Optimal Broadcast for Fully Connected Networks

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 develop and implement a new optimal broadcast algorithm for fully connected, bidirectional, one-ported networks under a linear communication cost model. For *any* number of processors p the number of communication rounds required to broadcast N blocks of data is $\lceil \log p \rceil - 1 + N$. For data of size m, assuming that sending and receiving m data units takes time $\alpha + \beta m$, the best running time that can be achieved is $(\sqrt{(\lceil \log p \rceil - 1)\alpha} + \sqrt{\beta m})^2$, meeting the lower bound under the assumption that the m units are sent as N blocks. This is better than previously known (and implemented) results, which achieve this only when p is a power of two (or other special cases), in particular, the algorithm is (theoretically) a factor two better than the commonly used, pipelined binary tree algorithm. The algorithm has a regular communication pattern based on simultaneous binomial-like trees, and when the number of blocks to be broadcast is one, degenerates into a binomial tree broadcast. Thus the same algorithm can be used for all message sizes m. The algorithm has been incorporated into a state-of-the-art MPI (*Message Passing Interface*) library. We demonstrate significant practical improvements of up to a factor 1.5 over several other, commonly used broadcast algorithms.

1 Introduction

There has recently been renewed interest in efficient, portable and easy to implement broadcast algorithms for use in *Message Passing Interface* (MPI) libraries [12], the current *de facto* standard for distributed memory parallel computers [3,9,10,13,15]. Earlier theoretical results, typically assuming a strict, one-ported communication model in which processors can either send or receive messages are summarized in [5,7]. Early implementations typically assume a homogeneous, fully connected network, and were often based on straightforward binary or binomial trees, which are inefficient for large data sizes. Better MPI libraries (for instance [6]) take the hierarchical communication system of current clusters of SMP nodes into account by broadcasting in a hierarchical fashion, and use pipelined binary trees, or algorithms based on recursive halving [13,15] that are much better as data size increases. However, these algorithms are all (at least) a factor two off from the theoretical optimum.

A theoretically better algorithm for hypercubes (later extended to incomplete hypercubes) was proposed by Johnsson and Ho [8,14]. Instead of pipelining the

L.T. Yang et al. (Eds.): HPCC 2005, LNCS 3726, pp. 45–56, 2005.
© Springer-Verlag Berlin Heidelberg 2005

blocks successively through a fixed-degree tree, in this so called *edge-disjoint spanning binomial tree algorithm* the root processor sends successive blocks to its children in a round-robin fashion, each of which functions as a root in a binomial tree that is edge-disjoint from the other binomial trees. Another interesting algorithm based on so called *fractional trees* [10] provides for a smooth transition from pipelined binary tree to a linear pipeline algorithm as data size increases, and gives an improvement over both for medium data sizes. In the arguably more realistic *LogP* model [1,4] an (near) optimal algorithm was given in [11], but without implementation results. Non-pipelined broadcast algorithms in hierarchical, clustered, heterogeneous systems were discussed in [2], which proposes a quite general model for such systems.

In this paper we first give an algorithm for one-ported, fully connected networks of a pipelined broadcast algorithm that is quite similar to the algorithm of Johnsson and Ho [8] when the number of processors is a power of two. Fully connected networks are realized by crossbars as in the Earth Simulator, and the assumption is a reasonable approximation for medium sized networks for high-end clusters like Myrinet or Quadrics. Our main result extends this algorithm to arbitrary numbers of processors. An important feature of the algorithm is that it degenerates towards the binomial tree algorithm as the number of blocks to be broadcast decreases. A smooth transition from short data to long data behavior is thus possible with one and the same algorithm. The optimal algorithm has been implemented in a state-of-the-art MPI library [6], and we present an experimental comparison to other, commonly used broadcast algorithms on a 32-node AMD cluster with Myrinet interconnect, showing a bandwidth increase of more than a factor 1.5 over these algorithms.

