Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture

Richard P. Martin, Amin M. Vahdat, David E. Culler and Thomas E. Anderson
Computer Science Division
University of California
Berkeley, CA 94720

Abstract

This work provides a systematic study of the impact of communication performance on parallel applications in a high performance network of workstations. We develop an experimental system in which the communication latency, overhead, and bandwidth can be independently varied to observe the effects on a wide range of applications. Our results indicate that current efforts to improve cluster communication performance to that of tightly integrated parallel machines results in significantly improved application performance. We show that applications demonstrate strong sensitivity to overhead, slowing down by a factor of 60 on 32 processors when overhead is increased from 3 to 103 μs. Applications in this study are also sensitive to per-message bandwidth, but are surprisingly tolerant of increased latency and lower per-byte bandwidth. Finally, most applications demonstrate a highly linear dependence to both overhead and per-message bandwidth, indicating that further improvements in communication performance will continue to improve application performance.

1 Introduction

Many research efforts in parallel computer architecture have focused on improving various aspects of communication performance. These investigations cover a vast spectrum of alternatives, ranging from integrating message transactions into the memory controller [4, 23, 26, 38] or the cache controller [1, 17, 29], to incorporating messaging deep into the processor [7, 8, 9, 14, 19, 20, 33, 37], integrating the network interface on the memory bus [6, 28], providing dedicated message processors [5, **?**, 34], providing various kinds of bulk transfer support [4, **?**, 26, 34], supporting reflective memory operations [6, 18], and providing lean communication software layers [32, 39, 40]. Recently, we have seen a shift to designs that accept a reduction in communication performance to obtain greater generality (e.g., Flash vs. Dash), greater opportunity for specialization (e.g. Tempest [35]), or a cleaner communication interface (e.g., T3E vs. T3D). At the same time, a number of investigations are focusing on bringing the communication performance of clusters closer to that of the more tightly integrated parallel machines [3, 9, 18, 32, 39]. Moving forward from these research alternatives, a crucial question to answer is how much do the improvements in communication performance actually improve application performance.

The goal of this work is to provide a systematic study of the impact of communication performance on parallel applications. It focuses on a high performance cluster architecture, for which a fast Active Message layer has been developed to a low latency, high bandwidth network. We want to quantify the performance impact of our communication enhancements on applications and to understand if they have gone far enough. Furthermore, we want to understand which aspects of communication performance are most important. The main contributions of this work are (i) a reproducible empirical apparatus for measuring the effects of variations in communication performance for clusters, (ii) a methodology for a systematic investigation of these effects and (iii) an in-depth study of application sensitivity to latency, overhead, and bandwidth, quantifying application performance in response to changes in communication performance.

Our approach is to determine application sensitivity to machine communication characteristics by running a benchmark suite on a large cluster in which the communication layer has been modified to allow the latency, overhead, per-message bandwidth and per-byte bandwidth to be adjusted independently. This four-parameter characterization of communication performance is based on the LogP model [2, 11], the framework for our systematic investigation of the communication design space. By adjusting these parameters, we can observe changes in the execution time of applications on a spectrum of systems ranging from the current high-performance cluster

This work originally appeared in the Proceedings of the 24th International Symposium on Computer Architecture, Denver CO, June 1997.

This work was supported in part by the Defense Advanced Research Projects Agency (N00600-93-C-2481, F30602-95-C-0014), the National Science Foundation (CDA 9401156), Sun Microsystems, California MICRO, Hewlett Packard, Intel, Microsoft, and Mitsubishi. Anderson was also supported by a National Science Foundation Presidential Faculty Fellowship. The authors can be contacted at {rmartin, vahdat, culler, tea}@cs.berkeley.edu.

to conventional LAN based clusters. We measure a suite of applications with a wide range of program characteristics, e.g., coarse-grained vs. fine-grained and read vs. write based, to enable us to draw conclusions about the effect of communication characteristics on classes of applications.

Our results show that, in general, applications are most sensitive to communication overhead. This effect can easily be predicted from communication frequency. The sensitivity to message rate and data transfer bandwidth is less pronounced and more complex. Applications are least sensitive to the actual network transit latency and the effects are qualitatively different than what is exhibited for the other parameters. Overall, the trends indicate that the efforts to improve communication performance pay off. Further improvements will continue to improve application performance. However, these efforts should focus on reducing overhead.

