Scalable NIC-based Reduction on Large-scale Clusters

Adam Moody[1,2] Juan Fernandez[2] Fabrizio Petrini[2]
Dhabaleswar K. Panda[1]

[1]Department of Computer & Information Science
The Ohio State University, Columbus, OH 43210, USA
{moody,panda}@cis.ohio-state.edu

[2]Performance and Architecture Laboratory (PAL)
Computer and Computational Sciences (CCS) Division
Los Alamos National Laboratory, NM 87545, USA
{juanf,fabrizio}@lanl.gov

Abstract

Many parallel algorithms require efficient reduction collectives. In response, researchers have designed algorithms considering a range of parameters including data size, system size, and communication characteristics. Throughout this past work, however, processing was limited to the host CPU. Today, modern Network Interface Cards (NICs) sport programmable processors with substantial memory, and thus introduce a fresh variable into the equation. In this paper, we investigate this new option in the context of large-scale clusters.

Through experiments on the 960-node, 1920-processor ASCI Linux Cluster (ALC) at Lawrence Livermore National Laboratory, we show that NIC-based reductions outperform host-based algorithms in terms of reduced latency and increased consistency. In particular, in the largest configuration tested —1812 processors— our NIC-based algorithm summed single-element vectors of 32-bit integers and 64-bit floating-point numbers in 73 μs and 118 μs, respectively. These results represent respective improvements of 121% and 39% over the production-level MPI library.

1 Introduction

Many high-performance computing applications depend critically on efficient reduction algorithms. Recent performance evaluation studies show that large-scale scientific simulations spend up to 60% of their time executing reductions [21]. Similar results have been provided by an in-depth analysis of the scientific workload at Lawrence Livermore National Laboratory [12]. Reduction algorithms which minimize latency will thus substantially reduce the overall run-time of such programs.

The problem of developing efficient reduction algorithms has proven to be a rather rich area of research. Reduction collectives involve both communication (data transfer) and processing (data reduction operations), and so efficient implementations must consider characteristics of the network, the processors, and the interplay between the two. In other words, the design space for developing efficient reduction algorithms is quite large. Over the years, many researchers have committed significant time in order to derive optimal and scalable algorithms [1, 2, 3, 4, 5, 8]. These algorithms differ in their assumptions of the underlying system characteristics. During all of this effort, however, designers have commonly assumed processing must be performed by the host CPU.

Network interface cards for modern cluster interconnects, such as the Quadrics Elan [20] or Myrinet NIC [7], provide programmable processors and substantial memory. This added capability allows the host processor to delegate certain tasks to the NIC processor. To differentiate where the task is actually performed, the terminology “host-based” and “NIC-based” has been introduced. There are various reasons to do such a thing, and in this paper we discuss two of them with regard to reduction. Namely, we find that NIC-based reductions can offer both significantly lower latency and better consistency than host-based

SC’03, November 15–21, 2003, Phoenix, Arizona, USA

algorithms.

This paper presents our scientific and technical contributions. We first derive a detailed model to predict the performance of various NIC-based reduction algorithms on the Quadrics network. Guided by this model, we then implement a NIC-based algorithm that uses emulated floating-point operations in the Quadrics NIC. This algorithm operates without the intervention of the host processors. Finally, we provide an enhanced version of our algorithm to be used when reducing larger vector sizes.

Experimental results show that our NIC-based reduction algorithm provides reduced latency and increased consistency in the common case. In particular, in the largest configuration tested on ALC [25] —1812 processors— our NIC-based algorithm summed single-element vectors of 32-bit integers and 64-bit floating-point numbers in 73 μs and 118 μs, respectively. These results represent respective improvements of 121% and 39% over the production-level MPI library. In addition, since the NIC-based implementation is not subject to certain host-level interference, we found that the performance of our algorithm is much more predictable. To the best of our knowledge, our reduction latency results are the best performance achieved on any large-scale parallel computer, both in terms of scalability and consistency.

The rest of this paper is organized as follows. Section 2 outlines relevant characteristics of the Quadrics network. Section 3 describes important trade-offs involved between implementing host-based and NIC-based collectives, and Section 4 discusses design constraints specific to NIC-based reductions. Section 5 presents the model and the algorithm we developed, while Section 6 provides the results we obtained. Finally, related work is discussed in Section 7, and some concluding remarks are given in Section 8.

2 The Quadrics Network

We implemented our NIC-based reduction algorithm on the Quadrics network, a modern cluster interconnect technology [20]. Quadrics is based on two building blocks: a programmable network interface card called the Elan [22, 23] and a low-latency high-bandwidth communication switch called the Elite [24].

The Elan resides on the PCI bus and interfaces a processing node, containing one or more CPUs, to the network. The Elan itself has respectable processing capabilities. It provides a user-programmable, multi-threaded, 32-bit, 100 MHz RISC-based processor; supported with a 64 MB bank of local SDRAM memory, along with an MMU and other sophisticated processing features. All of this hardware is available to the NIC to aid the implementation of higher-level message processing protocols without requiring explicit intervention from the host CPU. In order to better support this usage model, the processor's instruction set includes specialized instructions to construct network packets, manipulate events, and schedule threads.

The Elan divides messages into a sequence of fixed-length transactions for efficient transfer through the network. The primary communication primitive supported by the network is the Remote DMA (RDMA). RDMAs allow for one-sided data transfer between remote processes, i.e. the remote process need not explicitly participate in the exchange. Transfer operations include PUT, which transfers data to a remote address space, and GET, which acquires data from a remote address space. Both operations can access either host- or NIC-level memory.

The underlying network is circuit-switched and uses source-based, wormhole routing. It consists of Elite switches interconnected in a fat-tree topology [19]. Each Elite provides the following features: 8 bidirectional links each with a raw bandwidth of 400 MB/s (325 MB/s at the MPI-level), a full crossbar switch with a low 35 ns cut-through latency, and hardware support for collective communication including barriers and broadcasts.

3 NIC-based vs. Host-based – Pros and Cons

In this paper, we show how NIC-based reduction implementations can outperform host-based versions in two important ways: reduced latency and increased consistency. These benefits are not limited to reductions, and in this section, we describe how NIC-based collectives, in general, can attain such gains. We also discuss the major penalties encountered when implementing collectives at the NIC-level, namely, host-NIC synchronization cost and reduced computational performance.

3.1 Advantages – Reduced Latency and Increased Consistency

NIC-based collectives can be completed significantly faster than host-based versions on fast networks. Modern cluster interconnects, such as Quadrics, support very low message latencies; so low in fact, that PCI bus transaction time is substantial compared to the latency between nodes. By implementing collective communi-

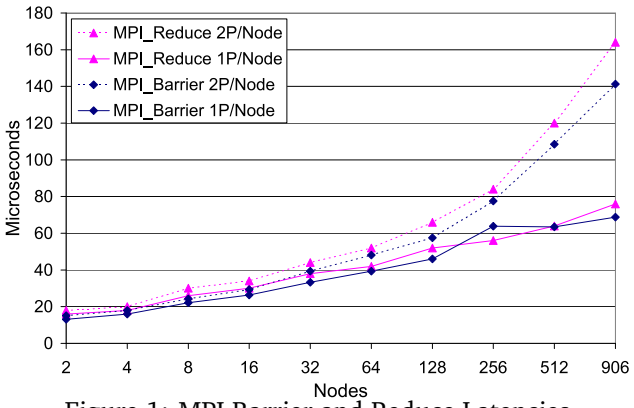


Figure 1: MPI Barrier and Reduce Latencies

cations in the NIC, as opposed to the host, many extraneous PCI bus transactions can be eliminated. This can significantly reduce the total operational latency.

Collective communications, by their very nature, require a series of related messages to be exchanged between nodes involved in the operation. In host-based implementations, the host processor explicitly handles each of these messages. In order to do so, each message must be relayed back and forth between the host processor and the network via PCI bus transactions. NIC-based implementations, on the other hand, handle messages immediately at the NIC, avoiding most of these trips through the PCI bus. In fact, NIC-based implementations suffer from such costs only while initiating and terminating the operation. Collectives involving many nodes entail many messages, which means that NIC-based collectives can scale substantially better than host-based versions as the size of the cluster increases.

Thus far, the majority of NIC-based research has taken focus on this advantage [6, 9, 10, 11, 14, 16, 18, 27, 28]. In the process of further investigating how this established advantage extends to the realm of reductions, we found a new and much more significant advantage that NIC-based collectives provide when running on large-scale systems:

NIC-based collectives show dramatically reduced latency and increased consistency over host-based versions when used in large-scale clusters.

It happens that process interference at the host level turns out to be a major problem on large clusters. To demonstrate this, observe Figure 1. This figure shows the host-based latencies measured for a barrier and a reduction when using both one and two processes per node. As the number of nodes is increased, note how the latency for each collective deviates drastically when two processes are involved on each node as opposed to just one.

This result is surprising since the underlying implementation efficiently reduces the two process problem to the one process problem by first performing a local shared memory step. Since shared memory operations take place quite quickly compared to typical network latencies, and because all nodes perform this brief step in parallel, the added overhead should be both small and constant with the number of nodes. Hence, the implementation can not be at fault; something else is to blame.

In this system, there are two physical processors per node. When the collective involves only one process per node, there is a spare processor on which the node may run various system threads. However, when both processors are used by the collective, at least one of the processes is forced to share its processor with the system threads. This process interference turns out to be responsible for the drastic drop in performance [21].

