcasting relative large messages. For such large message (especially in the *Abinit* case), the *fragmented chain* algorithm is usually the better choice. Nevertheless, we have seen (in case of the *HPL*) that point-to-point broadcasts which were faster in the microbenchmarks, have been outperformed by the new broadcast implementation. The already mentioned long-term statistics of the *HLRS* show that on average each broadcast operations transfers a message size of roughly $17\ KiB$ and a parallel job contains about 32.4 *MPI* processes. These parameters are perfectly suited for the new *ipmc* broadcast algorithm.

5 Conclusion and Future Work

This diploma thesis has analysed the network feature *multicast*, which is supported by several network technologies. It tried to evaluate different solutions for the various problems that appear when *multicast* is used to implement the *MPI_Bcast*() operation. A preceding analysis of existing applications had a big influence on the decisions. The resulting broadcast algorithm does not only scale perfectly with large communicators - it takes usually the same amount of time, whether it is used to broadcast a message to just ten nodes or to some hundred or thousand nodes - it also uses the intermediate phase of the broadcast very efficiently, making it even perform better for real-world applications than for synthetical benchmarks.

The final *ipmc* implementation for *Open MPI* can be easily installed (even afterwards to an existing installation as a binary object) and used by anyone. Users do not need to know anything about the broadcast behaviour of their application: they can simply check out this implementation and measure the direct change of their application performance. After this single test, they can immediately decide if it is useful to them. More interested users should read the paragraph about the adjustable parameters in section 3. The default settings can be changed easily with the help of command line parameters, which permit almost full control of the broadcast behaviour.

Developers should find it relatively easy to understand the well documented algorithm as well as the elaborately commented and legibly written source code which should comply to the *Open MPI* coding standards. This should facilitate quick modifications, or even ports of this implementation to different (i.e. non-IP-based) platforms.

Surely, this implementation is not yet fully optimized, leaving room for further improvements.

Open things for future and related work includes:

1. implementation and usage of a *MADCAP* server
2. *IPv6* support
3. a way to reduce copy overhead (”*zero copy*”?)
4. maybe a self-adapting decision function (at request of the user)
5. possibly support for *InfiniBand* or other network technologies
6. further analysis of applications to find ways to measure and parameterize *process skew*
7. utilization of *multicast* for other collective operations as well

A Appendix

A.1 A guide for Open MPI with the IPMC component

This is just an example how one can build, install and use a recent version of *Open MPI* together with the new *ipmc* component.

A.1.1 Installation of a single Open MPI instance

At the beginning we will start with the following steps:

1. get a recent (or special) subversion checkout of *Open MPI*
2. add the sources of the new *ipmc* component
3. build the *Open MPI* binaries (including *ipmc*)
4. install the results in a temporary directory
5. check if the installation is working correctly

The following script assumes a *Bash*-like shell, and has been tested with a recent version of *subversion* (v1.3.2) , *autoconf* (v2.59), *automake* (v1.9.6), *libtool* (v1.5.20) and *flex* (v2.5.33).

```
$ mkdir /tmp/openmpi
$ cd /tmp/openmpi
$ # get a recent version of Open MPI
$ # (add "-r 11682" after "co" to get revision 11682)
$ svn co http://svn.open-mpi.org/svn/ompi/trunk ompi-trunk
$ # add the new "ipmc" component
$ cd ompi-trunk/ompi/mca/coll
$ tar xzf $DOWNLOADS/ipmc_component.tar.gz
$ cd ../../..
$ # prepare for building
$ ./autogen.sh
$ mkdir build
$ cd build
$ # configure Open MPI
$ ../configure --prefix=/tmp/openmpi
$ # build Open MPI with "ipmc"
$ make all 2>&1 | tee make_all_with_ipmc_log.txt
$ # install the binaries
$ make install
$ # activate the binaries
$ export LD_LIBRARY_PATH=/tmp/openmpi/lib
$ export PATH=/tmp/openmpi/bin:$PATH
$ # check if Open MPI is installed correctly
$ ompi_info
$ # list all parameters of the "ipmc" component
$ ompi_info --param coll ipmc
```