2 Problem, Preliminaries and Previous Results

For the rest of this paper m denotes the amount of data to be broadcast, and p the number of processors which are numbered from 0 to $p-1$. Logarithms are to the base 2, and we let $n = \lceil \log p \rceil$. Without loss of generality, we assume that broadcast is from processor 0 (otherwise, processor 0 and the broadcast *root* processor exchange roles). In MPI, broadcast is a *collective operation* `MPI_Bcast(buffer,count,datatype,root,comm)` to be executed by all processors in the communicator `comm`.

We assume a fully connected, homogeneous network with one-ported, bidirectional communication, and a simple, linear cost model. A processor can simultaneously send and receive a message, possibly from two different processors, and the time to send m units of data is $\alpha + \beta m$ where α is the start-up latency and β the transfer time per unit. In the absence of network conflicts this model is somewhat accurate, although current communication networks and libraries typically exhibit a large difference in bandwidth between "short" and "long" messages. The extended *LogGP* model, for instance, attempts to capture this [1], and we discuss this problem further in Section 4. The model also does not match current clusters of SMP nodes that have a hierarchical communication system. We cater

for this by broadcasting hierarchically on such systems, but do not discuss this further in this paper; see instead the companion paper [16].

2.1 Lower Bounds

In the homogeneous, linear cost model, the lower bound for broadcasting the m data is $\max(\alpha n, \beta m)$. Assuming furthermore that the m data is sent as N blocks of m/N units, the number of rounds required 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 number of rounds can be found as

$$N_{\text{opt}} = \sqrt{\frac{(n-1)\beta m}{\alpha}}$$

and the optimal block size as

$$B_{\text{opt}} = \sqrt{\frac{m\alpha}{(n-1)\beta}} = \sqrt{\frac{m}{n-1}}\sqrt{\frac{\alpha}{\beta}} \tag{1}$$

for a total running time of

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

For proofs of these lower bounds, see e.g. [8,10]. Other algorithms are off from the lower bound either by a larger latency term (e.g. linear pipeline) or a larger transmission time (e.g. $2\beta m$ for a pipelined binary tree).

3 The Algorithm

In this section we give a high-level description of our new optimal broadcast algorithm. The details are filled in first for the easier case where p is a power of two, then for the general case. First, we assume that the data to be broadcast have been divided into N blocks, and that $N > n$. Note that the algorithm as presented here only achieves the $n-1+N$ rounds for certain combinations of p and N; in some cases up to $(n - N \bmod n - 1)$ extra rounds may be required for some processors (see Subsection 3.4).

The algorithm is pipelined in the sense all processors are 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 (processor 0) sends the first n blocks successively to its children. The root continues in this way sending blocks successively to its children in a round robin fashion. A sequence of n blocks is called a *phase*.

The non-root processors receive their first block from the 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 goes into 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 $0 \leq i < N$.

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. For the general case where p is not a power of two, small modifications of the fill phase and the last rounds are necessary.

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. In the next subsections we describe how to compute these functions. The functions next and prev determine the communication pattern. 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:

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 (2).*

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, with slight modifications for the last phase, 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*. The broadcast algorithm is *correct* if the full block schedule fulfills the conditions that either

$$\mathsf{recvblock}(i,r) \in \{\mathsf{recvblock}(j, \mathsf{prev}(i,r)) \mid 0 \leq j < i\} \tag{3}$$

or

$$\mathsf{recvblock}(i,r) = \mathsf{recvblock}(\mathsf{first}(\mathsf{prev}(i,r)), \mathsf{prev}(i,r)) \tag{4}$$

for $0 \leq i < n$, i.e. the block that processor r receives in round i from processor $\mathsf{prev}(i,r)$ has been received by that processor in a previous round.

When $N = 1$ the algorithm degenerates into an ordinary binomial tree broadcast, that 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 communication buffer space.