Basically, the problem arises since host-based processes in charge of handling intermediate messages during the collective may be subject to descheduling. That is, processes at intermediate nodes may be descheduled from the CPU just before handling an incoming message. In this case, the collective will stall until the process is rescheduled to handle the message. Such untimely context switching may lead to poor performance. The problem tends to manifest itself on large systems more so than on small systems, because larger collectives require larger communication tree structures. Larger trees, in turn, require more intermediate nodes. Thus, there are simply more chances that some intermediate processes will be interfered with on large-scale clusters.

In addition to increased latency, one may immediately understand that this is a rather non-deterministic phenomenon, which leads to a large variance in operational latency from one collective invocation to another. Thus, the same process interference problem simultaneously increases average latency and decreases operational consistency.

As host-level process interference is inherently a host-based problem, NIC-based implementations can avoid it altogether. As a result, NIC-based collectives can complete with drastically better latency and in a more consistent fashion.

3.2 Disadvantages – Overhead and Limited NIC Processor Capability

Even though the NIC carries out the actual collective in the NIC-based implementations, the host must communicate to the NIC, among other information, what operation is to be done, which data are to be processed, and when the operation is to start. Also, the

NIC must notify the host of the operation's completion and deliver any final results. This process is termed "host-NIC synchronization."

In the absence of process interference, host-NIC synchronization introduces overhead which may significantly reduce the benefits gained from implementing NIC-based collectives. Although we lack the space for discussion, it should be noted that this overhead can be largely avoided by overlapping it with other operations and is thus of minor concern.

The most important issue to be considered is that of the NIC processor. The user-programmable processor on the NIC is considerably slower than the host processor (more than 25 times slower on the Livermore machine). Different processing requirements by different algorithms and different operations make this a very significant difference. Basically, this difference places a limit on the complexities of the collectives and algorithms which may benefit from NIC-based implementations. To make matters more complicated, the NIC processor typically lacks substantial processing functionality as well. For example, there is no hardware-based floating-point support on the Quadrics Elan. The limitations of the NIC CPU proved to be the toughest design issue we encountered in our work.

4 NIC-based Reduction Design Constraints

Reductions are computationally intensive collectives, and as a result, the slower and less functional NIC CPU becomes a limiting factor. In this section, we probe the sensitivity of the Quadrics Elan to computational requirements. Fortunately, the common case reduction operation in many programs does not require large amounts of computation. Thus, even with limited processing power, NIC-based reductions present a viable option.

4.1 Complications – Processing Speed and Capability

As noted above, NIC CPUs are typically much slower than the CPU available at the host level, often by an order of magnitude or so. In addition, NIC CPUs provide less functionality. Knowing these limitations, most of the research in NIC-based work so far has concentrated on collectives which involve little processing. Collectives such as barriers, broadcasts, and multicasts simply require intermediate nodes to pass on the received message as is, with perhaps minor data restructuring. Because so little processing is required, these

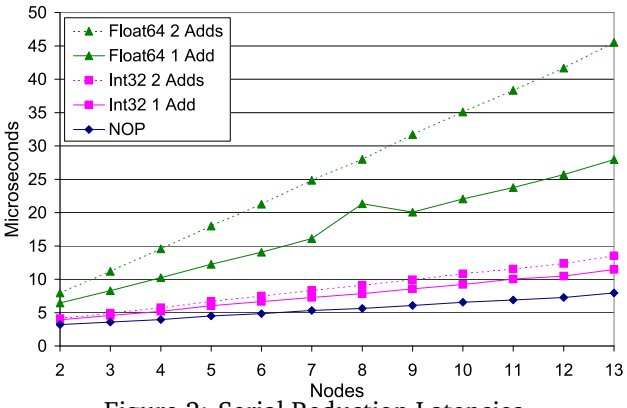


Figure 2: Serial Reduction Latencies

algorithms incur little penalty from running on slower processors, and the overall results have been quite positive.

The success obtained by simpler NIC-based collectives inspired us to investigate more complicated cases, namely reductions. Our design goals were to support NIC-based implementations of the standard MPI reduce and allreduce collectives for 32- and 64-bit integer and floating-point data types, each having minimum, maximum, and summation operations.

The first problem we encountered is the fact that the Elan CPU has no hardware support for floating-point operations. Thus, we were required to emulate floating-point operations in software with integer instructions. Of course, this isn't the first time such a problem has been posed, and others have provided sophisticated software libraries to serve as a solution. In particular, we tackled this problem by porting SoftFloat [15] to the Elan, an IEEE 754 compliant floating-point package written by John R. Hauser, which is freely available to the public domain.

After providing floating-point capability, we investigated the communication and computation characteristics of the Elan. This was accomplished by implementing a very simplistic reduction algorithm. Basically, a group of P nodes perform a reduction by designating one of the nodes as the root, which is solely responsible for receiving and reducing all of the data. After a synchronization phase, all non-root nodes simultaneously send their data to a corresponding RDMA buffer at the root. Upon receiving all of the messages, the root performs the reduction operation on them in serial order. We will refer to these results at later points in the paper, so it is convenient to provide a name for this algorithm. We simply call it the "serial reduction" algorithm.

Serial reduction tests involving 2-13 nodes for various reduction operations and data sizes produced Figure 2. In particular, we show latencies for 32-bit integer and 64-bit floating-point addition on both one-

and two-element vectors, as well as, a NOP operation which indicates the lowest achievable bound for any serial reduction involving data processing. There are a couple of important features to take note of.

First, regardless of the operation, all of the curves closely follow a linear trend as the number of nodes is increased. Such a tight trend makes it very easy to model performance, as latency can be predicted using only a handful of model parameters. We address this issue in more detail in Section 5, but essentially, the intercept is related to the message latency, while the slope represents the reception and reduction costs required to process a message.

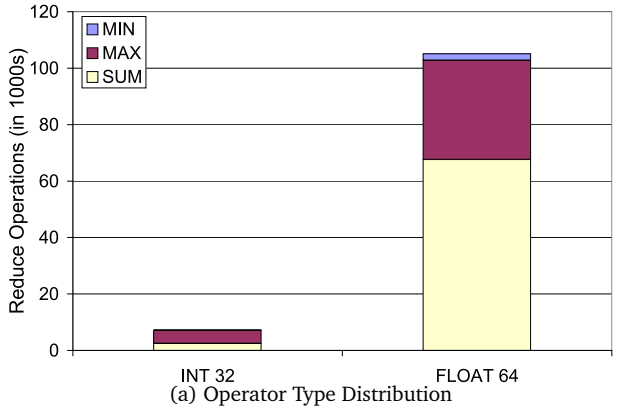
Second, it is more relevant at this time to take note of the reduction latency sensitivity to the operation being performed. Simpler operations scale considerably better than more complicated ones: compare integer addition to floating-point addition. Even fast operations are rather sensitive to small changes in data size: observe integer addition for one- and two-element vectors. And, the emulated floating-point operations are especially slow: the time to perform a single 64-bit floating-point addition is comparable to the message latency between nodes.

Certainly then, it will be essential to consider both communication and computation costs when designing efficient NIC-based reduction algorithms. It is also clear that NIC-based reductions, even for very simple operations, will execute with reasonably low latency only for small data sizes. Nevertheless, it turns out that even while this is a rather stringent restriction on the class of problems where NIC-based implementations may be valuable, a large majority of the problems posed by practical programs falls within this class.

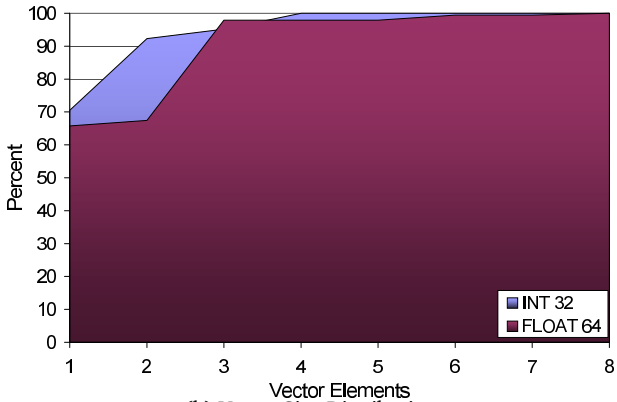
4.2 Simplifications – Simple Operations and Small Data Sizes

Reductions involving simple operations on small data sizes are the common case in many scientific applications. To demonstrate this, we profiled the MPI allreduce operations performed during the execution of SAGE [17]. SAGE is a program representative of the typical scientific applications running on large-scale, ASCI-class parallel machines. The results are shown in Figure 3.

Figure 3(a) shows the distribution of reduction operator types. First, note that only a few simple types of operations are used by SAGE: minimum, maximum, and summation. Typical reduction operations thus require limited processing. Second, note that floating-point operations far outnumber integer operations. This strongly suggests that, if no hardware-based floating-point support is provided on the NIC



(a) Operator Type Distribution



(b) Vector Size Distribution

Figure 3: Profile of MPI_Allreduce Operations in SAGE

CPU, the emulation software should be highly optimized in order to reduce costs in the common case.

Equally important is Figure 3(b), which shows the cumulative distribution of the data sizes for both integer and floating-point data types. Direct observation makes a striking point: 95% of all reductions use 3 or fewer elements and 100% use 8 or fewer.

Observations from these two figures are key. Together, they imply that typical reductions involve simple operations on small vectors, which so happens to be the same class of reductions for which one may benefit from NIC-based implementations. In other words, NIC-based reduction implementations will benefit the common case the most. Thus, given the substantial benefits previously mentioned, NIC-based reduction implementations promise to be quite valuable to typical programs, even while considering the limitations imposed by the NIC processor.

