Parallel Algorithms using Unreliable Broadcasts

John Matthews*  
UC Davis

Charles Martel†  
UC Davis

Abstract

Parallel computing on networks of workstations (NOWs) allows effective use of existing hardware and software. NOWs are often connected using a broadcast medium such as an Ethernet, so it is natural to use broadcast as a communication primitive. However, existing parallel programs usually rely on point-to-point protocols for performing broadcasts since native broadcast schemes (UDP broadcast) do not guarantee delivery. Thus the potential of the broadcast medium is lost.

Instead of making the broadcast reliable, we design algorithms which use an Unreliable Broadcast. Our algorithms are correct regardless of the reliability of the broadcast, but are more efficient the higher the reliability.

We provide experimental results which show that UDP broadcasts are quite reliable in practice and can be used to design efficient Minimum Spanning Tree and Load Balancing algorithms for NOWs.

1 Introduction

There is a tremendous interest in the design and use of multiprocessor computers. The most common and cost effective parallel system is currently a network of workstations (NOW) since a NOW allows existing hardware and software to be used as a parallel system. Since NOWs typically have higher message costs and lower bandwidth than dedicated parallel computers, it is particularly important on NOWs to limit interprocessor communication and to use the limited bandwidth effectively. There has been a great deal of recent work on parallel algorithms for NOWs in the area of both system support and algorithm design. Our focus in this paper is how to use existing architectural and system features to develop good algorithms for NOWs.

Most current NOWs are connected using Ethernet(s). Even with the emergence of ATM and other new technologies, Ethernet is likely to be in continued use either solely or as subnetworks of an ATM network. The development of faster Ethernet as well as substantial existing investment in Ethernet technology will keep Ethernet networks in use for some time to come.

Ethernet connections limit communication and bandwidth on a NOW since only one message at a time can be transmitted. However, an Ethernet also provides an efficient means to broadcast information to all other nodes on the same segment. Thus exploiting this opportunity can be important in the design of effective parallel algorithms for NOWs. In this paper we will describe some general techniques for using broadcast to design good algorithms, and illustrate this by applying our techniques to classical problems such as finding a Minimum Spanning Tree and Load Balancing.

Standard Ethernet protocols allow a processor to send a message to all other processors in the same segment. However, there is no guarantee that all processors will receive the message. Thus in order to guarantee that everyone gets the broadcast message, the sending processor must get an acknowledgment from all other processors. While in some cases the acknowledgments can be combined with other communication, in general this largely defeats the benefit of the broadcast. But, while there is no guarantee that all processors will receive the broadcast, in practice most processors do get it. Thus we propose to design algorithms using unreliable broadcasts where the algorithm will work correctly even if some processors lose a broadcast message, and the algorithm will be efficient as long as most processors receive the broadcasts. These unreliable broadcasts provide a low-overhead, fast way to distribute information. By using a single transmission to send to many processors, the effective bandwidth of the system is also increased.

In the next section we describe details of implementing the unreliable broadcast, and some experiments which show that most processors receive these broadcasts, but some messages are lost. In sections 3 and 4 we describe algorithms using unreliable broadcasts for Minimum Spanning trees and load balancing which provide good speedups. In each case we give experimental results showing the improvements which result using unreliable broadcast. In section 5 we discuss some additional promising applications.

* Address: Dept. of Computer Science, University of California at Davis, Davis, CA 95616. Email: matthewj@cs.ucdavis.edu. Research supported by NSF grants CCR-94-03651 and CCR-91-03937.
† martel@cs.ucdavis.edu.
2 Broadcast Implementations and Performance

Standard User Datagram Protocols (UDP) generally provide as a feature an unreliable broadcast to single ethernet segments [15]. When a message transmits on an ethernet segment, the workstations on that segment must check the message to see if they are the destination. This occurs at the interface card. A broadcast is a message with a special destination that identifies it as a broadcast message. All workstations will read a broadcast message and deliver it to the device driver.

