An Optimal Broadcast Algorithm Adapted to SMP Clusters

Jesper Larsson Träff and Andreas Ripke

C&C Research Laboratories, NEC Europe Ltd.,
Rathausallee 10, D-53757 Sankt Augustin, Germany
{traff, ripke}@ccrl-nece.de

Abstract. We describe and and evaluate the adaption of a new, optimal broadcast algorithm for "flat", fully connected networks to clusters of SMP nodes. The optimal broadcast algorithm improves over other commonly used broadcast algorithms (pipelined binary trees, recursive halving) by up to a factor of two for the non-hierarchical (non-SMP) case. The algorithm is well suited for clusters of SMP nodes, since intra-node broadcast of relatively small blocks can take place concurrently with inter-node communication over the network. This new algorithm has been incorporated into a state-of-the art MPI library. On a 32-node dual-processor AMD cluster with Myrinet interconnect, improvements of a factor of 1.5 over for instance a pipelined binary tree algorithm has been achieved, both for the case with one and with two MPI processes per node.

1 Introduction

Broadcast is a frequently used collective operation of MPI, the *Message Passing Interface* [8], and there is therefore good reason to pursue the most efficient algorithms and best possible implementations. Recently, there has been much interest in broadcast algorithms and implementations for different systems and MPI libraries [2,5,6,9,10], but none of these achieve the theoretical lower bound for their respective models. An exception is the *LogP* algorithm in [7], but no implementation results were given. A quite different, theoretically optimal algorithm for single-ported, fully connected networks was developed by the authors in [11]. This algorithm has the potential of being up to a factor two faster than the best currently implemented broadcast algorithms based on pipelined binary trees, or on recursive halving as recently implemented in `mpich2` [9] and elsewhere [5,10]. These algorithms were developed on the assumption of a "flat", homogeneous, fully connected communication system, and will not perform optimally on systems with a hierarchical communication system like clusters of SMP nodes. Pipelined binary tree algorithms can naturally be adapted to the SMP case [3], whereas the many of the other algorithms will entail a significant overhead.

In this paper we present the ideas behind the new, optimal broadcast algorithm; technical details, however, must be found in [11]. We describe how the

B. Di Martino et al. (Eds.): EuroPVM/MPI 2005, LNCS 3666, pp. 48–56, 2005.

implementation has been extended to clusters of SMP nodes, and compare the performance of the algorithm to a pipelined binary tree algorithm [3] on a 32-node dual-processor AMD based SMP cluster with Myrinet interconnect. Very worthwhile performance improvements of more than a factor of 1.5 are achieved.

2 The Broadcast Algorithm

We first give a high-level description of the optimal broadcast algorithm for "flat", homogeneous, fully connected systems. We assume a linear communication cost model, in which sending m units of data takes time $\alpha + \beta m$, and each processor can both send and receive a message at the same time, possibly from different processors. The p processors are numbered from 0 to $p-1$ as in MPI, and we let $n = \lceil \log p \rceil$. Without loss of generality we assume that broadcast is from *root* processor 0. Assuming further that the m data is sent as N blocks of m/N units, the number of communication rounds required (for any algorithm) to broadcast the N blocks is $n-1+N$, for a time of $(n-1+N)(\alpha+\beta m/N) = (n-1)\alpha + (n-1)\beta m/N + N\alpha + \beta m$. By balancing the terms $(n-1)\beta m/N$ and αN, the optimal running time can be found as

$$T_{\text{opt}}(m) = (n-1)\alpha + 2\sqrt{(n-1)\alpha}\sqrt{\beta m} + \beta m = (\sqrt{(n-1)\alpha} + \sqrt{\beta m})^2 \quad (1)$$

Proofs of this lower bound can be found in e.g. [4,6].