5 The Model and the Algorithm

Through the years, many efficient reduction algorithms have materialized, stressing the importance of these collective communication patterns. However,

the large majority of previous work performs the reduction at the host-level, thus missing out on the valuable benefits available to NIC-level implementations. In this section, we address details in the design of efficient NIC-based reductions on the Elan. We begin by applying a modified LogP [13] model to Quadrics, which can be used to choose among various NIC-based implementation alternatives. We then present an efficient algorithm based on this model, and conclude with a valuable optimization available for multi-element vectors.

5.1 The Model

Observations of the serial reduction data, as shown previously in Figure 2, suggest a simple parametric model. Namely, it is difficult to overlook the sharp linear trend that relates the reduction latency to the number of nodes involved. Using just the slope and intercept, such a tight trend provides a very simple but accurate analytical model to estimate the serial reduction latency. Furthermore, the serial reduction algorithm will form the basic building block of more sophisticated tree-based algorithms. Given an accurate model for the building blocks, we can piece together a model for more sophisticated algorithms. In other words, the slope and intercept of the serial reduction latency curves are sufficient to quite accurately predict the performance of any other proposed algorithm.

With this as our motivation, we delve a little deeper to define the slope and intercept in terms of more meaningful parameters. To account for the linear trend, we recall the implementation of the serial reduction algorithm: all nodes simultaneously send their data to the root, which receives all, and then reduces all messages in order. Since the nodes send to the root simultaneously, all messages worm their way to the root in parallel. Hence, regardless of the number of nodes, we suffer the cost of message latency only once. On the other hand, the root receives and reduces each message serially, which introduces reception and reduction cost on a per node basis.

With these observations, we define our model as given in Table 1. We will typically suppress the functional parameters M (message size) and OP (reduce operation) from the various terms.

Essentially, this model modifies LogP [13] to better serve our needs. We substitute the parameter r in place of o, the cost to receive a message; and represent the parameter g as $(r + c)$, the time required to fully process a message. Conceivably, the message latency, the reception costs, and the reduction costs may all differ between host-based and NIC-based implementations, and these redefinitions allow one to explicitly account for those differences using dedicated parameters. Additionally, since r and c may be general functions of the message size, one may better model nonlinearities, such as data packetization and caching, which are relevant for small data sizes. Finally, although we don't investigate it in this work, splitting g into two components, r and c, allows one to directly model any communication and computation overlap.

Parameter	Meaning
L	message latency
r(M)	reception cost of a message of size M
c(M, OP)	reduction cost of a message of size M, dependent on the operation OP
P	number of nodes
C(OP)	constant due to initial overhead, in general dependent on the operation OP

TABLE 1: Model Parameters

Note with this model it is simple to describe the linear form of the serial reduction latency curves as:

$$T_{serial}(P) \approx C + L + (P-1) \cdot (r + c)$$

This expression is shown pictorially in Figure 5.1, which presents a timeline depicting the time required for the root node of the serial reduction to receive and reduce $(P-1)$ vectors.

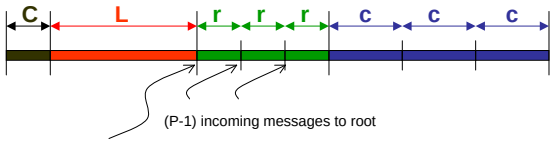


Figure 4: Model of Serial Reduction Latency

To assign numerical values to the parameters, we extracted the values of r and c from the serial reduction data for various values of M and OP. The terms L and C were fit to the data, and P is given for a particular problem. We note that while r is dependent on the message size in general, it turns out to be constant for cases we are interested in. This is because we focus

on reductions involving vector sizes of only a few elements, say up to eight, which typically fit into a single 64-byte fixed-length packet on the Quadrics network. Thus whether we are working with one-element vectors or eight-element vectors, the receive time is the same, i.e. the cost to receive one 64-byte packet.

The proposed model parameters also suggest the general form of efficient algorithms. Again looking at the serial reduction data in Figure 2, one may note that for small messages, the latency, L, is often significantly more than the receive time, r: e.g. compare the intercept of the NOP curve to its slope. This is relevant considering the circuit-switched nature of the network as the sender may only send a message every L units of time, while the receiver can receive one in every $r \ll L$ units. As a result of this asymmetry, nodes in efficient algorithms will tend to receive more often than they send, which leads to tree-shaped communication structures. Given that efficient algorithms will take the form of trees, we designed and implemented f-nomial tree algorithms, feeling they provide a good balance between structural simplicity and optimality.

5.2 f-nomial Trees – Generalized Binomial Trees

Our goal in this paper is not so much to present a new algorithmic communication structure via f-nomial trees (a.k.a., k-nomial trees), but rather to show that because of the limited NIC processor, efficient NIC-based reductions require a range of communication structures. We chose to implement f-nomial trees because they provide such a range of structures by generalizing a well-known reduction algorithm, binomial trees.

Binomial trees are commonly used in reduction algorithms because they offer two useful properties: 1) they have a regular structure, so they are easy to implement, and 2) they keep many nodes involved throughout the collective, so they are well-parallelized. In fact, binomial trees have been shown to be optimal communication structures for reduction in synchronous communication networks, i.e. those in which the sender and receiver have the same cost for message transfer [2].

f-nomial trees generalize binomial trees to add a third valuable property: they provide a range of different communication structures, so one may selectively balance communication against computation. This property is especially useful for NIC-based reductions, where the computation costs incurred by the slow processor may change substantially depending on the operation being performed and the amount of data being processed. We will describe f-nomial trees starting from a quick review of the operation of binomial trees, from which the generalization is trivial. Also, although reduction trees collapse to the root node, it is easier to describe the structure of a tree as it expands. For convenience then, say we desire to broadcast a message from the root to all nodes in the tree.

The operation of binomial trees can be described as follows. Starting from the root, the broadcast message is distributed through a series of communication phases. During each phase, each node that holds a copy of the broadcast message at the start of the phase sends to another node which doesn't. In this way, the number of nodes that hold a copy of the message doubles at the end of each phase. Thus, in a binomial tree, the number of nodes the message can reach grows as a power of 2 (hence the prefix "bi") with the number of phases.

An f-nomial tree generalizes this algorithm by having each node with a copy of the message at the start of a phase send to $(f-1)$ others who don't, as opposed to just one. For instance, during the first phase, the root sends to $(f-1)$ children, so that by the end of the first phase the message has spread from the root, 1 node, to the root and its $(f-1)$ children, a total of $1+(f-1) = f$ nodes. In the second phase, each of these f nodes becomes a parent to $(f-1)$ children who have yet to receive the message. By the end of the second phase, the message spreads from the f parent nodes to each of their $(f-1)$ children, reaching a total of $f + f(f-1) = f(1+(f-1)) = f^2$ nodes. Similarly, in the third phase, each of these f^2 nodes become a parent and each sends to $(f-1)$ children who have yet to receive the message, so that by the end of the third phase, the message spreads to a total of f^3 nodes, and so on. Thus now, the number of nodes the message can reach grows as a power of f with the number of phases. This is the structure of the algorithm we implemented; only remember, the tree collapses rather than expands since we desire a reduction rather than a broadcast. In Appendix A, we provide the psuedo/C code which initiates and drives a reduction algorithm using f-nomial tree communication structures.

As a concrete description of an f-nomial reduction, consider Figure 5, which shows a graph representing a 4-nomial tree overlaid atop a set of 16 nodes. In this example, we wish to reduce data distributed among the 16 nodes and place the result at the root node 0 using a 4-nomial tree communication structure. The arcs in the graph connect communication partners and are labeled with the phase number in which the corresponding communication takes place; all messages travel upward from children to parents. During the first phase, parent node 0 receives and reduces $(4-1) = 3$ messages from nodes 1, 2, and 3;

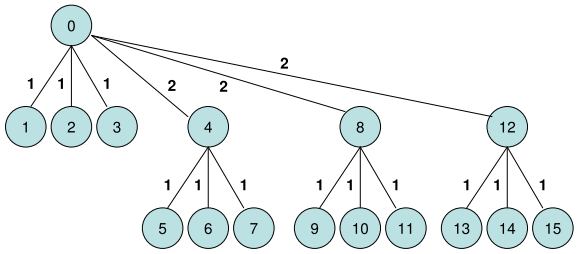


Figure 5: 4-nomial Reduction over 16 Nodes

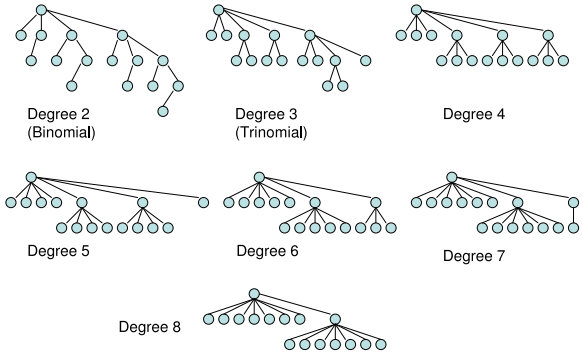


Figure 6: f-nomial Trees of Varying Degrees covering 16 Nodes

while likewise, nodes 4, 8, and 12 simultaneously receive and reduce data from their own three children. At the end of the first phase, the distributed data has been partially reduced and localized to the four parent nodes 0, 4, 8, and 12. To be precise, the number of nodes containing data relevant to the reduction has been cut by a factor of four, from 16 to 4. The algorithm completes after the second phase again cuts the number of nodes by a factor of four, from 4 to 1, when node 0 receives and reduces the partial results from the three, now child, nodes 4, 8, and 12. Thus, in two communication rounds, the 4-nomial tree is able to perform a reduction over $4^2 = 16$ nodes.

f-nomial trees offer a range of communication structures to select from through choice of the degree of the tree f. For example, Figure 6 shows f-nomial trees of various degrees, all which cover 16 nodes. This flexibility allows one to trade off between communication and computation costs, choosing an appropriate mix for a given problem. Each level of the tree corresponds to a communication phase, while the width is related to the amount of computation any one processor is required to do. Efficient algorithms will tend to balance the costs of communication and computation. Communicationally bound reductions will favor wide trees to minimize the number of tree levels, and hence, the number of communication phases. Computationally bound reductions, on the other hand, will fair better with tall trees which better parallelize the processing. Thus, the best choice for the degree of the tree depends on the relative costs established by a particular problem.