The UDP protocol does not guarantee delivery and some processors may not receive the broadcast. The major reason observed for losing packets is when the destination is receiving too many packets and is not able process the incoming traffic quickly enough. Thus, we should observe patterns in how messages are dropped.

2.1 Approaches to Broadcasting

Most parallel programming environments for NOWs, such as PVM [9], MPI [13] and Express by Parasoft do not take advantage of the available UDP broadcast. A PVM broadcast sends a single message to each workstation with a destination task along with a return acknowledgment. There are several possible reasons for this. A UDP broadcast has to be handled by every workstation which hears it and this may not be a good way to make friends with co-workers. A solution to this is to use an IP-multicast. Whereas UDP broadcasts do not include any information related to destination, IP-multicast includes a destination group. If a workstation is not in the destination group, the multicast can be discarded before it is sent to the software layer. It can still cause interruptions to workstations not involved in the multicast depending on the type of network card employed. An additional complication is that PVM is designed to handle heterogeneous environments which may support broadcast on some networks but not others.

There are also approaches to implementing reliable broadcasts using unreliable broadcasts. For instance, PCODE [4] is a protocol for use in certain types of parallel message passing programs. They provide reliability without requiring acknowledgments for each broadcast. Instead, they maintain information on the communication history with other processors and use this to decide if broadcasts have been lost. While this works well for applications with predictable message structures, it requires more complex protocols. Also, PCODE does not seem to be suitable for programs which broadcast at unpredictable intervals.

Other researchers have investigated broadcast schemes such as Transis[1] and the Trans algorithm[12], but none use the broadcast medium in the (unreliable) low overhead way we propose. In [7] they use an unreliable broadcast to implement efficient protocols for clock synchronization, membership, and file replication.

2.2 Experiments

We investigated the effectiveness of our unreliable broadcast. Our experiments were done with a modified version of PVM 3.3. One process was running on each machine (PVM daemon) which set up an agreed upon broadcast port. All of the daemons both broadcast and receive on this port using UDP. Broadcast messages were restricted to a single packet of 4096 bytes to avoid handling fragmented messages. No acknowledgments were made when a broadcast was received.

As an experiment, we set up a single workstation which continually broadcasts messages (8 bytes of data) consisting of unique numbers to 24 other workstations on the same 10Mb/s Ethernet segment. Broadcasts were issued as quickly as possible. The only limiting factor was the time required to generate a broadcast which depended on the type of broadcast used. All of the workstations were DECstation 5000/25’s. The machines and the ethernet segment used were relatively quiet. After 500 broadcasts, the process was stopped and we determined which broadcasts were received. We found that the overwhelming number of broadcasts were received. Table 1 gives the percentage of processors which received all broadcasts, 95% - 100%, etc. The majority of workers received all of the messages broadcast. This is not surprising since the major reason for loss is the overflowing of the receivers. Since only one workstation generated the broadcasts at the average rate of 1 broadcast/5ms, it was unable to overwhelm the other workstations. We can conclude that our workstations were able to receive approximately 1 broadcast/1.67ms without loss. This includes the overhead of receiving messages to the PVM daemon and forwarding it to the user process.

When the number of broadcasting workstations was increased, there were more drops overall. For up to three broadcasters, almost all of the workstations still experienced no loss. Instead, a handful of workstations (often just one), accounted for all of the dropped messages. When the number of broadcasters was increased to four, the number of processors receiving all broadcasts dropped dramatically. However, virtually all processors still received at least 75% of the broadcasts.

