$\log_n P$ and $\log_3 P$: Accurate Analytical Models of Point-to-Point Communication in Distributed Systems

Kirk W. Cameron, *Member*, *IEEE*, Rong Ge, *Student Member*, *IEEE*, and Xian-He Sun, *Senior Member*, *IEEE*

Abstract—Many existing models of point-to-point communication in distributed systems ignore the impact of memory and middleware. Including such details may make these models impractical. Nonetheless, the growing gap between memory and CPU performance combined with the trend toward large-scale, clustered shared memory platforms implies an increased need to consider the impact of middleware on distributed communication. We present a general software-parameterized model of point-to-point communication for use in performance prediction and evaluation. We illustrate the utility of the model in three ways: 1) to derive a simplified, useful, more accurate model of point-to-point communication in clusters of SMPs, 2) to predict and analyze point-to-point and broadcast communication costs in clusters of SMPs, and 3) to express, compare, and contrast existing communication models. Though our methods are general, we present results on several Linux clusters to illustrate practical use on real systems.

Index Terms—Distributed systems, middleware, performance modeling and prediction.

1 Introduction

For many scientific distributed applications, the cost of communication dominates overall execution time. Point-to-point communication requires moving data from the source process' local memory to the target process' local memory. Algorithm designers and application programmers use explicit communications to specify source and target buffer locations and amount (or type) of data transferred. For example, given source (B[i]) and target (A[i]) array elements of type double, $\mathrm{A[i]} = \mathrm{B[i]}$ specifies an explicit data transfer of 8 bytes within a single address space on many 32-bit systems. Explicit communications across address spaces (e.g., MPI send communications[1]) require additional information to identify source and target processes.

An *explicit communication* is an abstraction for a series of implicit communications. In a typical load-store architecture for a loop assigning $\mathrm{A[i]} = \mathrm{B[i]}$ for $0 < i < n - 1$, a series of block transfers between memory hierarchy levels brings data from main memory to cache to registers to complete this task. The user explicitly specifies source and target locations, but the assignment implicitly causes movement of data from memory to registers and back to memory. *Implicit communications* are the transmissions that occur "behind the scenes" to complete an explicit communication. This requires hardware (e.g., data replication from memory to cache) and system software support (e.g., demand paging) when the data does not reside in memory. The details of the implicit communication are hidden to ease programming efforts.

For message passing in a distributed system, sends and receives are explicit communications accomplished using implicit communication mechanisms provided in middleware. Communication middleware is systems software or libraries designed to support abstraction in explicit communications. In distributed communication using MPI, this includes the costs of operating system overhead and MPI software. For example, an MPI_Send() of a strided[2] message describes a point-to-point transfer explicitly. To ensure that packed data is actually sent across the network, MPI middleware performs a series of implicit communications to complete the transfer (i.e., packing strided data at the source and unpacking data by stride at the target). Some transmissions occur in user space, others via the operating system in kernel space.

Models of communication cost must balance abstraction and accuracy. The PRAM model [11] assumes unit cost for implicit communication. Optimal algorithm design using PRAM minimizes the number of explicit communications. Unfortunately, the "flat" cost in the PRAM model does not accurately reflect the characteristics of today's complicated, multilevel communication systems. This can lead to algorithm designs that perform poorly on real systems. Succeeding variants of PRAM [1] introduce complexity to

1. In this paper, we focus on message passing (e.g., MPI) due to its popularity in large-scale clusters. Our approach is general and applicable to shared memory communication as well, though our parameter measurement techniques may require extension (e.g., OpenMP).

- *K.W. Cameron and R. Ge are with the Department of Computer Science, 212 Knowledge Works II Bldg, Corporate Research Center, Virginia Tech, Blacksburg, VA 24061. E-mail: {cameron, ge}@cs.vt.edu.*
- *X.-H. Sun is with the Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616. E-mail: sun@iit.edu.*

Manuscript received 7 Sept. 2005; revised 19 Apr. 2006; accepted 26 July 2006; published online 22 Jan. 2007.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0306-0905.

2. The words "strided," "distributed," "unpacked," and "noncontiguous" are interchangeable. So are "unit-stride," "not distributed," "packed," and "contiguous." Each refers to the degree of spatial locality in message data in this paper.

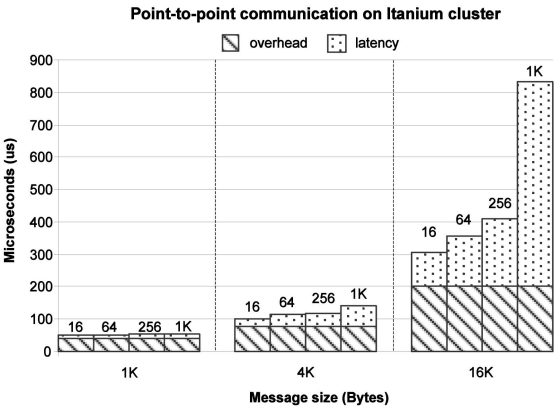


Fig. 1. Half round-trip time for point-to-point communication. Overhead is the communication cost of nonstrided message transfers. Latency is the additional time for strided message transfers. Latency can dominate transmission costs for strided communications.

improve accuracy considering additional application (and rudimentary system) characteristics such as arbitration of reads and writes, yet do not typically consider the effects of hardware or middleware.

Hardware-parameterized models ignore the increasing effects of middleware on communication cost. The LogP [8], [9] model uses hardware-specific measurements and an analytical machine model to predict [19] and analyze implicit communication more accurately than PRAM and at finer granularity than models of aggregate communication such as BSP [21]. The hardware costs of small message injection overhead (o), message repeat rate (g), and network latency (L) define the analytical model. Succeeding variants of LogP [3], [7], [12], [14], [16] reflect evolving architectures that diverge from the MPP designs (e.g., CM-5) upon which LogP is based. Examples include:

1. practical elimination of the repeat rate cost (g) parameter,
2. extension to long messages by adding the G parameter (LogGP [3]),
3. extension to active messages by adding the C parameter (LoPC [12], LoGPC [16]), and
4. incorporation of synchronization costs by adding the S parameter (LoGPS [14]).

There are compelling reasons to incorporate middleware costs into models of distributed communication. First, middleware can dominate communication cost. For example, Fig. 1 shows the software-parameterized costs of point-to-point communication on an Itanium cluster. The lower stack of each bar (overhead) is the total unit-stride transfer cost in microseconds between source and target nodes for various data message sizes (1K, 4K, and 16K bytes). This cost, an upper bound of the hardware transfer cost, does not change with a message's stride size (16, 64, 256, and 1K bytes). The communication cost is quickly dominated by the upper stack of each bar (latency) or the additional cost due to strided memory accesses. The impact of latency on communication varies with data size, data stride, and system implementation. These effects are ignored under hardware parameterized models.

Second, more accurate models of communication encourage efficient algorithm design. Existing hardware-parameterized models of communication ignore middleware as a potential performance bottleneck. This implies algorithms designed may be less than optimal. For example, an algorithm designed under LogP has no incentive to reduce the number of strided communications. Nonetheless, Fig. 1 shows such communications can easily grow to 4x the cost of unit-stride communications. Additionally, more accurate cost models encourage overlap in communications. The latency costs in Fig. 1 have the potential for overlap depending on system design, such as nonblocking memory accesses or aggressive prefetching and algorithm characteristics.

Since existing parallel programs often do not exhibit good performance on distributed systems, a large class of scientific applications (e.g., simulations) stands to benefit from the development of predictive models of distributed communication that incorporate system software characteristics and encourage reductions in middleware communication cost. For example, the 3D FFT applications described in Section 4.3.4 can spend as much as 50 percent of their execution time in middleware communication.

In this paper, we describe our approach to separate the costs of unit-stride and strided accesses at various points along the communication critical path. In Section 2, we discuss our general model of point-to-point communication $(\log_n P)$ which is accurate and robust yet cumbersome to use in practice. Hence, we also show how to apply the general model to existing clusters to create a practical model of point-to-point communication that incorporates the significant costs of middleware $(\log_3 P)$. Section 3 describes our experimental platforms and measurement techniques. Section 4 presents the practical application of our model $(\log_3 P)$ on various platforms to obtain parameters, analyze communication costs, and predict performance. In Section 4.3.4, we show how $\log_3 P$ can be used in algorithm design to optimize performance. Section 5 describes related work showing how the $\log_n P$ model can be used to derive most existing models of point-to-point communication prevalent in the literature. In this section, we also compare and contrast the complexity of all related models of communication to our models. Last, we discuss conclusions, including the limitations of our approach and future directions.

2 The LOG_NP Model of Point-to-Point Communication

In this section, we present a general model $(\log_n P)$ of communication that incorporates middleware costs. Previous models of communication (e.g., LogP) have been augmented to suit architectural evolution. Our general model is designed for flexibility. To ease use, we simplify our general model to create a model $(\log_3 P)$ that reflects the current architectural trend toward parallel systems consisting of clusters of SMPs. We then use $\log_3 P$ to predict and analyze communication costs on several clusters.

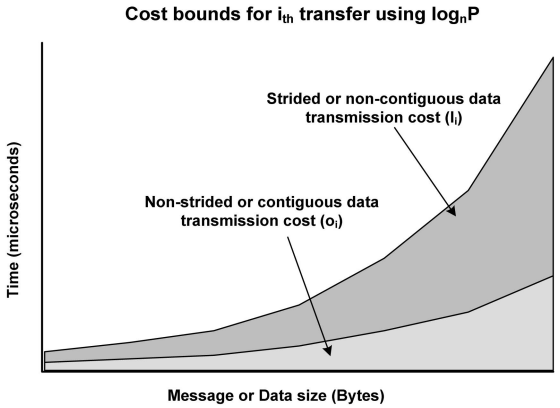


Fig. 2. Performance bounds with the $\log_n P$ model parameters. o and l are both functions of message size. l is additionally subject to variations due to stride size. l is shown for a single, fixed stride.