We would like to be able to choose the best tree through analytical methods, so now we apply our model to the algorithm. Since the root node of an f-nomial tree is involved in every phase of the algorithm, we can predict the latency of the entire operation by focusing on the work the root node must do. Assuming a full tree, an f-nomial tree generates $\log_f P$ phases, during each of which the root has $(f-1)$ children. Each phase will be of the linear, building-block form of the serial reduction algorithm previously discussed. In other words, the critical path consists of a series of $\log_f P$ serial reductions, each involving f nodes. Thus, inserting $T_{serial}(f)$ and adjusting for initial overhead, one arrives at the following expression as a quick analysis of the f-nomial reduction latency:

$$\begin{aligned} T^{full}_{fnomial}(P,f) &\approx C + T_{serial}(f)\cdot\log_f P \\ &\approx C + [L + (f-1)\cdot(r+c)]\cdot\log_f P \end{aligned}$$

An example application of the model to intermediate phases is shown pictorially in Figure 7. In this figure, the two horizontal timelines represent two intermediate parent nodes in the f-nomial tree, the bottom node being one of the children of the top node. To start, the initial overhead, C, is encountered in parallel across all nodes as a one time cost. Then, after waiting for a length of time L, the two parent nodes each receive and reduce the data from their $(f-1)$ children of the first phase. Starting the second phase, the bottom node, now a child to the top node, immediately sends its partial result to its parent. Again, after a length of time L, the top node receives and reduces the data from its $(f-1)$ children of the second phase. The reduction continues off the diagram as the top node, now a child to some higher node, sends its partial result to its parent to begin the third phase, which is not shown.

With this model, it is straight-forward to compute the optimal degree f to use for a particular problem. One seeks to find that value of f which minimizes the reduction latency. To do so, we take the derivative of $T^{full}_{fnomial}(P,f)$ with respect to f and find the condition which sets the result equal to zero. This analysis is left to Appendix B, but we provide the result here for discussion:

$$f\cdot(\ln f - 1) = L/(r+c) - 1$$

The value of f satisfying the expression above for L, r, and c will provide the minimal reduction latency

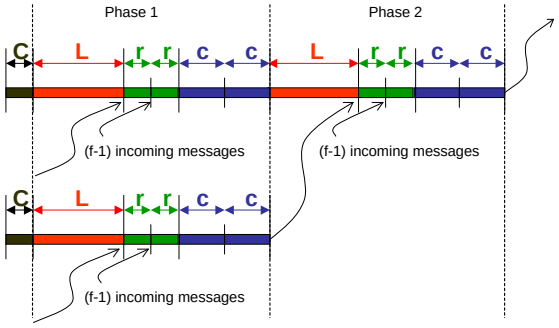


Figure 7: Model of f-nomial Reduction Latency

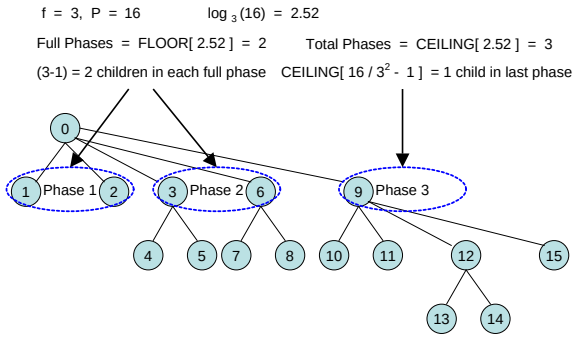


Figure 8: Application of Reduction Latency Model to 16-Node 3-nomial Tree

for a full f-nomial tree. One important observation is that the optimal value of f is not influenced by the number of processing nodes P. Only integer values $f \geq 2$ are meaningful, so one must choose f which gives the closest fit under these restrictions. Note that for these values, the left side of the equation strictly increases with f. Hence, low values of f are good when $(r+c) \gg L$ and high values of f are good when $(r+c) \ll L$. Since L and r are constants, this implies that low-degree trees are desireable when computation costs are high, and high-degree trees are desireable when computation costs are low, which agrees with our previous intuitive arguments. Note that if the computation cost c is negligible, the best value for f assumes an upper bound, given by a function involving just L and r, two constants. This means that, since computation costs are essentially negligble for the host processor when considering simple operations and small data sizes, the best degree for host-based algorithms is constant across all operations and data sizes and bounded from above only by the receive cost. The same can not be said for NIC-based algorithms where computation is much more costly. The best degree f for NIC-based algorithms may take on any value ranging from the lower bound to the upper bound depending on the exact computation costs.

Unfortunately, the simplistic expression above for $T^{full}_{fnomial}(P, f)$ does not accurately account for trees with an arbitrary number of nodes. The above expression was derived assuming a full tree, i.e. assuming $\log_f P$ is an integer. When the number of nodes is not an integer power of the degree f, the root may not have a full set of children during the final phase. In this case, the root still incurs the message latency cost, L, while waiting for the data of the last phase to arrive, however, there will be fewer than the full set of $(f-1)$ messages to receive and reduce. In general, more careful analysis will show that:

$$\begin{aligned} T_{fnomial}(P, f) \quad \approx \quad & C + L \cdot \lceil \log_f P \rceil + \\ & (r+c) \cdot (f-1) \cdot \lfloor \log_f P \rfloor + \\ & (r+c) \cdot \lceil P / f^{\lfloor \log_f P \rfloor} - 1 \rceil \end{aligned}$$

Here, $\log_f P$ represents roughly the number of phases in the f-nomial tree. In particular $\lceil \log_f P \rceil$ is the *total* number of phases, while $\lfloor \log_f P \rfloor$ is the number of *full* phases, i.e. those involving a full set of $(f-1)$ children for the root. The L term accounts for the message latency cost incurred from each phase of the tree. The last two $(r+c)$ terms together sum the reception and reduction costs incurred for processing each child. Of these two terms, the first counts the number of children we process due to full phases, while the second counts the number of children in the final phase, if less than a full set. An example given in Figure 8 demonstrates how the various terms refer to a 16-node, 3-nomial tree.

When using this expression for $T_{fnomial}(P, f)$, it is non-trivial to express the best degree f in terms of the other model parameters, as done before. However, in practice the best degree tends to be a small value, so one can simply cycle through a limited set of values and evaluate the expression to find the best one numerically. This approach is illustrated graphically in Section 6 when we validate the model.

5.3 Vector Split Optimization

The slower and less functional NIC CPU is quite sensitive to the vector size of the reduction, especially for floating-point operations which must be emulated in software. To reduce this effect, it helps to increase the processing parallelism. In other words, we would often like to keep as many of the NIC processors working as possible. To do so, we are often willing to suffer a

little extra communication cost in favor of a substantial reduction in computation cost.

For multi-element vectors, we can increase parallelism through an optimization proposed by Van de Geijn [26]. Basically, the idea is to split the vector and assign the different pieces to different groups of nodes. The groups then reduce the distributed pieces in parallel and recombine the vector from the partial results in the last step. In other words, presented with this optimization, we now have two options available to reduce multi-element vectors: 1) reduce one large vector serially through a single tall tree, or 2) distribute and reduce smaller pieces of the vector in parallel through shorter trees, incurring the added overhead of splitting and recombining. In the second approach, we suffer from extra communication to distribute and recombine the vector pieces, however, if computation is expensive, we save significantly by processing smaller pieces of data during each phase of the tree. For tall trees, which require many phases, this savings can quickly amount to a lot.

As an example, which is diagrammed in Figure 9, say we would like to use this optimization to reduce a two-element vector over 8 nodes. Here, the vector elements are shown as small rectangles located adjacent to circles representing the nodes on which they reside. As shown in the left section of the figure, we first split the group of 8 nodes into two groups of 4, represented with the dotted line bisecting the circles. We wish to assign the the top element of the vector to the top group of 4 nodes and the bottom element to the bottom group. To do so, nodes in the two groups send the appropriate element to a partner in the opposite group, as represented by the arrows, and reduce the received data with their local copy of the corresponding element. At this point, the two-element vector has been split among the two groups. The top group contains all information about the top element, and the bottom group contains all information about the bottom element. Once this distribution is complete, the two groups simultaneously perform group-wise reductions on the element assigned to them, represented with the dotted boxes in the middle section of the figure. Finally, as shown in the right section of the figure, the two fully-reduced elements are recombined to produce the fully-reduced two-element vector.