A.1.2 Make Open MPI available and build an application

Since we have installed *Open MPI* in a local temporary directory, we need to make it explicitly available to all other cluster nodes. An alternative would be the use of a distributed or parallel file system, but our large test system *CLiC* had sometimes problems with its *AFS* file system. Therefore we will continue with the following steps:

1. build a binary package of *Open MPI*
2. install those binaries on all cluster nodes
3. build and install a test application
4. run the test application

We assume that $NODEFILE is a variable with a name of a file containing a list of all cluster nodes, and $NUMNODES is a variable holding the number of cluster nodes.

```
$ # build a binary package
$ rm -rf ompi-trunk
$ tar -cjf /tmp/openmpi_r11682_with_ipmc.tar.bz2 *
$ # install those binaries on all nodes
$ for node in ‘cat $NODEFILE‘; do
$  echo "installing Open MPI on node $node ...";
$  ssh $node rm -rf /tmp/openmpi;
$  ssh $node mkdir /tmp/openmpi;
$  scp -q /tmp/openmpi_r11682_with_ipmc.tar.bz2 \
$     $node:/tmp/openmpi/;
$  ssh $node "cd /tmp/openmpi ; \
$     tar xjf openmpi_r11682_with_ipmc.tar.bz2";
$ done
$ # get and build a test application
$ wget www.tu-chemnitz.de/~chsi/bcast_bench.tar.gz
$ tar xzvf bcast_bench.tar.gz
$ cd bcast_bench
$ make
$ # install this test application
$ for node in ‘cat $NODEFILE‘; do
$  echo "copying test application to node $node ...";
$  scp -q bcast_bench $node:/tmp/openmpi/test_app;
$ done
$ # run this test application
$ mpiexec -np $NUMNODES --hostfile $NODEFILE \
$    --prefix /tmp/openmpi /tmp/chsi-tmp/test_app \
$    2>&1 | tee results_bcast_bench_1st.txt
```

A.1.3 Playing around with the IPMC parameters

If all steps until here observed no problems, then the installation seems to be working correctly. Now we can start to play around with some parameters of the *ipmc* component to influence its behaviour and performance.

First, you might try to disable this new component to know how the performance of your application changes when it is used with the original *Open MPI* components. Disabling can be achieved by lowering the priority of the *ipmc* component. This can be done by adding the

```
--mca coll_ipmc_priority 0
```

parameter to the *mpiexec* call.

Second, you might try to adjust the decision boundaries for the alternative broadcast algorithms. Large message are usually broadcasted using the *fragmented chain* algorithm, and on small communicators it falls back to the *linear* broadcast algorithm. You can adjust these boundaries by modifying the following two parameters:

```
--mca coll_ipmc_crossover_size 2097152
--mca coll_ipmc_crossover_nodes 8
```

The first example sets the maximum message size (for the multicast-based broadcast algorithm) to $2\ MiB$, and the second example forbids the usage of the multicast-based broadcast when the communicator contains less than 8 *MPI* nodes.

Third, you might try to optimize some parameters to further influence the performance of your application.

```
--mca coll_ipmc_fragment_size 8192
--mca coll_ipmc_root_wait_time 10
--mca coll_ipmc_use_crc_checking 0
```

The first example increases the payload size of the *IP multicast* datagrams to $8\ KiB$, the second line causes the root node to wait $10\ \mu s$ before issuing the multicast. The last example disables the additional CRC checking of all datagram packets. You should only do this when you are sure that no corrupt datagrams are possible.

Finally, you can also turn on some useful statistical output, which can help you to get some more details:

```
    --mca coll_ipmc_print_statistics 1
```

This will print some counters for every node, when a communicator is finally destroyed. Here you can see how many broadcast operations were called by your application and how many multicast datagrams were sent, or useful at the receivers side.

A.2 Fragmented Tree vs. Fragmented Chain

Theorem: For any fixed communicator size, there exist a message size with a corresponding fragment size, so that for all larger message sizes the *fragmented chain* broadcast is always faster than any tree-based broadcast algorithm implemented on top of point-to-point communication.

