Parallel Processing Letters

A PIPELINED BROADCAST FOR MULTIDIMENSIONAL MESHES

JERRELL WATTS

Scalable Concurrent Programming Laboratory, California Institute of Technology, Pasadena, California 91125, USA

and

ROBERT VAN DE GEIJN

Department of Computer Sciences, University of Texas at Austin, Austin, Texas 78712, USA

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

ABSTRACT

We address the problem of performing a pipelined broadcast on a mesh architecture. Meshes require a different approach than other topologies, and their very nature puts a tighter bound on the performance that one can hope to achieve. By using the appropriate techniques, however, one can obtain excellent performance for sufficiently long messages. The resulting algorithm will work on meshes of any dimension with any number of nodes. Our model assumes that the mesh is a torus and/or that it has bidirectional links and uses wormhole routing. Performance data from the Cray T3D are included.

Keywords: broadcast, pipelining, communication, mesh, torus

1. Introduction

The broadcast is a fundamental routine in any communication library. As a result, its implementation should be as efficient as possible. To that end, no single algorithm will suffice: A routine which minimizes the number of message operations will perform well for short messages, when start-up latencies are the primary time component. For longer messages, data transmission time dominates, so the best algorithm minimizes data flow and contention, even if additional message operations are required to do so. Therefore, one must use a suite of algorithms (and their hybrids) to achieve piecewise optimality over a wide range of parameters [2, 3].

Previously, we presented the recursive-splitting (RS) and scatter-collect (SC) broadcasts, efficient algorithms for short and long messages, respectively [1]. These routines could then be hybridized to produce an algorithm that is efficient over a wide range of message lengths, as we have shown in [2, 3]. Although the strategy presented in those papers is very effective, the pure SC broadcast is still a factor of two worse than the theoretical optimal for very long messages. One way to improve performance in this range would be to switch from the SC broadcast to a pipelined broadcast when the message reaches a certain size. (This crossover point can be accurately determined by using the execution time formulae for the algorithms used.) This strategy was not adopted in [2, 3] because no good mesh-based pipelined algorithm was then available; theoretically superior pipelined algorithms were outperformed by the simpler SC broadcast. Also, in those papers, we investigated the implementation of an entire collective communication library, for which the SC broadcast was more convenient.

The objective of this paper is thus to find an efficient pipelined broadcast for a given mesh-based architecture. While the idea of a pipelined broadcast is certainly not a new one, methods for implementing them efficiently on a modern mesh interconnect topology have, to our knowledge, not yet been found. Attempts have been made in [1, 5, 12, 13], but these techniques either rely on assumptions that are not applicable to today's mesh-based architectures, or they do not achieve the same performance possible with pipelined broadcasts on other topologies, such as hypercubes [7]. Furthermore, these methods are designed for two-dimensional meshes and do not readily generalize to meshes of higher dimension.

Meshes present two difficulties for pipelined broadcasts: First, the minimum pipeline depth is considerably greater than many other topologies. Namely, the pipeline depth of a d-dimensional mesh of p processors is at least $\Omega(dp^{1/d})$, whereas on a hypercube the pipeline depth is $\Theta(\log_2(p))$. Second, edge-disjoint spanning trees embedded in meshes often lead to synchronization difficulties which do not occur in hypercube-based algorithms such as Ho and Johnsson's edge-disjoint spanning tree (EDST) algorithm [7]. Typically, various nodes receive multiple messages simultaneously, greatly increasing the running time of the broadcast and producing very erratic behavior. This phenomenon was illustrated by the EDF broadcast in [1]. Explicit synchronization can alleviate this problem but would greatly increase the running time of the algorithm. The best approach would then seem to be one that reduces pipeline depth and requires as little explicit synchronization as possible to function properly. Such an approach should also generalize to meshes of any dimension. Furthermore, the algorithm should stress simplicity over peak theoretical performance. (Many proposed algorithms have been ignored because their complexity and subtleties are impediments to their implementation.) This paper presents an algorithm that meets all of the above requirements.

2. Assumptions and Conventions

The following assumptions are made regarding the appropriate architecture for the algorithm presented in this paper:

(i) The interconnect topology is a d-dimensional mesh with $m_0, m_1, \ldots, m_{d-1}$ nodes in each dimension, respectively. The total number of nodes, p is thus $m_0 \times m_1 \times \ldots \times m_{d-1}$. A node is identified by the Cartesian d-tuple that represents its position in each dimension.

