Automatic Tuning of Collective Communication Operations in MPI

Rajesh Nishtala, Neil Patel, Kaushal Sanghavi, Kushal Chakrabarti
December 2003: CS262A Final Project
Computer Science Division
University of California, Berkeley
rajeshn@cs.berkeley.edu {neilp,kaushal,kushalc}@uclink.berkeley.edu

Abstract

In this paper we present an adaptive approach to tuning MPI collective communications algorithms. The approach was arrived at in two separate steps. In the first, we observed the standard vendor implementation of several collective communications operations to be naive and inflexible. To make these operations faster and more efficient we developed a family of algorithms for each of four collective operations that often showed impressive improvement over the standard implementations. While observing that some of these new algorithms performed better, we also noticed that their level of performance changed relative to each other over time. These changes persisted when we varied a number of factors affecting the context of the operation, including the number of processes over which to perform the operation, the size of the data to be communicated, and the physical cluster on which the operation was run. These observations led us to believe that the best approach to optimizing collective communication operations is to dynamically choose best-performing algorithms based on empirical results on recent performance. In the second step, we developed a lottery scheduler that would manage these results and probabilistically choose a globally optimal algorithm. We observed that with a scheduler, a long-running application would choose algorithm implementations whose performance was near the optimum performance.

1 Introduction

Over the past decade, computer systems have gotten significantly faster and more powerful. One of the consequences of this rapid technological advancement, however, is that the complexity of modern systems has increased dramatically. Although most system administrators, developers, and researchers possess sufficient knowledge of computer science to be able to fully exploit the power of these machines, there are many problems to restricting their use to such individuals. For instance, distributed systems are widely used by scientists from fields as diverse as physics, statistics, and chemistry. However, these users cannot be expected to manually tune either their applications or the underlying systems to fully exploit the available computational power. At the same time, however, it is important to tune these applications because the tuned versions can result in significant performance improvement (up to 800% in our work). [5]

In practice, distributed systems and applications are often manually and tediously tuned by professional system administrators. This approach is unfortunate because such tuning is not only very expensive but is often outpaced by the rate of technological innovation. At the same time, it is interesting to examine the notion of optimization: systems are not designed to be optimal for every possible application; indeed, they cannot be. All these problems motivate the need for a system that can automatically tune these applications based on run-time parameters.

Here, we focus on the automatic optimization of collective communication operations – the transfer of data across many processes – on distributed memory computing clusters. The complex architecture of these systems, which are characterized by the presence of a high bandwidth, low latency interconnection that networks together many heterogeneous machines, creates significant opportunity for optimization. For instance, as one can see in Table 1, different clusters are associated with different processor speeds, physical memory sizes, and network topologies, all of which can be used to produce specifically tuned implementations. Even more interestingly, many hardware vendors implement their own versions of point-to-point communication software which creates the possibility of optimizing over another dimension in the tuning space.

The Message Passing Interface (MPI) is a commonly used library for inter-process communication on these systems[7]. We therefore concentrate on its optimization. In fact, most scientific computing applications use the collective communications implemented by MPI for bulk data transfers and distribution of data across

different nodes for processing. For instance, parallel scientific applications that perform matrix multiplication could use the `scatter()` function in MPI to distribute submatrices to different machines on the cluster, and recollect them with `gather()` after completion of processing.

However, the presence of such a large parameter space precludes the possibility of manual tuning and motivates the need for automatic tuning. The choice of the optimal implementation of the algorithm varies across the topology of the cluster, the number of processes in the operation, and the size of the message that we wish to transfer. In this paper we analyze four common collective communications operations: broadcast, scatter, gather, and reduce. For each of these operations we have implemented a family of implementations. These implementations vary the tree structure used to disseminate the data as well as the minimum unit of transfer (heretofore called *segment size*). Because the optimal choice of implementation is based on many different run time parameters such as network load and layout of the processes within the network, we present a mechanism that will dynamically chose the correct algorithm based on the lottery scheduling mechanism [11].

The structure of this report is as follows: in Section 2, we further examine the intricacies of MPI and operations on which we worked. In Section 3, we discuss the variations on the different implementations of the operations. We then present our experimental methodology in Section 4 and an initial evaluation of the data in Section 5 and give the motivations for a dynamic choice of algorithm. In Section 6, we discuss the implementation of a lottery scheduler and present its results. We conclude with related and future work in Section 7.

2 Relevant Message Passing Interface (MPI) Functions

The Message Passing Interface (MPI) is used for high-performance clustered computing. Particularly popular communications using MPI involve the transfer of information between one process (the *root* process in the MPI context) and every other process in its *communication group*. These transfers, henceforth referred to as *collective communication operations*, are described below.

2.1 Broadcast

`Broadcast()` is intuitively network broadcast. The root process sends the same message to all the other processes in the communication group. It is defined as

```
BROADCAST (buffer, count, datatype, root, comm):
          IN/OUT   buffer     starting address of buffer
          IN       count      number of entries in buffer
          IN       datatype   data type of entries in buffer
          IN       root       rank of broadcast root
          IN       comm       communication group
```

For the purposes of this paper, we say that `broadcast()` is an *unspecialized* operation because the data received by each node is not specific (or specialized) to it. The spirit of this definition is that each process, after receiving the appropriate amount of data, need only transmit one piece of data to its receiver.

2.2 Reduce

`Reduce()` is structurally similar to `Broadcast()`, but is, in fact, its inverse. Here, every process sends data to the root process – instead of the root transmitting to each process. An important, difference, however, is that `reduce()` also takes in an aggregation operation that combines the data received from each of the processes into a single data set.

```
REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm)
       OUT      recvbuf     address of receive buffer; significant only at root
       IN       sendbuf     address of send buffer
       IN       count       number of elements in send buffer
       IN       datatype    data type of elements of send buffer
       IN       op          aggregation operation handle
       IN       root        rank of root process
       IN       comm        communication group
```

An important feature of these aggregation operation is that they are *global*. In other words, these operations can be performed on all the data sent by all the processes in the communication group. For instance, the MPI interface supplies default implementations of `sum`, `min`, and `max`. These operations are commutative and associative, since the order in which the root receives data from the processes is not defined.

Because of these constraints, the `reduce()` operation can considered unspecialized. Here, the operation corresponds to the spirit of the definition of unspecialization because each process, upon receipt of its senders' data, need only forward a single piece of data to the corresponding recipient. This aggregation can, in fact, be performed in arbitrary sequence because of the constraint that the aggregation function be associative and commutative.