Fig. 2 provides an illustrative view of the succeeding discussion. We formally characterize data transfer costs using five parameters:

l: the effective latency (the letter "ell"), defined as the effective delay[3] in the transmission or reception of a strided message over and above the cost of a unit-stride transfer. The system-dependent l cost is a function of the message data size (s) under a variable stride or distribution (d). We denote this function as $l = f(s,d)$, where variable s corresponds to a series of discrete message sizes in bytes, variable d corresponds to a series of discrete stride distances in bytes between array elements, and function f is the additional time for transmission in microseconds over and above the unit-stride cost for variable message data size s and stride d. This cost is bounded above by the cost of data transfers without computational overlap and bounded below by 0 or full computational overlap.

o: the effective overhead, defined as the effective delay in the transmission or reception of a unit-stride message. The system-dependent o cost is a function of the message data size (s) under a fixed unit-stride (i.e., when $d = 1$ *array element*). We denote this function as $f(s,d) = f(s,1) = o$, where variable s corresponds to a series of discrete message sizes in bytes,[4] variable $d = 1$ *array element* corresponds to the unit-stride between adjacent array elements, and function f is the time for transmission in microseconds for variable message data size s and stride $d = 1$ *array element*. This average, unavoidable overhead represents the best case for data transfer on a target system. This cost is bounded below by the data size divided by the hardware bandwidth.

g: the gap, is unit-stride point-to-point effective communication cost, including additional system delays. o is the cost of a unit-stride point to point transfer without resource contention. $g - o$ is the additional cost of contention. g provides flexibility for expansion of our model to consider effects of multiple messages not covered by o and l. For now, we assume this parameter has no impact on communication cost, effectively using $o = g$. At times, we use $max(o,g)$ for completeness, but this cost simply reduces to o under our assumption.

n: the number of implicit transfers along the data transfer path between two endpoints of communication. Endpoints can be as simple as two distinct local memory arrays or as complex as a remote transfer between source and target memories across a network. o_i or l_i are the average costs for the ith implicit transfer along the data transfer path where $0 < i < n-1$. As n increases, so does the accuracy and complexity of the model of implicit communication.

P: the number of processor/memory modules. This parameter is used when determining the cost of collective communications estimated as a series of point-to-point transfers.

All parameters are measured as multiples of processor clock cycles converted to microseconds. Conversion to rates of cycles or microseconds per byte is straightforward. In our discussion, we assume typical load/store architectures with hierarchical memory implementations. Clusters may be composed of single processor or multiprocessor nodes communicating on a shared bus or through a network interface card (NIC) attached to interconnect. Our analyses and predictions are at the application level, so nondeterministic characteristics of memory access delay at the microarchitecture level are not considered. We assume deterministic access delay and use minimums of average values as inputs to our model. This assumption is validated if our predictions are accurate. In this paper, our predictions for common collective communications are typically within 3 percent. As is customary, we assume the receiving processor may access a message only after the entire message has arrived. At any given time, a processor can either be sending or receiving a single message.

2.1 Usage of the Model

$\log_n P$ estimates point-to-point communication cost (or time, T) as:

$$T = \sum_{i=0}^{n-1} \{\max(o_i, g_i) + l_i\}. \quad (1)$$

T is the cost of an explicit communication consisting of n implicit transfers numbered 0 to $n-1$. Any transfer has data characteristics of size (s) and stride (d). o_i is the cost for unit-stride transfers for implicit communication number i. o_i is a function of the size (s) and unit-stride $(d = 1)$, so $o_i = f(s,d)_i = f(s,1)_i$. We assume $g_i - o_i = 0$ since we do not consider system contention in this work. l_i is the additional cost for strided transfers for implicit communication number i. l_i is a function of the size and stride of a message, so $l_i = f(s,d)_i$. Equation (1) can be expressed as:

$$T = \sum_{i=0}^{n-1} \{o_i + l_i\} = \sum_{i=0}^{n-1} \{f(s,1)_i + f(s,d)_i\}. \quad (2)$$

The main drawback to using $\log_n P$ directly as described in (2) is complexity. This approach allows consideration of

3. Our model parameters do not explicitly model communication overlap such as prefetching. However, the use of "effective" measurements allows inclusion of these effects provided overlap occurs in steady state communication. For example, when we measure source node overhead, prefetching reduces the effective overhead cost. In this case, we measure the repeatable, steady-state effective rate per byte for transmissions.

4. In our results, we refer to unit stride for doubles as $d = 8$ bytes. We use $d = 1$ *array element* in this discussion to maintain independence from an underlying architecture or array element type.

costs previously ignored by hardware models of communication. It is also flexible enough to apply to any point-to-point transfer. However, such complexity can prohibit practical use. Too many parameters may keep nonexperts from drawing conclusions and isolating communication cost bottlenecks. Too few parameters do not provide enough information. Additionally, the parameters must be measurable. We analyze the complexity of our model in relation to others in Section 5.

2.2 The $\log_3 P$ Model of Point-to-Point Communication

Current debate notwithstanding,[5] the convergence of distributed architectures to clusters of SMPs implies we can make some assumptions to reduce the complexity of the communication model described by (2). We assume $o = g$, and $n = 3$ points of implicit communication. The first reduction corresponds to ignoring the extraneous effects of multiple messages competing in the system. This is reasonable as our initial intent is to model point-to-point and collective communication operations, not nondeterministic effects of resource contention. This assumption is validated by the resulting accuracy of our techniques. The second assumption is a starting point for our analyses. If a further breakdown of costs is necessary, we can refine the model by increasing $n > 3$ and applying (2).

For $n = 3$, (2) reduces to:

$$T = \sum_{i=0}^{2} \{f(s,1)_i + f(s,d)_i\} = \{f(s,1)_0 + f(s,d)_0\} + \{f(s,1)_1 + f(s,d)_1\} + \{f(s,1)_2 + f(s,d)_2\}, \tag{3}$$

$$T = \{o_0 + l_0\} + \{o_1 + l_1\} + \{o_2 + l_2\}. \tag{4}$$

Equation (4) describes implicit communication points 0, 1, and 2, respectively. Each implicit point is broken into costs $(o + l)$ under our model. The implicit points correspond to hops between endpoints as follows: 0) Middleware communication within user space to the network interface buffer; this includes the effects of hierarchical memory. 1) Communication across the interconnect. 2) Middleware communication from the network interface buffer to user space; this includes the effects of hierarchical memory. The number of implicit communications may be larger (e.g., across the memory hierarchy); we abstract them into three single points for simplicity. This assumption is validated by the resulting usefulness of our techniques for performance prediction and analysis.

We combine source and target overhead—common practice in communication models such as LogP. It is practically cumbersome to measure and separate source and target overheads individually. Additionally, since any point-to-point communication inherently requires source and target overhead, separation as such doesn't provide information of enough interest to warrant the complexity of separate parameters in the model. On the other hand, maintaining the separation of costs as overhead (unit-stride cost) and latency (additional strided data costs) seems warranted given the previous discussion regarding middleware costs and the impact of memory. Hence, we reformulate (4):

$$T = \{o_0 + o_2\} + \{l_0 + l_2\} + \{o_1 + l_1\}. \tag{5}$$

More precisely, though the costs at points $i = 0$ and $i = 2$ are distinct, for simplicity we group by pairs and provide more meaningful subscripts: $middleware\ overhead = o_{mw} = \{o_0 + o_2\} = \{f(s,1)_0 + f(s,1)_2\}$ and $middleware\ latency = l_{mw} = \{l_0 + l_2\} = \{f(s,d)_0 + f(s,d)_2\}$ under the preceding simplifications. For point $i = 1$ or network transfer cost, $o_1 = f(s,1)_1$ is a linear function of a fixed packet size transfer cost across the interconnect and $l_1 = f(s,d)_1$ is assumed to be zero since packets are unit-stride and fixed size. The resulting linear function is the $network\ overhead = o_{net} = f(s,1)_1$. The $\log_3 P$ model can be expressed semantically as:

$$T = o_{mw} + l_{mw} + o_{net}. \tag{6}$$

3 Experimental Details

3.1 Platforms

In the next section, we provide detailed results for an IA-64 Linux cluster referred to as Titan. Each node of Titan has two 800 MHz Intel Itanium I processors running Red Hat Linux 7.1. Each processor is equipped with L1, L2, and L3 caches of 32 KB, 96 KB, and 4 MB, respectively. Each node has 2 GB ECC SDRAM and nodes are connected using Myrinet 2000 technology.

We are able to measure our model parameters on nearly any platform. Measurement of model parameters allows accurate prediction of communication algorithm costs. Our analysis techniques are general, but, at times, system implementation details are needed to explain performance trends. For example, to best analyze distributed communication costs, we need some understanding of the buffer transmissions performed by the middleware. The open-source characteristics of MPICH allow us to study implementation specifics. This was a significant motivation for the use of Linux-based platforms. The machines studied use a version of the MPICH implementation of the MPI standard. For analysis on nonopen source platforms, policy decisions must be inferred if not provided directly to the public. All of the following discussions refer to MPICH.

3.2 Techniques

We created a set of micro benchmarks using a modified version of mpptest [11]. The mpptest tool provides platform independent, reproducible measurement of message passing experiments such as ping-pong and memory copy and is part of the MPICH distribution. It can be used to benchmark systems for determining MPICH platform dependent parameters. To ensure reproducible results, we do the following:

1. preload data sets to "warm up" the cache so we do not measure start-up costs,
2. repeat an explicit communication operation n times (n is an input parameter, usually > 100) and take the

5. US supercomputing strategy is in flux. Evolving systems will be large and complex. Since experts cannot agree on which fundamental technology should dominate, it is likely systems will be diverse as well.

average as one sample to ensure we are measuring a steady state,
3. take m samples (m is an input parameter set to 100) and choose the minimum of these m samples as the measured value to select the best case transmission, and
4. repeat this full set of $m \times n$ measurements at least two different times at varied hours to ensure system loads do not perturb results.

The mpptest tool provides various functions of use in our experiments. Specifically, we control the message size and type, call type (e.g., blocking or nonblocking send), and the precision or tolerance desired. We modified the tool to provide further granularity such as specifying the stride of a message. We use the resulting control to vary the data type (char, integer, and double), message size (s), and stride (d). The modified tool[6] is portable to all systems under study (and any system running MPI). For simplicity, we only present results for data type double and common communication functions MPI_Send and MPI_Recv unless mentioned explicitly. For all measurements related to strided data, we consider only regular access patterns and, if using derived data types, use MPI_Type_vector.