We believe that there are several advantages to our approach of running real programs with realistic inputs on a flexible hardware prototype that can vary its performance characteristics. The interactions influencing a parallel program's overall performance can be very complex, so changing the performance of one aspect of the system may cause subtle changes to the program's behavior. For example, changing the communication overhead may change the load balance, the synchronization behavior, the contention, or other aspects of a parallel program. By measuring the full program on a modified machine, we observe the summary effect of the complex underlying interactions. Also, we are able to run applications on realistic input sizes, so we escape the difficulties of attempting to size the machine parameters down to levels appropriate for the small problems feasible on a simulator and then extrapolating to the real case [41]. These issues have driven a number of efforts to develop powerful simulators [35, 36], as well as to develop flexible hardware prototypes [21].

The drawback of a real system is that it is most suited to investigate design points that are "slower" than the base hardware. Thus, to perform the study we must use a prototype communication layer and network hardware with better performance than what is generally available. We are then able to scale back the performance to observe the "slowdown" relative to the initial, aggressive design point. By observing the slowdown as a function of network performance, we can extrapolate back from the initial design point to more aggressive hypothetical designs. We have constructed such an apparatus for clusters using commercially available hardware and publicly available research software.

The remainder of the paper is organized as follows. After providing the necessary background in Section 2, Section 3 describes the experimental setup and our methodology for emulating designs with a range of communication performance. In addition, we outline a microbenchmarking technique to calibrate the effective communication characteristics of our experimental apparatus. Section 4 describes the characteristics of the applications in our benchmark suite and reports their overall communication requirements, such as message frequency, and baseline performance on sample input sets. Section 5 shows the effects of varying each of the four LogP communication parameters for our applications and, where possible, builds simple models to explain the results. Section 6 summarizes some of the related work and Section 7 presents our conclusions.

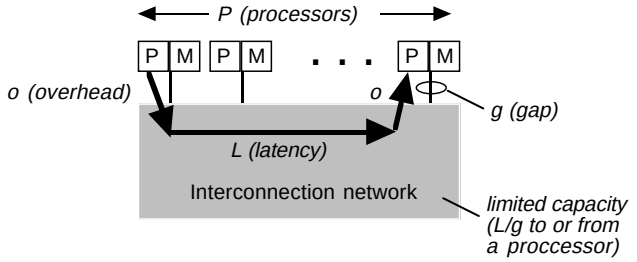

Figure 1: **LogP Abstract Machine.** *The LogP model describes an abstract machine configuration in terms of four performance parameters: L, the latency experienced in each communication event, o, the overhead experienced by the sending and receiving processors, g, the gap between successive sends or successive receives by a processor, and P, the number of processors/memory modules.*

2 Background

When investigating trade-offs in communication architectures, it is important to recognize that the time per communication operation breaks down into portions that involve different machine resources: the processor, the network interface, and the actual network. However, it is also important that the communication cost model not be too deeply wedded to a specific machine implementation. The LogP model [11] provides such a middle ground by characterizing the performance of the key resources, but not their structure. A distributed-memory multiprocessor in which processors physically communicate by point-to-point messages is characterized by four parameters (illustrated in Figure 1).

- L: the *latency*, or delay, incurred in communicating a message containing a small number of words from its source processor/memory module to its target.
- o: the *overhead*, defined as the length of time that a processor is engaged in the transmission or reception of each message; during this time, the processor cannot perform other operations.
- g: the *gap*, defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a module; this is the time it takes for a message to cross through the bandwidth bottleneck in the system.
- P: the number of processor/memory modules.

Platform	o (μs)	g (μs)	L (μs)	MB/s(1/G)
Berkeley NOW	2.9	5.8	5.0	38
Intel Paragon	1.8	7.6	6.5	141
Meiko CS-2	1.7	13.6	7.5	47

Table 1: **Baseline LogGP Parameters.** *This figure shows the performance of the hardware platform used, the Berkeley NOW. Two popular parallel computers, the Intel Paragon and the Meiko CS-2 are included for comparison.*

L, o, and g are specified in units of time. It is assumed that the network has a finite capacity, such that at most $\lceil L/g \rceil$ messages can be in transit from any processor or to any processor at any time. If a processor attempts to transmit a message that would exceed this limit, it stalls until the message can be sent without exceeding the capacity limit.

The simplest communication operation, sending a single packet from one machine to another, requires a time of $L + 2o$. Thus, the latency includes the time spent in the network interfaces and the actual transit time through the network, which are indistinguishable to the processor. A request-response operation, such as a read or blocking write, takes time $2L + 4o$. The processor issuing the request and the one serving the response both are involved for time $2o$. The remainder of the time can be overlapped with computation or sending additional messages.