The broadcast operation involves all *MPI* processes in the specified communicator. ”Work-optimal” would mean that all those processes are communicating all the time until the operation has finally finished, while there are no duplicate or senseless message transfers. Unfortunately, since only the root node owns the data at the beginning, an additional startup- and/or ending-step is definitely necessary. In the case of fragmented broadcast algorithms, this can be seen as ”filling” the pipeline and/or ”emptying” the pipeline. Pipelining uses the fact that a given large message can be split into several smaller fragments.

Now the easy-to-understand reasoning why a tree structure cannot be better (in regards to ”work-optimal”) than the corresponding chain variant: Both variants have a single node that does only sending: the root node. But contrary to the tree variant where there are around $\sharp nodes/2$ leave nodes, the chain variant has only a single node that does only receiving. Therefore for a large enough communicator and a large enough message (crossover for pipeline), the chain variant will get up to twice as fast as the tree variant. The binomial tree version is even worse because the root node sends to more than two children when the size of the communicator increases above 4. Therefore the resulting bandwidth will be divided by the fan-out of the root node.

Another explanation is based on the fact that we originally assumed that only a single message can be injected into the network. If a node is serving several children instead of only one, then it can only issue the fragments in an interleaved fashion. A single child node will always only receive data halve of the time, effectively halving the available bandwidth.

A.2.1 Example

A larger message, say $1\ MiB$ $(= 1024 * 1024\ byte)$, which can be split into 1024 fragments, should be broadcasted to 8 nodes. We will just count the number of fragment-transfer rounds for simplicity. The binomial tree structure (which is BTW optimal for power of two node numbers) needs exactly 3 rounds to fill the pipeline and 3 additional rounds to empty the pipeline which is quite good. Unfortunately, the root node needs to supply 3 children with the data. Therefore the overall bandwidth within the pipeline stage gets reduced by a factor of 3! The chain version needs 7 rounds to fill the pipeline and 7 rounds to empty it again. But since the root node only supplies a single child node, the achieved bandwidth within the pipeline does not reduce. The number of rounds with a full pipeline is much higher in this example than this negligible startup overhead. Here are the measurements (medians with a minimal deviation) for the different implementations (on CLiC with LAM/MPI):

original *LAM/MPI* $6.5.6$ implementation (binary tree version):

- node 1 receives the broadcast message after $464877\ \mu s$
- node 2 receives the broadcast message after $377973\ \mu s$
- node 3 receives the broadcast message after $565710\ \mu s$
- node 4 receives the broadcast message after $293223\ \mu s$
- node 5 receives the broadcast message after $486061\ \mu s$
- node 6 receives the broadcast message after $388784\ \mu s$
- node 7 receives the broadcast message after $577007\ \mu s$

The average broadcasting time over all 7 receivers using the *binary tree* broadcast is $450519\ \mu s$.

binomial tree implementation (without fragmentation):

- node 1 receives the broadcast message after $283172\ \mu s$
- node 2 receives the broadcast message after $283697\ \mu s$
- node 3 receives the broadcast message after $284005\ \mu s$
- node 4 receives the broadcast message after $294555\ \mu s$
- node 5 receives the broadcast message after $294327\ \mu s$
- node 6 receives the broadcast message after $295074\ \mu s$
- node 7 receives the broadcast message after $305870\ \mu s$

Fortunately, new version of *LAM/MPI* use this *binomial tree* broadcast too. The average broadcasting time over all 7 receivers is here $291529\ \mu s$.

binomial tree implementation (with 1024 fragments):

- node 1 receives the broadcast message after $273319\ \mu s$
- node 2 receives the broadcast message after $273678\ \mu s$
- node 3 receives the broadcast message after $274590\ \mu s$
- node 4 receives the broadcast message after $274324\ \mu s$
- node 5 receives the broadcast message after $274580\ \mu s$
- node 6 receives the broadcast message after $273669\ \mu s$
- node 7 receives the broadcast message after $274560\ \mu s$

Although we are using fragmentation now, the average broadcast duration over all 7 receivers only slightly decreases to $274103\ \mu s$, saving only 5.98%.

simple chain implementation (without fragmentation):