4 Experimental Results

Our experimental objectives are threefold. First, we verify how we obtain parameter values. Second, we use our model to study the communication costs of high-end clusters. Third, we apply our model to predict communication cost for use in algorithm design.

4.1 Parameter Evaluation

Before we evaluate our model parameters, we discuss the MPICH implementation for the common send operation (MPI_Send) on a Linux cluster. Fig. 3 provides an abstract flow chart for the implicit communication on the sender side for long messages. Unit-stride messages do not require packing. Strided data is packed into a contiguous buffer and sent across the network to its destination. In either case, the "send contiguous" function is invoked.

At this point, one of three size-dependent protocols is selected to ensure good performance. Messages are classified as short $(s < 1\ Kbytes)$, long $(1\ Kbytes < s < 128\ Kbytes)$, and very long $(s > 128\ Kbytes)$. For short and long messages, no message handshakes or acknowledgments establish communication as the data has been saved in an intermediate local buffer on the sender side.[7] For very long messages, handshakes or acknowledgments are required. We denote two cases for MPI_Send:

Case 1: Same source and destination (send to self).

Short/Long messages. Data copy from send buffer to intermediate buffer to receive buffer.

Very long messages. Streaming data copy between send buffer and receive buffer.

6. Our implementation is open-source and available (cameron@vt.edu) upon request.

7. Astute readers will notice that we simplify this discussion slightly by ignoring tiny buffer characteristics which are ignored in our MPICH implementation. While analyses could be extended to include these details, this level of granularity is beyond the scope our work.

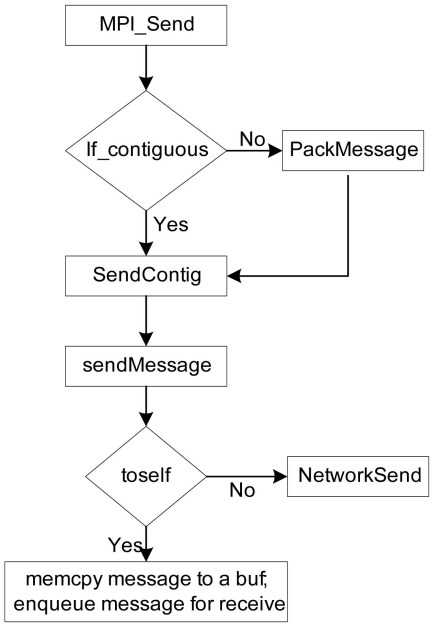


Fig. 3. Sender distributed communication. This flow chart shows MPICH implementation of a blocking MPI_SEND for long messages. Any strided message is packed prior to transmission. Messages sent in shared memory (or to self) avoid the use of sockets. Messages sent across the network use sockets and require additional size-dependent buffering.

Case 2: Different source and destination (remote send).

All messages. Data sent in socket to destination.

Fig. 4 shows our model parameters for the sender and receiver (generally) and costs for long messages (16 Kbyte) and long, strided messages (1 Kbyte stride) on the IA-64 Linux cluster (Titan). Figs. 4a and 4b show costs for unit-stride send and receive pairs. For simplicity, we use symmetrical parameters (e.g., the values for o_0 and o_2 from (5) are averaged for sender and receiver and expressed as a single value such as $o_0 = o_{mw}/2$ and $o_2 = o_{mw}/2$). The o_{mw} term used in later graphs refers to the total overhead ($o_{mw} = o_{mw}/2 + o_{mw}/2$) on sender and receiver as described by (6). Figs. 4c and 4d show the additional latency for strided communications (see the "pack message" in Fig. 3). The l_{mw} term is the total middleware latency on sender and receiver. As discussed earlier, network transfer costs have $l_{net} = 0$, so it is not necessary to break o_{net} down further (since sender and receiver network latency is intuitively a single cost).

We identify each term of Fig. 4 for our model as follows: We begin by obtaining the round trip costs of contiguous transfers for two cases (send to self or 0 sends to 0 and remote send or 0 sends to 1, respectively) as a function of $size = s$ (denoted as "(s)"): send to self $(2T_{0,0(s)})$ and remote send $(2T_{0,1(s)})$. Next, we measure T_{mem} (the cost of memory copy) for different message sizes. We then solve the send to self equation $(T_{0,0(s)} = o_{mw}/2 + T_{mem} + o_{mw}/2)$ to obtain o_{mw}. Last, we use the remote send equation $(T_{0,1(s)} = o_{mw}/2 + o_{net} + o_{mw}/2)$ to solve for o_{net}. Both o_{mw} and o_{net} are functions of size (s). At this point, we have individual costs for Figs. 4a and 4b.

Using these costs, we perform similar comparisons to separate the costs for noncontiguous data l_{mw}. We begin by obtaining the round trip cost of noncontiguous transfers as a function of $size = s$ and $stride = d$ (denoted as "(s, d)") for

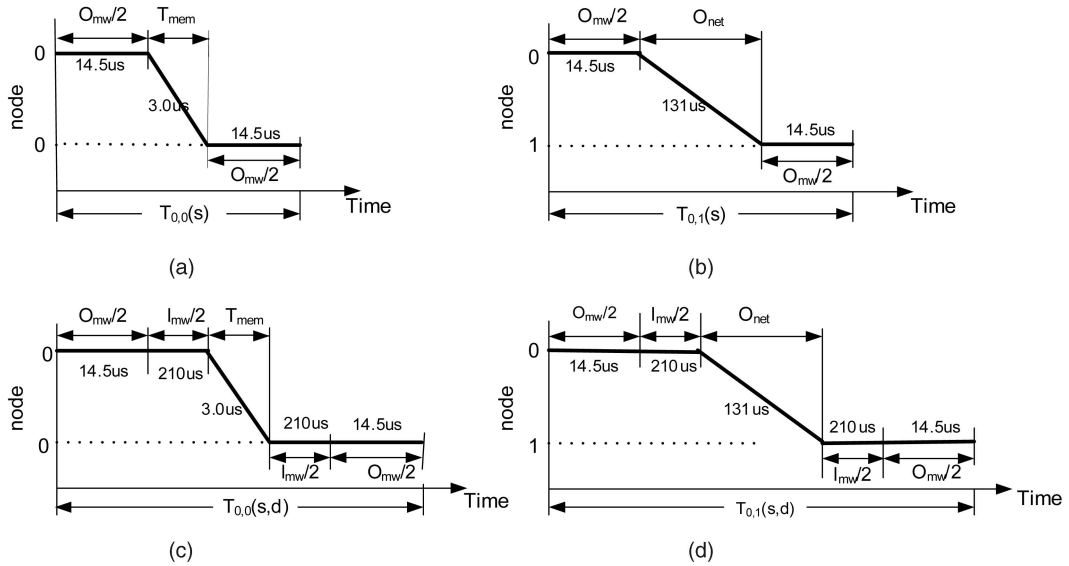


Fig. 4. Half-round trip sender/receiver communication cost. Case 1 (same source and destination or send to self) shown for nonstrided and strided costs in (a) and (c), respectively. Case 2 (different source and destination) shown for nonstrided and strided costs in (b) and (d), respectively. Actual costs (in microseconds) shown for message size of 16 Kbytes, stride of 1 Kbyte for (c) and (d) on IA-64 cluster. Note: Costs not drawn to scale for illustrative purposes. (a) Send contiguous data to self. (b) Half-round trip of contiguous data. (c) Send noncontiguous data to self. (d) Half-round trip of noncontiguous data.

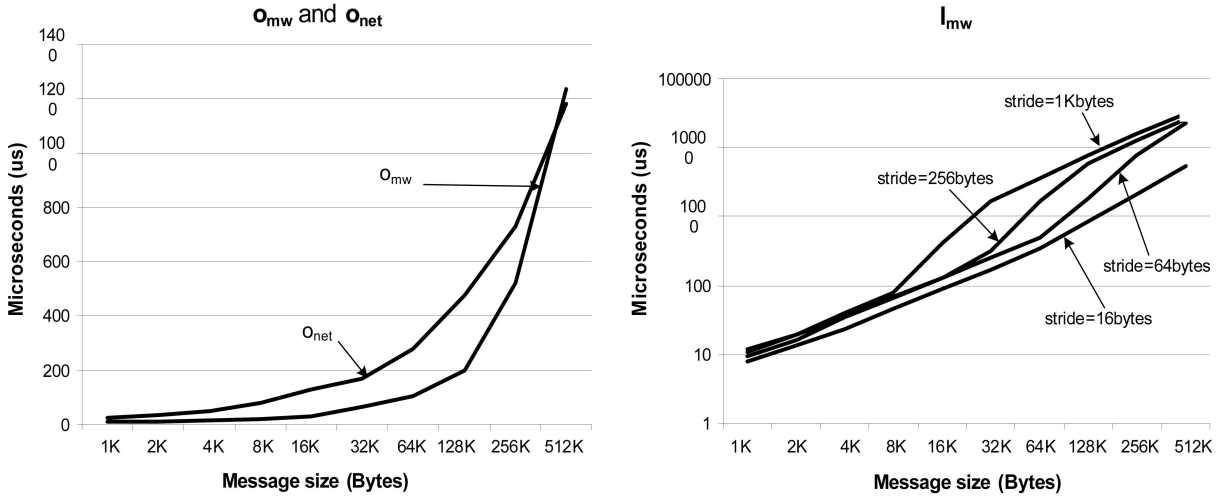


Fig. 5. Measured overhead and latency on the IA-64 (Titan) cluster. The left picture shows how middleware overhead (o_{mw}) and network overhead (o_{net}) vary with size. Though the trends are linear as expected (x axis is in log), the slopes are different, indicating trade-offs occur at some crossover point that varies with data size. The right picture illustrates how the latency (additional costs for strides) varies substantially (x and y axes are in $\log_2$) with size and stride. The varied magnitude of this cost implies crossover points in the left figure will vary with size and stride.

send to self $(2T_{0,0(s,d)})$. Next, we use our previously measured and calculated costs to solve the send to self equation for noncontiguous data $(T_{0,0(s,d)} = o_{mw}/2 + l_{mw}/2 + T_{mem} + l_{mw}/2 + o_{mw}/2)$ to obtain l_{mw}. By definition, l_{mw} is a function of size (s) and data stride (d). We can then predict the cost of remote send $(2T_{0,1(s,d)})$ using the parameters. Later, we show the accuracy of this prediction by comparing to direct measurement of the half-round trip remote send for strided data.

