Efficient Broadcast on Computational Grids

Gabriel Mateescu [a] and Ryan Taylor [b]

[a] Information Management Sservices Branch, National Research Council Canada, 100 Sussex Drive, Ottawa ON K1A 0R6, Canada
E-mail: gabriel.mateescu@nrc.gc.ca

[b] School of Computer Scince, Carleton University, 1125 Colonel By Drive, Ottawa, K1S 5B6, Canada
E-mail: rtaylor@scs.carleton.ca

Collective communication operations such as broadcast, gather and reduce are potential performance bottlenecks for scientific computing software. With the advent of wide-area distributed computing and computational Grids, achieving efficient collective communication for applications running on geographically distributed computers becomes of paramount importance. We propose an efficient broadcast algorithm, that is applicable to any connected network and show that it improves the performance of the MPI broadcast operation.

Les opérations de communication inter-processus telles la diffusion (broadcast), la collecte des données (gather), et les opérations mathématiques inter-noeuds (reduce) peuvent aisément limiter la performance des codes de calcul scientifique. L'intérêt croissant que suscite le calcul géographiquement distribué de type grille rend d'autant plus primordial le développement de méthodes collectives de communication efficaces. Nous proposons ici un algorithme de diffusion efficace des données qui peut être utilisé sur tout type de réseau actif. Cet algorithme améliore la performance de l'opération MPI de diffusion.

1 Introduction

Parallel and distributed scientific computing software applications employ various forms of collective communication, such as broadcast, reduce, scatter, and gather. Collective communication is a potential performance bottlenecks, especially when the communication occurs over shared networks with large latencies. For computational Grids, achieving efficient collective communication for applications spanning geographically distributed computers is of paramount importance. We propose an efficient broadcast algorithm, that is applicable to any connected network and show that its performance is competitive with that of the MPICH implementation [1,2] of the Message Passing Interface [3].

We consider a set of networked computer resources, and represent the network as a directed graph $G = (V, E)$, where V is the set of vertices, and E is the set of edges. Vertices represent computer nodes and edges represent the interconnect. With each edge $(u, v) \in E$, we associate the weight $w(u, v) > 0$, which represents the latency of the communication from vertex u to v. A message of size m is to be sent from a designated root to all the other vertices such that: (i) for each edge $e = (u, v) \in E$, it takes the time $\Delta(u) > 0$, called the *injection time* for the node u to inject the message for delivery to v; (ii) a sender vertex can inject only a message at a time, i.e., point-to-point communication. The graph is strongly connected, i.e., for each pair of vertices $u, v \in V$, there is a path from u to v and from v to u.

The problem is to to find the *broadcast schedule* that minimizes the time at which all the vertices have the message, where the broadcast schedule specifies the vertices that each vertex sends messages to, and the order in which the sends are performed. Notice that the problem contains undirected graphs as a special case. The problem can be shown to be NP-complete [4,5]. We present here an approximate solution.

2 Related Work

The published literature on collective communication for wide-area networks [6,7] typically follows the approach of dividing the network in a hierarchy of levels, and define collective operations in terms of collective operation within the levels. Often, a simplified view of the wide are network is assumed, in which point-to-point latencies are equal.

An approach that supports general latencies has been proposed by Mandal, Kennedy and Mellor-Crummey [5]. The authors consider the special case when Δ is a constant, and employ Dijkstra's algorithm together with a broadcast schedule derived from node labeling. Our approach has some common points with the method proposed in [5]. Unlike our method, the authors build the communication tree by applying the original Dijkstra's algorithm. However, simply using Dijkstra's algorithm does not include the effect of the insertion time on the communication topology. This may produce a shortest path three that is far from the optimal communication topology, as shown by the example in Fig. 1, where all the edges have the weight of one.