The time required by our implementation of unreliable broadcast was compared to two reliable alternatives available within PVM. We report the total wall clock time di-
vided by the number of broadcasts. One alternative is a simple loop of single point-to-point sends which uses TCP/IP (pvm_send). The other alternative is the use of PVM’s multicast routine (pvm_mcast), which is not implemented using a UDP broadcast. Instead, a UDP packet is sent to each daemon and an acknowledgment is sent back. One important detail is how soon the task which is broadcasting is able to continue. With our unreliable broadcast, the task continued as soon as the data was sent to it’s local daemon. In the loop of single sends, each send blocks until the message is safely on its way to the destination. For PVM’s multicast routine, the broadcasting task continues once the data is received by it’s local daemon. PVM’s multicast routine was slower at the broadcaster primarily since the local daemon handled the broadcast. This kept the local daemon busier than with the unreliable broadcast scheme and thus slowed down the broadcaster.

We measured the time spent by the broadcaster in generating all of the broadcasts and the total time required for all receivers to read the broadcast (see figure at end of paper). We look at the case of a single broadcaster. In the case of the unreliable broadcast and PVM’s multicast, the broadcaster requires basically a fixed amount of time independent of the number of receivers (see figure at end of paper). As expected, our unreliable broadcast greatly outperforms the reliable PVM broadcast mechanisms.

Given that the broadcast was often 99% reliable, it may be tempting to simply send broadcasts more than once. If the probability of success for each broadcast was independent, this would be reasonable. However, since misses are correlated at single workstations, this will not work. If the broadcasts are separated in time then there will usually be less correlation between misses, and thus is likely to result in higher overall receive rates.

3 Minimum Spanning Tree

We now show that unreliable broadcasts can be used to create a simple, efficient MST algorithm for NOWs.

Prior parallel algorithms include those designed for PRAM models [5] [2] as well as distributed settings [10] [8] [16]. Consider $G = (V, E)$, where $V$ is the set of nodes and $E$ is the set of undirected edges. $|V| = n$ and $|E| = m$ and $w_{i,j}$ is the weight of the edge $i, j$.

Our NOW algorithm has one master processor and $p$ worker processors. The master processor starts up the workers and reports the MST created. The graph is broken into $p$ arbitrary equal size disjoint sets of edges $\{E_1, \ldots , E_p\}$. Let $G_j = (V, E_j)$, the $j$th worker processor has $G_j$ local to it. The master has no information about $G$ on startup.

The master algorithm is similar to Kruskal’s algorithm. The workers help process the edges, so the master may only have to look at a subset of edges, and the workers help the master sort the edges. The master begins constructing the MST as soon as there is an edge from each worker. It may continue as long as there is at least one unused edge from each worker or a worker has processed all of its edges.

The master algorithm examines the edges in the same order as Kruskal’s algorithm. Since each worker has $m$ edges, initially constructing the min-heap at each worker takes $O\left(\frac{m}{p}\right)$ time. Since the graph is assumed to be distributed prior to startup, there is no communication necessary, and thus we obtain linear speedup for this first phase.

The master examines the same number of edges from the min-heap as in the sequential algorithm. The master maintains a min-heap of size $p$ containing the smallest unprocessed edge from each worker. Therefore the work for this second phase is reduced from $O(l \lg m)$ to $O(l \lg p)$, where $l$ represents the actual number of edges that need to be examined for the given input. If $\frac{m}{p} \ll m$, the first phase dominates and we get a good speedup with this simple approach, but often $l \approx m$.

A filtered version of the algorithm allows workers to reject many edges which will not be part of the MST. As the algorithm proceeds, the master constructs a set of connected components $S$. Each time the master algorithm processes the next cheapest edge $\{i, j\}$, it rejects $\{i, j\}$ if $v_i$ and $v_j$ are in the same component in $S$. The master can broadcast information about $S$ incrementally to the workers and let the workers reject edges which fail the test. This can greatly reduce the number of edges sent, resulting in less communication and less work for the master. However, this increases the communication from the master since the MST is sent to workers as it is being formed.

Edges which are discarded by the workers would have been discarded by the master so the filtered version is also correct.