2.3 Scatter

`Scatter()` is the operation where the root needs to send different sets of data to all processes in its communication group. Hence, its `sendbuf` is broken up into different sets of data, and sendbuf is the starting point for the data that the root needs to send to process with rank `i`.

```
SCATTER(sendbuf,  sendcount,  sendtype,  recvbuf, recvcount, recvtype,  root,  comm)
        IN   sendbuf     address of send buffer;  significant only at root
        IN   sendcount   number of elements sent to each process; ...
        IN   sendtype    data type of send buffer elements; ...
        OUT  recvbuf     address of receive buffer
        IN   recvcount   number of elements in receive buffer
        IN   recvtype    data type of receive buffer elements
        IN   root        rank of sending process
        IN   comm        communication group
```

For the purposes of this paper again, we consider `scatter()` to be a *specialized* operation because each process receives a piece of data specific to it. When compared to unspecialization, the spirit of specialization is essentially that nodes must transmit data specific to each recipient.

An example of the usage of `scatter()` has been provided in Section 1.

2.4 Gather

`Gather` is the exact inverse operation of `Scatter`. The root collects a different piece of information from each of the processes in the communication group. Gather is defined as:

```
GATHER (sendbuf,  sendcount,  sendtype,  recvbuf,  recvcount, recvtype,  root,  comm)}
        IN   sendbuf     starting address of send buffer
        IN   sendcount   number of elements in send buffer
        IN   sendtype    data type of send buffer elements
        OUT  recvbuf     address of receive buffer choice; significant only at root
        IN   recvcount   number of elements in any single receive; ...
        IN   recvtype    data type of recv buffer elements; ...
        IN   root        rank of receiving process
        IN   comm        communication group
```

We state, without further elaboration, that `gather()` is a specialized operation by analogy to the previously mentioned operations.

3 Static Optimizations: Trees & Pipelining

For each of the operations described in Section 2 we have created a family of implementations. These implementations vary on the tree structure used to disseminate the data, along with the segment size. In this section, we first discuss the structure and ramifications of the various trees, and continue onto describing the characteristics and importance of pipelining data through the tree structure.

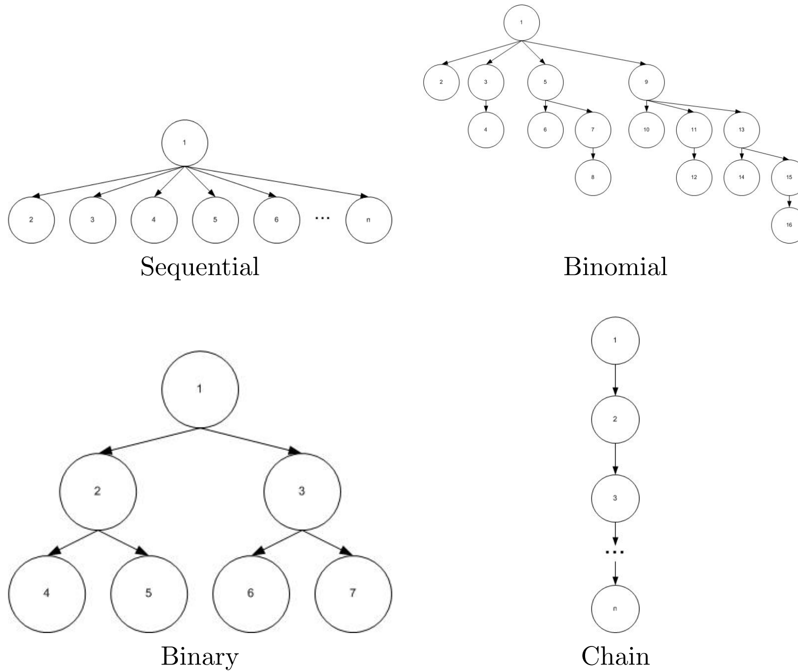

Figure 1: The different trees used to disseminate the data.

3.1 Tree Structures

In order to improve upon the existing vendor-provided implementation, we have implemented each of the previously mentioned collective communication operations on four different tree structures. We expect that, for certain operations, the trees will allow better parallelization of both processing and network bandwidth usage. These four different tree structures are described (as shown as directed graphs in Figure 1) below. In each of these trees, if the root process is transmitting data, every process sends to its children in the corresponding tree. On the other hand, if the root process is collecting data, every process sends to its parents in the tree. For simplicity, the following discussion focuses on the former case – the latter case follows straightforwardly.

Binary The standard binary tree is important for a number of reasons. First, it allows meaningful parallelization of processing and network bandwidth usage, while enforcing that no process incurs the cost of sending to more than two other processes. This network bandwidth parallelization is important because, at each time step, an increasing number of processes can use their network links to send data. Second, it limits the length of longest of chain of consecutive sends to $O(\lg N)$, where N is the number of processes in the communication group. Other scaling characteristics of this tree is similar to binomial, and is discussed below. No known standard MPI implementation uses the binary tree.[1]

Binomial The binomial tree extends the parallelization seen in the binary tree by (1) allowing a process to send to an increased number of nodes, and (2) creating a natural order in which these sends can take place. With regard to the former, a process could be required to send to or receive from up to other $O(\lg N)$ processes. This is particularly meaningful for large communication groups, where a larger number of processes can begin to simultaneously use available network bandwidth (relative to binary). At the same time, if the process sends to the child with the greatest number of descendants (see Figure 1) with blocking sends, it can be shown that every process receives the data at the same time. The binomial tree requires that a process receive data for all of its descendant processes in the tree, like binary. Here, however, there is no straightforward expression for the amount of data that every node receives – in the worst case, however, a node might receive exponentially more data than it needs. As before, no additional data is sent in the special cases of `broadcast()` and `reduce()`. The standard MPICH implementation uses the binomial tree for the `broadcast()` and `reduce()` operations.

[1] This observation is conditioned upon our inability to access the proprietary MPI implementation on the IBM Seaborg cluster. This condition holds for the remaining trees.