The single source shortest path tree (left graph, the shortest path edges are marked with arrows) determined assuming $\Delta = 0$ has the longest path of 1. However, if Δ is included, the length of the longest path is actually $1 + 5\Delta$, since all the messages are sent by vertex 0 and there is insertion delay of Δ for each message. The tree shown on the right side (the shortest path edges are marked with arrows) has the longest path $2 + 3\Delta$, which is better than $1 + 5\Delta$, when $\Delta > 0.5$. Moreover, in the labeling stage of the algorithm, [5] uses an expensive method for computing the

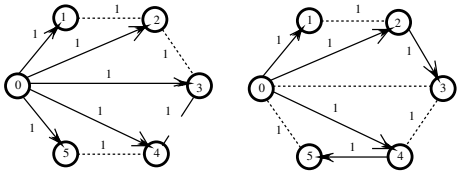

Figure 1. Single Source Shortest Path (left) is not optimal for broadcast. The broadcast schedule defined by the tree on the right side is better for $\Delta > 0.5$.

labels and the schedule, performing a level-ordering of the shortest path tree, followed by two level-order tree traversals.

3 Proposed Method

We propose a two stage algorithm for determining the schedule of the broadcast. In the first stage, a tree T is extracted from the graph G. The tree defines the communication topology such that each vertex receives the message m from its parent in T and sends the message to all its children in T.

In the second stage, we determine the order in which each vertex sends the messages along the edges of the tree T. As discussed earlier, the order affects the time to perform the broadcast because of the cost Δ of inserting a message. The order is performed in terms of a vertex *labeling* which is defined recursively as follows:

$$label(u) = \begin{cases} 0, & \text{if } u \text{ is a leaf in } T \\ \max_{1 \le i \le n} \{label(v_i) + w(u, v_i) + i\Delta(u)\} & \text{otherwise} \end{cases}$$

where n is the number of children of node u in T, $v_1, v_2, \ldots, v_n$ are the children of u, and $label(v_1) + w(u, v_1) \ge label(v_2) + w(u, v_2) \ge \cdots \ge label(v_n) + w(u, v_n)$.

The label of a vertex u gives the maximum time it takes a message sent from u to reach the vertices in the subtree rooted at u. We perform the sends from a vertex u to its children such that u sends the messages in order of decreasing label of the vertices. Choosing this order means that, for a given set of labels of the child vertices of u, the label of u is minimized.

The first stage of the algorithm is an extension of Dijkstra's single source shortest path algorithm, whereby we incorporate in the cost of the paths the cost of inserting the message. Taking into account the insertion time is critical for finding a good communication topology, as illustrated by the example given in Fig. 1. The pseudocode of the algortithm is shown in Fig. 2.

The second stage of the algorithm is implemented using a post-order traversal of the tree to compute the label of the vertices and the schedule, as shown in Fig. 3.

```
extended_single_source_shortest_path{
/*
 Inputs:
     G      = graph representing the
              network of host machines
     V      = number of vertices in G
     E      = number of edges in G
     root   = the vertex from which data
              is broadcast
     w[E]   = array of edge weights
   DELTA[E] = array of insertion delays

 Outputs:
    dist[V] = array holding shortest distance
              from root to every vertex
  parent[V] = array holding the parent of each
              vertex in the shortest path tree
*/

 dist[0:V-1] = infinity;
 dist[root]  = 0;
 parent[root] = null;

 // priority queue
 queue = init_queue(G, dist);

 while( (u=dequeue_min(queue)) != NULL ){
  // compute shortest distance from u
  // to all its neighbors in the queue
  while((v=get_next_neighbor(G,u)) != NULL){
   e = (u,v);   // edge connecting u and v

   if( (v in queue) &&
       (dist[v] > dist[u] + w[e] + DELTA[e])){
     // new minimum found for dist[v]
     dist[v] = dist[u] + w[e] + DELTA[e];
     // v is the child of u, so u incurs an
     // injection time along e = (u,v)
     dist[u] = dist[u] + DELTA[e];
     parent[v] = u;
   } // end if

  } // end while
 } // end while
}
```