(ii) If the mesh is a true torus, the links need only be unidirectional. If not a torus, the mesh must have bidirectional links and use wormhole routing. The latter allows one to treat a non-torus mesh as a logical, unidirectional torus by restricting all communication to one direction in each dimension. For example, on a two-dimensional mesh, if all message sends are either to the "south" or "east," then all logical wraparounds proceed "north" or "west," respectively, and do not conflict. Furthermore, the presence of wormhole routing implies that a message traveling between two nodes utilizes all of the channels on the path determined by the routing method (e.g., e-cube or dimensional-order).

(iii) Nodes are single-ported. This implies that a node can send to only one other node at a time. It can simultaneously receive messages from the same or different nodes, but if more than one message is being received at any given time, then Assumption 4 applies.

(iv) Assuming no conflicts occur, if communication is limited to physical nearest neighbors on true torii and/or if wormhole routing is used (in which case communication need not be restricted to nearest neighbors), sending a message of length n from one node to another requires time $\alpha + n\beta$, where α and β represent the communication start-up time and per item (byte) transfer time, respectively.[a] If $i > 1$ messages cross a channel simultaneously, then β is increased by a factor of i.

(v) The message-passing library available to implement the algorithm provides a synchronous send function, **ssend**(x,a), for a message, x, and a destination node, a.[b] It must also provide a non-blocking receive function, **irecv**(y,b), for a message, y, and a source node, b, as well as a blocking function, **wait**(y), that returns when y is received. Examples of such functions include **MPI_Ssend**(), **MPI_Irecv**() and **MPI_Wait**() under the MPI standard [11].

The following notational conventions are used in this paper:

- c_i indicates node c's position in the i^{th} dimension.
- $x_{(i,j)}$ indicates the i^{th} portion of a vector of data, x, divided into j pieces of approximately equal length.

3. Target Architectures

The assumptions listed in Section 2 are applicable to some of today's prevailing architectures, such as the Cray T3D, which has a three-dimensional torus interconnect [6]. They only partially apply to the Intel Touchstone Delta and Paragon systems, because the nodes cannot send and receive simultaneously at full bandwidth, contradicting Assumption 3 [9, 10].

[a] In the case of wormhole routing, this assumption is empirically valid for known architectures. The variation in α is typically greater than the time variation between sending to nearest and most distant nodes on a very large mesh. Furthermore, due to the small amount of buffering at each router (typically a few bytes), the time for the router buffer pipeline to fill is negligible for reasonable length messages.

[b] A synchronous send is one that does not initiate until a matching receive has been executed at the destination processor. On machines that provide only blocking send functions, this property can be simulated by having the destination processor send a zero-length synchronization message to be received by the source processor prior to sending the actual message.

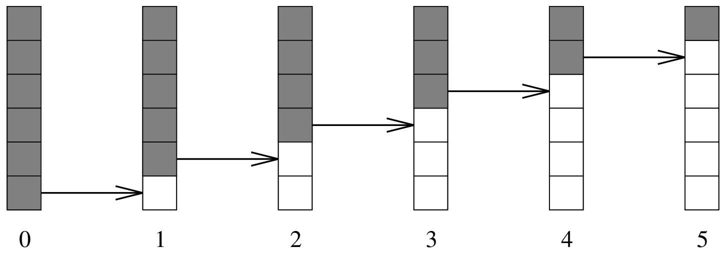


Fig. 1. A pipelined broadcast on a linear array of nodes numbered 0 through 5, originating at node 0. The pieces of the message are indicated by the boxes comprising the nodes. Darkened squares indicate those pieces that have already been received; arrows indicate the transmission of a piece to the next node in the array.

4. Previous Work: Motivations and Limitations

The motivation for pipelining is simple. Given an operation comprised of multiple, distinct steps, one can improve the running time of the operation by overlapping its steps as much as possible. In modern microprocessors, for example, the execution of an instruction is partitioned into multiple steps so that the execution stages of several instructions can be performed concurrently.

With broadcasts, there are distinct communication rounds. On any p-node machine for which Assumption 3 holds, a broadcast will require $\Omega(\log_2(p))$ steps. (Given Assumption 3, the number of nodes to which a message has been sent at each stage in an algorithm can at most double.) Optimally, this sequence of message sends forms a conflict-free logical minimum spanning tree within the network [1]. On meshes and hypercubes, the depth of this spanning tree is $\Omega(\log_2(p))$.