Using a broadcast, the component information is disseminated quickly and efficiently. The information sent is edges as they are added to the MST. Thus there need be at most $n - 1$ messages broadcast. If a broadcast message is not received, the worker will possibly send more edges than necessary to the master. These extra edges will still be discarded by the master.

The structure of $G$ plays an important role in the performance of MST algorithms. For purposes of testing the algorithms presented here, $G$ will be a clustered random graph. It will be split into clusters with light weights used for intra-cluster edges and heavy weights used for inter-cluster edges. For instance, for a graph with $c$ clusters, it will be generated by first separating the nodes into $c$ disjoint sets $(V_1, V_2, \ldots , V_c)$. Intra-cluster edges are created with probability $p_1$ and weights form a random permutation of these edges. Inter-cluster edges are created with proba-
This speedup compares favorably to previous studies [6] [8] using the filtered algorithm and 16 processors. We believe this is due to the increased memory available using multi-processors. Note that we get a slight super linear speedup in most cases where the number of clusters in the graph, and the weights form another random permutation. All intra-cluster edges are smaller than inter-cluster edges. We have chosen this class of graphs both because it provides a more general setting than classical random graphs (where \( c = 1 \)) and because it models several natural applications of MST where nodes form clusters. The initial distribution of edges is arbitrary, so clusters are spread across processors.

Below we summarize the effect of incorporating the broadcast into this algorithm. Since there is a single process generating broadcasts, virtually all broadcast messages were received. In these tables, \( p \) is the number of workers, \( n \) is the number of nodes, \( m \) is the number of edges, \( c \) is the number of clusters in the graph, and \( k \) is the number of edges sent in a single message from the workers to the master. Speedup is defined as \( \frac{\text{sequential time}}{\text{distributed time}} \) where the sequential time was computed on the same workstation used for the master using Kruskal’s algorithm which is one of the best sequential algorithms for this type of graph[14]. The sequential version did not have any overhead due to PVM. Note that we get a slight super linear speedup in most cases using the filtered algorithm and 16 processors. We believe this is due to the increased memory available using multiple processors which allows better locality of data access. This speedup compares favorably to previous studies [6] [8] which obtained speedups of 3-4 using 8-9 processors.

4 Load Balancing

The MST algorithm used a single broadcaster so very few broadcasts were dropped. We now consider a setting with many broadcasters and show that we still get good performance. The load balancing technique presented here has been used successfully in real applications.

We want to complete a set of \( n \) equal length tasks which are part of a parallel computation. Each task is independent of all others and no intertask communication is required. It is not harmful if a task is performed more than once. For the computation to finish, at least one processor must know that all the tasks completed. The set of tasks may be an entire computation or one phase of a computation. This models problems such as evaluating a function at many points, phases of a grid computation and graphics evaluations.

The time required to compute a single task is \( C_t \). For communication, we model only the overhead of sending or receiving a single message, \( C_m \). For broadcasts, the cost of sending is still \( C_m \) and each receiver must pay \( C_m \) also. When a processor sends a message, it does not wait for a reply, but continues immediately. A receive may be either blocking or non-blocking.

We assume that different processors may be working on these tasks at widely variable rates of speed over the course of the program. This could be due to variations in processor power or load. Thus, the total time required may vary greatly. We concentrate on determining the total amount of work required. We break work down into three categories: Useful work is work performed in the successful completion of the tasks. Redundant work occurs when the same task is performed by more than one processor. For instance, if two processors complete the same task, one processor may count its work as useful, but the other is considered redundant. Communication work is any work used either in supporting the protocol or in sending/receiving messages.

4.1 Load Balancing Algorithm

We assume that all workers know the entire set of tasks that need to be completed. In many different applications, this can be done without requiring communication comparable to the number of tasks. We now describe our load balancing algorithm.

Once a worker knows all the tasks, it begins selecting tasks at random to work on.