The optimal broadcast algorithm is pipelined in the sense that all processors are (after an initial fill phase of n rounds) both sending and receiving blocks at the same time. For sending data each processor acts as if it is a root of a(n incomplete, when p is not a power of 2) binomial tree. Each non-root processor has n different parents from which it receives blocks. To initiate the broadcast, the root first sends n successive blocks to its children. The root continues in this way sending blocks successively to its children in a round-robin fashion. The non-root processors receive their first block from their parent in the binomial tree rooted at processor 0. The non-roots pass this block on to their children in this tree. After this initial *fill phase*, each processor now has a block, and the broadcast enters a *steady state*, in which in each round each processor (except the root) receives a new block from a parent, and sends a previously received block to a child.

A more formal description of the algorithm is given in Figure 1. The buffer containing the data being broadcast is divided into N blocks of roughly m/N units, and the ith block is denoted $\texttt{buffer}[i]$ for $0 \leq i < N$.

As can be seen each processor receives N blocks of data. That indeed N different blocks are received and sent is determined by the $\mathsf{recvblock}(i,r)$ and $\mathsf{sendblock}(i,r)$ functions which specify the block to be received and sent in round i for processor r (see Section 2.2). The functions next and prev determine the *communication pattern* (see Section 2.1). In each phase the same pattern is used, and the n parent and child processors of processor r are $\mathsf{next}(j,r)$ and $\mathsf{prev}(j,r)$ for $j = 0, \ldots n-1$. The parent of processor r for the fill phase is $\mathsf{first}(r)$, and the first round for processor r is likewise $\mathsf{first}(r)$. With these provisions we have:

Root processor 0:

```
/* fill */
for i ← 0, n − 1 do
    send(buffer[sendblock(i, 0)], next(i, 0))
/* steady state */
for i ← 1, N do
    j ← (i − 1) mod n
    send(buffer[sendblock(n − 1 + i, 0)], next(j, 0))
```

Non-root processor r:

```
/* fill */
i ← first(r)
recv(buffer[recvblock(i, r)], prev(i, r))
for i ← first(r) + 1, n − 1
    send(buffer[sendblock(i, r)], next(i, r))
/* first block received, steady state */
for i ← 1, N
    j ← (i − 1) mod n
    if next(j, r) ≠ 0 then /* no sending to root */
        send(buffer[sendblock(n − 1 + i, r)], next(j, r))
    || /* send and receive simultaneously */
    recv(buffer[recvblock(n − 1 + i, r)], prev(j, r))
```

Fig. 1. The optimal broadcast algorithm

Theorem 1. *In the fully-connected, one-ported, bidirectional, linear cost communication model, N blocks of data can be broadcast in $n-1+N$ rounds reaching the optimal running time (1).*

The algorithm is further simplified by the following observations. First, the block to send in round i is obviously

$$\mathsf{sendblock}(i,r) = \mathsf{recvblock}(i, \mathsf{next}(i,r))$$

so it will suffice to determine a suitable recvblock function. Actually, we can determine the recvblock function such that for any processor $r \neq 0$ it holds that

$$\{\mathsf{recvblock}(0,r), \mathsf{recvblock}(1,r), \ldots, \mathsf{recvblock}(n-1,r)\} = \{0,1,\ldots,n-1\}$$

that is the recvblock for a phase consisting of rounds $0,\ldots n-1$ is a permutation of $\{0,\ldots,n-1\}$. For such functions we can take for $i \geq n$

$$\mathsf{recvblock}(i,r) = \mathsf{recvblock}(i \bmod n, r) + n(\lfloor i/n \rfloor - 1 + \delta_{\mathsf{first}(r)}(i \bmod n))$$

where $\delta_j(i) = 1$ if $i = j$ and 0 otherwise. Thus in rounds $i+n, i+2n, i+3n, \ldots$ for $0 \leq i < n$, processor r receives blocks $\mathsf{recvblock}(i,r), \mathsf{recvblock}(i,r) + n, \mathsf{recvblock}(i,r)+2n, \ldots$ (plus n if $i = \mathsf{first}(r)$). We call such a recvblock function

a *full block schedule*, and to make the broadcast algorithm work we need to show that a full block schedule always exists, and how it can be computed. Existence is proved in [11], while the construction is outlined in the following sections.