3.1 Powers of Two Number of Processors

For the case where p is a power of two, the communication pattern and block schedule is quite simple. For $j = 0,\ldots n-1$ processor r receives a block from processor $\mathsf{prev}(j,r) = (r - 2^j) \bmod p$ and sends a block to processor $\mathsf{next}(j,r) = (r + 2^j) \bmod p$, that is the distance to the previous and next processor doubles in each round.

The root sends n successive blocks to processors $1, 2, 4, \ldots, 2^j, \ldots 2^{n-1}$ for $j = 0,\ldots n-1$ with this pattern. The subtree of child processor $r = 2^j$ consists of the processors $(r + 2^k) \bmod p, k = j+1, \ldots, n-1$. 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 all processors in group j. Figure 2 shows the

group:	0	1	2				3							
proc r:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`schedule`:	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0

Fig. 2. First block schedule for $p = 16$

blocks received in rounds 0 to 3 for $p = 16$. We call the sequence of first blocks received by the processors the *first block schedule*, i.e. `schedule`$[r]$ is the first block that processor r will receive (in round $\mathsf{first}(r)$). It is easy to compute the `schedule` array: assume `schedule`$[i]$ computed for groups $0, 1, \ldots, j-1$, that is for $1 \leq i < 2^j$; then set `schedule`$[2^j] = j$, and `schedule`$[2^j + i] =$ `schedule`$[i]$ for $i = 1, \ldots, 2^j - 1$ (incidentally this sequence is a palindrome). We need the following property of the `schedule` array.

Lemma 1. *Any segment of size 2^j of the first block schedule for p a power of two contains exactly $j + 1$ different blocks.*

The proof is by induction on j.

Using the first block schedule we can compute a full block schedule $\mathsf{recvblock}$ as follows. In round n (the first round after the fill phase) processor r receives a block from processor $r' = (r - 1) \bmod p$; this can only be the block that processor r' received in its first round $\mathsf{first}(r')$, so take $\mathsf{recvblock}(0, r) =$ `schedule`$[r']$. For $0 < i < \mathsf{first}(r)$ we take $\mathsf{recvblock}(i, r)$ to be the *unique* block in `schedule`$[\mathsf{prev}(i, r) - 2^i + 1, \mathsf{prev}(i, r)]$ which is *not* in `schedule`$[\mathsf{prev}(i, r) + 1, r]$ and is *not* `schedule`$[r]$. These two adjacent segments together have size 2^{i+1}, and by Lemma 1 contain exactly $i + 2$ different blocks, one of which is `schedule`$[r]$ and another i of which have already been used for $\mathsf{recvblock}(i-1, r), \mathsf{recvblock}(i-2, r), \ldots, \mathsf{recvblock}(0, r)$. The first block received by processor r is `schedule`$[r]$ so $\mathsf{recvblock}(\mathsf{first}(r), r) =$ `schedule`$[r]$. Finally, for $i > \mathsf{first}(r)$ take $\mathsf{recvblock}(i, r)$ to be the unique block in the interval `schedule`$[\mathsf{prev}(i, r) - 2^{i-1} + 1, \mathsf{prev}(i, r)]$. By construction either $\mathsf{recvblock}(i, r) \in \{\mathsf{recvblock}(j, \mathsf{prev}(i, r)) \mid j < i\}$ or $\mathsf{recvblock}(i, r) = \mathsf{recvblock}(\mathsf{first}(\mathsf{prev}(i, r)), \mathsf{prev}(i, r))$. We have argued for the following proposition.

Proposition 1. *When p is a power of two the full block schedule constructed above is correct (and unique).*

The full block constructed above furthermore has the property that for all processors in group j

$$\{\mathsf{recvblock}(0, r), \mathsf{recvblock}(1, r), \ldots, \mathsf{recvblock}(j, r)\} = \{0, 1, \ldots, j\} \qquad (5)$$

and $\mathsf{recvblock}(j, r) = j$ for the first processor $r = 2^j$ in group j.