The available per-processor message bandwidth, or communication rate (messages per unit time) is $1/g$. Depending on the machine, this limit might be imposed by the available network bandwidth or by other facets of the design. In many machines, the limit is imposed by the message processing rate of the network interface, rather than the network itself. Because many machines have separate mechanisms for long messages(e.g. DMA), it is useful to extend the model with an additional gap parameter, G, which specifies the time-per-byte, or the reciprocal of the bulk transfer bandwidth [2]. In our machine, G is determined by the DMA rate to or from the network interface, rather than the network link bandwidth.

The LogGP characteristics for our cluster, the Berkeley NOW, are summarized in Table 1. For reference, we also provide measured LogGP characteristics for two tightly integrated parallel processors, the Intel Paragon and Meiko CS-2 [12].

3 Methodology

In this section we describe the empirical methodology of our study. The experimental apparatus consists of commercially available hardware and system software, augmented with publicly available research software that has been modified to conduct the experiment. The experimental apparatus is calibrated using a simple microbenchmarking technique.

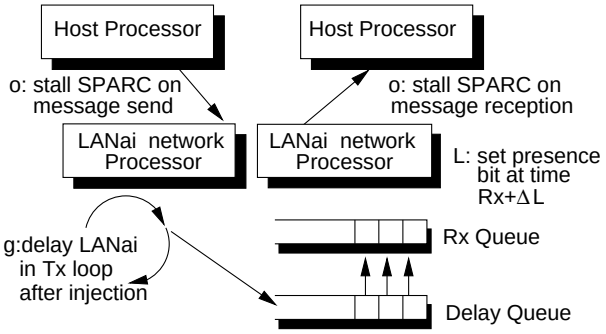

Figure 2: **Varying LogGP Parameters.** *This figure describes our methodology for individually varying each of the logP parameters. The interaction between the host processor, the network processor (LANai) and the network is shown for communication between two nodes.*

3.1 Experimental setup

The apparatus for all our experiments is a cluster of 32 UltraSPARC Model 170 workstations (167 MHz, 64 MB memory, 512 KB L2 cache) running Solaris 2.5. Each has a single Myricom M2F network interface card on the SBUS, containing 128 KB SRAM card memory and a 37.5 MHz "LANai" processor [**?**]. This processor plays a key role in allowing us to independently vary LogGP parameters. The machines are interconnected with ten 8-port Myrinet switches (model M2F, 160 MB/s per port).

All but two of the applications are written in an SPMD model using Split-C [10], a parallel extension of the C programming language that provides a global address space on distributed memory machines. Split-C (version 961015) is based on GCC (version 2.6.3) and Generic Active Messages (version 961015), which is the base communication layer throughout the study. While the programming model does not provide automatic replication with cache coherence, a number of the applications perform application-specific software caching. The language has been ported to many platforms [2, 31, 39, 40]. The sources for the applications, compiler, and communication layer can be obtained from a publically available site [1].

3.2 Technique

The key experimental innovation is to modify the communication layer so that it can emulate a system with arbitrary overhead, gap, or latency. We then vary these parameters independently and observe the effect on application performance. Our technique is depicted in Figure 2 which illustrates the interaction of the host processor, the

[1] `ftp.cs.berkeley.edu/pub/CASTLE/Split-C/-release/sc961015`

LANai (network interface processor) and the network for communication between two nodes.

The majority of the overhead is the time spent writing the message into the network interface or reading it from the interface. Thus, varying the overhead, o, is straightforward. For each message send and before each message reception, the operation is modified to loop for a specific period of time before actually writing or reading the message.

The gap is dominated by the message handling loop within the network processor. Thus, to vary the gap, g, we insert a delay loop into the message injection path after the message is transferred onto the wire and before it attempts to inject the next message. Since the stall is done after the message is actually sent the network latency is unaffected. Also, since the host processor can write and read messages to or from the network interface at its normal speed, overhead should not be affected. We use two methods to prevent excessive network blocking from artificially affecting our results. First, the LANai is stalled at the source rather than the destination. Second, the firmware takes advantage of the LANai's dual hardware contexts; the receive context can continue even if the transmit context is stalled. To adjust G, the transmit context stalls after injecting a fragment (up to 4KB) for a period of time proportional to the fragment size.