- node 1 receives the broadcast message after $192910\ \mu s$
- node 2 receives the broadcast message after $294266\ \mu s$
- node 3 receives the broadcast message after $396343\ \mu s$
- node 4 receives the broadcast message after $499811\ \mu s$
- node 5 receives the broadcast message after $602176\ \mu s$
- node 6 receives the broadcast message after $706036\ \mu s$
- node 7 receives the broadcast message after $717621\ \mu s$

The average broadcasting time over all 7 receivers is $487023\ \mu s$, which is even worse than the binary tree implementation! So avoid the chain version if you cannot use fragmentation.

simple chain implementation (with 1024 fragments):

- node 1 receives the broadcast message after $95092\ \mu s$
- node 2 receives the broadcast message after $97027\ \mu s$
- node 3 receives the broadcast message after $97471\ \mu s$
- node 4 receives the broadcast message after $97894\ \mu s$
- node 5 receives the broadcast message after $98190\ \mu s$
- node 6 receives the broadcast message after $98615\ \mu s$

- node 7 receives the broadcast message after $98944\ \mu s$

The simple (and usually bad performing) *chain* broadcast becomes a very fast broadcast algorithm when it is used with fragmentation. The average broadcasting time over the 7 receivers is $97605\ \mu s$, making this broadcast algorithm around 64.4% faster than the fragmented *binomial tree* implementation.

A.3 IP over InfiniBand

Our new big cluster in Chemnitz, called *CHIC*, will be equipped with an *InfiniBand* interconnection network. Therefore my *MPI_Bcast()* implementation, which is based on *IP multicast*, is only of limited use for this cluster. InfiniBand itself can support native multicast too, so a further work could adapt the *ipmc* implementation and get it running natively with InfiniBand. Up to then it might be an option to use *IP over InfiniBand* (*IPoIB*, see [(IB06]), an encapsulation of IP packets in native InfiniBand. Since we are planning to establish an InfiniBand-only cluster, *IPoIB* is required in any case (e.g. for the management). Oded Bergman (Project Manager at Voltaire) assured me that the IP multicast will be mapped to InfiniBand multicast and is therefore working as expected. He kindly sent me the following performance numbers:

- Native IB latency - $1.2\ \mu s$ up to $4\ \mu s$ (and more on old server platforms)
- IPoIB latency - $6\ \mu s$ for ping RC
- IPoIB latency - $20 - 30\ \mu s$ for TCP
- IPoIB MCE latency - $9\ \mu s$ using UDP multicast sockets APIs
- IPoIB bandwidth - 1.5 to $2\ Gbps$
- Native IB bandwidth - over $7\ Gbps$

As you can see, the IPoIB penalty both latency and bandwidth is quite huge (a factor of 4 to 5), making this idea (using the *ipmc* component with IPv4 on InfiniBand) obsolete. Although the new *ipmc* broadcast should work correctly using IPoIB, the expected performance gain up to some hundred of nodes would be eliminated by the performance penalty of the IPoIB encapsulation.

Abbreviations and Acronyms

ACK	Acknowledgment *– page 30*
ATLAS	Automatically Tuned Linear Algebra Software *– page 23*
BID	Broadcast Identifier *– page 29*
BLAS	Basic Linear Algebra Subprograms *– page 23*
BSD	Berkeley Software Distribution *– page 16*
BSP	Bulk Synchronous Parallel (model) *– page 4*
CPU	Central Processing Unit *– page 23*
CRC	Cyclic Redundancy Check *– page 29*
FAQ	Frequently Asked Questions *– page 1*
FLOPS	Floating Point Operations Per Second
GFLOPS	gigaFLOPS (10^9 FLOPS) *– page 4*
GID	Global Identifier *– page 16*
HPC	High Performance Computing *– page 2*
HPL	High-Performance Linpack (Benchmark) *– page 4*
IEEE	Institute of Electrical and Electronics Engineers *– page 19*
IETF	Internet Engineering Task Force *– page 30*
IP	Internet Protocol *– page 2*
IPv4	Internet Protocol Version 4 *– page 18*
IPv6	Internet Protocol Version 6 *– page 33*
IRTF	Internet Research Task Force *– page 30*
KiB	kibibyte ($1\ KiB = 2^{10}\ byte$) *– page 4*
MADCAP	Multicast Address Dynamic Client Allocation Protocol *– page 19*
MCA	Modular Component Architecture *– page 20*
MD5	Message-Digest algorithm 5 *– page 33*
MiB	mebibyte ($1\ MiB = 2^{20}\ byte$) *– page 36*