In a pipelined broadcast, one divides the message into many parts and overlaps the transmission stages. For example, on a p-node linear array, one would send the first piece of the message to the next node in the array. Then, while the second node sends the first piece to the third node, one would send the second piece to the second node, and so forth [12]. (See Fig. 1.) In this case, if a length n message is divided into k parts, the time for completion would be:

$$T_{1\mathrm{D}} = (p-1)(\alpha + \frac{n}{k}\beta) + (k-1)(\alpha + \frac{n}{k}\beta) \tag{1}$$

The first term is the time required for the first packet to reach the far end of the array, and the second term is the time for the remaining pieces to be received. The optimal k, determined by setting $\partial T_{1D}/\partial k = 0$ and solving for k, is:

$$k_{\mathrm{opt}} = \begin{cases} 1 & \text{if } \sqrt{(p-2)n\beta/\alpha} < 1 \\ n & \text{if } \sqrt{(p-2)n\beta/\alpha} > n \\ \sqrt{(p-2)n\beta/\alpha} & \text{otherwise} \end{cases} \tag{2}$$

k_{opt} would, of course, be rounded to the nearest integer.

In non-array topologies, the depth of the pipeline can be greatly reduced. Ho and Johnsson's EDST algorithm accomplishes this by embedding $\log_2(p)$ edge-disjoint spanning trees rooted at the nearest neighbors of the source node [7]. (See Fig. 2.) The source node alternates among the spanning trees, sending pieces of the message along each of them. This reduces the pipeline depth to $\log_2(p) + 1$.

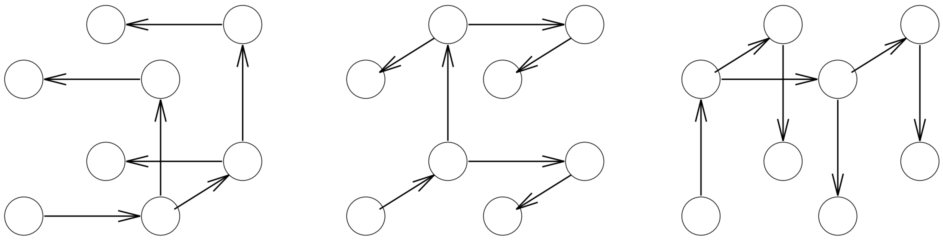


Fig. 2. The spanning trees for an EDST broadcast on a 3-cube with the origin at the front, lower lefthand node.

One cannot, however, embed a $\log_2(p)$-depth edge-disjoint minimum spanning tree in a mesh. While the logical edges of a such a tree are certainly disjoint, the physical paths that result are not. Thus, the message pipelines will overlap and interfere with one another. The best one can do is to embed a spanning tree in which all communication occurs between physical nearest neighbors. In this case, the minimum pipeline depth of a d-dimensional mesh is approximately the sum of its dimensions: $m_0 + m_1 + \ldots + m_{d-1} = \Omega(dp^{1/d})$.[c]

In previous work, we demonstrated how to perform a pipelined broadcast on a $r \times c$ two-dimensional mesh [1]. In theory, the execution time for our EDF broadcast should be quite good for long messages:

$$T_{\text{EDF}} = (k + r + c - 2)(\alpha + \frac{n}{k}\beta) \tag{3}$$

with:

$$k_{\text{opt}} = \begin{cases} 1 & \text{if } \sqrt{(r+c-2)n\beta/\alpha} < 1 \\ n & \text{if } \sqrt{(r+c-2)n\beta/\alpha} > n \\ \sqrt{(r+c-2)n\beta/\alpha} & \text{otherwise} \end{cases} \tag{4}$$

The EDF broadcast did not attain is expected performance, however, due to synchronization problems. (See Fig. 3.) In fact, its running time is fifty percent greater than what one would expect. One could explicitly synchronize each message send to eliminate this problem, but doing so would similarly increase the running time since the message start-up latency would effectively be doubled and steps would be delayed due to the fact that there are inherent conflicts between the two pipelines.[d]

The synchronization problem arises whenever the root node alternates between the spanning trees. By alternating less frequently, one is able to attain much of the theoretical performance of the EDF broadcast while avoiding the high costs that result from its practical difficulties.