When $N = 1$ the algorithm is just an ordinary binomial tree broadcast, which is optimal for small m. The number of blocks N can be chosen freely, e.g. to minimize the broadcast time under the linear cost model, or, which is relevant for some systems, to limit the amount of communication buffer space.

2.1 The Communication Pattern

When p is a power of two the communication pattern of the allgather algorithm in [1] can be used. In round j for $j = 0, \ldots, n-1$ processor r receives a block from processor $(r - 2^j) \bmod p$ and sends a block to processor $(r + 2^j) \bmod p$, so for that case we take

$$\mathsf{next}(j, r) = (r + 2^j) \bmod p$$
$$\mathsf{prev}(j, r) = (r - 2^j) \bmod p$$

With this pattern the root successively sends n blocks to processors $1, 2, 4, \ldots, 2^j$ for $j = 0, \ldots n-1$. The subtree of child processor $r = 2^j$ consists of the processors $(r+2^k) \bmod p, k = j+1, \ldots, n-1$. We say that processors $2^j, \ldots 2^{j+1}-1$ together form *group* j, since these processors will all receive their first block in round j. The *group start* of group j is 2^j, and the *group size* is likewise 2^j. Note that $\mathsf{first}(r) = j$ for a processor in group j.

For the general case where p is not a power of two, the processors are divided into groups of size approximately 2^j. To guarantee existence of the full block schedule, the communication pattern must satisfy that the total number of processors in groups $0, 1, \ldots, j-1$ plus the root processor must be at least the number of processors in group j, so that all processors in group j can receive their first block in round j. Likewise, the size of the last group $n-1$ must be at least the size of groups $0, 1, \ldots, n-2$ for the processors of the last group to deliver a block to all previous processors in round $n-1$. To achieve this we define for $0 \leq j < n$

$$\text{groupsize}(j, p) = \begin{cases} \text{groupsize}(j, \lceil p/2 \rceil) & \text{if } j < \lceil \log p \rceil - 1 \\ \lfloor p/2 \rfloor & \text{if } j = \lceil \log p \rceil - 1 \end{cases}$$

and

$$\text{groupstart}(j, p) = 1 + \sum_{i=0}^{j-1} \text{groupsize}(j, p)$$

When p is a power of two this definition coincides with the above definition, eg. $\text{groupsize}(j, p) = 2^j$.

We now define the next and prev functions analogously to the powers-of-two case:

$$\mathsf{next}(j, r) = (r + \text{groupstart}(j, p)) \bmod p$$
$$\mathsf{prev}(j, r) = (r - \text{groupstart}(j, p)) \bmod p$$

We note that this communication pattern leads to an exception for the fill phase of the algorithm in Figure 1, since it may happen that $\mathsf{next}(j,r) = \text{groupstart}(j+1,p) = r'$ and $\mathsf{prev}(\mathsf{first}(r'),r') = 0 \neq \mathsf{next}(j,r)$. Such a **send** has no corresponding **recv**, and shall not be performed.

2.2 Computing the Full Block Schedule

The key to the algorithm is the existence and construction of the *full block schedule* given the communication pattern described in the previous section. Existence and correctness of the construction is discussed in [11]. For the construction itself a greedy algorithm almost suffices. Based on what we call the *first block schedule* which determines the first block to be received by each processor r, the construction is as follows.