The values of our model parameters for 16KB message and 1K stride on the Titan (IA-64) machine are obtained from Fig. 4: $o_{mw} = 29us$, $l_{mw} = 420us$, $T_{mem} = 3us$, and $o_{net} = 131us$. We stress that, using our methodology, all values are measured many times and are repeatable.

Half-round-trip time for send-to-self of noncontiguous data ($T_{0,0(s,d)}$, where $s = 16KB$, $d = 1KB$) is $452us$ on the IA-64 systems. Though one would not ordinarily send data to oneself using MPI, this is a measure of the MPI overhead. In contrast, the half-round-trip time for remote send of noncontiguous data ($T_{0,1(s,d)}$, where $s = 16\ KB$, $d = 1\ KB$) is $580\ us$ on the IA-64 systems.

4.2 System Performance Analysis

4.2.1 System Performance Analysis of the IA-64 Cluster

The left-hand picture in Fig. 5 shows the measured middleware overhead (o_{mw}) and network overhead (o_{net}) on the Itanium cluster. From the figure, we see both middleware overhead (o_{mw}) and network overhead (o_{net}) increase with all message size, while middleware overhead

(o_{mw}) increases faster for large message sizes. For small message sizes, network overhead (o_{net}) dominates the cost. With the increase in message size, middleware overhead (o_{mw}) or the memory/middleware communication cost dominates communication cost. For small message sizes, when data fits in cache, hit rate is high and middleware overhead (o_{mw}) is small. However, once the message size exceeds the cache size, capacity misses increase the average memory access time. Additional costs in middleware determine a system-specific intersection of the two curves. This *crossover point* is the point at which memory or middleware delays on the source and target nodes dominate overall communication cost.

Latency costs, depicted on the right side of Fig. 5, are particularly susceptible to cache characteristics such as associativity. This figure depicts various strides over increasing message sizes. The larger the stride and message size, the more this cost dominates communication. Note the x and y-axes are expressed in log_2. Cache characteristics are evident in the large differences between various strides and the relationship between $(size \times stride)$ and cost. As distances between accesses increase, average memory access times increase.

The left-hand graph of Fig. 5 illustrates an important use of our $\log_3 \mathrm{P}$ model for application and system analysis. In the graph, the crossover point occurs at about 512K. Message transmissions larger than 512K are dominated by memory/middleware communication cost. We have additional data that shows o_{net} versus $(o_{mw} + l_{mw})$ for various stride sizes. As the costs due to data strides increase (reflected in the l_{mw} parameters on the right side of Fig. 5), crossover points will move steadily to the left (in the graph on the left in Fig. 5) to smaller data sizes. For example, $stride = 16$ bytes results in a crossover point at message size of 32K and $stride = 256$ bytes results in a crossover point at message size of 8K.

Applications that limit messages falling to the right of a crossover point may improve performance. If such messages are unavoidable, then system improvements in the middleware or hardware should target reducing memory communication costs. In such applications and systems, decreasing network transmission latency will not address the dominant bottleneck of the communication. A corollary to this observation is that our analyses could influence machine design to support a single type of application that only exhibits characteristics on one side of this crossover point.

Next, we analyze the performance of the middleware implementation. Fig. 6 shows the cost separation by our model parameters for strided message transfers into three parameters: middleware overhead (o_{mw}), network overhead (o_{net}), and middleware latency (l_{mw}). These three parameters increase with message size, while middleware latency (l_{mw}) increases the fastest. Moreover, middleware latency (l_{mw}) varies with stride as well as data size.

The larger the stride size, the larger the middleware latency (l_{mw}) due to plateau cache performance already discussed. For large message size with large stride size, middleware latency (l_{mw}) dominates communication time. MPICH is responsible for the middleware latency (l_{mw}), allocating extra buffers for pack and unpack operations on the sender and receiver. These additional memory copies impact performance severely. This indicates where MPICH performance can be targeted for optimization.[8]

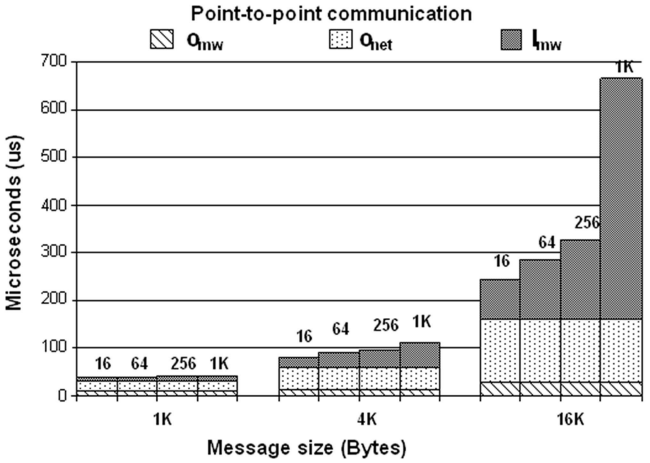


Fig. 6. Parameterized strided costs broken down for the NCSA IA-64 cluster (Titan): middleware overhead (o_{mw}), network overhead (o_{net}), and middleware latency (l_{mw}). Data characteristics determine which parameter dominates communication cost and should be targeted for optimization. For each message size (1K, 4K, 16K), costs for four different stride sizes (16, 64, 256, and 1K) are measured.

4.3 Cost Prediction

4.3.1 Point-to-Point Communication

In this section, we predict the performance of point-to-point communications using the derived $\log_3 \mathrm{P}$ model parameters (from Section 4.1) and compare this prediction to the LogP/LogGP model. We calculate cost predictions per message using the LogGP model as $2o + L + (k-1)G$, where k is the message size in bytes. In all of our direct comparisons with LogP/LogGP, we consider changes between rates (cycles per message in LogGP) and direct cost (microseconds in $\log_3 \mathrm{P}$) when predicting transfer time. We use the MPI LogP/LogGP benchmark tool [15] to gather the parameters[9] presented in Table 1.

Fig. 7 shows the point-to-point communication prediction using the $\log_3 \mathrm{P}$ model and LogGP model on the IA-64 Linux cluster (Titan). For a given message size, data stride does not affect LogGP predictions. LogGP captures hardware characteristics and ignores the effects of middleware. As shown (note the y axis is in log), middleware has a significant effect on the communication cost. The bigger the stride size, the larger this extra cost. The average relative error of LogGP prediction for contiguous data communication is 28 percent. The proposed $\log_3 \mathrm{P}$ model can predict the cost with average error of 5 percent for all the

8. We do not mean to imply that middleware latency is the fault of the MPI implementation. Our model isolates the costs resulting from the interaction of application and middleware. An application may require strided accesses, resulting in significant middleware latency. Our model quantifies the impact and identifies a possible culprit (MPI). However, it may be more appropriate in some cases to modify the application.

9. Such tool measurements make obtaining LogGP parameters as easy as obtaining the parameters of our model. We assume the tool is accurate and the predictions are comparable to our own. Our hope is to present the LogP/LogGP models in the best light possible to underscore the contribution of our model. An alternative would be to use hardware parameters for LogP/LogGP specified by the system manufacturer. These predictions are inaccurate since they do not consider any software effects on performance.

TABLE 1
LogP/LogGP Parameters

Parameters	Values (μs)
L	13.62
o	5.9
G	0.00448
g	14.6

measurements. Our model prediction is slightly more accurate for short messages less than 256 bytes and large messages bigger than 128K bytes. This observation can be explained by the fact that, under these situations, data can be fit in cache totally or out of cache, and the data transfer cost is slightly more stable and predictable. The $\log_3 P$ model captures the cost of memory communication through parameters such as middleware overhead (o_{mw}) and middleware latency (l_{mw}). Predictions are more accurate with these parameters.

4.3.2 Derived Data Type Analyses

Derived data types provide an abstraction to ease programming; some implementations (e.g., MPICH) may suffer poor performance when DDTs are employed. An alternative often embraced by users is to pack and unpack data manually (using simple optimizations for block size, loop unrolling, etc.). One implementation of packing and unpacking can be simulated by copying indexed items in a buffer to a contiguous buffer, for instance: for $(i = 0, j = 0; j < count; i+ = stride, j++)$, $a[j] = b[i]$. Unpacking is copying items from a contiguous buffer to a noncontiguous buffer by index. The sum of packing and unpacking is the cost of the explicit communication.

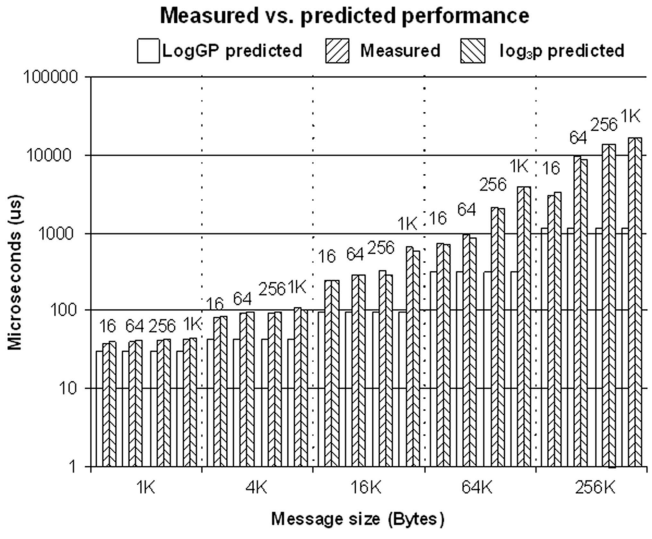


Fig. 7. Measured versus predicted cost of half-round trip for derived data type on Itanium cluster using LogGP (first bar) and $\log_3 P$ (third bar) are presented. The x-axis is the message size in bytes and the y-axis is the cost in microseconds on a log scale. For each message size, we measure and predict a regular stride of 16, 64, 256, and 1,024 bytes.