[c]The spanning trees presented in [5] are actually of depth $\sqrt{p}$ for a square, two-dimensional p-processor mesh. However, the authors assume a true torus interconnect under the store-and-forward model. The tree will thus not work on a logical unidirectional torus simulated with wormhole routing on a non-torus mesh. Furthermore, their trees are dependent on the assumption that a node can send/receive simultaneously in all four directions, contradicting Assumption 3 in Section 2.

[d]The algorithm in [13] uses the same spanning trees as the EDF broadcast. However, as above, the author assumed that nodes could simultaneously send/receive in all four directions. Under such circumstances, there is no need to alternate between spanning trees. When the algorithm is modified to conform to the assumptions in Section 2, it encounters the same synchronization problems as the EDF broadcast.

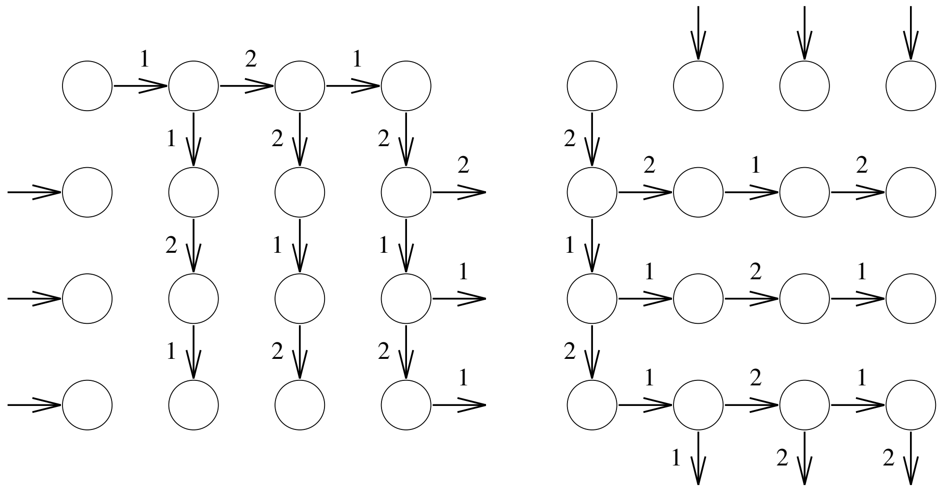


Fig. 3. The spanning trees for an EDF broadcast on a 2-D mesh with the origin as the upper lefthand node. The labels 1 and 2 on the edges indicate which ones are active during odd and even steps, respectively. Notice that the two nodes to the right and below the root node receive two messages simultaneously during the same step. This will delay the message coming in from the root, creating a bubble in the pipeline. These bubbles result in more synchronization problems, as other nodes will begin to receive more than one message at a time.

5. Dimensional-Order Pipelined Broadcast

The dimensional-order pipelined (DOPL) broadcast generalizes the EDST broadcast to arbitrary meshes in way very different from the EDF broadcast. The basic idea is quite simple: Instead of continuously alternating between dimensions, the root node pipelines part of the message along all of the nodes in each dimension, one dimension at a time. Doing so ensures that all of the nodes in a dimension are in the same phase. This prevents a node from receiving messages from more than one direction simultaneously.

Consider a two-dimensional mesh. One might pipeline the entire message along the first dimension then along the second dimension. In this case, the running time of the broadcast, T_{DOPL}, would have a lower bound of roughly $2n\beta$, which is no better than the SC broadcast we presented previously in [1]. By cutting the message in half, one can improve on this bound considerably. To start, pipeline the first half along the root node's row. Then, pipeline the second half along the root node's column, and at the same time, pipeline the first half along the other columns. Finally, pipeline the second half across all the rows, and simultaneously send the first half across from the last column to the root node's column. The lower bound on T_{DOPL} would then be $\frac{3}{2}n\beta$. (See Fig. 4.) Extending the idea to alternating between dimensions multiple times, the lower bound on a two-dimensional mesh would be:

$$\frac{k_{\text{alt}}+1}{k_{\text{alt}}}n\beta \tag{5}$$

Here, k_{alt} is the number of pieces into which the message is divided. The "+1" term is for the distribution of the last piece. In the general, d-dimensional case, $d-1$ extra steps are required to distribute the last piece. (See Fig. 5.)