When a worker completes a group of \( q \) tasks, it broadcasts a message indicating which task group it just completed. It then handles any incoming broadcasts and updates its task list. A worker then selects a task group at random from its set of uncompleted tasks. It will work on this task group till completion. Without this grouping, if we have \( n \) tasks, each worker will have to maintain this status information for each of the \( n \) tasks and will have to perform some form of book keeping on each one. This would require \( O(n) \) time. This is not desirable, therefore, we perform this grouping to reduce this to \( O(\frac{n}{q}) \). Combining tasks into groups also allows us to balance the amount of computation done between broadcasts.

No worker will ever stall waiting for any messages. Therefore, this scheme can tolerate all but one worker failing.

This randomized scheme works correctly even if no
broadcasts are successful since work can be duplicated safely. It is inefficient if the broadcasts do not work well. Generally, the broadcasts were reliable, but if the task size was small enough, workers broadcast frequently and the reliability of broadcasts decreases. Therefore, there is a practical lower limit on the time spent computing a single task or group of $q$ tasks.

4.2 Performance of Load Balancing Algorithm

The amount of useful work is simply $n C_t$. For the idealized setting where each processor learns of a completed task group as soon as it is completed, we can use the result in [11] which shows that the amount of redundant work is expected to be $O(pq \log^2 p C_t)$ where $p$ is the number of processors. For each completed task group, there will be a broadcast and everyone is assumed to receive it. We have $\frac{n}{q}$ useful task group completions and expected $O(\frac{pq \log^2 p C_t}{q})$ redundant task group completions. Therefore, the expected communication work is $O(\frac{2}{q} + p \log^2 p) p C_m$.

In experiments, we implemented the broadcast using our unreliable version and PVMs multicast routine ($pvm_mcast$). We show the speedup for both protocols in a figure at the end of paper.

Using reliable protocols like point-to-point creates a dependency between processors. A slow processor is then able to slow others down. This also has an effect on the amount of redundant work if the broadcasting process is able to continue before the broadcasts repeat. The unreliable broadcast results on duplicate work are in agreement with the predicted amount even with a slow processor. The results in Figures ? show that unreliable broadcast is better than point-to-point even when all processors are contributing at roughly equal rates. When there is a large variation among processors, the unreliable broadcast is much better.

5 Other Applications

Using unreliable broadcasts to quickly and cheaply convey information can be useful for many other parallel applications. We give a brief overview of two applications we have begun investigating.

In work stealing approaches to load balancing for multi-threaded computation [3], idle processors steal threads from others with excess threads. This works well using randomized solutions, but in some settings broadcasts from processors with heavy loads can improve performance.

Broadcasting can also be useful for parallel branch and bound algorithms. In best-first search implementations, the goal is to always expand the nodes of lowest weight. In a parallel setting, the difficulty is in identifying and distributing the cheapest nodes. For a parallel branch and bound algorithm to be effective, several types of global state information is important: best solution found to date, cheapest unexpanded nodes’ values and their location. Unreliable broadcasts can be helpful for disseminating this information. Broadcasts are adaptable to blackboard like schemes, as well as work stealing [3].

6 Conclusions

We have shown that unreliable broadcasts can be used effectively to design algorithms for NOWs connected by a broadcast medium.

Further work is suggested on several fronts. The first is to consider more applications, and in particular to extend the preliminary work we have done on Branch-and-Bound and for Work-stealing. Another area to consider is further experiments on settings with multiple segments. Our work also suggests that better support for broadcast and multicasts is helpful for making NOWs more effective. Currently a UDP broadcast disturbs all nodes on the segment. Better hardware support to allow multicasts which are only processed by participating nodes makes the use of broadcast more generally attractive. Similarly, better support to make broadcasts more reliable would make the use of unreliable broadcasts more effective.

References


Number of Receivers

Average Time per Broadcast (ms)

- unreliable
- pvm_sends
- pvm_mcast
Number of Workers

Speedup

unreliable

pvm_mcast