Figure 2. Extended Single Source Shortest Path Algorithm

4 Modelling Real Networks

The algorithm works for any directed acyclic graph with positive edge weights and is thus able to handle all real-world networks. The network topology and the latencies (edge weights) are made available to the algorithm through external tools such as the Network Weather Service [8]. Thus, the approach proposed here can be applied to topologies that are dynamic and the global graph can be assem-

```
label_nodes{
/*
Inputs:
     u    = vertex whose label is computed
     T    = the tree determined by
            extended_single_source_shortest_path
     V    = number of vertices in T
     E    = number of edges in G
     w[E] = array of edge weights
 DELTA[E] = array of injection times

Output:
 label[V] = array containing the label of
            each vertex in T
*/

 label[u] = 0;

 // return at bottom of recursion
 if ( u is a leaf in T ) return;

 // get adjacency list of u
 list = adjacency_list(T, u);

 while ( v = get_next_vertex(list) ){
  // visit all children of u
  label_nodes(T, V, v, label);
 }

 // find label of u
 //  sort adjacency list of u by
 //  decreasing label(v) + w(T,u,v)
 sort_decreasing( list, label, w);
 count = 1;
 for ( get_next_vertex(list) ){
  e = (u,v);   // edge connecting u and v
  tmp = label[v] + w[e] + count*DELTA[e];
  if ( tmp > label[u] )  label[u] = tmp;
  count++;
 }
}
```

Figure 3. Vertex Labeling algorithm

bled from querying several information services.

In other words, the graph itself is not static; instead the algorithm uses the graph that provides the most recent model of the network topology. This allows to support real networks where the latency is non-constant and non-uniform.

5 Injection Time Impact

The implementation of the algorithm uses for point-to-point communication the standard send function of MPI [3], called `MPI_Send`. The effect of the injection time on the broadcast time is significant when `MPI_Send` is used. For example, for a message size of 100000 bytes sent between two machines located about 6 miles away, the value of the measured injection time has been $\Delta = 7.3\,ms$, while the value of latency measured was $1.6\,ms$.

The injection time is larger when the function that performs the send operation returns control after some movement of data out of the send buffer occurrs (e.g., for `MPI_Send`) than when the function only initiates the send operation and immediately returns control (e.g., for `MPI_Send` [3], nonblocking *immediate* send). In particular, `MPI_Send` returns control when the the send buffer is free for reuse by the program. In the MPI [3] specification, `MPI_Send` is called a *blocking send* whereas, traditionally, a blocking send is defined as an operation that completes when an acknowledgement (such as "ready to receive message" or "message received") from a matching receive process is received. Nonblocking send functions such as `MPI_Isend` rely on buffering, either system or additional user area buffers. For example, a program using `MPI_Isend` to invoke a large number of send operations and to send large amounts of data relies on the existence of buffering space, either at the sender's or receiver's end, to hold the pending data.

Buffer space is a finite resource; the amount of buffer space depends on several factors including the MPI implelemenation and the amount of memory available on the host machines. Programs relying on the existence of buffer space may deplete the buffers and may run correctly in some conditions but fail sometimes. Hence, these programs are unsafe, unless care is taken to make sure that enough buffer space is available. To achieve safety, the use of nonblocking sends should be combined with some kind of handshaking between the sends and receives, e.g., the sender process should first wait for a *ready to receive* message from the receiver before posting the send.

Therefore, writing correct programs with `MPI_Isend` requires additional synchronization operations beyond waiting for the completion of the posted sends by invoking the functions `MPI_Waitall` or `MPI_Test` [3]. As the number of processors over which an MPI program is distributed increases and the distance between the processors increases (so the communication latency increases), the cost of the additional synchronization increases and the advantage of `MPI_Isend` having a lower injection time than `MPI_Send` may be offset by the additional synchronization operations required to guarantee program safety when using `MPI_Isend` (particularly for processors located hundreds or thousands of miles away). In addition, the smaller the message size, the smaller the difference between the injection time of `MPI_Isend` and that of `MPI_Send`.