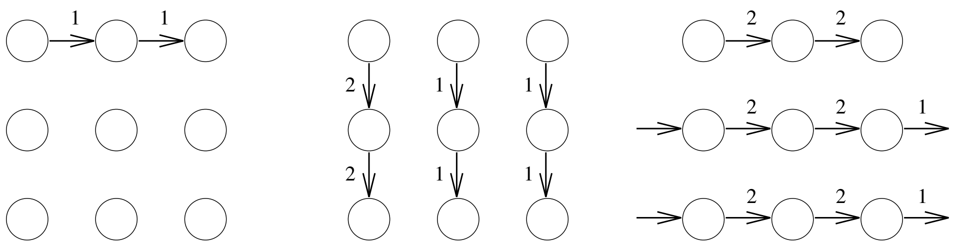


Fig. 4. The DOPL pipelines for a two-dimensional mesh with $k_{\mathrm{alt}} = 2$ and the origin at the upper left. The arrows labeled 1 and 2 indicate, respectively, the channels over which the first and second half of the message are being pipelined. The only exceptions are the wraparound 1 sends on the far right; these are direct sends and are not pipelined.

Generalizing from the example given above, the DOPL algorithm for a d-dimensional mesh is as follows: At step i, $0 \leq i \leq k_{\mathrm{alt}} + d - 2$, let the "head" node of each row in the $(i \bmod d)^{th}$ dimension be the node with the same coordinate as the source node in that dimension, and let the "tail" node be the node immediately preceding the "head" node (wrapping around to the end of the row, if necessary). The first $m_{i \bmod d} - 1$ nodes in each row pipeline the last piece received by the "head" node to the last $m_{i \bmod d} - 1$ nodes in their row. At the same time, the "tail" node sends the next piece needed by "head" node directly to it. Pseudo-code for the algorithm is given in Fig. 6.

Obviously, a higher value of k_{alt} will decrease the lower bound on the running time of the DOPL broadcast. The coefficient k_{alt} cannot be arbitrarily high, however. Each time the broadcast alternates from one dimension to another, synchronization is necessary to prevent the problems that occur with the EDF broadcast. Since there is a diminishing return for increasing k_{alt}, the additional synchronization costs will eventually outweigh this gain. To determine the optimal value of k_{alt}, one can simply iterate over values of k_{alt}, starting at d and stopping when an increase in predicted running time occurs. The running time for the DOPL broadcast is given by:

$$T_{\mathrm{DOPL}} = \sum_{i=0}^{k_{\mathrm{alt}}+d-2} (m_{i \bmod d} + k_{i \bmod d} - 2)(\alpha + \frac{n}{k_{\mathrm{alt}} k_i}\beta) \tag{6}$$

where k_i, $0 \leq i < d$ is determined in the same manner as Eq. 2:

$$k_i = \begin{cases} 1 & \text{if } \sqrt{(m_i - 2)(n/k_{\mathrm{alt}})\beta/\alpha} < 1 \\ n/k_{\mathrm{alt}} & \text{if } \sqrt{(m_i - 2)(n/k_{\mathrm{alt}})\beta/\alpha} > n/k_{\mathrm{alt}} \\ \sqrt{(m_i - 2)(n/k_{\mathrm{alt}})\beta/\alpha} & \text{otherwise} \end{cases} \tag{7}$$

6. Comparison to Other Algorithms

In this section, the DOPL broadcast is compared with the SC and EDF broadcasts originally presented in [1]. The comparisons are made for running times for variable message length and fixed mesh size (i.e., raw performance) and for fixed message length and variable mesh size (i.e., scalability). In terms of both predicted and empirical times, the DOPL