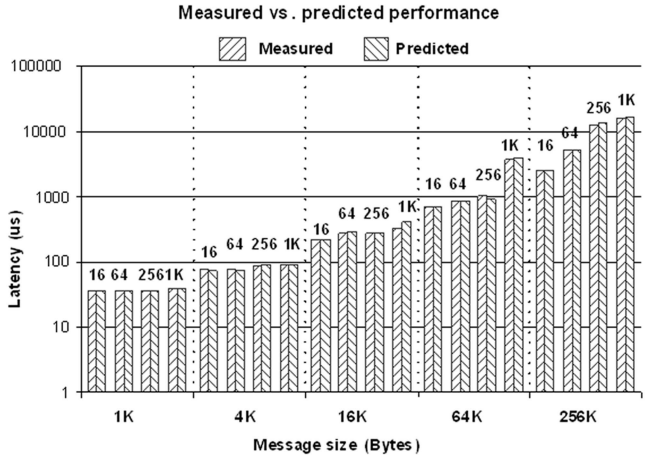


Fig. 8. Measured versus predicted half-round trip time for packing and unpacking. The x-axis is message size in bytes and y-axis is time in microseconds. For each message size, we measure and predict a regular stride of 16, 64, 256, and 1,024 bytes. The first bar is measured cost, while the second bar is the predicted cost.

We use our $\log_3 P$ model to predict the cost of packing and unpacking for various size and strides of data. Fig. 8 shows the measured and predicted latency using our model. The average relative error of prediction is 3.5 percent. The prediction is slightly more accurate for short messages less than 256 bytes and large messages bigger than 128K bytes for all the strides. One interesting observation is that, instead of providing much better performance, which we expected, manual packing and unpacking doesn't always guarantee much better performance on this IA-64 cluster. The average improvement over derived data types is about 15 percent. The maximum improvement is as much as 50 percent, while, in some cases, the improvement is just below 2 percent.

Researchers at Argonne National Laboratory have used our model to improve the general performance of derived data types. This is done as follows: When a derived data type is used, the size and stride information is embedded in the DDT representation. At runtime, the size and stride information is used as input to our model to predict the performance of various algorithm implementations. The prediction is used to suggest the best algorithm implementations for various blocking and array padding factors. By selecting the best performing algorithm at runtime, derived data type performance was improved significantly (at times more than 50 percent) over both MPICH and proprietary IBM MPI implementations for various systems. Further details can be found in a related paper [6].

4.3.3 Collective Communication

In this section, we illustrate use of the point-to-point $\log_3 P$ communication model to analyze two collective communication algorithms: linear broadcast and tree structured broadcast. The communication patterns are depicted in Fig. 9. The actual MPI broadcast is implemented in MPICH by integrating these two algorithms. For example, for an 8-node broadcast, MPICH uses linear broadcast (Fig. 9a) for group[10] $\text{size} = 8$ and tree-structured broadcast (Fig. 9b) for group $\text{size} = 1$. For other group sizes, a tree-structured

10. A "group" in MPI parlance refers to a group of processes.

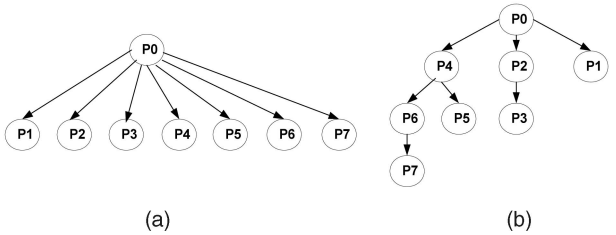


Fig. 9. Communication patterns of 8-way broadcast. Here, the numbering denotes the order of the communications. A typical MPICH broadcast implements a hybrid version of these algorithms where processes are grouped. Linear broadcast is used within the groups and tree-structured broadcast is used across groups. We show linear and tree-structured broadcast only since the hybrid scheme is a subset of these two extremes. (a) Linear broadcast algorithm. (b) Tree-structured broadcast algorithm.

algorithm is used to broadcast a message between groups of processes and then the linear algorithm is used to broadcast the message from the first process in a group to all other processes. These examples serve several purposes: 1) They quantify the impact of middleware costs for simple algorithm cost models and 2) they illustrate how to apply the model to algorithm cost analysis for comparison.

The linear broadcast algorithm is based on point-to-point communication in which $(P-1)$ individual consecutive MPI_Sends are used at the source/root node to transfer data to each remaining node, where P is the number of processors. The cost of this implementation of broadcast includes the overhead at the source, the cost of network transmission, and the cost of delays until the last node receives the message. We implement a linear broadcast, as one would implement an algorithm, predict the cost analytically, and compare this prediction to the measured cost. For data transmission, the cost should be the sum of contiguous data communication and the extra latency introduced by strided data. Using $\log_3 \mathrm{P}$, the cost is $P^*(o_{mw}/2 + l_{mw}/2) + o_{net}$, where $P^*(o_{mw}/2 + l_{mw}/2)$ is the middleware overhead and middleware latency occurring at the source node for sending data to other $(P-1)$ nodes and the last receiving node and o_{net} is the network overhead. The prediction of broadcasting a message size of $(k+1)$ bytes with LogGP is $2o + L + (P-1)Gk + (P-2)g$, where $2o$ is the overhead at the source node and the last receiving node, L is the network latency, $(P-1)Gk$ is the cycles to send $(P-1)$ messages with each of them taking Gk cycles, and $(P-2)g$ is the cost of $(P-2)$ gaps between $(P-1)$ messages. The values of parameters o, L, G, and g are given in Table 1.

Fig. 10 shows predictions for linear broadcast using the $\log_3 \mathrm{P}$ model and LogGP model on the IA-64 Linux cluster. For small message sizes and small strides, the LogGP prediction is accurate. But, for large message sizes and larger strides, LogGP prediction error is considerable and it increases with data size and stride. For the data points measured, the maximum relative error of LogGP is 54 percent and the average relative error is 20.3 percent. The average error of $\log_3 \mathrm{P}$ predictions is about 3 percent and the maximum relative error is 11 percent for the data points measured.

For the tree-structured broadcast algorithm, each node sends data to its children after receiving from its parent. The root node is the source. This algorithm has a characteristic that the message latency is determined by the height of the tree. Using the $\log_3 \mathrm{P}$ model, the latency of this algorithm is $(o_{mw} + l_{mw} + o_{net})$ times the height of the tree, which is $h = log_2(P)$, where P is the number of processors. The prediction for LogGP is $h(2o + L + kG) + (h-1)g$, where $(k+1)$ is the message size in bytes and other parameters are given in Table 1.

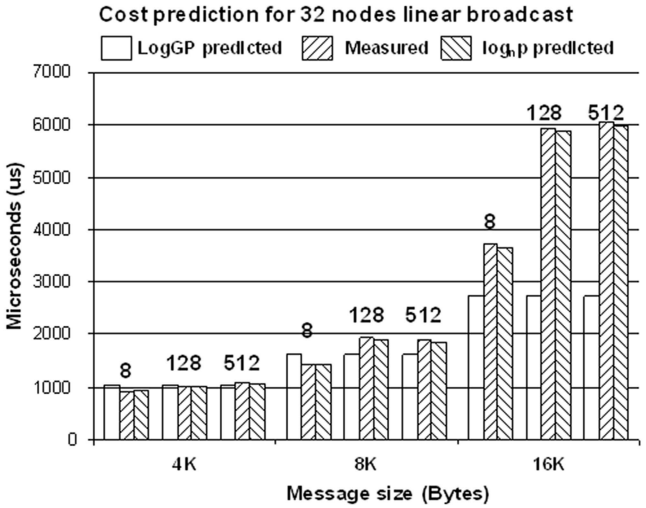


Fig. 10. Cost prediction of linear broadcast. The x-axis is message size in bytes and the y-axis is time in microseconds. For each message size, we measure and predict regular stride of 8, 128, and 512 bytes. The first bar is LogGP predicted cost, the second bar is the measured cost, while the third bar is the predicted cost by the $\log_3 \mathrm{P}$ model.

Fig. 11 shows the predictions for tree-structured broadcast using the $\log_3 \mathrm{P}$ model and LogGP model on the IA-64 Linux cluster. The average relative error of LogGP prediction is 46 percent for the data points measured. The error increases with data size and stride. The minimum relative error is 16 percent for contiguous data with size of 4 Kbytes, and maximum relative error is 72 percent at size 16 Kbytes

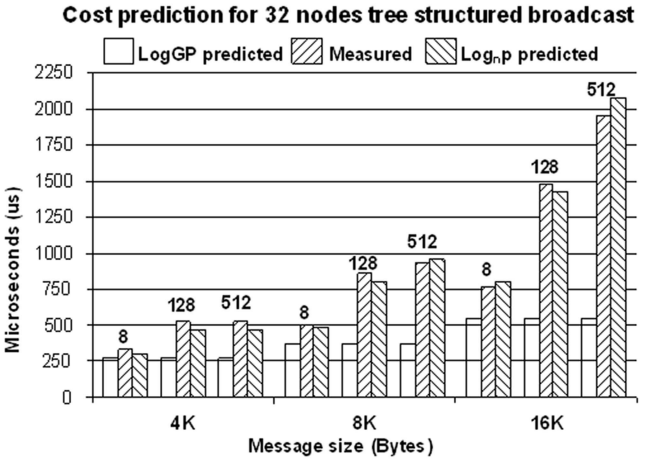


Fig. 11. Cost prediction of tree-structured broadcast. The x-axis is message size in bytes and y-axis is time in microseconds. For each message size, we measure and predict a regular stride of 8, 128, and 512 bytes. The first bar is LogGP predicted cost, the second bar is the measured cost, while the third bar is the predicted cost by the $\log_3 \mathrm{P}$ model.