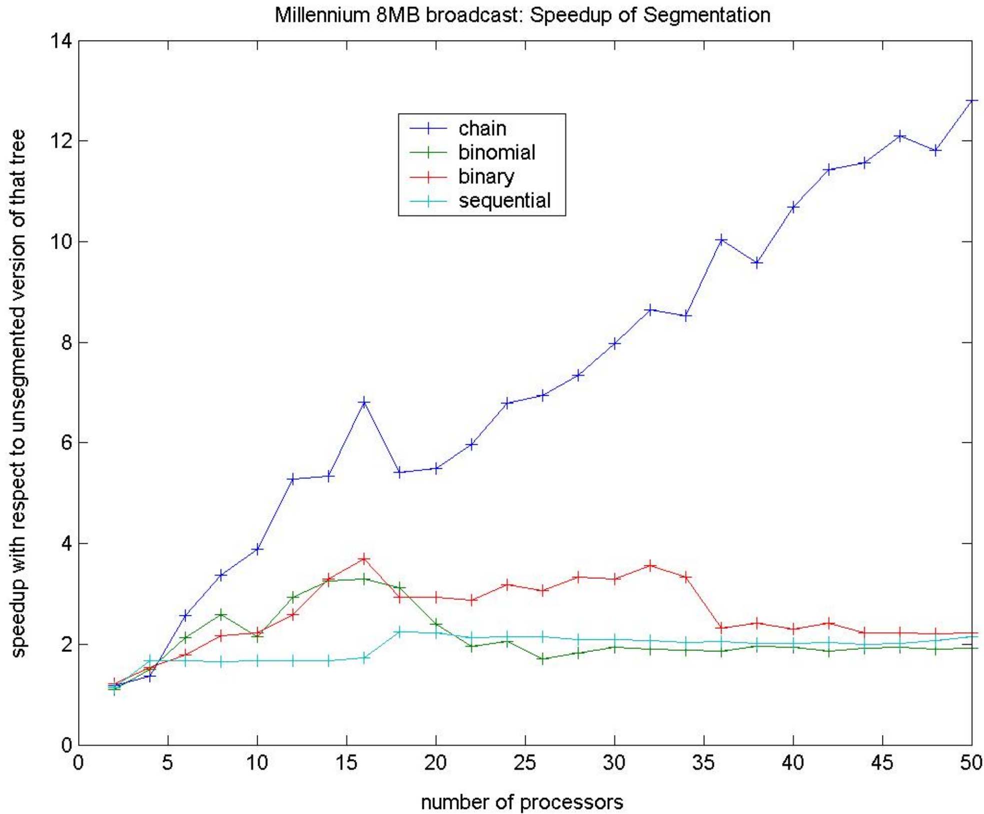

Figure 2: **Effect of Segmenting**. This plot shows the effect of segmenting on the Millennium Cluster. Each of the lines shows the speedup of the best segment size for each tree versus the unsegmented implementation of that tree.

Chain (Linear) The straightforward chain tree causes every process to send to the process that has the next highest identifier. In this case, every process first receives the data for every process with an identifier greater than or equal to its own. Given this data, it cleaves the data intended for it, and forwards the remaining data to the appropriate process. In the special cases of `broadcast()` and `reduce()`, the chain tree does not induce any increased overhead in terms of network bandwidth. In the general case, however, it requires the transmission of $O(NM)$ data, where M is the size of each message. No known standard implementation uses the chain tree.

Sequential The sequential tree is the naive tree in which the root process directly transmits data to every other process in its communication group. There are no parallelization benefits from this tree that are apparent to the authors of this report. In fact, this tree seems to enforce that the root transmit independent streams of data to each of its children in a way that precludes parallelization of both network bandwidth and processing. The standard MPICH implementation of `scatter()` and `gather()` uses the sequential tree.

3.2 Pipelining

The observation that trees allow parallelization can be further leveraged by the use of pipelining. Such pipelining involves the (1) segmenting of messages and (2) the simultaneous non-blocking transmission and receipt of data. Messages can be segmented by breaking up the larger message into smaller segments and sending these smaller messages through the network. The main advantage of segmenting is that it allows the receiver to begin forwarding a segment while receiving another segment.

Data pipelining produces a number of significant improvements. First, pipelining masks the processor and

network latencies that are known to be an important bottleneck in high-bandwidth networks, such as those found there. Second, because it allows the simultaneous transmission and receipt of data, pipelining exploits the full duplex nature of the interconnect links. Third, because these links are known to support very high throughput, they could in fact support the simultaneous transmission of data to multiple children, thereby decreasing the total time of transmission.

The pipeline for `broadcast()` and `reduce()` is very straightforward because the aggregated data is not specific to individual processes. We can thus leverage the parallelism of having processes receive one segment of data and resend multiple copies of that same segment. Thus, network bandwidth can be readily parallelized. For `reduce()`, even processing can be parallelized within the network.

However in this model it is very difficult to pipeline the scatter and gather operations because every messages are not generic and must be routed properly through the network. Thus if we use anything besides a sequential tree to disseminate the information, there will be additional network traffic and unnecessary transmission in the network. With respect to our tree implementations, intermediary nodes act as "packet forwarders" that route the data to their children – it is this procedure that is pipelined. We believe that, despite the dramatic increase of data on the network, pipelining on these operations still allows meaningful optimization through a greater number of processes simultaneously receiving and sending data. For instance, if there is sufficient bandwidth in the network, the transmission of additional data can occur without significantly increased cost, while still allowing the masking of network and processor latencies. In fact, In Section 5 we will show that performance increases are indeed observed in practice. Figure 3.2 shows the effect of this segmenting

4 Experimental Methodology

We have developed a variety of performance profilers to evaluate the performance of our implementations. These profilers, in particular, measure the performance of one of the collective communication operations across a range of segment sizes, message sizes, and communication group sizes. The ranges for these variables were as follows: message sizes ranged from 1KB to 1MB for scatter and gather, increasing by factor of 4. For broadcast and reduce, the upper limit was extended to 8MB. Segment sizes ranged from 1KB to the size of the message, increasing by a factor of 2. The size of the communication group ranged from 2 to 32, 50, and 64 for CITRIS, Millennium, and Seaborg, respectively, increasing by 2.

Performance was measured in terms of median running time on each of the clusters. These times were measured on Millennium and Seaborg with standard MPI wall time clock interface, whereas times on CITRIS were measured with PAPI [1]. We were required to implement the PAPI measurement on CITRIS because the resolution of standard MPI wall clock implementation is approximately 4 milliseconds – far too inaccurate for our measurements. Although the former measures total elapsed time between operation initiation and termination, and the latter measures processor ticks during actual execution, ie. excluding time spent outside of a context switch, we believe this difference to be negligible because we never directly compare absolute times across clusters. Each operation was executed for each parameter set ten times to account for experimental error. The data shown in the following graphs display the median run times of these runs. It is important to note, however, that the CITRIS and Millennium clusters do not employ load balancing and prevent the user from declaring acceptable load levels. For these reasons, experiments on these two clusters are not as repeatable as those on the Seaborg cluster. Table 1 shows a summary of the clusters that were used in our experiments.

5 Initial Data Analysis

Although our profilers supported analysis of operations across a number of different parameters, the entire parameter space was explored only for broadcast profiling. For `scatter()`, `reduce()`, `gather()`, we only analyzed performance across the four trees, segment sizes, and communication group sizes – message sizes and physical clusters were not varied. We did this for several reasons. First, the availability of computational resources were limited on the shared clusters and forced us to be selective in gathering data. Second, we observed from early trials that broadcast exhibited interesting variation across the entire range of parameters. Finally, analysis on broadcast alone simplified our dataset while still providing suitable evidence that

	Millennium [2]	CITRIS [2]	Seaborg (dense) [3]	Seaborg (sparse) [3]
Processor Type	Pentium II Xeon	Itanium 2	IBM Power3	IBM Power3
Processor Clock Rate	500-700 MHz	900 MHz - 1.3 GHz	375 MHz	375 MHz
Processors per Node	2-4	2	16	1
Physical Memory	2-5 GB	4-5 GB	16-64 GB	16-64 GB
Network Topology	Star (ie. symmetric links)	Star	Two Level	Star
Interconnect Type	TCP/IP Gigabit Ethernet	TCP/IP Gigabit Ethernet	CSS	CSS
Nodes in Network	50	32	4	64

Table 1: **Cluster Summary**. This table shows a summary of the pertinent facts about the clusters used in our experiments. Note that there are two different versions of the Seaborg cluster here. The Seaborg (dense) cluster is 64 processors across 4 nodes were used, while Seaborg (sparse) indicates 64 processors across 64 nodes. Since the interconnects within the processor are presumably faster than the network we say that Seaborg (dense) is in essence a two level cluster. The first level is all the processors within a node while the second level is the interconnect of all the nodes.

variation in cluster environment exists.