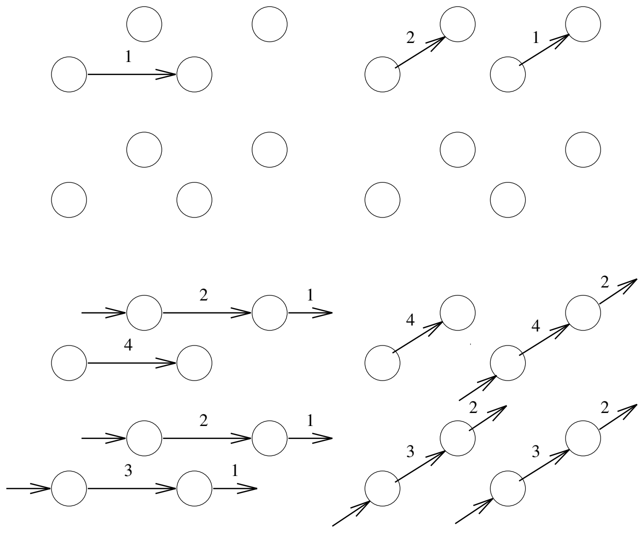


Fig. 5. The DOPL pipelines for a three-dimensional mesh with $k_{\mathrm{alt}} = 4$ and the origin at the front, upper lefthand corner. The arrows labeled 1 through 4 indicate, respective, the channels over which the quarters of the message are being pipelined. (The mesh shown is $2 \times 2 \times 2$ to save space. In practice, one would not divide the quarters of the message if there were only two nodes in a dimension; instead, one would send the entire quarter as a single message.)

broadcast compares very favorably to the SC and EDF algorithms when long messages are involved.

The empirical data that follows was taken from the Cray T3D [6]. The message-passing routines used in the implementations were layered directly on the explicit shared memory routines provided by Cray [4]. While the T3D comes very close to the ideal model presented in Section 2, it deviates in two significant ways: First, pairs of processors share a single network interface [6]. As a result, if both processors are sending and/or receiving simultaneously, they do so at half normal bandwidth. This particularly diminishes the performance of the DOPL broadcast, which depends on the ability of all nodes to send and receive simultaneously at full bandwidth. To compensate for this shortcoming, all broadcasts were conducted with only the even-numbered processors participating. This guarantees each processor exclusive access to its network interface. The second difference is that bandwidth varies with message length; shorter messages are transmitted at somewhat less bandwidth than longer ones. (See Fig. 7.) This affects all of the broadcasts, since they all break a message up into smaller pieces. As a result, the value of β must be adjusted to compensate.

The predicted execution times are given by Eq. 3 and 6 as well as those in [1]. For the T3D, $\alpha = 4.5\mu\mathrm{sec}$ and β is varied according to the inverse of the bandwidths shown in Fig. 7. Fig. 8 and 9 show the predicted versus actual times for various message lengths and partition sizes, respectively.

As one can see, the DOPL broadcast significantly outperforms the SC broadcast. Theoretically, the EDF broadcast should be the best of the three, but due to the problems ex-

```
subroutine DOPLbcast(x, src, kalt, k)
    for i := 0 to kalt + d − 2 do
        head := me                                      /* "Head" node in the (i mod d)^th dimension */
        head_{i mod d} := src_{i mod d}
        tail := head                                    /* "Tail" node in the (i mod d)^th dimension */
        tail_{i mod d} := (head_{i mod d} + m_{i mod d} − 1) mod m_{i mod d}
        if me = head then                               /* Receiving a wraparound? */
            if me = src then                            /* The source? */
                r := −1
            else                                        /* Not the source */
                r := i − d
                kr := 1
        else                                            /* Not receiving a wraparound */
            r := min(kalt, i+ level(head, src, i mod d) −d)
            kr := k_{i mod d}
        if me = tail then                               /* Sending a wraparound? */
            if head = src then                          /* Sending to the source? */
                s := −1
            else                                        /* Not sending to the source */
                s := i − d
                ks := 1
        else                                            /* Not sending a wraparound */
            s := min(kalt, i+ level(head, src, i mod d) −d)
            ks := k_{i mod d}
        if r ≥ 0 and s ≥ 0                              /* Receiving and sending */
            pipe(x_(r,kalt), kr, x_(s,kalt), ks, i mod d)
        elseif r ≥ 0                                    /* Just receiving */
            pipe(x_(r,kalt), kr, nil, 0, i mod d)
        elseif s ≥ 0                                    /* Just sending */
            pipe(nil, 0, x_(s,kalt), ks, i mod d)

subroutine pipe(x, kr, y, ks, j)
    prev := me                                          /* "Previous" node in the j^th dimension */
    prev_j := (prev_j + m_j − 1) mod m_j
    next := me                                          /* "Next" node in the j^th dimension */
    next_j := (next_j + 1) mod m_j
    for i := 0 to kr − 1 do                             /* Post receives for pieces */
        irecv(x_(i,kr), prev)
    if x⃗ = y⃗ then                                      /* Sending and receiving the same pieces? */
        for i := 0 to ks − 1 do                         /* Wait for each piece, then send it */
            wait(x_(i,ks))
            ssend(y_(i,ks), next)
    else                                                /* Sending and receiving different pieces */
        for i := 0 to ks − 1 do                         /* Send pieces */
            ssend(y_(i,ks), next)
        for i := 0 to kr − 1 do                         /* Wait for pieces */
            wait(x_(i,kr))

function level(a, b, j)
    i := 0
    while i < d and a_{(j+i) mod d} = b_{(j+i) mod d} do
        i := i + 1
    return i
```