1. Construct the *first block schedule* `schedule`:
   set `schedule`[groupstart(j,p)] $= j$, and `schedule`[groupstart$(j,p)+i$] $=$ `schedule`$[i]$ for $i = 1,\ldots,\text{groupstart}(j,p)-1$.
2. Scan the first block schedule in descending order $i = r-1, r-2, \ldots 0$. Record in `block`$[j]$ the first block `schedule`$[i]$ different from `block`$[j-1]$, `block`$[j-2], \ldots$ `block`$[0]$, and in `found`$[j]$ the index i at which `block`$[j]$ was found.
3. If $\mathsf{prev}(j,r) <$ `found`$[j]$ either
   - if `block`$[j] >$ `block`$[j-1]$ then swap the two blocks,
   - else mark `block`$[j]$ as unseen,

   and continue scanning in Step 2.
4. Set `block`[$\mathsf{first}(r)$] $=$ `schedule`$[r]$
5. Find the remainder blocks by scanning the first block schedule in the order $i = p-1, p-2, \ldots r+1$, and swap as in Step 3.

For each r take

$$\mathsf{recvblock}(i,r) = \texttt{block}[i]$$

with `block` as computed above.

Space for the full block schedule is $O(p \log p)$, and as described above the construction takes $O(p^2)$ time. However, by a different formulation of the algorithm, the computation time can be reduced to $O(p \log p)$ steps [11]. The correctness proof can also be found in [11]; as anecdotal evidence we mention that we computed and verified all schedules up to 100 000 processors on a 2 GHz AMD Athlon PC. Construction time for the largest schedule was about 225 ms. This is of course prohibitive for on-line construction of the schedule at each `MPI_Bcast` operation. Instead, a corresponding schedule must be precomputed for each MPI communicator. In applications with many communicators the space consumption of $O(p \log p)$ can become a problem. Fortunately, it is possible to store the full block schedule in a distributed fashion with only $O(\log p)$ space for each processor: essentially each processor i needs only its own $\mathsf{recvblock}(i,j)$ and $\mathsf{sendblock}(i,j)$ functions, assuming that process 0 is the broadcast root (if this is not the case, the trick is that some other processor sends the needed schedule to processor 0, so each processor has to store schedules for two processors which is still only $O(\log p)$ space. The sending of the schedule to processor 0 can be hidden behind the already started broadcast operation and thus does not cost extra time).

2.3 Properties

We summarize the main properties of the broadcast algorithm as described above for flat systems.

1. The algorithm broadcasts N blocks in $n-1+N$ communication rounds, which is best possible.
2. The number of blocks can be chosen freely. In the linear cost communication model, the best block size is $\sqrt{(m\alpha)/((n-1)\beta)}$ resulting in optimal running time (1).
3. The number of rounds can also be chosen such that a given maximum block size, eg. internal communication buffer, is not exceeded.
4. Small messages should be broadcast in $N = 1$ rounds, in which case the algorithm is simply a binomial tree algorithm.
5. The required *full block schedule* can be computed in $O(p \log p)$ time.
6. Space for the full block schedule is likewise $O(p \log p)$ which can be stored in a distributed fashion with $O(\log p)$ space per processor.

2.4 Adaption to Clusters of SMP Nodes

The flat broadcast algorithm can be adapted to systems with a hierarchical communication structure like clusters of SMP nodes as follows. For each node a local root processor is chosen. The flat broadcast algorithm is run over the set of local root processors. In each communication round each root processor receives a new block which is broadcast locally over the node. For SMP clusters a simple shared memory algorithm can be used. Furthermore, the steady-state loop of the algorithm in Figure 1 can easily be rewritten such that in each round a) a new block is received, b) an already received block is sent, and c) the block received in the previous round is broadcast locally. This makes it possible – for communication networks supporting this – to hide the local, shared memory broadcast completely behind the sending/receiving of new blocks. Only the node local broadcast of the last block cannot be hidden in this fashion, which adds time proportional to the block size $\sqrt{(m\alpha)/((n-1)\beta)}$ to the total broadcast time. For broadcast algorithms based on recursive halving [2,9,10], the size of the "last block" received may be proportional to $m/2$ causing a much longer delay.