Our data is organized as follows: Section 5.1 shows and analyzes the performance of each of the collective communications operations with each tree implementation ranging across the number of processes, all on the CITRIS cluster. Section 5.2 describes similar data for the broadcast operation across four different clusters (CITRIS, Millennium, Seaborg (sparse), and Seaborg (dense)). Section 5.3 examines the performance of segmentation as a function of communication group size.

5.1 Data Analysis Across Operations

We observed the performance of each of the four collective communications operations (broadcast, gather, scatter, gather) on the CITRIS cluster with each of the four tree implementations. Plots of these performance measurements are shown in Figure 3.

There are substantial performance gains relative to the vendor implementation for `broadcast()`, `reduce()`, and `gather()`; however, our various implementations of `scatter()` exhibited no improvements from the standard. Nevertheless, it is interesting to note that in every operation – including `scatter()` – there is an implementation that performs as well as, if not better than, the vendor implementation. For instance, the chain `reduce()` implementation scales independently of the number of processors, whereas the standard MPICH binomial `reduce()` implementation scales logarithmically. Similarly, we see that even for `scatter()`, the chain and binomial implementations perform just as well as the standard MPICH sequential implementation.

In general, we see that specialized operations scale linearly with the number of processors, whereas sensible implementations of unspecialized operations scale logarithmically or independently of the number of processors. The additional constant cost for specialized sends is incurred because adding an additional node simply means sending up (or down) one additional level of the tree. With a pipelined implementation this is constant cost per process. Extra unspecialized sends add little or no cost because including an additional node to the operation involves adding a single send or receive made in parallel with the original sends and receives. The sequential tree, moreover, incurs the cost because this parallelism is not present.

The similar performance of different implementations of particular collective communication operations is significant because transient network conditions could cause one implementation to suddenly perform significantly better than its comparable implementation. This is particularly relevant for cases where many comparable implementations are, in fact, the best implementations: for instance, a chain `scatter()` could be adversely affected by network traffic at node 2, in which case a comparable binomial implementation could significantly outperform it (see Figure 3). This observation is critical to the motivation behind lottery scheduling and is discussed further in Section 6.1.2.

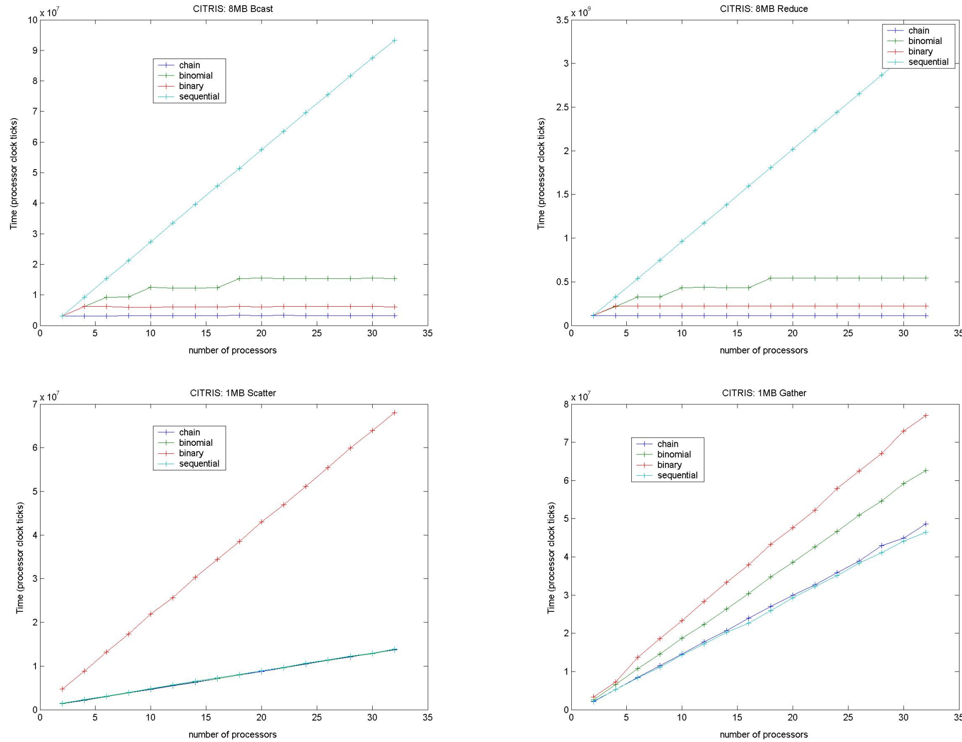

Figure 3: **Varying the Operation on CITRIS**. These plots show the differences across operations on the CITRIS cluster. For a given number of processors the graph shows the time taken for the best segment size for each of the given trees.

5.2 Data Analysis Across Clusters

The broadcast operation was run on four different clusters. Figure 4 shows the performance of the four tree implementations on each of these clusters. On the CITRIS and Millennium clusters, the binary tree and chain tree implementations perform the best, and are comparable with respect to each other in the time taken to complete the operation. Nevertheless for the Millennium cluster, binary tree would be preferred for broadcasting on a relatively small communication group (between 2 and 32 processes), while chain is always preferred over binary on CITRIS. This suggests that small changes in the network environment could result in one tree structure being better than an another for a period of time. Thus we need a way to automatically choose the optimum MPI operation based on network and processor parameters.

Similarly, we see that the binary tree implementation is the best performer on Sparse Seaborg but does relatively poorly on the Dense Seaborg. Additionally the cost of additional processes on the Sparse cluster for all algorithms but sequential tree is close to zero. This indicates that we must strive to achieve a way to make the same implementation of MPI operations use the most efficient algorithm, regardless of which type of architecture it is installed on. This is in stark contrast to the current solution, which involves manually fine-tuning the implementation before installing it on a system.

One of the important observations is that the different trees perform better across different clusters. For example on CITRIS, the optimal chain implementations dominate the other implementations while in the other clusters the other trees take comparable times as chain. A speculation to this observation is that the CITRIS cluster is a bandwidth limited cluster while the others are not[2]. The observation is that the higher fanout factor of a tree the more the send operations that get queued at some level of the software stack.

[2]The reason that we believe CITRIS is bandwidth limited is the processors used in the network are significantly faster than the processors used in the other clusters. The rate at which the Itanium2 processors can ship data to the network cards is a lot higher than the rate at which the network cards can put the data on the link, therefore the processors are not the limiting factor. However on the other processors, this is not the case, implying that the processors themselves are the bottleneck and the network cards can send data at the same rate at which the processor can feed the network card.