Without further proof we note that it is possible using the bit pattern of the processor numbering to compute for each processor r each block $\mathsf{recvblock}(i, r)$ of the full block schedule in $O(\log p)$ bit operations (and no extra space), for a total of $O(\log^2 p)$ operations per processor. This may be acceptable for a practical implementation where B_{opt} and N_{opt} are considerable.

3.2 Arbitrary Number of Processors

When p is not a power of two, the uniqueness of the first block schedule as guaranteed by Lemma 1 no longer holds (the segment of size 4 starting at processor 8 for the first block schedule for $p = 22$ in Figure 3 has $4 > 3$ different blocks). This is one obstacle for generalizing the construction to arbitrary number of processors, and solving this is the main contribution of this paper.

group:	0	1	2			3					4										
proc r:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
`schedule`:	0	1	2	0	1	3	0	1	2	0	4	0	1	2	0	1	3	0	1	2	0

group:	0	1	2			3					4										
proc r:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
`block`	**0**	0	1	2	0	1	3	0	1	2	0	4	0	1	2	0	1	3	0	1	2
	2	**1**	0	1	2	0	1	3	0	1	2	2	4	0	1	2	0	1	3	0	1
	1	2	**2**	**0**	**1**	2	2	2	3	3	3	1	2	4	4	4	2	2	2	3	3
	3	3	3	3	3	**3**	**0**	**1**	**2**	**0**	1	3	3	3	3	3	4	4	4	4	4
	4	4	4	4	4	4	4	4	4	4	**4**	**0**	**1**	**2**	**0**	**1**	**3**	**0**	**1**	**2**	**0**

Fig. 3. First (top) and full block schedules (bottom) for $p = 22$. The part of the full block schedule corresponding to the first block schedule is shown in bold.

The communication pattern must satisfy for each j that the total size of groups $0, 1, \ldots, j-1$ plus the root processor must be at least the size of 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

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

Figure 3 illustrates the definition with $p = 22$: $\text{groupsize}(0) = 1, \text{groupsize}(1) = 1, \text{groupsize}(2) = 3, \text{groupsize}(3) = 5, \text{groupsize}(4) = 11$. It obviously holds that both $\text{groupsize}(j,p) \leq 2^j$ and $\text{groupstart}(j,p) \leq 2^j$. For p a power of two $\text{groupsize}(j,p) = 2^j$, so the definition subsumes the power of two case.

Now we can define the next and prev functions analogously to (and subsuming) the powers-of-two case:

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

This communication pattern leads to an exception for the fill phase of the algorithm as formalized in Figure 1. 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. For an example consider the full block schedule of Figure 3. Here processor 1 would attempt to send to processor 3 in fill round 2; processor 3, however, will become active in round 3 and receive from root 0.

3.3 Computing the Block Schedule

A greedy algorithm almost suffices for computing the full block schedule for the non-powers of two case. For each processor r the construction is as follows.

1. Construct the *first block schedule* `schedule` as described in Subsection 3.1: set $\texttt{schedule}[\text{groupstart}(j,p)] = j$, and $\texttt{schedule}[\text{groupstart}(j,p)+i] = \texttt{schedule}[i]$ 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 $\texttt{block}[j]$ the first block $\texttt{schedule}[i]$ different from $\texttt{block}[j-1], \texttt{block}[j-2], \ldots \texttt{block}[0]$, and in $\texttt{found}[j]$ the index i at which $\texttt{block}[j]$ was found.
3. If $\mathsf{prev}(j,r) < \texttt{found}[j]$ either
   - if $\texttt{block}[j] > \texttt{block}[j-1]$ then swap the two blocks,
   - else mark $\texttt{block}[j]$ as unseen,

   and continue scanning in Step 2.
4. Set $\texttt{block}[\mathsf{first}(r)] = \texttt{schedule}[r]$
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.

To see that the full block schedule thus constructed satisfies the correctness conditions (3) and (5) we need the following version of Lemma 1.