MPI	Message Passing Interface – *page 2*
MTU	Maximum Transmission Unit – *page 36*
NACK	Negative Acknowledgment – *page 30*
PDF	Portable Document Format – *page 5*
PRAM	Parallel Random Access Machine – *page 4*
RFC	Request for Comments – *page 19*
SHA-1	Secure Hash Algorithm - 1 – *page 33*
TCP	Transmission Control Protocol – *page 16*
UDP	User Datagram Protocol – *page 16*

References

[AISS97] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser and Chris Scheiman: *LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation*. *Journal of Parallel and Distributed Computing*, 44(1), 1997:pp. 71–79. [p. 4]

[AMP04] J. Liu A. Mamidala and D. K. Panda: *Efficient Barrier and Allreduce on IBA clusters using hardware multicast and adaptive algorithms*, September 2004. IEEE Cluster Computing 2004. [p. 23]

[BWRS03] Andrew M. Rudoff By W. Richard Stevens, Bill Fenner: *UNIX Network Programming Volume 1, Third Edition: The Sockets Networking API*. Addison Wesley, 2003. ISBN 0-13-141155-1. [p. 17]

[Deu96] P. Deutsch: *GZIP file format specification version 4.3*. Request for Comments: 1952, May 1996. Contains a sample code for the CRC-32 calculation. [p. 37]

[FK99] Hans-Hermann Frese and Harald Knipp: *Performance Evaluation of MPI and MPICH on the Cray T3E*, 1999. [p. 41]

[For95] Message Passing Interface Forum: *MPI: A Message-Passing Interface Standard (version 1.1)*, 1995. Technical report.
URL `http://www.mpi-forum.org` [p. 2]

[For97] Message Passing Interface Forum: *MPI-2: Extensions to the Message-Passing Interface*, 1997. Technical report.
URL `http://www.mpi-forum.org` [p. 2]

[GDBC03] Qing Huang Gregory D. Benson, Cho-Wai Chu and Sadik G. Caglar: *A Comparison of MPICH Allgather Algorithms on Switched Networks.*. In *Proceedings, Recent Advances in Parallel Virtual Machine and Message Passing Interface, 10th European PVM/MPI Users' Group Meeting*. Springer, Venice, Italy, September 2003, Lecture Notes in Computer Science, pp. 335–343, pp. 335–343. [p. 14]

[GWS05] Richard L. Graham, Timothy S. Woodall and Jeffrey M. Squyres: *Open MPI: A Flexible High Performance MPI*, September 2005. [p. 20]

[HACA00] Yvette O. Carrasco Hsiang Ann Chen and Amy W. Apon: *MPI Collective Operations over IP Multicast*, 2000. IPDPS Workshop. [p. 28]

[HR05] T. Hoefler and W. Rehm: *A short Performance Analysis of Abinit on a Cluster System*, July 2005. [p. 25]

[HSB$^+$06] T. Hoefler, J. Squyres, G. Bosilca, G. Fagg, A. Lumsdaine and W. Rehm: *Non-Blocking Collective Operations for MPI-2*, August 2006. [p. 27]

[HTM05] Torsten Hoefler and W. Rehm T. Mehlan, F. Mietke: *Evaluation of publicly available Barrier-Algorithms and Improvement of the Barrier-Operation for large-scale Cluster-Systems with special Attention on InfiniBand Networks*, March 2005. Diploma Thesis. [p. 28]

[IAN06] IANA: *Internet Multicast Addresses*, 1988-2006. A comprehensive and up-to-date list of reserved multicast addresses.
URL `http://www.iana.org/assignments/multicast-addresses` [p. 19]

[(IB06] V. Kashyap (IBM): *IP over InfiniBand (IPoIB) Architecture*. Request for Comments: 4392, April 2006. [p. 60]