6 Evaluation

The first stage of the algorithm has the same complexity as Dijkstra algorithm, i.e., $O(V \log V + E)$, and the second stage has $O(V \log V)$ operations. Thus, the complexity of the algorithm is $O(V \log V + E)$. While the complexity is

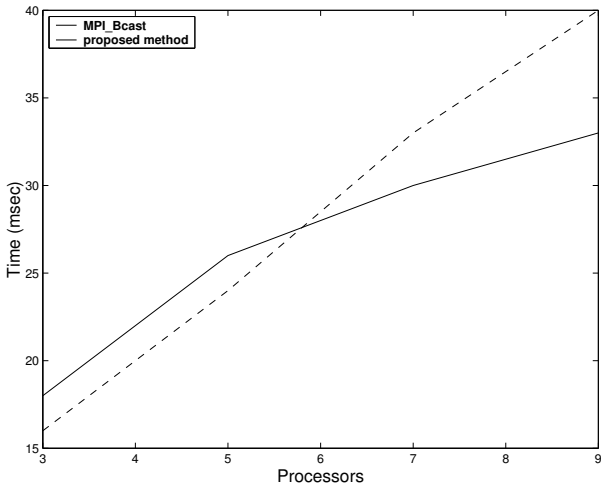

Figure 4. Broadcast time versus the number of processors for the proposed method (continuous line) and for MPICH MPI_Bcast x(dashed line)

a significant characteristics of the algorithm, the algorithm itself is executed only once at the beginning of the application and produces the broadcast schedule. The quality of the algorithm is assessed also by the improvement it brings to the wall-time of the broadcast operation.

We have compared the running time of the broadcast operation implelemted in MPICH [1,2] (the `MPI_Bcast` function) with the time given by the proposed method. In Fig. 4 we show the results on a network of five Intel workstations, four of which are dual processors, and the fifth is single processor. The workstations used in the experiment are located on different local area networks, as follows: two workstations located on a 100 Mb/s local area network, and the other three located on another local area network, with a distance of about 6 miles between the two networks. We have compared against the `MPI_Bcast` implementation of MPICH that uses the ch_p4 device. The message size used has been of 100000 bytes. The results indicate that as the number of processors increases, our method outperforms `MPI_Bcast`.

7 Conclusion

We have introduced an efficient broadcast algorithm that is applicable to any connected network. First experiments indicate that, for a large enough number of processors, it outperforms the broadcast function implemented in MPICH. We plan to conduct more experiments, using larger testbeds and benchmarking real applications. We also intend to generalize the method along three directions: the first is to replace `MPI_Send` with a more efficient method, the second is to take into account the link bandwidth; the third is to use runtime network performance information provided by tools such as the Network Weather Service [8].

References

1. W. Gropp and E. Lusk. A high performance MPI implementation on a shared-memory vector supercomputer. *Parallel Computing*, 22(11):1513–1526, 1997.
2. W. Gropp and E. Lusk. User's guide for MPICH, a portable implementation of MPI. Technical Report ANL 96/6, Argonne National Laboratory, 2002.
3. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. MIT Press, Cambridge, MA, USA, 1999.
4. M. R. Garey and D. R. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, NY, 1979.
5. A. Mandal, K. Kennedy, and J. Mellor-Crummey. An approximate balanacing algorithm for efficient broadcast. http://www.cs.rice.edu/∼anirban/acads, September 2001.
6. T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. *ACM SIGPLAN Notices*, 34(8):131–140, 1999.
7. M. Banikazemi, V. Moorthy, and D. Panda. Efficient collective communication on heterogeneous networks of workstations. In *International Conference on Parallel Processing, Minneapolis, MN*, pages 460–467, 1998.
8. Richard Wolski. Forecasting network performance to support dynamic scheduling using the network weather service. In *6th IEEE Symp. on High Performance Distributed Computing*, pages 316–325, 1997.