Lemma 2. *Any segment of size* $\sum_{i=0}^{j} \text{groupstart}(i,p)$ *of the first block schedule contains at least* $j+1$ *different blocks.*

Again the proof is by induction on j, but is omitted due to limited space.

When $\mathsf{prev}(j,r) \geq \texttt{found}[j]$ the next $\texttt{block}[j]$ is within the segment already scanned by processor $\mathsf{prev}(j,r)$, and taking this block as $\mathsf{recvblock}(j,r)$ is therefore correct. The violation $\mathsf{prev}(j,r) < \texttt{found}[j]$ means that the block that has been found for processor r for round j has possibly not been seen by processor $\mathsf{prev}(j,r)$. To ensure correctness we could simply mark the block as unseen and continue scanning; Lemma 2 guarantees that a non-violating block can be found that has been seen by processor $\mathsf{prev}(j,r)$ before round j. However, since we must also guarantee Condition (5), large numbered blocks (in particular block j for processors in group j) cannot be postponed till rounds after $\mathsf{first}(r)$. The two alternatives for handling the violation suffice (as per this proof sketch) for the main theorem to hold.

Theorem 2. *For any number of processors p the full block schedule constructed above is correct. The full block schedule can be constructed in $O(p \log p)$ steps.*

Since the whole first block schedule has to be scanned for each r, the construction above takes $O(p)$ steps per processor and $O(p^2)$ steps for constructing the full schedule. It is relatively easy to reduce the time to $O(p \log p)$ steps for the full block schedule. The idea is to maintain for each possible block $0 \leq b < n$ a set of processors that have not yet included b as one of its found blocks `block`$[i]$. Scanning the first block schedule as above, each new processor is inserted into the block set for all blocks, and for each $b =$ `schedule`$[r]$ all processors now in the set for block b are ejected and the condition of Step 3 is checked for each. Two scans of the `schedule` array are necessary, and since each r is in at most n queues the $O(p \log p)$ time bound follows.

Neither is, of course, useful for on-line construction at each broadcast operation, so instead the block schedule must be constructed in advance. For an MPI implementation of the broadcast algorithm this is not a problem because collective operations can only be executed by processors belonging to the same communication domain (*communicator* in MPI terminology), which must have been set up prior to the `MPI_Bcast(...)` call. The full block schedule can be constructed at communicator construction time and cached with the communicator for use in later broadcast operations. With a small trick to cater for the general case where the broadcast is not necessarily from root processor 0, it is possible to store the full block schedule in a distributed fashion with only $O(\log p)$ space per processor.

3.4 The Last Phase

To achieve the claimed $n - 1 + N$ number of rounds for broadcasting N blocks, modifications to the full block schedule for the last phase are necessary. Using the full block schedule defined in Section 3, after $n-1+N$ rounds each processor has received N different blocks. Some of these may, however, be larger than N (that is, $\mathsf{recvblock}(i, r) \geq N$ for some rounds i belonging to the phase from $N-1$ to $n-1+N$), and processors for which this happens will miss some blocks $< N$. To repair this situation, a mapping of the blocks $\geq N$ to blocks $< N$ has to be found such that after $n-1+N$ rounds all processors have received all N blocks. In the rounds of the last phase where the root would have sent a block $b > N$, the block onto which b is mapped is sent (again) instead. In cases where such a mapping does not exist, the communication pattern for the last phase must also be changed. We do not describe this here.

4 Performance

The optimal broadcast algorithm has been implemented and incorporated into NEC's state-of-the-art MPI implementations [6]. We compare this implementation to implementations in the same framework of a simple binomial tree algorithm, a pipelined binary tree algorithm, and a recently developed algorithm based on recursive halving of the data [15].