This optimization was prepended to the f-nomial algorithm to create a new algorithm we call "f-nomial split". During the beginning, the vector is recursively split in half a specified number of times, with the pieces being distributed among the appropriate number of groups. The f-nomial tree algorithm is then used within each of the groups to reduce the smaller pieces. As discussed, these partial reductions occur in parallel across the multiple trees. The root of the f-nomial tree in each group will receive a fully-reduced piece of the vector, which is then sent to the primary root of the overall reduction during the last step. The improvement due to this optimization proved to be dramatic and is discussed in Section 6. Basically, it allows NIC-based reductions to scale substantially better than they otherwise would have for larger vector sizes.

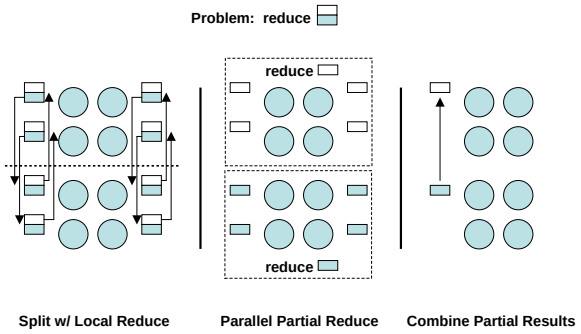


Figure 9: Vector Split Optimization

6 Experiments

In this paper, we aim to highlight the attractive advantages NIC-based reductions achieve over host-based versions in large-scale clusters. We developed our algorithm and our initial performance evaluation on the "crescendo" cluster at Los Alamos National Laboratory, which consists of 32 dual-processor nodes with 1.0 GHz Pentium IIIs and the Quadrics network. We completed our scalability analysis on the ALC cluster [25] located at Lawrence Livermore National Laboratory. The ALC uses 960 dual-processor nodes with 2.4 GHz Xeons and the Quadrics network.

To begin, we verify the accuracy of the newly proposed model. Then, we show results indicative of the reduced latency and increased consistency we observed when using NIC-based reductions. To end, we present the benefits obtained with the vector split optimization. Yet before discussing the measurements, we explain several points of our testing procedures in more detail.

6.1 Procedural Details

Here, we discuss certain details about our implementation and the testing methods we used which are relevant for proper interpretation of the results given in the following subsections.

First, regardless of the number of host processes per node, each node implements the NIC-based reduction

using a single thread running on a single NIC. Whenever there are multiple processes per node, we first use the host processor and shared memory to reduce the local data vectors before we initiate the NIC-based portion of the algorithm. In NIC-based reduction, one accepts the increased computational cost associated with performing reduction processing on the slower NIC processor in return for elimination of extraneous data transfers to and from the host. However, if a collection of data (e.g., vectors for multiple local processes) is already located at the host, one may as well use the faster host processor to reduce it. In addition to the obvious computational savings, less data needs to be sent through the PCI bus to the NIC, just the locally reduced result rather than each of the local vectors.

Second, while our goal is to investigate both reduce and allreduce operations, our NIC-based reduction results present just reduce. Our focus was to optimize reduce, which is simpler to implement, model, and analyze. Admittedly, since allreduce tends to be used more frequently than reduce in many parallel programs, its inspection is more relevant. However, the hardware-based broadcast provided by Quadrics allows us to simplify things. The hardware-based broadcast, which scales very well (almost constant) as the number of nodes is increased, can be tacked on to the end of an efficient reduce operation to implement an efficient allreduce operation. Thus, our measurements for reduce are representative of what one may expect for allreduce, since the observed reduce latencies can be extrapolated to estimate allreduce latencies with the addition of a small constant.

Third, for testing purposes, we insert a barrier between each of our NIC-based reductions in order to serialize consecutive reduction invocations. As Quadrics provides a hardware-based barrier mechanism, these barriers tend to keep the distributed nodes very tightly synchronized. This simplifies the measurement procedure since we need not worry about pipelining effects associated with nodes which escape ahead to start the next operation before the previous one has completed. While this extra synchronization adds unnecessary overhead to the reduction operation, we include the cost of the barrier in the NIC-based latency measurements to make a fairer comparison to the host-based results, whose implementation, as to be discussed, includes such global synchronization as a final step.

Fourth, we used the MPI reduce collective for our host-based tests. The MPI implementation internally delegates the work to a reduction function, called elan_reduce(), supported in the lower-level Quadrics Elan library [22]. The Elan algorithm, in turn, performs a reduction via a 4-ary tree communication structure followed by a hardware-based broadcast of the result. This trailing broadcast simultaneously serves as a global synchronization step and acts to extend the reduce into an allreduce. Thus, really the Elan library function implements an allreduce operation, rather than the simpler reduce operation which we investigate with our NIC-based scheme. Even so, the tests remain more or less fair, since the cost of the barrier inserted between each of the NIC-based reductions effectively offsets the broadcast which completes each of the host-based reductions.

Finally, when taking measurements, we found a large variance in the operational latency from one reduction invocation to another, especially for host-based reduction. Unless otherwise stated, we compensate for this variance by reporting the average reduction latency: computed as the total time required to complete 100,000 iterations, divived by 100,000.

6.2 Model Validation and Algorithm Inspection

Before running tests on large-scale systems, we wanted to inspect the accuracy of the model. We extracted the model parameters from the serial reduction data as previously mentioned and applied them to various f-nomial trees for different reduction problems.

To provide a context of typical values, we list some of the NIC-based model parameter values in Table 2. From these numbers one may also note many of the design characteristics previously mentioned. For example, note that r is a fifth of L which highlights the asymmetry issue we discussed. Also, take note of the significant computation costs, especially for floating-point, and the nonlinearities introduced by caching effects as the number of elements increases for a given operation. Again, the model was intentionally designed to detail these characteristics, which are relevant in the design of efficient reduction algorithms.

To provide some confidence in this model, in Figure 10, we show the predicted and measured NIC-based f-nomial reduction latencies as a function of the degree f for 64-bit floating-point addition on a 31-node system using vectors sizes of 1, 2, 4, and 8 elements. There are a few items of interest here.

First, as one might guess, we were of course quite pleased to see how well the model aligns with actual measurements. Because the model fits the data so closely, one may make theoretical estimates of the behavior of various reduction algorithms with a good deal of confidence. Thus, in future reduction algorithm design, one has a detailed model by which one may be able to consider and eliminate many design choices without the need to run extensive tests. This is

Parameter	Value
L	2.10
r	0.42
C	9.20

(a) Communication and Initialization (μs)

Operation	1-elem	2-elem	4-elem	8-elem
Int32 Max	0.27	0.46	0.84	1.60
Int32 Add	0.25	0.44	0.76	1.44
Float64 Max	0.67	1.27	2.44	4.80
Float64 Add	1.50	2.95	5.80	11.56

(b) Computation (μs)

TABLE 2: NIC-based Model Parameter Values

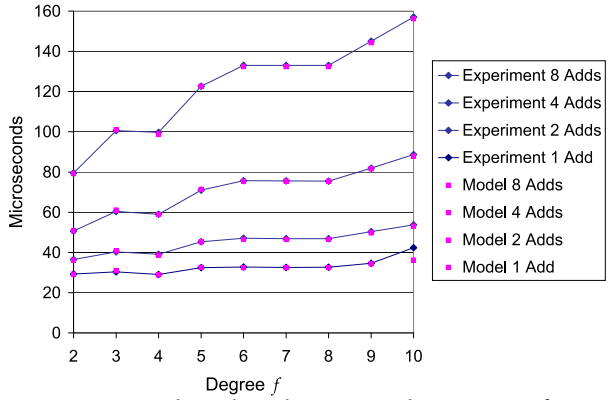


Figure 10: Predicted and Measured Latencies for 64-bit Floating-Point Addition over a 31-Node f-nomial Tree

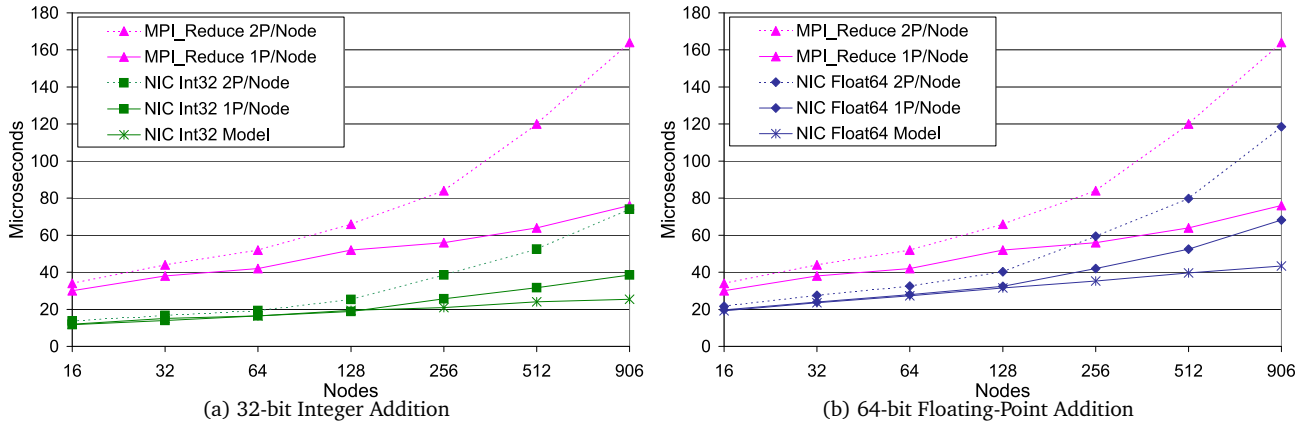
valuable since opportunities to run tests on large-scale machines are hard to come by.