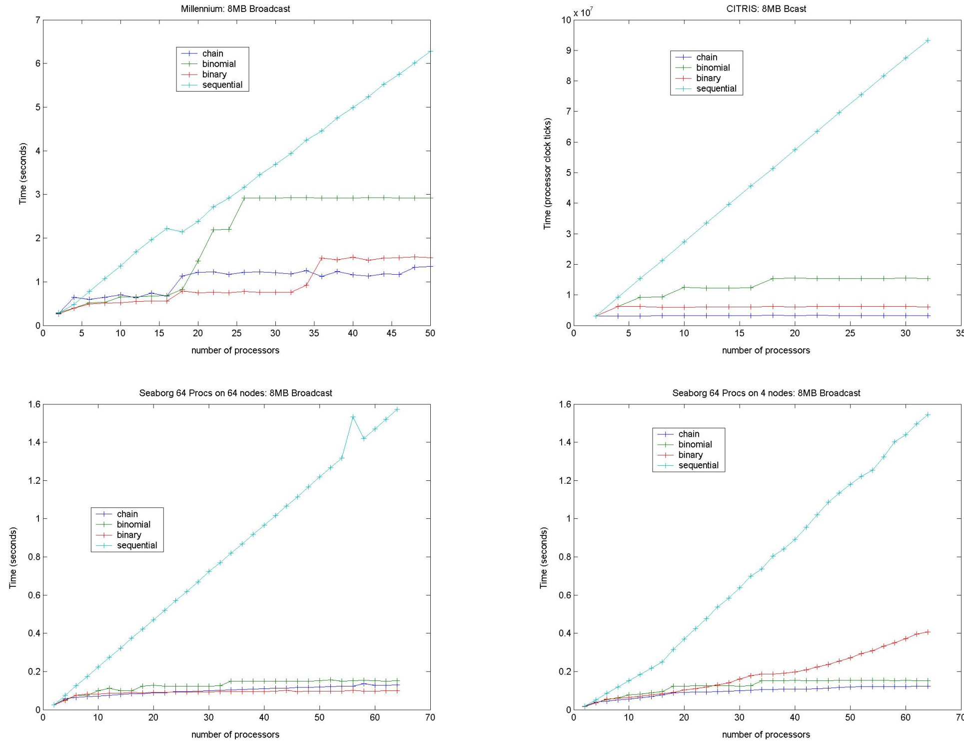

Figure 4: **Varying the Cluster on a Broadcast**. These plots show the differences across clusters for a broadcast operation. For a given number of processors the graph shows the time taken for the best segment size for each of the given trees.

Thus on bandwidth limited networks the length of this send queue could be an an important factor, implying that chain has the best performance since it has the shortest queues at every processor. However when the network bandwidth is not a limiting factor the time spent in the send queues for the various segments is a negligible effect, implying that the parallelism that the trees can provide can be leveraged. This would explain why the chain trees dominate on CITRIS but not the other processors.

5.3 Across Parameters

Pipelining the tree algorithms is most effective when it is done with a segment size that is agreeable with the cluster's underlying architecture. We see from Figure 5 that clusters respond in unique ways to segment sizes. On Millennium, segmenting is crucial for the broadcast on an 8MB message, where a 16KB segment performed the best for all four tree implementations. As Millennium is a processor-constrained cluster, sending with smaller segment sizes is probably necessary to utilize link bandwidth and to mask to latency created by slow processors. Figure 6 shows the effect of segmenting on the Millennium cluster. The figure shows that each of the different trees has different valleys which imply an optimal segment size. On the other hand the same operation on CITRIS did not rely on fine-grain segmenting for best performance, as a 2MB segment size was optimal for all trees. On smaller message sizes, the optimal segment was observed to be half the size of the full message. Finally, we see that on Seaborg (Sparse and Dense), no single segment size was settled on as a best size across all comm group sizes. This variation in behavior with respect to segment size among the four clusters is initially surprising given that all four saw the same implementation of broadcast, chain tree, perform best. This means that even if an algorithm implementation emerges as the universal best performer, there are cluster-specific parameters that must be tuned to ensure the optimal performance. Thus an adaptive approach to determining the best algorithm for a given cluster would be ideal. If we could dynamically determine the best algorithm (and parameters) given the operation to execute, commgroup size, and message size, then we could avoid the repetitive re-implementation that is prevalent in current vendor-implemented systems to deliver optimal performance on physically unique clusters. An empirical approach to choosing the optimal would be preferred to a modelling approach, as models cannot

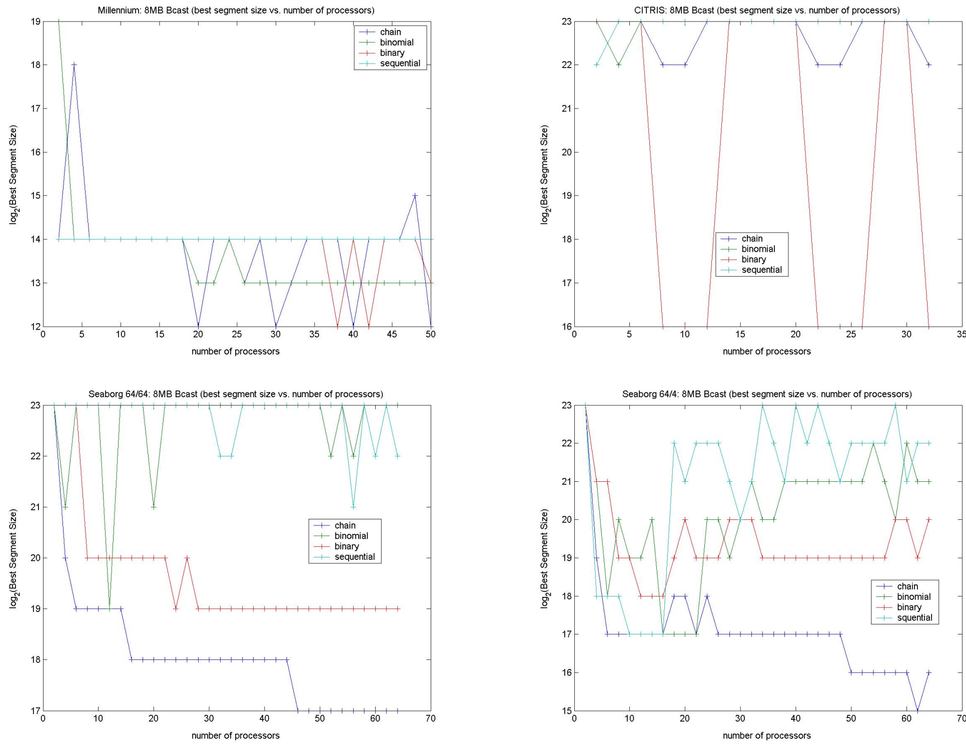

Figure 5: **Varying the Cluster on a Broadcast**. These plots show the differences across clusters for a broadcast operation. For a given number of processors the graph shows the best segment size for each of the given trees.