3 Performance

Both the flat and the SMP cluster broadcast algorithms have been implemented in a state-of-the art MPI implementation for PC clusters. We give results for a dual-processor AMD cluster with Myrinet interconnect.

Figure 2 compares the performance of various broadcast algorithms for the flat case with one MPI process per SMP node. The new, optimal algorithm is compared to an algorithm based on recursive halving [10], a pipelined binary tree algorithm (with a binomial tree for short messages), and a binomial tree algorithm. The theoretical bandwidth improvement over both the recursive halving

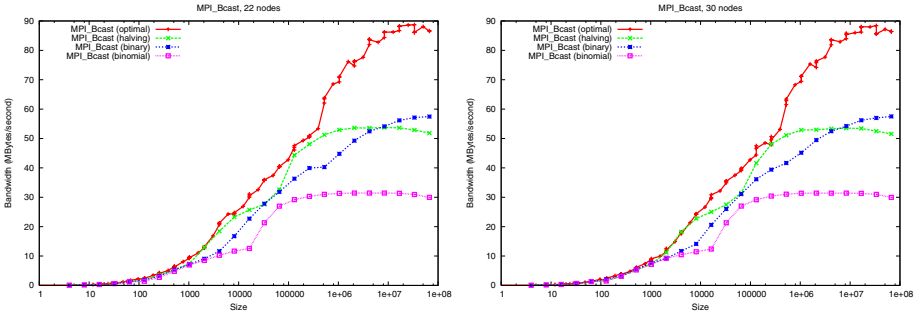

Fig. 2. Bandwidth of 4 different "flat" broadcast algorithms for fixed number of processors $p = 22$ (left) and $p = 30$ (right), one MPI process per node, and data size m up to 64MBytes

and the pipelined binary tree algorithm is a factor of 2, over the binomial tree algorithm a factor $\lceil \log_2 p \rceil$. The actual improvement is more than a factor 1.5 even for messages of only a few K bytes. It should be noted that to obtain the performance and relatively smooth bandwidth growth shown here, the simple, linear cost model is not sufficient. Instead a piecewise linear model with up to 4 different values of α and β is used for estimating the best block size for a given message size m.

Figure 3 compares the new SMP-adapted broadcast algorithm to a pipelined binary tree algorithm likewise adapted to SMP clusters with two MPI processes per node. For both algorithms, lower bandwidth than in the one process/node case is achieved, but also in the SMP case the new broadcast algorithm achieves about a factor 1.4 higher bandwidth than the pipelined binary tree.

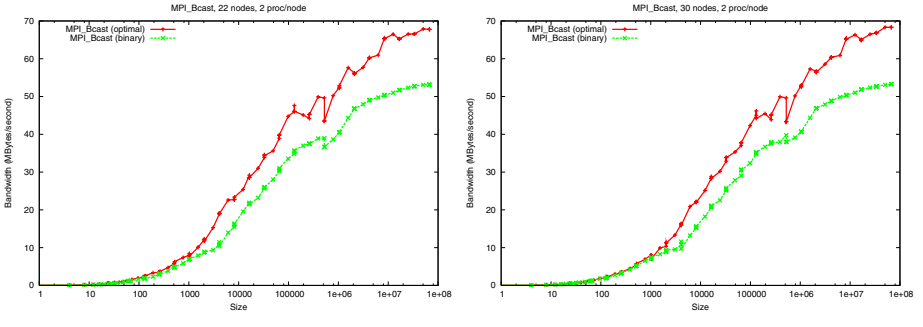

Fig. 3. Bandwidth of optimal and pipelined binary tree with two MPI processes per node, $p = 22$ (left) and $p = 30$ (right), and data size m up to 64MBytes

Finally, Figure 4 illustrates the SMP-overhead of the current implementation. The flat version of the algorithm is compared to the SMP-algorithm with one process/node and to the SMP-algorithm with two processes/node. Even with one process/node, the SMP-adapted algorithm (in the current implementation) has