Fig. 6. The DOPL algorithm for a message, x, originating at some node, src, and divided according to $k_{\rm alt}$ and k; me is the coordinate ID of the node on which the algorithm is running. The **pipe**() routine receives message x in kr pieces from the previous node in the j^{th} dimension, and sends message y in ks pieces to the next node in that dimension, waiting for each piece of x first if x and y point to the same message. The **level**() function determines node a's depth in the dimensional spanning tree rooted at node b, starting in the j^{th} dimension.

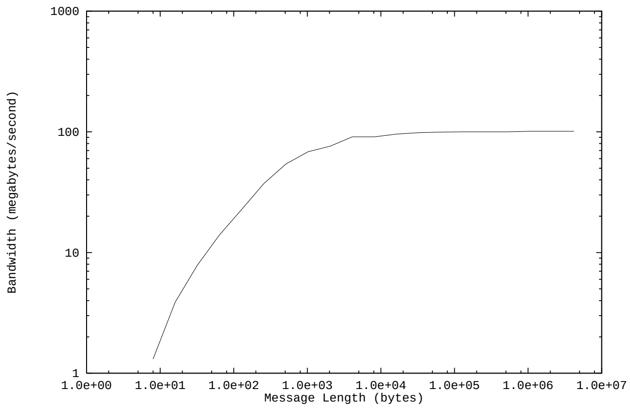


Fig. 7. Bandwidth vs. message length

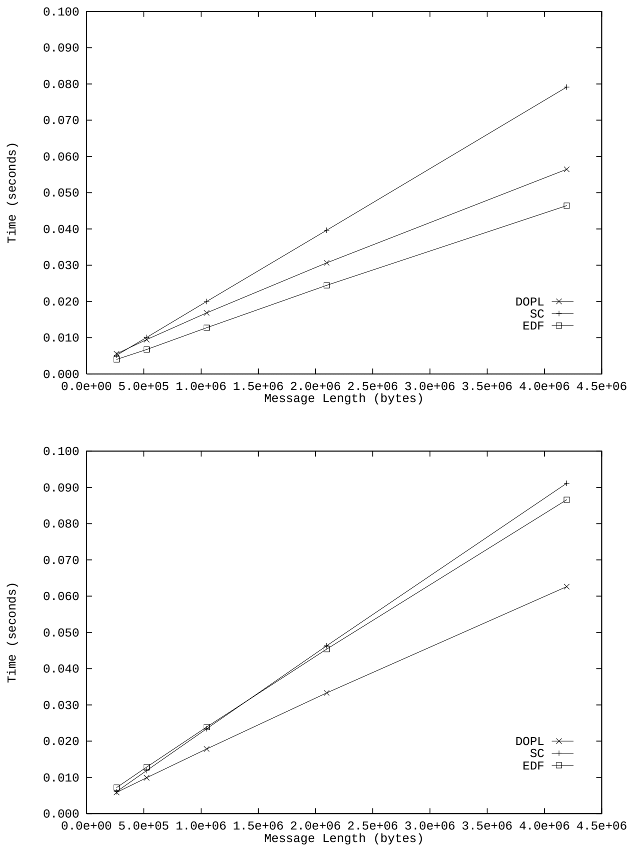
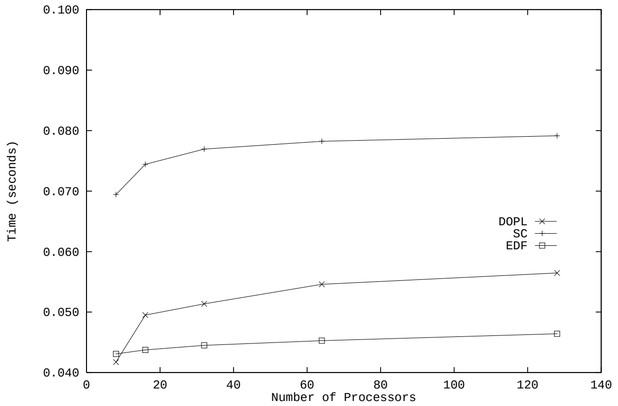


Fig. 8. Predicted (top) vs. actual (bottom) performance ($4 \times 4 \times 8$ torus).