Second, note that because of the high susceptibility to computation costs, the degree of the f-nomial tree may make a significant difference in the latency of the reduction. Intuition suggests that expensive computation should be spread among as many processors as possible, implying that efficient algorithms will tend to produce low-degree trees for problems that require much computation. Reassuringly, that is what is observed in the plots. Small vectors, which require less processing time, lead to curves that are essentially flat for the degrees tested, while larger vectors tend to heavily favor lower-degree trees: compare the one-element curve to the eight-element curve. On the other hand, for reduction operations simpler than floating-point addition, it pays more dividends to use higher-degree trees to save on the relatively more costly communication. Once again, because the host processor is so much faster, such drastic latency variation would not be observed as the degree of the tree is varied in host-based reductions. Efficient host-based trees are effectively independent of both the vector size and the computation being performed. NIC-based reductions are clearly not. Forutnately, since the model accurately predicts actual performance, one may use it to determine the best degree f for a particular reduction problem. For example, looking at the figure, the model correctly tells us that a degree of 4 is best for 64-bit floating-point addition of 1-element vectors, and a degree of 2 is best for 2, 4, and 8-element vectors on this 31-node system.

6.3 Reduced Latency

We timed the latencies for host-based and NIC-based reduction over a variety of operations and data sizes, using both one and two processes per node. In all measurements we consider a 4-nomial tree, which provides a good performance trad-off for the configurations used in the experiments (see Appendix 8). We show the single-element vector results obtained for host-based and NIC-based 32-bit integer addition in Figure 11(a) and 64-bit floating-point addition in Figure 11(b). The NIC-based curves scale considerably better than the host-based results. Indeed, as one may infer from the 32-bit integer addition plot, our NIC-based implementation was able to perform simple integer reductions in about half the time it takes the host to do so. Further, even while incurring the expensive cost of emulating floating-point addition on a much slower processor, our NIC-based implementation was able to substantially improve the host-based reduction. When reducing over 906 nodes, we were able to obtain latencies as low as 40 μs for integer operations and a slightly higher time of 65 μs for floating-point. In the largest configuration tested —1812 processors— our NIC-based algorithm summed single-element vectors of 32-bit integers and 64-bit floating-point numbers in 73 μs and 118 μs, respectively. These results represent respective improvements of 121% and 39% over the production-level MPI library.

We should also point out the deviation in the NIC-based latencies when involving two processes per node, as opposed to one. Unfortunately, the NIC-based curves follow the same trend which we earlier noted in the host-based latencies. As similarly discussed for the host-based case, we blame this occurrence on system noise, i.e. the kernel threads and system dæmons that interfere with the execution of the collective communication. The NIC-based implementation is subject to host-level process interference during the time it takes the distributed host processes to initiate the reduction operation. Once initiated, however, the NIC-based algorithm is able to avoid process interference throughout all of the intermediate phases while actu-



(a) 32-bit Integer Addition

(b) 64-bit Floating-Point Addition

Figure 11: Host-based and NIC-based Reduction Latencies for Single-Element Vectors

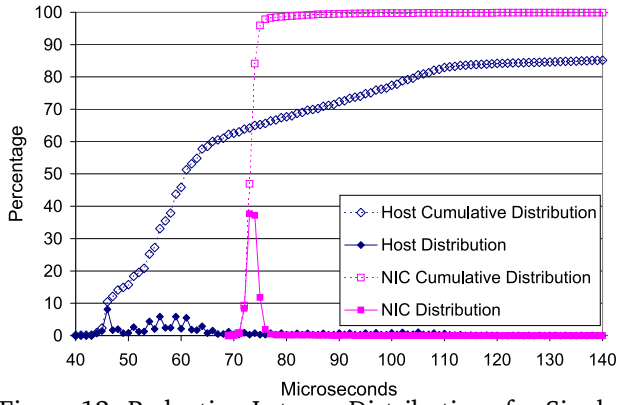


Figure 12: Reduction Latency Distributions for Single-Element 64-bit Floating-Point Addition over 900 Nodes

Reduction	Average (μs)	Std. Deviation (μs)
host-based	89.30	65.26
NIC-based	73.67	0.29

TABLE 3: Reduction Latency Statistics for Single-Element 64-bit Floating-Point Addition over 900 Nodes

ally executing the reduction. As a result, one may note that our NIC-based reduction implementation is only marginally affected by the system noise when compared to the host-based results.

6.4 Increased Consistency

The host-based MPI reduce latencies varied substantially from one invocation to another. The best times we observed were about three times better than the average time. The NIC-based results, on the other hand, were quite steady. This is related to the consistency advantage we have noted for NIC-based reductions.

To clarify this point, Figure 12 shows a distribution graph of the latencies recorded for NIC-based and host-based 64-bit floating-point addition of a single-element vector over 900 nodes. Unlike measurements for the average reduction latency, to obtain these data points, we timed 100,000 reduction invocations individually and grouped the resulting set of 100,000 times into bins of a histogram to produce a distribution.

Though at first glance the NIC-based reduction appears to take more time than the host-based reduction, the NIC-based latencies are largely contained within a sharp spike, while the host-based latencies are spread smoothly across a wide range of values. To be precise, 97% of the NIC-based reductions fall with a spread of only 4 μs, while for host-based reductions, only 57% fall within a spread of 20 μs. Indeed, a substantial number of host-based latencies extend far past the right-hand limit of the distribution graph. After pitching out the highest 1% of the samples, one arrives at the statistics in Table 3. The average host-based latency is 89 μs, while the NIC-based latency is 74 μs. The drastic, two order-of-magnitude difference in the standard deviations is perhaps most telling. This notably large contrast in consistency is quite indicative of the non-deterministic effect that process interference imposes on host-based reduction implementations. As expected, NIC-based reductions are more consistent than host-based versions on large-scale systems.

6.5 Vector Split Optimization

Earlier we noted that, while NIC-based reductions can provide reduced latency and increased consistency, they are especially sensitive to computational cost due to the slow NIC CPU. The vector split optimization is a way to counteract this shortcoming by increasing parallelism when reducing multi-element vectors.

We measured the performance of the f-nomial split

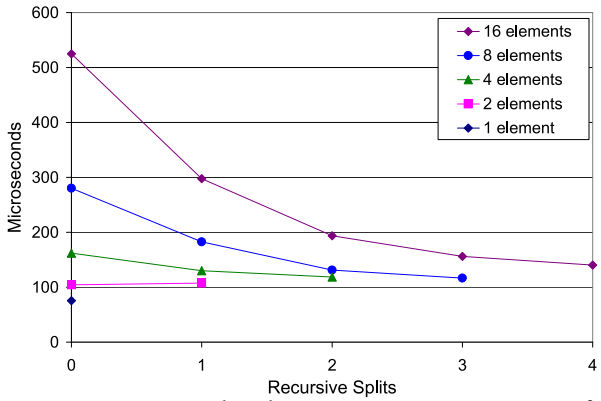


Figure 13: f-nomial Split on Various Vector Sizes for 64-bit Floating-Point Addition over 512 Nodes

algorithm for 64-bit floating-point addition on 512 nodes using various vector sizes. The results are shown in Figure 13. The value of the vector split optimization is quite pronounced. After 3 recursive splits, the 8-element latency is improved by nearly a factor of three, while for 4 recursive splits, the 16-element case is over three times faster. The trend obviously suggests the larger the vector, the better the benefit.

Although the vector split optimization enables NIC-based reductions to scale better than they otherwise would have, there is still a limit on the performance it can achieve. Note that a latency of 140 μs for a 16-element reduction may still be much more than what a host processor can provide. And interestingly, one may carefully note that the latency for a 2-element vector actually increases slightly after one split. This of course will happen if the total savings in computation over the height of the tree is less than the added communication cost of the recombine step. However, the cross-over point can be computed so as to always pick the better of the two options. Van de Geijn discusses the details in [26].

7 Related Work

Huang and McKinley were perhaps the first to realize the potential of NIC-based collectives [16]. They examined the benefits gained by implementing broadcast and barrier operations on Asynchronous Transfer Mode (ATM) network adapters to avoid the excessive processing overhead incurred throughout the protocol stack. In order to develop implementations portable across a variety of ATM hardware, they placed rigid restrictions on the processing and memory requirements of their algorithms so that even the most limited ATM devices could adequately support them. Namely, they designed algorithms which were table-driven and performed only a small number of arithmetic and logical operations while using just a few scalar variables. In addition, they considered only static communication tree structures. Yet even with such limitations, they showed that certain NIC-based collectives scaled substantially better than host-based versions due to significantly reduced message processing (software) overhead.

While the advent of zero-copy, user-level protocols lessened the dramatic improvement shown in the above work, modern cluster interconnects, such as Quadrics and Myrinet, have reduced wire and switch latencies to the point where the cost of a PCI-bus transaction is significant. Simultaneously, the processing capability and memory available on the network interface cards have increased. Thus, the concept of NIC-based collectives remains a hot topic and researchers continue to investigate more complicated collectives and algorithms.

Many researchers have considered NIC-based multicast algorithms [6, 11, 14, 16, 18, 27, 28]. Multicast can be used as a building block to implement other collectives, such as broadcasts or barriers, and thus an efficient multicast implementation is desirable. In addition, the problem is rather rich since the message size and destination set can be different with each invocation, and often one must design flow control and acknowledgment collection schemes for reliability. Each of the numerous publications put forth demonstrates a different approach, all of which have found success.