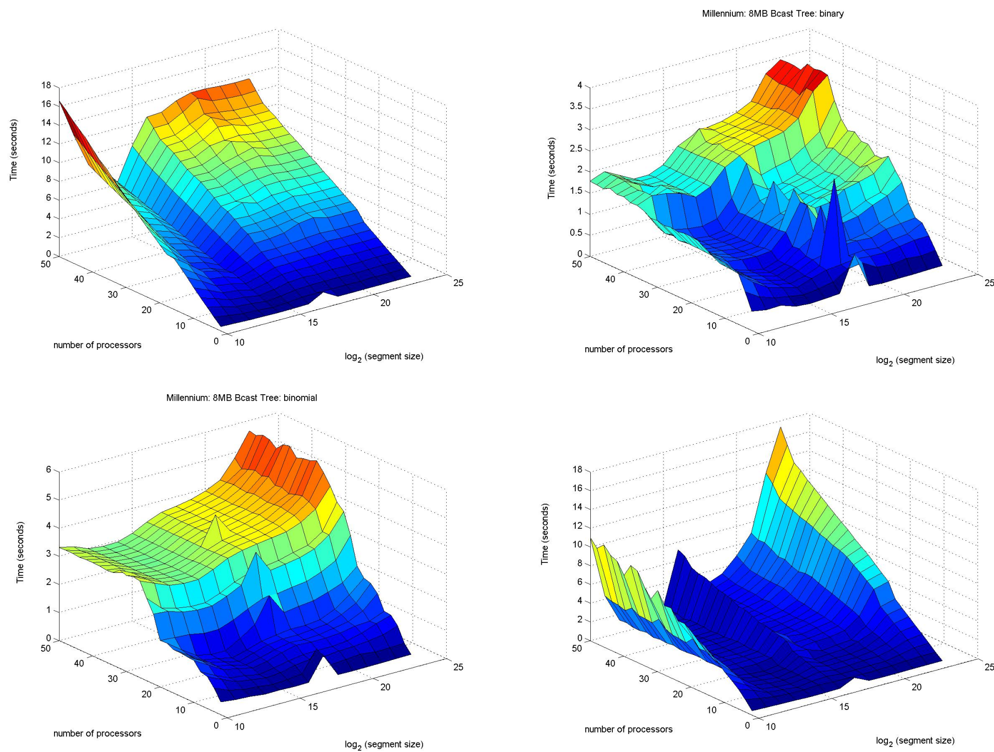

Figure 6: **Effect of Segment Size Across Processors**. These plots show the differences across trees for different segment sizes and different numbers of processors involved in the communication

capture real-time changes in network conditions accurately. This reasoning motivated the development of a probabilistic lottery scheduler for choosing best-performing operation implementations. Needed: - some reasoning about CITRIS's lack of segmenting and Seaborg's fluctuation best segment - some reasoning about the heavy use of Millennium vs. light use of CITRIS and how this may affect latency and bandwidth

6 Dynamic Optimizations: Lottery Scheduler

We have implemented a naive version of a broadcast lottery scheduler and can, using this, show performance results that consistently outperform the equivalent MPI broadcast implementation (Figure 7). Similar results are expected for lottery scheduler implementations for `gather()` and `reduce()`.[3]

6.1 Theory and Implementation

6.1.1 Architecture

Intuitively speaking, the lottery scheduler should be able to adapt to particular clusters and transient cluster conditions by preferring efficient implementations[4] This, within our framework, requires that the lottery scheduler (1) occasionally explore the implementation space and attempt to discover the most efficient implementation available to it (*exploration phase*), and (2) disproportionately select this most efficient implementation (*execution phase*). In the following subsection, we describe the theory and implementation behind the development of such a lottery scheduler.

First, we explain the actual process undergone by collective communication lottery scheduling. Upon execution of a specific collective communication operation, the originating node selects a particular implementation of the operation according to a previously defined probability distribution. In practice, each implementation is allocated, at any particular time, a certain number of lottery tickets – the probability that a particular implementation is chosen is exactly the ratio of the number of tickets that it holds over the total number of tickets. After having chosen the implementation, the originating node also chooses the number of times that this choice will be valid, i.e. a time-to-live (TTL), as a function of the implementation's relative number of tickets. The originating node then transmits, according to a statically known implementation of broadcast, e.g. the vendor-supplied `broadcast()`, an encoding of the function choice and TTL to every other node in its communication group. Upon receipt of this encoding, every node in the communication group (1) executes the particular collective communication operation according to the specified implementation, and (2) stores it and the TTL into memory. Every future call to the same operation checks if the TTL is positive: if it is, the call decrements it, and executes the same implementation; otherwise, it chooses a new implementation based upon the aforementioned protocol, and continues similarly. Throughout the execution of the operation, the lottery scheduler measures, at each node, performance characteristics that are consistent throughout the communication group, e.g. total time until completion. Based upon these measured characteristics, each node independently updates its independent ticket allocation.

6.1.2 Ticket Allocation

To the extent that this ticket allocation determines the frequency at which particular implementations are chosen, the policy is critical to the sensible operation of a lottery scheduler. Early observation of this and the fact that there are a wide variety of such policies led us to implement a lottery scheduler that allows the straightforward incorporation of diverse policies. For the purposes of this report, however, we have implemented a simple ticket allocation policy that nevertheless performs satisfactorily in practice. The lottery scheduler, in particular, maintains for every implementation an exponential average of its running times. (It is this statistic that is updated at the end of every collective communication operation.) Based upon these averages, the lottery scheduler allocates a large proportion α, e.g. 80%, of the tickets to the implementation that has the smallest average time. The remaining implementations are uniformly allocated the residual tickets.

[3]On the other hand, increased performance is not expected for MPI_Scatter(). To the extent that, (1) we have been unable to develop a better implementation of this collective communication, and (2) our lottery scheduler discovers the best implementation, it should always prefer the equivalent MPI implementation.

[4]Here, we define an *implementation* of a collective communication operation to be a corresponding procedure that disseminates data to the processes in its communication group according to a particular tree structure and segment size.

This ticket policy has at least a few subtle benefits. Most obviously, it ensures that the fastest algorithm is chosen a disproportionately high number of times. This is important because the data shows that there are a large number of algorithms that perform very poorly – this policy minimizes their negative contribution. Said in another way, this policy ensures that the average running time of lottery scheduled implementations is not a weighted average of all the algorithms; such a weighted average would be unsatisfactory because (1) there are a large number of non-optimal algorithms, and (2) these non-optimal algorithms have extremely poor running times (see Figure 7). A somewhat more subtle benefit is that this policy enforces that the lottery scheduler only improve its prediction: If a particular implementation is allocated 80% of the tickets at a particular time, another implementation can be allocated 80% of the tickets at a future time only if it has a faster running time.