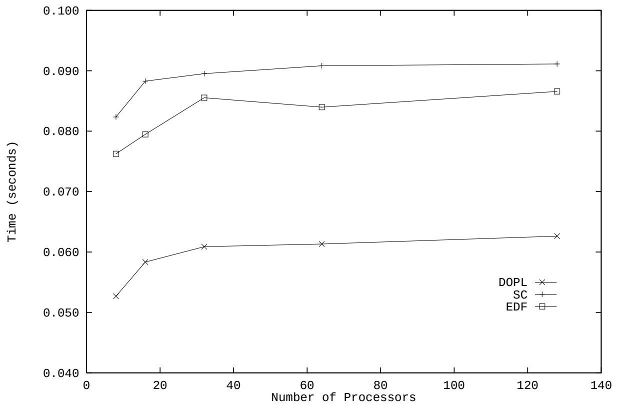


Fig. 9. Predicted (top) vs. actual (bottom) scalability (4 megabyte message).

plained in Section 4, it actually comes in second. The DOPL broadcast also shows excellent scalability as the execution time grows slowly as the mesh size increases.

7. Conclusion

This paper demonstrates that an efficient pipelined broadcast is possible on a modern, mesh-based architecture. Proper techniques are required, however, to obtain acceptable performance. Both predicted and simulated results show that the DOPL broadcast would make an excellent method for loading operating system code, user programs or large data sets in situations where hardware support for such operations is unavailable. As such, it is appropriate for use in an algorithm suit to implement an efficient broadcast over a wide range of parameters.

Acknowledgements

This research was partially sponsored by the University of Texas Center for High Performance Computing. Access to a Cray T3D was provided by the NASA Jet Propulsion

Laboratory and the California Institute of Technology.

References

1. M. Barnett, D. Payne, R. van de Geijn and J. Watts, Broadcasting on meshes with worm-hole routing, Technical Report TR-93-24, Department of Computer Sciences, the University of Texas at Austin, Nov. 1994.
2. M. Barnett, S. Gupta, D. Payne, L. Shuler, R. van de Geijn and J. Watts, Interprocessor Collective Communications Library (InterCom), in *Proc. Scalable High Perf. Comput. Conf.*, (IEEE Computer Society Press, 1994) 357–364.
3. M. Barnett, S. Gupta, D. Payne, L. Shuler, R. van de Geijn and J. Watts, Building a Collective Communications Library, to appear in *Proc. Supercomputing '94*, Washington, DC, Nov. 14–18, 1994.
4. R. Barriuso and A. Knies, *SHMEM user's guide for C*, Cray Research, Inc., 1994.
5. J.-C. Bermond, P. Michallon and D. Trystram, Broadcasting in wraparound meshes with parallel monodirectional links, *Parallel Comput.* **18(6)** (1992) 639–648.
6. Cray Research, Inc., *CRAY T3D System Architectural Overview*, 1993.
7. C.-T. Ho and S. L. Johnsson, Distributed routing algorithms for broadcasting and personalized communication on hypercubes, in *Proc. 1986 Int. Conf. on Parallel Processing*, (IEEE Computer Society Press, 1986) 640–648.
8. Intel Corporation, *Paragon OSF/1 User's Guide*, 1993.
9. S. L. Lillevik, The Touchstone 30 gigaflop Delta prototype, in *Sixth Distributed Memory Comput. Conf. Proc.*, (IEEE Computer Society Press, 1991) 671–677.
10. R. Littlefield, Characterizing and tuning communications performance on the Touchstone Delta and iPSC/860, in *Proc. 1992 Intel User's Group Meeting*, Dallas, TX, Oct. 4–7, 1992.
11. Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, May 5, 1994.
12. Y. Saad and M. H. Shultz, Data communication in parallel architectures, *Parallel Comput.* **11(2)** 1989 131–150.
13. M. Simmen, Comments on broadcast algorithms for two-dimensional grids, *Parallel Comput.* **17(1)** 1991 109–112.