The work most closely aligned with our own is that by Buntinas and Panda [10]. They investigated the potential of NIC-based reduction on clusters interconnected with Myrinet. In particular, they modified the network drivers to implement binary AND and OR operations, as well as, integer and floating-point addition on a single 64-bit value via binomial trees. For these cases, they found that NIC-based reduction has better scalability than host-based reduction and shows performance gains in clusters as small as 8 nodes. Although they only used binomial trees, with each reduction invocation, the host processor passes an operation descriptor, including the list of communication partners, to the NIC so their implementation could be easily generalized to use other tree structures. However, they leave the investigation of other communication trees, as well as, larger reduction vectors to future work.

Our work serves two purposes. In part, this paper picks up where the previous work left off. We investigate the effect of using different tree structures and various vector sizes for an expanded set of reduction operations, as well as, an optimization for multi-element vectors. We also propose a parameterized model which can be accurately used to dynamically

select the best available tree for a given instance of a reduction. In remainder, we believe our paper is the first to show the dramatically reduced latency and increased consistency which NIC-based reductions may achieve through avoidance of host-level process interference on large-scale clusters.

8 Conclusions and Future Work

In this paper we showed that NIC-based reductions outperform host-based versions in two important ways: reduced latency and increased consistency. While NIC-based reductions are able to gain on the host by eliminating many PCI bus transactions, we discovered that the major benefit on large-scale clusters is due to decreased susceptibility to host-level process interference.

NIC-based implementations are potentially valuable, however, they don't come for free. Namely, one must deal with host-NIC synchronization overhead and perform processing on a much slower and less functional processor. Even so, modern NICs are powerful enough to handle the processing required by typical reductions and should benefit practical programs.

We presented a simple model, derived from LogP, which can be used to design efficient reduction algorithms on Quadrics. We then presented the f-nomial tree reduction algorithm, and demonstrated how to choose the best degree f for a given problem based on the model parameter values. We also added the vector split optimization to improve performance when reducing larger vectors. These issues, which may often be neglected when using host-based implementations, must be considered in order to design efficient NIC-based reductions.

The experimental results show low latency and impressive scalability. In the largest configuration tested —1812 processors— our NIC-based algorithm summed single-element vectors of 32-bit integers and 64-bit floating-point numbers in 73 μs and 118 μs, respectively. These results represent respective improvements of 121% and 39% over the production-level MPI library.

Future work will involve exploration of additional communication structures. It is also possible to optimize the software performing the reduction on the NIC, especially for floating-point operations. Finally, we intend to investigate the value of asynchronous (non-blocking) reductions, as well as, hybrid host/NIC-based reductions.

Acknowledgments

We would like to thank Robin Goldstone, Jim Garlick, Moe Jette and Ryan Braby at Lawrence Livermore National Laboratory and David Addison at Quadrics who provided us with the opportunity to run experiments on the ASCI Linux Cluster [25]. We would also like to thank our anonymous reviewers for their time and many instructive comments.

This work was supported by the U.S. Department of Energy through Los Alamos National Laboratory contract W-7405-ENG-36.

References

[1] V. Bala, J. Bruck, R. Cypher, P. Elustondo, A. Ho, C.T. Ho, S. Kipnis, and M. Snir. CCL: A Portable and Tunable Collective Communication Library for Scalable Parallel Computers. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 835–844, Cancun, Mexico, April 1994. Available from `http://citeseer.nj.nec.com/bala95ccl.html`.

[2] A. Bar-Noy, S. Kipnis, and B. Schieber. Optimal Computation of Census Functions in the Postal Model. *Discrete Applied Mathematics*, 58(3):213–222, April 1995.

[3] M. Barnett, R. Littlefield, D.G. Payne, and R.A. van de Geijn. Global Combine on Mesh Architectures with Wormhole Routing. In *Proceedings of the 7th International Parallel Processing Symposium*, pages 156–162, Newport Beach, California, April 1993. Available from `ftp://ftp.cs.utexas.edu/pub/rvdg/ipps.ps`.

[4] M. Barnett, L. Shuler, S. Gupta, D.G. Payne, R.A. van de Geijn, and J. Watts. Building a High-Performance Collective Communication Library. In *Proceedings of the Supercomputing Conference*, pages 107–116, Washington D.C., November 1994. Available from `http://www.scp.syr.edu/~jwattszSzSC94.ps.gz`.

[5] M. Bernaschi and G. Iannello. Collective Communication Operations: Experimental Results vs. Theory. *Concurrency: Practice and Experience*, 10(5):359–386, April 1998. Available from `ftp://www.grid.unina.it/pub/Papers/iannello/ps-files/ours-art/cc-exp.ps`.

[6] R. Bhoedjang, T. Ruhl, and H. Bal. Efficient Multicast on Myrinet Using Link-Level

Flow Control. In *Proceedings of the International Conference on Parallel Processing*, pages 381–390, Minneapolis, Minnesota, August 1998. Available from http://citeseer.nj.nec.com/bhoedjang98efficient.html.

[7] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawick, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, January 1995. Available from http://www.myri.com/research/publications/Hot.ps.

[8] J. Bruck, L. de Coster, N. Dewulf, C.T. Ho, and R. Lauwereins. On the Design and Implementation of Broadcast and Global Combine Operations Using the Postal Model. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):256–265, March 1996.

[9] D. Buntinas and D.K. Panda. Fast NIC-Based Barrier over Myrinet/GM. In *Proceedings of the International Parallel and Distributed Processing Symposium*, San Francisco, California, April 2001. Available from ftp://ftp.cis.ohio-state.edu/pub/communication/papers/ipdps01-NIC-barrier.pdf.

[10] D. Buntinas and D.K. Panda. NIC-Based Reduction in Myrinet Clusters: Is It Beneficial? In *Proceedings of the Workshop on Novel Uses of System Area Networks*, pages 22–33, Anaheim, California, Febuary 2003. Available from ftp://ftp.cis.ohio-state.edu/pub/communication/papers/san-2-NIC-reduction.pdf.

[11] D. Buntinas, D.K. Panda, J. Duato, and P. Sadayappan. Broadcast/Multicast over Myrinet using NIC-assisted Multidestination Messages. In *Proceedings of the Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing, High Performance Computer Architecture Conference*, pages 115–129, Toulouse, France, January 2000. Available from ftp://ftp.cis.ohio-state.edu/pub/communication/papers/canpc00-nic-multicast.pdf.

[12] M. Collette. LLNL User Briefings. In *ASCI Q LANL/HP Technical Quarterly Meeting*, Santa Fe, New Mexico, March 2003.

[13] D.E. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, San Diego, California, May 1993. Available from http://www.cs.berkeley.edu/~culler/papers/logp.ps.

[14] M. Gerla, P. Palnati, and S. Walton. Multicasting Protocols for High-Speed, Wormhole-Routing Local Area Networks. In *Proceedings of the ACM SIGCOMM Symposium*, pages 184–193, Stanford, California, August 1996. Available from http://citeseer.nj.nec.com/gerla96multicasting.html.

[15] J.R. Hauser. SoftFloat. Available from http://www.jhauser.us/arithmetic/SoftFloat.html.

[16] C. Huang and P.K. McKinley. Efficient Collective Operations with ATM Network Interface Support. In *Proceedings of the International Conference on Parallel Processing*, volume 1, pages 34–43, Bloomingdale, Illinois, August 1996. Available from ftp://ftp.cps.msu.edu/pub/crg/PAPERS/icpp96.ps.gz.

[17] D.J. Kerbyson, H.J. Alme, A. Hoisie, F. Petrini, H.J. Wasserman, and M. Gittings. Predictive Performance and Scalability Modeling of a Large-Scale Application. In *Proceedings of the Supercomputing Conference*, Denver, Colorado, November 2001. Available from http://www.sc2001.org/papers/pap.pap255.pdf.

[18] R. Kesavan and D.K. Panda. Optimal Multicast with Packetization and Network Interface Support. In *Proceedings of the International Conference on Parallel Processing*, pages 370–377, Bloomingdale, Illinois, August 1997. Available from ftp://ftp.cis.ohio-state.edu/pub/communication/papers/icpp97-packet_mcast.ps.Z.

[19] C. E. Leiserson. Fat-trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, C-34(10):892–901, October 1985.

[20] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, January/Febuary 2002. ISSN 0272-1732. Available from http://www.computer.org/micro/mi2002/pdf/m1046.pdf.

[21] Fabrizio Petrini, Darren Kerbyson, and Scott Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *Proceedings of SC2003*, Phoenix, Arizona, November 10–16, 2003. Available from `http://www.c3.lanl.gov/~fabrizio/papers/sc03_noise.pdf`.

[22] Quadrics Supercomputers World Ltd. *Elan Programming Manual*, 2nd edition, December 1999.

[23] Quadrics Supercomputers World Ltd. *Elan Reference Manual*, 1st edition, January 1999.

[24] Quadrics Supercomputers World Ltd. *Elite Reference Manual*, 1st edition, November 1999.

[25] M. Seager. Planned Machines: ASCI Purple, ALC and M&IC MCR. In *Proceedings of the 7th Workshop on Distributed Supercomputing*, Durango, Colorado, March 2003. Available from `http://www.cs.sandia.gov/SOS7/presentations/seager.ppt`.

[26] R.A. van de Geijn. On Global Combine Operations. Technical Report CS-91-129, Available from `ftp://ftp.netlib.org/lapack/lawns/lawn29.ps`, April 1991.

[27] K. Verstoep, K. Langendoen, and H. Bal. Efficient Reliable Multicast on Myrinet. In *Proceedings of the International Conference on Parallel Processing*, volume 3, pages 156–165, Bloomingdale, Illinois, August 1996. Available from `ftp://ftp.cs.vu.nl/pub/amoeba/orca_papers/icpp96.ps.Z`.