Furthermore, every sensible lottery scheduler policy allows adaptive reaction to transient network or processor conditions. Unlike static compile-time and installation approaches, which can only utilize static performance data, lottery scheduling allows the cluster to dynamically choose the implementation that is best suited for current cluster conditions. For instance, suppose that (1) a static approach had decided upon the `scatter()` chain implementation, and (2) that the first child (process 2), because of some massive transfer at the corresponding node, suffered severe loss of bandwidth. In this case, the huge amount of data transferred through this child in `scatter()`, i.e. data for nodes 1 through N, would cause huge bottlenecks. On the other hand, a lottery scheduled approach would discover the poor performance of chain and choose, say, the binomial `scatter()` implementation: here, (1) process two receives only its data, and (2) cannot affect the performance of any other node. This is particularly relevant because the scientific computing operations that operate on such clusters generally execute for large amounts of time – this (1) simultaneously creates a huge cost for the static implementation, and (2) allows sufficient time for the lottery scheduler to discover another optimal implementation.

6.1.3 Optimizations

Here, it is important to note that, over time, the lottery scheduler will find the implementation that, on average, is optimal. Although a theoretical analysis of this time follows, we can ensure that the lottery scheduler can rapidly choose optimal - or, at least, nearly optimal - implementations. Upon program initiation, the lottery scheduler reads from a predefined disk location, e.g. file, previously generated exponential averages, ticket allocations, and other bookkeeping information. Lottery schedulers, throughout their execution, continue to update - only within memory - this information. Upon program termination, however, the lottery scheduler writes back the updated information to this disk location. This approach has the beneficial consequence of allowing the use of prior knowledge without significant overhead: because program initiation requires several disk accesses anyways, an additional read is not particularly significant; because program termination does not affect the performance of the program, disk accesses there are relatively inconsequential. On the other hand, this allows the possibility of having different programs overwrite updates from other programs, i.e. the critical section problem. We, however, believe that the steps necessary to counteract this problem are far too costly, e.g. disk writes at the completion of every operation, and consider this approach to be an appropriate balance.

In fact, the amount of time necessary for discovery of the correct implementation can be meaningfully computed according to straightforward statistical and probabilistic techniques. This characteristic is important as it provides a theoretically meaningful approach to determining and optimizing the degree of responsiveness of the lottery scheduler. In general, the expected number of iterations $E[I]$ necessary to converge to the optimal implementation is

$$E[I] \propto \frac{|Z| \lg |Z|}{1 - \alpha},$$

where I is the number if iterations, $|Z|$ is the number of implementations, and $(1 - \alpha)$ is the aforementioned probability of exploration. The intuition behind this expression lies in the observation that the lottery scheduler must "touch" every implementation some number of times. From probability theory (and the *coupon collector problem*), we know that, on average, $|Z| \lg |Z|$ attempts are necessary to randomly select each of $|Z|$ items. Because the lottery scheduler will, in fact, be using non-optimal entries with probability $(1-\alpha)$, the lottery scheduler will iterate, on average, $\frac{1}{1-\alpha}$ times before attempting a non-optimal implementation. Hence, to try each of the $|Z|$ implementations at least once, the lottery scheduler will require $\frac{|Z| \lg |Z|}{1-\alpha}$. Finally,

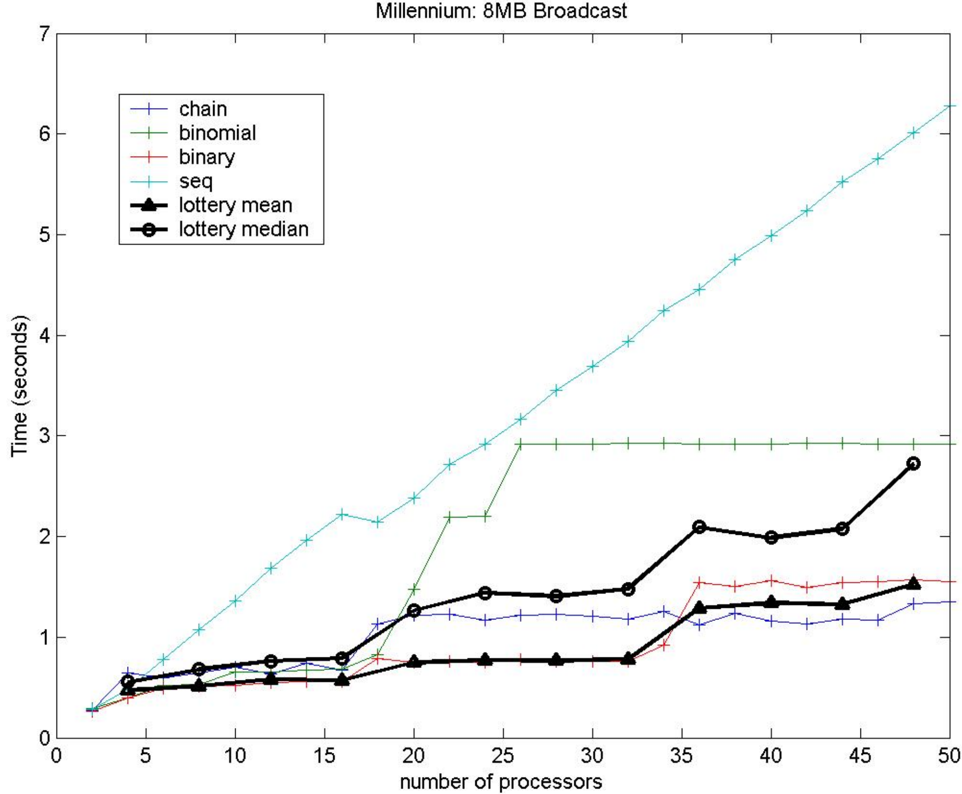

Figure 7: **Millennium Overlay Plot**. Performance of the broadcast operation as described in the previous sections with the performance of the Lottery Scheduler superimposed.

because measurements are essentially noisy observations, the lottery scheduler will need to try each of the $|Z|$ implementations some constant (and bounded) number of times. In practice, the statistics community uses ten to fifteen observations as a general rule of thumb.

6.2 Results

Looking at Figure 7 and Figure 8, it is apparent that the lottery scheduler performs relatively well. In general, its median performance closely follows that of the optimal implementation throughout the domain. On the other hand, it is important to note that the average performance slowly diverges from the median performances of the optimal algorithm as the number of processors increase.[5]

This divergence is readily explained through Figure 8 where we can see the presence of a small number of extremely poorly performing iterations. For instance, in Figure 8b, we see that the overwhelming majority of lottery scheduler iterations require less than two seconds – a reasonable upper performance bound for well-performing implementations, i.e. chain and binary. The problem, however, is that a very small number of extremely poorly-performing iterations (see implementations that require more than four seconds in Figure 8b) push up the average performance measurement; because there are so few calls to these implementations, the median performance measurement is not affected.

Although this problem cannot be entirely avoided, there are a couple improvements that warrant inspection.