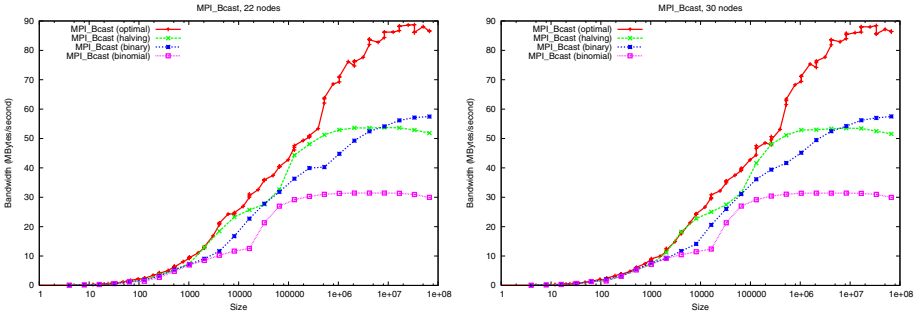

Fig. 4. Bandwidth of 4 different broadcast algorithms for fixed number of processors $p = 22$ (left) and $p = 30$ (right) with data size m up to 64MBytes

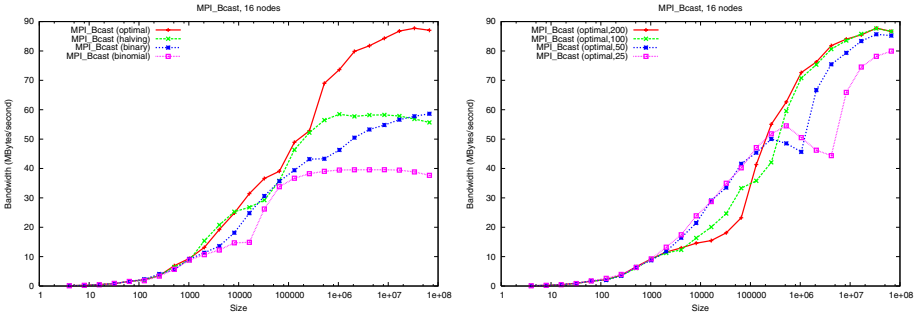

Fig. 5. Bandwidth for $p = 16$, data size m up to 64MBytes, for the piecewise linear (left) and the linear model (right) with four different values for the $\sqrt{\alpha/\beta}$ factor of equation (1)

Experiments have been performed on a 32-node, dual-processor AMD cluster with Myrinet interconnect. We consider only the homogeneous, non-SMP case here, that is the case where only one processor per node is active.

Figure 4 compares the four algorithms for fixed number of processors $p = 22$ and $p = 30$ and data size m from 0 to 64MBytes. For large data sizes the new optimal broadcast algorithm is more than a factor 1.5 faster than both the pipelined binary tree and the halving algorithm. To cater for the fact that communication bandwidth is not independent of the message size, we have, instead of using the simple, linear cost model, modeled the communication time as a piecewise linear function $t(m) = \alpha_1 + m\beta_1$ for $0 \leq m < \gamma_1$, $t(m) = \alpha_2 + m\beta_2$ for $\gamma_1 \leq m < \gamma_2$, $\ldots$, $t(m) = \alpha_k + m\beta_k$ for $\gamma_{k-1} \leq m < \infty$ with $k = 4$ pieces for our cluster. Finding the optimum block size in this model is not much more complicated or expensive than in the linear cost model (case analysis). Figure 5 contrasts the behavior of the algorithm under the piecewise linear cost model to the behavior under the linear model with four different values for the factor $\sqrt{\alpha/\beta}$ that determines the optimum block size in equation (1). A smooth bandwidth increase