[28] W. Yu, D. Buntinas, and D.K. Panda. High Performance and Reliable NIC-based Multicast over Myrinet/GM-2. In *Proceedings of the International Conference on Parallel Processing*, page to be presented, Kahosiung, Taiwan, October 2003.

Appendix A

This appendix provides a psuedo/C code listing of how to initialize and use an f-nomial tree communication structure for reduction operations.

We assume the processes in the collective group are numbered with IDs starting from 0. Without loss of generality, we assume that process ID 0 is the root of the reduction. (The algorithm can be used to describe the communication structure for a general root if one performs a logical remapping by shifting all process IDs cyclically down by an amount equal to the ID of the root.)

Each process must first initialize the f-nomial tree communication data structure by calling init_fnomial_tree() with the degree f, the number of processes involved, and the local process ID, as well as, a pointer to an empty communication data structure to be filled-in by the function. Having done this, a process may use the communication structure to carry out an f-nomial reduction operation as shown in reduce_fnomial_tree(). One must pass in a pointer to the local data, a pointer to a memory segment to be used as receive buffers, the data size, and the filled-in communication structure.

Reducing by an f-nomial Tree:

```
procedure reduce_fnomial_tree

  // Inputs:
  //   data_local:   pointer to local data
  //   data_remote:  pointer to memory segment for receive buffers
  //   data_size:    size of reduction vector
  //   fnomial_tree: filled-in communication data structure

  // Temporaries:
  uint data_temp = 0;
    // index into receive buffers
  uint num_recvs;
    // convience variable of the number of receives in a phase

  // Process any children
  for(phase = 0; fnomial_tree->array_receives[phase] != 0; phase++)
  {
    // number of messages we'll receive from children of this phase
    num_recvs = fnomial_tree->array_receives[phase];

    // wait for messages from children
    wait for messages to fill buffers at
    [data_temp, data_temp + num_recvs - 1] * data_size + data_remote;

    // reduce the data from children with local data
    reduce the data in buffers at
    [data_temp, data_temp + num_recvs - 1] * data_size + data_remote
    with             data_local
    store result in  data_local;

    // point to head of receive buffers for next phase
    data_temp += num_recvs;
  }

  // Send reduction data to parent
  send       data_local
  to         fnomial_tree->id_parent
  at buffer  fnomial_tree->child_num * data_size + data_remote;

end procedure
```

Initializing an f-nomial Tree:

```
procedure init_fnomial_tree
  // Maps process ID == 0 as root.

  // Inputs:
  //   degree:   degree, f, of the f-nomial tree
  //   id_count: number of processes involved
  //   id_local: local process ID, counts from 0

  // Outputs:
  //   fnomial_tree: communication data structure
  // i.e.,
  //   fnomial_tree->array_receives[]:
  //      number of children during each phase
  //   fnomial_tree->id_parent:
  //      parent process ID (equals id_local for root)
  //   fnomial_tree->child_num:
  //      which number child we are to our parent

  // Temporaries:
  uint phase  = 0;
    // which communication phase we are computing
  uint stride = 1;
    // how many process IDs are skipped between
    // adjacent children within a phase

  // Initialization
  // assume we are the root
  fnomial_tree->id_parent = id_local;

  // While we haven't covered the entire tree...
  while(stride < id_count)
  {
    // assume we have no children in this phase
    fnomial_tree->array_receives[phase] = 0;

    // If we are a parent in this phase...
    if(FLOOR(id_local / stride) MOD degree == 0)
    {
      // For each of our (possible degree-1) children...
      for(uint index = 0; index < (degree-1); index++)
      {
        // If the possible child really exists,
        // increase number of receives for this phase
        if(id_local + (index+1) * stride < id_count)
          fnomial_tree->array_receives[phase]++;
      }

    // Else, we must be a child in this phase...
    } else {
      // Note our parent's id
      fnomial_tree->id_parent =
        FLOOR(id_local / (stride * degree)) * (stride * degree);

      // and which number child we are to our parent
      fnomial_tree->num_child =
        phase * (degree-1) + (FLOOR(id_local / stride) MOD degree) - 1;

      // After determining our parent, our part is done
      break;  // out of the while
    }  // end if

    // Move on to the next phase of the tree
    stride *= degree;
    phase++;
  }  // end while

end procedure
```

Appendix B

This appendix illustrates the analysis to express the best degree f for an f-nomial tree based on the model derived in Section 5 for a full tree, i.e. those trees for which $\log_f P$ is an integer.

Basically, we desire to find that value of f which minimizes the following expression:

$$\begin{aligned} T^{full}_{fnomial}(P, f) &\approx C + T_{serial}(f) \cdot \log_f P \\ &\approx C + [L + (f-1) \cdot (r+c)] \cdot \log_f P \end{aligned}$$

To do so, we first take the derivative of $T^{full}_{fnomial}(P, f)$:

$$\begin{aligned} &\frac{\partial}{\partial f} T^{full}_{fnomial}(P, f) \\ &\approx \frac{\partial}{\partial f}\{C + T_{serial}(f) \cdot \log_f P\} \\ &= \frac{\partial T_{serial}(f)}{\partial f} \cdot \log_f P + T_{serial}(f) \cdot \frac{\partial \log_f P}{\partial f} \\ &= \frac{\partial [L + (f-1) \cdot (r+c)]}{\partial f} \cdot [\ln P / \ln f] \\ &\quad + [L + (f-1) \cdot (r+c)] \cdot \frac{\partial [\ln P / \ln f]}{\partial f} \\ &= (r+c) \cdot [\frac{\ln P}{\ln f}] \\ &\quad + [L + (f-1) \cdot (r+c)] \cdot [-\frac{\ln P}{f \cdot \ln^2 f}] \\ &= (r+c) \cdot \frac{\ln P}{\ln f} - [L + (f-1) \cdot (r+c)] \cdot \frac{\ln P}{f \cdot \ln^2 f} \end{aligned}$$

Then, we set this expression equal to zero and isolate f:

$$\begin{aligned} (r+c) \cdot \frac{\ln P}{\ln f} - [L + (f-1) \cdot (r+c)] \cdot \frac{\ln P}{f \cdot \ln^2 f} &= 0 \\ (r+c) \cdot \frac{\ln P}{\ln f} &= [L + (f-1) \cdot (r+c)] \cdot \frac{\ln P}{f \cdot \ln^2 f} \\ f \cdot \ln f \cdot (r+c) &= L + (f-1) \cdot (r+c) \\ f \cdot \ln f &= L/(r+c) + (f-1) \\ f \cdot \ln f - f &= L/(r+c) - 1 \\ f \cdot (\ln f - 1) &= L/(r+c) - 1 \end{aligned}$$

The above expression gives the best value of f to use with L, r, and c. It is a transcendental equation, so one must solve for f numerically by finding the intersection of $f \cdot (\ln f - 1)$ with the function $L/(r+c) - 1$, which is constant once c is decided by the operation and data size for a particular problem. We set $L = 2.10$ μs and $r = 0.42$ μs, the same values listed in Section 6, and ploted the intersection of these two functions for various values of c in Figure 14.

Only integers $f \geq 2$ produce valid f-nomial trees. For intersection points which are between two integers, one must choose the best of the two. For the values used for L and r note that the best degree may fall anywhere in the range $[2, 6]$ depending on the value of c. The upper bound is reached somewhere between 5 and 6 when $c = 0$. Note when $f = 6$, a parent node receives 5 messages so that the reception costs accumulate to exactly balance the message latency. As computation cost increases, the best degree decreases.

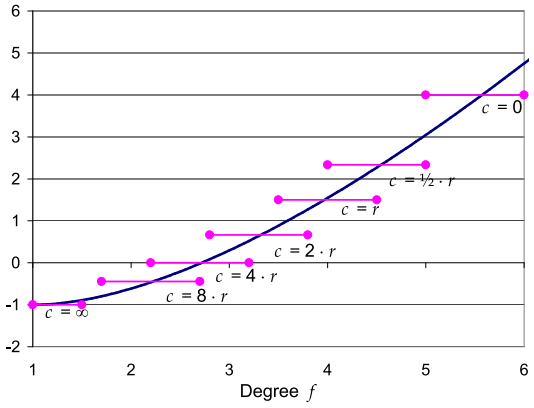


Figure 14: Plot of $f \cdot (\ln f - 1)$ and $L/(r+c) - 1$ for $L = 2.10$ μs, $r = 0.42$ μs, and various c

It is interesting to consider the range $[1, 2]$. Values of f smaller than 2 do not produce meaningful f-nomial trees, however, if we take some number in this range, say $f = 1.5$, and plug back it back into $T^{full}_{fnomial}(P, f)$ we get:

$$\begin{aligned} T^{full}_{fnomial}(P, 1.5) &\approx C + [L + ((1.5) - 1) \cdot (r+c)] \cdot \log_{(1.5)} P \\ &= C + [L + 0.5 \cdot (r+c)] \cdot \log_{1.5} P \end{aligned}$$

When compared to binomial trees, these values of f tend to construct trees which have more communication phases, since $\log_{1.5} P < \log_2 P$. They do so in return for a reduced amount of reception and computation costs, $0.5 \cdot (r+c)$ instead of $(r+c)$. Thus, trees in this range wish to suffer extra communication in order to save on computation, so this is the range in which optimizations like the vector split are valuable.

This analysis applies only to full f-nomial trees; it can not accurately be applied to arbitrary trees. However, it helps to build our intuition and establishes reasonable expectations by using actual parameter values, so it is worthwhile to study.