[5]Although it might seem that we should measure median lottery scheduler performance with the other medians, this is not true. In general, a median of measurements is taken when it is believed that (1) the measurements are generated independently of each other, and (2) the presence of noisy (and extreme) measurements would inappropriately bias the actual observation. Here, however, extreme measurements are a deterministic and intended result of lottery scheduler exploration. To the end that these extreme measurements are non-erroneous, a measurement of lottery scheduler performance should include them.

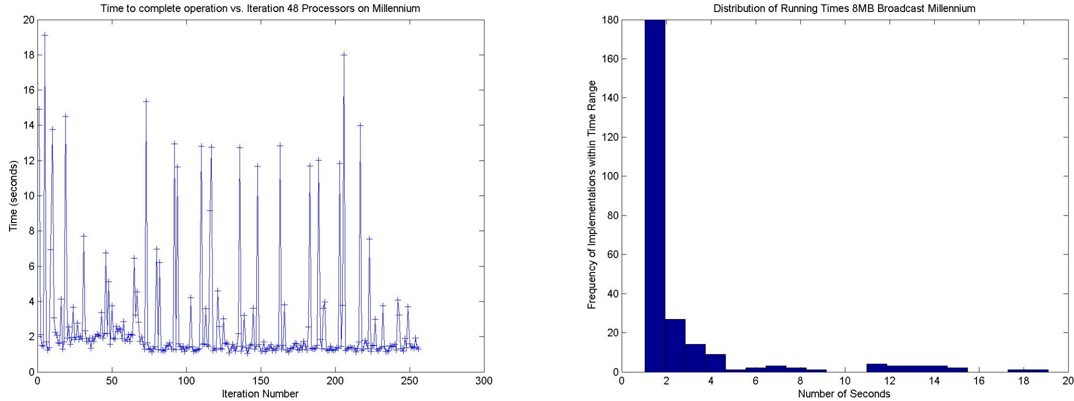

Figure 8: Performance of Lottery scheduler with 48 processors. The first plot shows the time taken by every iteration while the second plot shows a distribution

The most important proposal involves the allocation of exploratory tickets to non-optimal implementations in inverse proportion to their running time. Doing so acts to decrease the rate at which poor implementations are selected. This solution is, unfortunately, incomplete: it still causes the average performance of the lottery scheduler to be biased by very poorly performing implementations. For instance, in Figure 7, we see that exploration of sequential broadcast implementations would cause the inclusion of implementations that are several orders of magnitude worse than the optimal implementation. In order to circumvent this problem, we propose that the lottery scheduling algorithm only explore implementations whose running times are at most two standard deviations more than the optimal running time. This heuristic allows for the exploration of approximately optimal implementations, while precluding the implementations that are extremely poor. Although this approximation may preclude discovery of the globally optimal implementation, we believe that this could be a valid tradeoff against the tremendous cost of exploring very poor implementations.

7 Related Work

1. The work by Gabriel, Resch, and Rhle [4] optimize the the `broadcast and reduce` operations only. Moreover, they optimize only the binomial tree that is used in the MPI implementation. We improve upon this by testing with different trees, as decribed in Section 3.1.

2. In [10], the automatic tuning is done by automatically re-arranging the nodes so that it matches how the cluster is structured. Also, the root dynamically sends the messages to each node informing them what they should do with the data that they have received. We improve upon this by choosing the algorithm to run by running a lottery, as decribed in Section 6. Hence, each different algorigthm type gets a chance to perform well. Moreover, by calculating the time taken by an algorithm based on an exponential average, a new algorigthm type is not chosen based on a knee-jerk reaction.

3. [8] improve the existing set of MPICH implementations by optimizing the message size. However, as we explained in Section 6, this 'optimial' message size can change with changes in the environment of the cluster. As a result, automatic tuning is essential for the collective operations to perform optimally.

4. Karonis, et. al ([6]) have optimized collective operations, but they have done it with a view of wide-area networks, not clusters. As a result, many of their links are much slower than the traditional high-speed clusters that we have concentrated on.

5. Shroff and Geijn have set benchmarks for common MPI operations by comparing different implementatins by different vendors; many of which are customized for different types of clusters. We believe that by letting the lottery scheduler dynamically pick the best algorithm, we have provided a generalized solution that would work well on most systems. [9].

8 Future Work

- Adding different types of trees. We have currently implemented the trees as a forwarding tree, i.e.: each process receives many pieces of data. It realizes if the data is meant for it, and if not, immediately passes it on to the required child process.
- Although we have optmized a few of the more oft used MPI operations, we will optimize more MPI operations, such as MPI_ALLGATHER, MPI_ALLREDUCE, etc.
- Extensions to the current simplistic version of Lottery Scheduler.
- Test more extensively, especially with more clusters (like Lemieux, Clerc) and different interconnects like Myrinet; to make sure that the automatic tuning works across clusters, and with more varied architectures.

9 Conclusion

The gains from our work emerged in two steps. In the first, we found the standard implementations for several MPI collective communications operations to be under-performing. To improve performance we developed a family of implementations for each such operation that performed remarkably well relative to the naive implementations. In the process we observed that the algorithms behaved differently with respect to a variety of variables, including the operation, the size of the communication group, the size of the message, and the cluster that the operation was to run on. Additionally we found that these algorithms responded to a pipelined approach with varying degrees of success across these same variables. Reasoning about the non-uniformity of performance led us to believe that the results reflect the fact that (1) cluster hardware architectures are diverse and thus give us little hope of finding globally optimal implementations, and (2) conditions on networks are often unpredictably transient in nature and thus locally optimal implementations may differ over time. These observations led us to conclude that the best way to optimize collective communications in a flexible way is to adaptively choose locally optimal algorithms based on empirical data. This approach would be more flexible than the current approach of re-implementing the operations for each cluster to maintain optimal performance. The lottery scheduler was an initial attempt at achieving this goal, and its results were quite encouraging. In the future we envision that a suitably general but accurate scheduler be used across all clusters, readily incorporating new, more effective algorithms for the collective communications operations than the current set of tree-based implementations.

References

[1] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of Supercomputing*, November 2000.

[2] UC Berkeley Millennium Cluster. http://www.millennium.berkeley.edu.

[3] NERSC High Performance Computing Facility, LBNL, and IBM. http://hpcf.nersc.gov/computers/sp/.

[4] E. Gabriel, M. Resch, and R. Rhle. Implementing mpi with optimized algorithms for metacomputing, 1999.

[5] Katherine A. Yelick Jack J. Dongerra, James W. Demmel. Automatic tuning for large scale scientific applications. In *NSF ITR Grant Proposal*, 2003.

[6] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. pages 377–386.

[7] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, 1993.

[8] Operations In Mpich. Improving the performance of collective.

[9] Mohak Shroff and Robert A. van de Geijn. Collmark: Mpi collective communication benchmark.

[10] Sathish S. Vadhiyar, Graham E. Fagg, and Jack Dongarra. Automatically tuned collective communications. pages 46–46, 2000.

[11] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Operating Systems Design and Implementation*, pages 1–11, 1994.