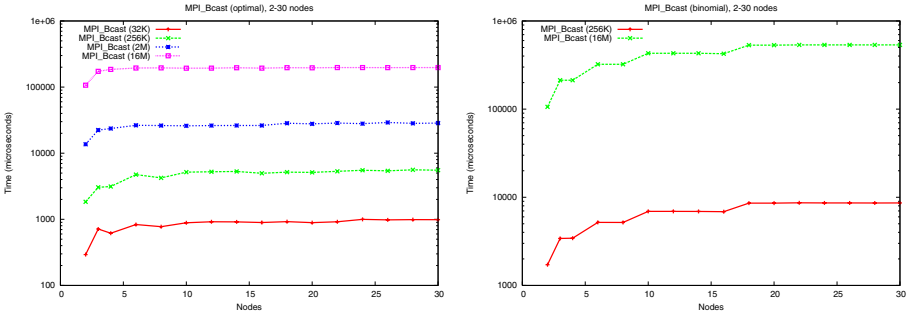

Fig. 6. Running time of the optimal broadcast algorithm for $p = 2, \ldots, 30$ processors and fixed message sizes $m = 32\text{KBytes}, 256\text{KBytes}, 2\text{MBytes}, 16\text{MBytes}$ (left). For comparison the running time for the binomial tree algorithm is given for $m = 256\text{Kbytes}$ and $m = 16\text{MBytes}$ (right).

can be achieved with the piecewise linear model which is not possible with a fixed, linear model.

Finally, Figure 6 shows the scaling behavior of the optimal algorithm with four fixed data sizes and varying numbers of processors. For reference, the results are compared to the binomial tree algorithm. Already beyond 3 processors the broadcast time for $m > 32\text{KBytes}$ is independent of the number of processors.

5 Conclusion

We gave a new, optimal broadcast algorithm for fully connected networks for arbitrary number of processors that broadcasts N blocks over p processors in $\lceil \log p \rceil - 1 + N$ communication rounds. On a 32-node Myrinet PC-cluster the algorithm is clearly superior to other widely used broadcast algorithms, and gives close to the expected factor of two bandwidth improvement over a pipelined binary tree algorithm.

The algorithm relies on off-line construction of a communication schedule, which can be both space and time consuming. We would therefore like to be able to compute for any p and each r the $\mathsf{recvblock}(\cdot, r)$ and $\mathsf{sendblock}(\cdot, r)$ functions fast and space efficiently as is possible for the case where p is a power of two.

References

1. A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. J. Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.
2. F. Cappello, P. Fraigniaud, B. Mans, and A. L. Rosenberg. HiHCoHP: Toward a realistic communication model for Hierarchical HyperClusters of Heterogeneous Processors. In *15th International Parallel and Distributed Processing Symposium (IPDPS01)*, pages 42–47, 2001.

3. E. W. Chan, M. F. Heimlich, A. Purkayastha, and R. A. van de Geijn. On optimizing collective communication. In *Cluster 2004*, 2004.
4. D. E. Culler, R. M. Karp, D. Patterson, A. Sahay, E. E. Santos, K. E. Schauser, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, 1996.
5. P. Fraigniaud and E. Lazard. Methods and problems of communication in usual networks. *Discrete Applied Mathematics*, 53(1–3):79–133, 1994.
6. 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.
7. S. M. Hedetniemi, T. Hedetniemi, and A. L. Liestman. A survey of gossiping and broadcasting in communication networks. *Networks*, 18:319–349, 1988.
8. S. L. Johnsson and C.-T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1268, 1989.
9. 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.
10. P. Sanders and J. F. Sibeyn. A bandwidth latency tradeoff for broadcast and reduction. *Information Processing Letters*, 86(1):33–38, 2003.
11. E. E. Santos. Optimal and near-optimal algorithms for k-item broadcast. *Journal of Parallel and Distributed Computing*, 57(2):121–139, 1999.
12. 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.
13. 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.
14. J.-Y. Tien, C.-T. Ho, and W.-P. Yang. Broadcasting on incomplete hypercubes. *IEEE Transactions on Computers*, 42(11):1393–1398, 1993.
15. 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.
16. J. L. Träff and A. Ripke. An optimal broadcast algorithm adapted to SMP-clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. 12th European PVM/MPI Users' Group Meeting*, volume 3666 of *Lecture Notes in Computer Science*, 2005.