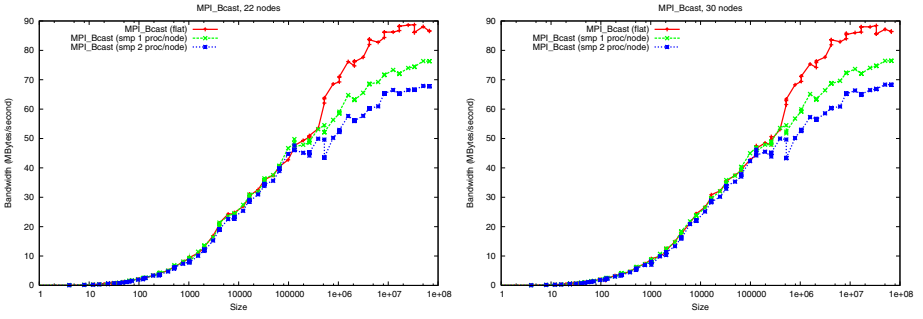

Fig. 4. New broadcast algorithm for the flat case vs. the SMP algorithm with one process/node vs. the SMP-algorithm with two processes/node, $p = 22$ (left) and $p = 30$ (right), and data size m up to 64MBytes. Note that the SMP algorithm even for the one process/node case performs an extra memory copy compared to the flat algorithm.

to perform an extra memory copy compared to the flat algorithm, and Figure 4 estimates the cost of this. Up to about 100KBytes the performance of the three versions is similar, after that the cost of the extra copying and node-internal, shared memory broadcast becomes visible, and degrades the performance. However, improvements in the implementation are still possible to overlap intra-node communication and node-internal broadcast as described in Section 2.4.

4 Conclusion

We described the main ideas behind a new, theoretically optimal broadcast algorithm for "flat", homogeneous, fully connected networks, and discussed an easy adaption to hierarchical systems like clusters of SMP nodes. On a small Myrinet cluster significant bandwidth improvements over other, commonly used broadcast algorithm were demonstrated, both for the "flat" case with one MPI process/node, and for the case with more than one process per node. Further implementation improvements to better overlap network communication and intra-node communication are still possible, and will be pursued in the future.

References

1. J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, 1997.
2. E. W. Chan, M. F. Heimlich, A. Purkayastha, and R. A. van de Geijn. On optimizing collective communication. In *Cluster 2004*, 2004.
3. M. Gołebiewski, H. Ritzdorf, J. L. Träff, and F. Zimmermann. The MPI/SX implementation of MPI for NEC's SX-6 and other NEC platforms. *NEC Research & Development*, 44(1):69–74, 2003.

4. S. L. Johnsson and C.-T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1268, 1989.
5. S. Juhász and F. Kovács. Asynchronous distributed broadcasting in cluster environment. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 164–172, 2004.
6. P. Sanders and J. F. Sibeyn. A bandwidth latency tradeoff for broadcast and reduction. *Information Processing Letters*, 86(1):33–38, 2003.
7. E. E. Santos. Optimal and near-optimal algorithms for k-item broadcast. *Journal of Parallel and Distributed Computing*, 57(2):121–139, 1999.
8. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference*, volume 1, The MPI Core. MIT Press, second edition, 1998.
9. R. Thakur, W. D. Gropp, and R. Rabenseifner. Improving the performance of collective operations in MPICH. *International Journal on High Performance Computing Applications*, 19:49–66, 2004.
10. J. L. Träff. A simple work-optimal broadcast algorithm for message passing parallel systems. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 11th European PVM/MPI Users' Group Meeting*, volume 3241 of *Lecture Notes in Computer Science*, pages 173–180, 2004.
11. J. L. Träff and A. Ripke. Optimal broadcast for fully connected networks. In *High Performance Computing and Communications (HPCC'05)*, Lecture Notes in Computer Science, 2005.