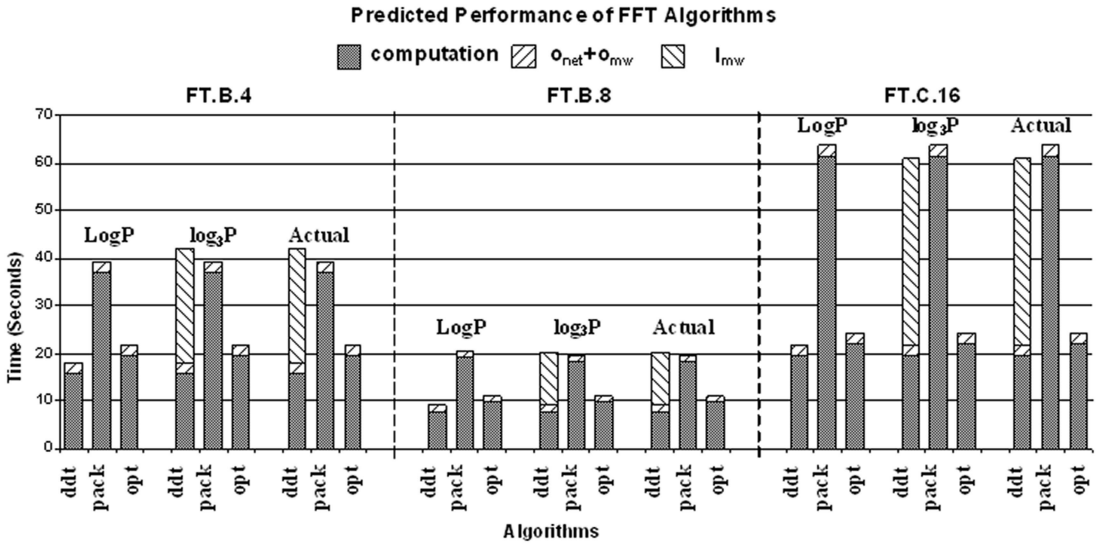


Fig. 12. LogP and $\log_3 P$ predicted and *actual* performance on 4, 8, and 16 processors with *ddt*, *pack*, and *opt* algorithms. LogP assumes middleware latency (l_{mw}) is negligible and suggests the *ddt* algorithm always performs best. $\log_3 P$ suggests the *opt* algorithm will perform best and suggests optimizing middleware cost may result in cost savings. Memory communication for these FFT codes is as much as 59.3 percent of total time.

with stride of 512 bytes. The average relative error of $\log_3 P$ prediction is about 6 percent for the measured data points. The maximum error is 18 percent for 16 nodes broadcast and 11 percent for 32 nodes broadcast.

4.3.4 Algorithm Analysis

In this section, we show how the use of the $\log_3 P$ model leads to efficient parallel algorithm design for a layered 3D FFT application. The 3D FFT algorithm partitions a 3D array of data in the z direction and performs three 1D FFT operations in the x, y, and z dimensions. The 1D FFT in the x and y dimensions can be completed locally on each node, but the 1D FFT in the z dimension requires all-to-all exchanges between nodes and a transpose between endpoints.

We first consider communication using a derived data type (*ddt*) algorithm to exchange strided data and perform the transpose. The *ddt* algorithm relies on middleware to pack the strided data and map strided data to contiguous locations at the destination. This results in middleware latency (l_{mw}). A second algorithm design (*pack*) manually packs and transposes the matrix and then exchanges the packed message data with other processors. Both designs are naive [5] in that they operate on entire rows or columns and introduce significant latency due to strided memory communication. A third optimized (*opt*) algorithm design uses blocking to manually pack and transpose the matrix. The NAS PB FT benchmark uses a similar implementation and blocking.

We used the NAS Parallel Benchmark (FT) for the *opt* algorithm and created our own versions of FT for the *ddt* and *pack* algorithms. These codes were executed on an NERSC IBM 1.9 GHz p575 POWER 5 system of 122 8-processor nodes, each with 32 GB shared memory connected by a high-bandwidth low-latency switching network. Each processor has a 64 KB/32 KB Instruction/Data L1 cache, 1.92 MB L2 cache, and 36 MB L3 cache. Fig. 12 shows costs for the *ddt*, *pack*, and *opt*[11] algorithms for three sets of problem-size+processor combinations (FT.B.4, FT.B.8, and FT.C.16). Within a single set, there are three groups of three bars. The groups refer to predicted communication cost combined with measured computation cost for LogP (i.e., LogGP) and $\log_3 P$ and actual measured values, respectively. Each bar in a group provides values for the three algorithms under study.

For each bar in Fig. 12, we divide the actual or predicted execution time into three costs as appropriate: FFT *computation time*,[12] contiguous data communication time $(o_{net} + o_{mw})$, and strided data communication time (l_{mw}). We also measured FFT setup, checksum, and synchronization time, but omit these in our graphs since they represent a small fraction of total time and are constant across all algorithm implementations.

We first observe that the memory communication cost in these implementations is significant. Fig. 12 shows *actual* measurements in all three data sets for the *ddt* and *pack* algorithms. The *actual* cost of packing strided data in middleware (l_{mw}) in the *ddt* algorithm is 51.6 percent of the total execution time[13] for FT.B running on four processors. The *actual* cost of manually packing strided data in our pack algorithm is 48.5 percent of total execution time on four processors—this cost is included in the "computation" cost. The *opt* algorithm improves the *ddt* algorithm performance by 43.5 percent on four nodes. The percentages of packing cost to total cost are 59.3 percent for the *ddt* algorithm and 11.3 percent for the *opt* algorithm on 16 processors. In all

11. Block size = 512 bytes.

12. In this section, we include the memory cost of packing/unpacking manually in "computation." LogP variants (strictly speaking) estimate computation cost using a simple RAM model (or computation count x time per computation). Thus, manual packing/unpacking is basically ignored or considered additional "computation," motivating simplification in our discussion. We note that application of $\log_n P$ or recursive application of $\log_3 P$ can be used to further model packing/unpacking, as shown in Section 4.3.2.

13. All percentages of total execution time in this section use total application execution time including FFT setup, checksum, and synchronization costs. As mentioned, these additional costs are not included in Fig. 14 for ease of discussion.

cases, the best (i.e., shortest) *actual* execution time is found using the *opt* algorithm.

Now that we have identified the best cost for the *actual* measurements on a real system, we can use LogP and $\log_3 \mathrm{P}$ to identify the best algorithms suggested by model prediction. Fig. 12 shows the predicted execution times for LogP and $\log_3 \mathrm{P}$. For FT.B.4, LogP suggests the best execution time is obtained using the *ddt* algorithm, while $\log_3 \mathrm{P}$ suggests the best execution time is obtained using the *opt* algorithm. Since *opt* is the *actual* best in all cases, $\log_3 \mathrm{P}$ suggests the appropriate algorithm. In the *ddt* case, LogP underpredicts since it ignores the middleware costs and $\log_3 \mathrm{P}$ predicts accurately and quantifies the costs of middleware that can be reduced with optimization. In the *pack* case, LogP and $\log_3 \mathrm{P}$ provide good estimates of *actual* cost since packing costs are absorbed in the computational cost.[14] In the *opt* case, LogP and $\log_3 \mathrm{P}$ provide accurate estimates since middleware costs have been minimized via blocking.

The results for FT.B.8 and FT.C.16 are similar. In both cases, LogP suggests the *ddt* algorithm performs best, while the $\log_3 \mathrm{P}$ model suggests the *opt* algorithm performs best. Again, LogP and $\log_3 \mathrm{P}$ are accurate for the *pack* algorithm and the *opt* algorithm, but the lack of middleware estimates by LogP significantly underestimates the *actual* cost of the *ddt* algorithm, which leads to an incorrect conclusion.

5 Model Derivations Using $\mathrm{LOG_NP}$

The $\log_\mathrm{n} \mathrm{P}$ model can describe any data transfer. We have already shown how to derive a practical model $(\log_3 \mathrm{P})$ for point-to-point communication prediction in clusters of SMPs. The $\log_\mathrm{n} \mathrm{P}$ (and thus $\log_3 \mathrm{P}$) model is inspired by three existing models. First, the previous LogP model and its variants inspire the 3-points of implicit communication used in $\log_3 \mathrm{P}$ for modeling distributed communications at a practical level of granularity. Second, the flexibility of modeling a number of successive transfers with varying distributions is similar to the approach of the copy-transfer model by Stricker and Gross [18]. Third, the simplicity of dividing communications into contiguous and noncontiguous costs was inspired by the memory logP model of shared memory communication [7]. We begin by expressing LogP variants in $\log_\mathrm{n} \mathrm{P}$ terminology and then we derive the copy transfer model and PRAM to show all of these models are, in fact, special cases of $\log_\mathrm{n} \mathrm{P}$.

Fig. 13a provides an illustrative view of the succeeding discussion. To derive any LogP variant, we first observe that these models define three points of implicit communication for an explicit data transfer. That is, they use $n = 3$ in $\log_\mathrm{n} \mathrm{P}$ terminology. This is a natural reflection of point-to-point communication in massively parallel architectures. Communication originates on the source node (point 0), is transmitted between two (or more) network interfaces (point 1) and is then serviced by the target node (point 2). All variations on LogP use this approach.

For convenience, we will use N, S, and D to describe the complexity of $\log_\mathrm{n} \mathrm{P}$ and other models. N is the number of terms needed to describe the communication hops; S_i and D_i for $0 \leq i \leq (N-1)$ are the number of terms needed to describe the discrete sizes and the discrete distributions (strides) used by the model, respectively, at communication hop i. A $\log_\mathrm{n} \mathrm{P}$ communication is described explicitly by (1). Expression of the costs in $\log_\mathrm{n} \mathrm{P}$ requires $\sum_{i=0}^{N-1}(S_i + S_i * D_i)$ terms or $N^* S^* (1 + D)$ if $S_0 = S_1 = \ldots = S_{N-1} = S$ and $D_0 = D_1 = \ldots = D_{N-1} = D$.

Thus, the number of hops and the number of discrete sizes and strides impacts the complexity. Other models make assumptions about the number and type of transmissions resulting in reduced complexity (in number of terms), as shown in Fig. 13b. The $\log_3 \mathrm{P}$ model, for example, assumes $N = 3$, resulting in $3^* S^* (1 + D)$ terms. Complexity in $\log_3 \mathrm{P}$ can be controlled by minimizing the number[15] of sizes and distributions.

LogP. Though the proposed $\log_\mathrm{n} \mathrm{P}$ denotes parameters similarly to the original LogP model,[16] the models are quite different. The *o* parameter of LogP is actually a lower bound for the o_i parameter of $\log_\mathrm{n} \mathrm{P}$. The *o* parameter of LogP is the cost for a fixed message size. This limits the context of LogP to small messages. Since the $\mathrm{o_i}$ parameter of $\log_\mathrm{n} \mathrm{P}$ includes all costs for contiguous transfers as size varies, any additional repeat rate costs (or assist costs) due to limited buffer sizes are additionally incorporated. Hence, the gap parameter, g, of LogP is also absorbed in the o_i parameter of $\log_\mathrm{n} \mathrm{P}$. LogP ignores data distribution, so l_i cost from $\log_\mathrm{n} \mathrm{P}$ is not present.