[JLP04] A. Mamidala J. Liu and D. K. Panda: *Fast and Scalable MPI-Level Broadcast using InfiniBand's Hardware Multicast Support.*, April 2004. Int'l Parallel and Distributed Processing Symposium (IPDPS 04). [p. 31]

[LBS86] M. Blum L. Blum and M. Shub: *A Simple Unpredictable Pseudo-Random Number Generator*. *SIAM Journal on Computing*, 15(2), May 1986:pp. 364–383. [p. 33]

[MSD98] Steven Huss-Lederman Marc Snir, Steve Otto and Jack Dongarra: *MPI - The Complete Reference, 2nd edition*. The MIT Press, 1998. ISBN 0-262-69215-5. Volume 1, The MPI Core; second edition. [p. 1]

[PGAB$^+$05] J Pjesivac-Grbovic, T Angskun, G Bosilca, G E Fagg, E Gabriel and J J Dongarra: *Performance Analysis of MPI Collective Operations*, 2005. [p. 14]

[PMG95] Lance Shuler Prasenjit Mitra, David Payne and Robert van de Geijn: *Fast Collective Communication Libraries, Please*. In *Proceedings of the Intel Supercomputing Users' Group Meeting*. January 1995. [p. 14]

[PPY06] A. Faraj P. Patarasu and X. Yuan: *Pipelined Broadcast on Ethernet Switched Clusters*, April 2006. [p. 42]

[Rab99] Rolf Rabenseifner: *Automatic Profiling of MPI Applications with Hardware Performance Counters*. In *Proceedings, Message Passing Interface and High-Performance Cluster Developer's and User's Conference*. Atlanta, USA, March 1999, pp. 35–42, pp. 35–42. [p. 4]

[RTG05] Rolf Rabenseifner Rajeev Thakur and William Gropp: *Optimization of Collective Communication Operations in MPICH*. *International Journal of High Performance Computing Applications*, 19(1), 2005:pp. 49–66. [p. 14]

[SHS99] B. Patel S. Hanna and M. Shah: *Multicast Address Dynamic Client Allocation Protocol (MADCAP)*. Request for Comments: 2730, December 1999. [p. 19]

[SL04] Jeffrey M. Squyres and Andrew Lumsdaine: *The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms*, July 2004. [p. 21]

[SR06] Christian Siebert and Wolfgang Rehm: *'Memory-Mapped Messages' - eine implizite Technik zur Überlappung von Kommunikation und Berechnung*, January 2006. KiCC Workshop 2005. [p. 27]

[Tan03] Andrew S. Tanenbaum: *Computer Networks*. Pearson, 2003. ISBN 3-8273-7046-9. [p. -]

[TKV00] Henri E. Bal Thilo Kielmann and Kees Verstoep: *Fast Measurement of LogP Parameters for Message Passing Platforms*. *Lecture Notes in Computer Science*, 1800, May 2000:p. 1176ff. [p. 5]

[Trä04] Jesper-Larsson Träff: *A simple Work-optimal Broadcast Algorithm for Message-Passing Parallel Systems*. In *EuroPVM/MPI 2004*. Springer, 2004, volume 3241 of *Lecture Notes in Computer Science*, pp. 173–180, pp. 173–180. [p. 47]

[WG95] Jerrell Watts and Robert Van De Geijn: *A pipelined broadcast for multidimensional meshes*. *Parallel Processing Letters*, 5(2), 1995:pp. 281–292. [p. 47]

[WYG05] Dhabaleswar K. Panda Weikuan Yu, Sayantan Sur and Rich L. Graham: *High Performance Broadcast Support in La-Mpi Over Quadrics*. *International Journal of High Performance Computing Applications*, 19(4), 2005:pp. 453–463. [p. 16]

[ZAS01] D. Meyer Z. Albanna, K. Almeroth and M. Schipper: *IANA Guidelines for IPv4 Multicast Address Assignments*. Request for Comments: 3171, August 2001. [p. 19]

[ZLGS99] Omer Zaki, Ewing Lusk, William Gropp and Deborah Swider: *Toward Scalable Performance Visualization with Jumpshot*. *International Journal of High Performance Computing Applications*, 13(2), 1999:pp. 277–288. [p. 23]