Point-to-point communication cost in LogP is typically modeled as $\{o\}_0 + \{L\}_1 + \{o\}_2$ for implicit communication points 0, 1, and 2, respectively. Formally, the $\log_\mathrm{n} \mathrm{P}$ equivalent model is:

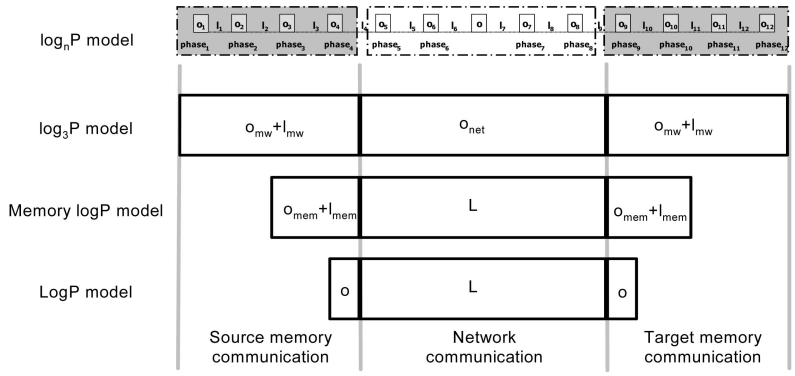
$$T = \sum_{i=0}^{2} f(s,1)_i + f(s,d)_i. \tag{7}$$

Equations (3) and (4) are more detailed expressions of (7). To derive LogP formally from (7) requires applying assumptions to individual terms as we did to derive $\log_3 \mathrm{P}$. Hardware-parameterized costs will be less than their software-parameterized counterparts. For point $i = 0$ or source overhead, $f(s,1)_0$ is reduced from a function to a constant, *o*, and $f(s,d)_0$ is assumed to be zero. For point $i = 1$ or network transfer cost, $f(s,1)_1$ is reduced from a function to a constant, L and $f(s,d)_1$ is assumed to be zero. Both of these assumptions are reasonable for point $i = 1$ since packets consist of contiguous data and packet size is typically fixed. For point $i = 2$ or target overhead, $f(s,1)_2$ is reduced from a function to a constant, *o* and $f(s,d)_0$ is assumed to be zero. Again, contiguous and noncontiguous

14. We use the measured (*actual*) computation cost to "predict" the computational cost for both models. This way, we are able to present total cost estimates for both models, quantify the impact of memory communication on total cost, and avoid prediction errors that are artifacts of the use of the RAM model or simple computation counts to approximate the computational cost.

15. We purposely make no claim as to the number of discrete sizes and strides needed since these numbers are application dependent. For our FFT example in Section 4.3.4, the complexity of our FFT predictions is given as follows: For FT.B.4, $d = 512$ Kbytes, $s = 128$ Mbytes, $S = 1$, $D = 1$, $complexity = 6$ terms; for FT.B.8, $d = 256$ Kbytes, $s = 64$ Mbytes, $S = 1$, $D = 1$, $complexity = 6$ terms; and, for FT.C.16, $d = 256$ Kbytes, $s = 128$ Mbytes, $S = 1$, $D = 1$, $complexity = 6$ terms.

16. Of course, this was our intention. While the models differ fundamentally for analysis purposes, the result is cost predictions that are similar in look and feel.



(a)

Model	Complexity (in # terms)	Assumptions (vs. $\log_n P$)
$\log_n P$	N*S*(1+D)	
copy-transfer	N*S*(1+D)	
$\log_3 P$	3*S*(1+D)	N=3
memory logP	3*S*(1+D)	N=3, f(s,d)=T_{mem}(s,d)
LoPC	3+|C_n|	N=3, S=1, D=0, |C_n| for i=2
LogGP	4	N=3, S_0=2, S_1=S_2=1, D=0
LogP	3	N=3, S=1, D=0
PRAM	1	N=1, S=1, D=0

accuracy ↑ complexity ↑

** For complexity: N = # terms for transmission hops; 0<=i<=(N-1); S = # terms for discrete sizes; D = # terms for discrete distributions (strides); |C_n| = # terms for contention (C_n) estimate.

(b)

Fig. 13. Model comparisons. (a) The $\log_n P$ model can be used to express any point-to-point communication. LogP-variants are special cases of $\log_n P$ where $n = 3$ and the o and l parameters are simplified under various assumptions. To highlight model differences, the actual costs are not drawn to scale. See Fig. 1 for cost comparisons drawn to scale that highlight the impact of memory communication on point-to-point communication cost. (b) The $\log_n P$ model is similar in complexity to the copy transfer model. The $\log_3 P$ model balances accuracy and complexity by limiting the number of terms used and assumptions applied.

memory costs are ignored. In practice, the o term of LogP is an average of the source and target overheads. This implies the costs from point $i = 0$ and point $i = 2$ are actually $o = [f(s,1)_0 + f(s,1)_2]/2$ under the preceding simplifications. In LogP terms then, (7) can be expressed as $T = o + L + o$ without loss of generality. Parameters in LogP ignore the effects of data size and distribution. Expression of the costs in LogP requires $N^*S^*(1+D)$ terms, where $N = 3$, $S = 1$, and $D = 0$ resulting in three total terms.

LogGP. Evolving systems were able to reduce network injection overhead using additional hardware support. Long messages sent as streams instead of a series of smaller messages incur overhead (o) only at the beginning and end of a message transfer (assuming $n = 3$ implicit communication points). Additional overhead per byte (G) was added to reflect costs in longer messages.

Point-to-point communication cost in LogGP is modeled as $\{o + ((k-1)G)\}_0 + \{L\}_1 + \{o\}_2$ for implicit communication points 0, 1, and 2, respectively. Here, k is the number of bytes in a long message. The $\log_n P$ equivalent model is similar to (7). The reductions are the same for o and L, and, once again (for small messages), $o = [f(s,1)_0 + f(s,1)_2]/2$. The difference is in the estimate of implicit communication for point $i = 0$. Under the original LogP model, a long message would be sent as a series of explicit, short messages. These explicit messages each incurred the original $2o + L$ cost. For longer messages, a single explicit message can be initiated and hardware supports implicit transfers costing $G = f(s,1)'$ per byte. $f(s,1)'$ is an additional constant representation of long message cost. We could represent the point $i = 0$ as a simple step function of costs for small and large messages to express LogGP in terms similar to (though simpler than the continuous $f(s,1)$ function in) (3). LogGP increases the accuracy of LogP at the expense of an additional parameter step function. Expression of the costs in LogGP requires $\sum_{i=0}^{N-1}(S_i + S_i * D_i)$ terms, where $N = 3$, $S_0 = 2$, $S_1 = S_2 = 1$, and $D_0 = D_1 = D_2 = 0$, resulting in four total terms. This reflects the use of $S_0 = 2$ message sizes that changes the injection rate of messages; this cost is only incurred when the message is sent $(i = 0)$ since the cost propagates ($i = 1$, and $i = 2$) along the critical path to the destination.

LoPC, LoGPC. One-sided communication requires another LogP variant. In this case, active messages are modeled by an additional parameter to incorporate resource contention at the source and target of the one-sided communication. LoPC and LoGPC model contention in the interrupt queues among all network interfaces involved in communication for small and large messages, respectively. In both cases, point $i = 0$ or point $i = 2$ in $\log_n P$ terminology are extended to incorporate contention at source or target endpoints, respectively (corresponding to get or put one-sided communication operations). All parameters ignore the effects of data distribution. For small messages,

point-to-point communication cost in LoPC is modeled as $\{o\}_0 + \{L\}_1 + \{C_n + o\}_2$ for implicit communication points 0, 1, and 2, respectively. Queuing methods are used to approximate C_n. This cost estimates contention among contiguous data packets; thus, data size and distribution are fixed (for small messages). This cost can be folded into the constant approximation (as in LogP) of the $f(s,1)$ function. Once the value for C_n is estimated using the model, the resulting cost can be expressed using (7). Extending to large messages is slightly more complicated [16], but basically involves modified estimates of the C_n term. LoPC and LoGPC increase accuracy of LogP at the expense of an additional parameter queuing function (C_n).

Expression of the costs in LoPC requires $N^*S^*(1+D)$ terms, where $N=3$, $S=1$, $D=0$, and $|C_n|$ for $i=2$, resulting in $3+|C_n|$ total terms, assuming $|C_n|$ denotes the number of terms necessary for the contention model. This reflects the use of C_n to model contention as a queue at the message destination $(i=2)$. The use of C_n in LoPC (and LoGPC) affects the relative ranking of accuracy and complexity (see Fig. 13b). In this figure, we assume $|C_n| < S(1+D)$, which may or may not be true depending on the use of either model.

Memory logP. This model differs from other LogP variants. Memory logP attempts to augment the o parameter of LogP with a model of memory performance. For shared memory communication, the number of implicit communication operations is modeled as $n=1$. This abstracts out the performance of memory hierarchies and system software optimizations such as prefetching. For point $i=0$ or source/target memory communication, performance is divided into contiguous and noncontiguous costs using $f(s,1)_0$ and $f(s,d)_0$. These estimates are then combined with the L parameter of LogP to create a model of point-to-point communication. The resulting cost estimate for small messages is

$$T = \{f(s,1)_0 + f(s,d)_0\} + \{L\} + \{f(s,1)_2 + f(s,d)_2\}.$$

As in the LoPC model, memory logP focuses on small messages. It does not consider feedback (or buffer copies in middleware) due to limited resources at the network interface (i.e., assumes an infinite network interface buffer).

Memory logP uses parameters that consider the effects of size and distribution resulting in improved accuracy at the expense of additional parameter functions. Expression of the costs in memory logP requires $N^*S^*(1+D)$ terms, where $N=3$, resulting in $3^*S^*(1+D)$ total terms. We note that a primary difference between memory logP and $\log_3 \mathrm{P}$ is the use of T_{mem} (or memory copy cost) to approximate $f(s,d)$ in memory logP.

Copy-transfer model. This low-level model of communication cost is not a LogP variant. Where LogP variants attempt to minimize the number of implicit communication points (n) for simplicity, the copy-transfer model from Stricker and Gross [18] attempts to maximize n for accuracy. Communications are described as they occur in hardware. This model formally describes a transmission in terms of the end-to-end throughput or bandwidth of the hardware. Reduction of $f(s,1)$ and $f(s,d)$ to the underlying hardware bandwidth for each hardware data transfer i in (2) results in an expression of the copy-transfer model. The copy-transfer model increases the number of implicit communications to improve accuracy at the expense of complexity. This model considers the effects of data distribution on cost. Expression of the costs in the copy-transfer model requires $N^*S^*(1+D)$ terms, making it as complex as $\log_\mathrm{n} \mathrm{P}$ but not directly applicable to distributed communication.

PRAM. This model of point-to-point communication is the easiest to use since it assumes a unit cost. To express PRAM in terms of $\log_\mathrm{n} \mathrm{P}$, we set $N=1$ and assume a unit cost for overhead and latency, so $T = o + l = 1$. Expression of the costs in the PRAM model requires $N^*S^*(1+D)$ terms, where $N=1$, $S=1$, $D=0$, resulting in one term and making PRAM the simplest model of communication. PRAM variants (EREW, CREW, etc.) require modifying the unit cost to include contention costs for shared data. PRAM and its variants ignore hardware characteristics and software characteristics such as data size and distribution in favor of a simple representation.

6 Other Related Work

To the best of our knowledge, this is the first model to consider the middleware cost and the effect of strided data when evaluating distributed communication performance. Nevertheless, previous work (other than the approaches discussed in the previous section) modeling either interconnect performance or memory hierarchy performance is prevalent.

Analytical techniques to predict cache performance can be used to estimate model parameters as desired. But, accurate models of memory hierarchy performance are necessarily complicated [2], [10]. Other models that attempt to capture the effects of spatial locality have been proposed. Of these, Sivasubramaniam et al. [17] and Stricker and Gross [18] address memory communication. The former consider spatial locality in a CC-NUMA architecture applying the traditional LogP model to estimate communication cost for optimizing simulation performance. The latter approach uses a fine grain model of the critical data path. As mentioned, this motivated the granularity of $\log_\mathrm{n} \mathrm{P}$.

The Hierarchical Memory Model (HMM) [4], [13] applies the characteristics of memory hierarchies to network communication. Cost estimates are accurate for very large sets of streaming data [20], [22], but ignore the network attributes common to parallel and distributed systems. Our work provides impetus for combining hierarchical memory performance with estimates of network communication cost, distinguishing the two approaches.

7 Conclusions

Previous hardware-parameterized models of communication cost do not consider the influence of middleware on performance. To address this problem, we defined the $\log_\mathrm{n} \mathrm{P}$ model of communication by combining the memory communication cost estimates of an earlier model (memory logP) with the throughput-driven copy-transfer model. The $\log_\mathrm{n} \mathrm{P}$ model incorporates middleware costs in a model of point-to-point communication on emergent systems at the expense of complexity. To address complexity, we derived

the $\log_3 P$ model by applying reductions to $\log_n P$ inspired by LogP variants. We then show how to measure $\log_3 P$ parameters and use them for accurate analysis, prediction of point-to-point (within 5 percent error) and collective broadcast communications (within 3 percent error), and to predict, analyze, and suggest algorithms when memory communication cost is significant. Since our techniques are fast and accurate, they have been used to improve MPICH performance [5].

Although our analysis techniques show promise in performance evaluation and prediction, there are some limitations. Our analyses were limited to regular access patterns. Prediction is more cumbersome for the irregular patterns present in some codes that use sparse matrices. We are exploring techniques used by the copy-transfer model to handle irregular accesses, though it is not clear at present whether this is applicable to middleware cost estimation. For more realistic communication schemes embedded in full applications, analyses will be additionally complicated. For instance, we are exploring incorporation of contention in the g parameter of our model. We are attempting to refine our approach for improved accuracy in such a context.

ACKNOWLEDGMENTS

The authors would like to thank the US Department of Energy (Grant No. DE-FG02-04ER25608) and the US National Science Foundation (Grant Nos. 0347683, 0509133, 0224377, 0305355, 0406328, and 0504291). They would also like to thank the anonymous reviewers for their insightful comments and NCSA, NPACI, ORNL, and NERSC for account access to their machines.

REFERENCES

[1] A. Aggarwal, A.K. Chandra, and M. Snir, "Communication Complexity of PRAMs," *Theoretical Computer Science,* vol. 71, pp. 3-28, 1990.
[2] D.H. Albonesi and I. Koren, "A Mean Value Analysis Multiprocessor Model Incorporating Superscalar Processors and Latency Tolerating Techniques," *Int'l J. Parallel Programming,* vol. 24, pp. 235-263, 1996.
[3] A. Alexandrov, M.F. Ionescu, K. Schauser, and C. Scheiman, "LogGP: Incorporating Long Messages into the LogP Model," *Proc. Seventh Ann. Symp. Parallel Algorithms and Architecture,* 1995.
[4] B. Alpern, L. Carter, E. Feig, and T. Selker, "The Uniform Memory Hierarchy Model of Computation," *Algorithmica,* vol. 12, pp. 72-109, 1994.
[5] S. Byna, W. Gropp, X.-H. Sun, and R. Thakur, "Improving the Performance of MPI Derived Datatypes by Optimizing Memory-Access Cost," *Proc. IEEE Int'l Conf. Cluster Computing,* 2003.
[6] S. Byna, X.-H. Sun, W. Gropp, and R. Thakur, "Improving the Performance of MPI Derived Datatypes by Optimizing Memory-Access Cost," *Proc. IEEE Int'l Conf. Cluster Computing,* 2003.
[7] K.W. Cameron and X.-H. Sun, "Quantifying Locality Effect in Data Access Delay: Memory logP," *Proc. IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS '03),* 2003.
[8] D.E. Culler, R. Karp, D.A. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, and T. von Eicken, "LogP: A Practical Model of Parallel Computation," *Comm. ACM,* vol. 39, pp. 78-85, 1996.
[9] D.E. Culler, J.P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann, 1999.
[10] X. Du and X. Zhang, "Memory Hierarchy Considerations for Cost-Effective Cluster Computing," *IEEE Trans. Computers,* vol. 49, pp. 915-933, 2000.
[11] S. Fortune and J. Wyllie, "Parallelism in Random Access Machines," *Proc. 10th Ann. ACM Symp. Theory of Computing,* 1978.
[12] M.I. Frank, A. Agarwal, and M.K. Vernon, "LoPC: Modeling Contention in Parallel Algorithms," *Proc. Sixth Symp. Principles and Practice of Parallel Programming,* 1997.
[13] T. Hey and J. Ferrante, *Portability and Performance for Parallel Processing.* John Wiley & Sons, 1993.
[14] F. Ino, N. Fujimoto, and K. Hagihara, "LogGPS: A Parallel Computational Model for Synchronization Analysis," *Proc. Principles and Practice of Parallel Processing Symp. (PPoPP '01),* 2001.
[15] T. Kielmann, H.E. Bal, and K. Verstoep, "Fast Measurement of LogP Parameters for Message Passing Platforms," *Proc. IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS),* 2000.
[16] C.A. Moritz and M.I. Frank, "LoGPC: Modeling Network Contention in Message-Passing Programs," *Proc. SIGMETRICS '98,* 1998.
[17] A. Sivasubramaniam, A. Singla, U. Ramachandran, and H. Venkateswaran, "Abstracting Network Characteristics and Locality Properties of Parallel Systems," *Proc. Int'l Symp. High Performance Computer Architecture (HPCA '95),* 1995.
[18] T. Stricker and T. Gross, "Optimizing Memory System Performance for Communication in Parallel Computers," *Proc. Int'l Symp. Computer Architecture (ISCA '95),* 1995.
[19] D. Sundaram-Stukel and M.K. Vernon, "Predictive Analysis of a Wavefront Application Using LogGP," *Proc. Symp. Principles and Practice of Parallel Programming (PPOPP '99),* 1999.
[20] R. Thakur, W. Gropp, and E. Lusk, "Optimizing Non-Contiguous Accesses in MPI-IO," *Parallel Computing,* vol. 28, pp. 83-105, 2002.
[21] L.G. Valiant, "A Bridging Model for Parallel Computation," *Comm. ACM,* vol. 33, pp. 103-111, 1990.
[22] J.S. Vitter and E.A.M. Shriver, "Algorithms for Parallel Memory II: Hierarchical Multilevel Memories," *Algorithmica,* vol. 12, pp. 148-169, 1994.

Kirk W. Cameron received the PhD degree in computer science from Louisiana State University. He is an associate professor in the Department of Computer Science and director of the Scalable Performance (SCAPE) Laboratory at Virginia Polytechnic Institute and State University. His research interests include high-performance and grid computing, parallel and distributed systems, computer architecture, power-aware systems, and performance evaluation and prediction. He is a member of the IEEE and the IEEE Computer Society.

Rong Ge received the BS degree and MS degree in fluid mechanics from Tsinghua University, China, and the MS degree in computer science from the University of South Carolina. She is a PhD candidate in the Department of Computer Science and a researcher at the SCAPE Laboratory at Virginia Tech. Her research interests include performance modeling and analysis, parallel and distributed systems, power-aware systems, high-performance computing, and computational science. She is a student member of the IEEE and the IEEE Computer Society and a member of the ACM and Upsilon Pi Epsilon.

Xian H. Sun is a professor of computer science at the Illinois Institute of Technology (IIT), a guest faculty member at the Argonne National Laboratory, and the director of the Scalable Computing Software (SCS) laboratory at IIT. Before joining IIT, he was a postdoctoral researcher at the Ames National Laboratory, a staff scientist at ICASE, NASA Langley Research Center, an ASEE fellow at the Naval Research Laboratory, and an associate professor at Louisiana State University-Baton Rouge. He is currently on sabbatical at the Fermi National Laboratory as a visiting scientist. He is a senior member of the IEEE and the IEEE Computer Society.