The latency, L, requires care to vary without affecting the other LogGP characteristics. It includes time spent in the network interface's injection path, the transfer time, and the receive path, so slowing either the send or receive path would increase L. However, modifying the send or receive path would have the side effect of increasing g. Our approach involves adding a delay queue inside the LANai. When a message is received, the LANai deposits the message into the normal receive queue, but defers setting the flag that would indicate the presence of the message to the application. The time that the message "would have" arrived in the face of increased latency is entered into a delay queue. The receive loop inside the LANai checks the delay queue for messages ready to be marked as valid in the standard receive queue. Modifying the effective arrival time in this fashion ensures that network latency can be increased without modifying o or g.

3.3 Calibration

With any empirical apparatus, as opposed to a discrete simulator, it is important to calibrate the actual effect of the settings of the input parameters. In this study, it is essential to verify that our technique for varying LogGP network characteristics satisfies two criteria: first, that the communication characteristics are varied by the intended amount and second that they can be varied independently. Such a calibration can be obtained by running a set of Active Message micro-benchmarks, described in [12]. The basic technique is to measure the time to issue a sequence of m messages with a fixed computational delay, Δ between messages. (The clock stops when the last message is issued by the processor, regardless of how many

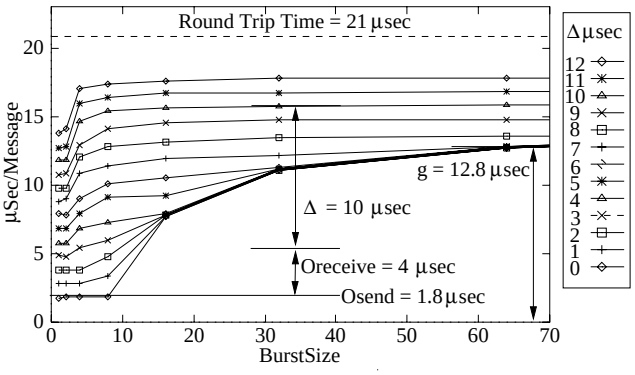

Figure 3: **Calibration of LogP Parameters.** *The LogP* signature *is visible as the isobaric plot of burst size vs. fixed computational delay, Δ. This signature was a calibration made when the desired g was 14 μsec. The send overhead, receive overhead and gap can be read from the signature. Overhead is modeled as the average of the send and receive overhead. Latency is computed as $\frac{1}{2}$(round trip time)-2o.*

requests or responses are in flight.) Plotting the average message cost as a function of the sequence size and delay generates a LogP *signature*, such as that shown in Figure 3. Each curve in the figure shows the average initiation interval seen by the processor as a function of the number of messages issued, m, for a fixed Δ. For a short sequence, this shows the send overhead. Long sequences approach the steady-state initiation interval, g. For sufficiently large Δ the bottleneck is the processor, so the steady state interval is the send overhead plus the receive overhead plus Δ. Finally, subtracting the two overheads from half the round-trip time gives L.

Table 2 describes the result of this calibration process for three of the four communication characteristics. For each parameter, the table shows the desired and calibrated setting for that parameter. For overhead, the calibrated value is within 1% of the desired value. Observe that as o is increased, the effective gap increases because the processor is the bottleneck, consistent with the LogP model. As desired, the value of L is independent of o. The calibrated g is somewhat lower than intended and varying g has little effect on L and no effect on o. Increasing L has little effect on o. A notable effect of our implementation is that for large values of L, the effective g rises. Because the implementation has a fixed number of outstanding messages independent of L, when L becomes very large the implementation is unable to form an efficient network pipeline. In effect, the capacity constraint of our system is constant, instead of varying with L and g as the LogGP model would predict.

To calibrate G, we use a similar methodology, but instead send a burst of bulk messages, each with a fixed size. From the steady-state initiation interval and message size we derive the calibrated bandwidth (not shown). We increase the bulk message size until we no longer observe an increase in bandwidth (which happens at a 2K

Desired	Observed			Desired	Observed			Desired	Observed		
o	o	g	L	g	g	o	L	L	L	o	g
2.9	2.9	5.8	5.0	5.8	5.8	2.9	5.0	5.0	5.0	2.9	5.8
4.9	5.1	10.1	5.0	8.0	7.5	2.9	5.1	7.5	8.1	2.9	6.3
7.9	8.1	16.0	4.7	10	9.6	2.9	5.5	10	10.3	2.9	6.4
12.9	13.0	26.0	5.0	15	14	3.0	5.5	15	15.5	2.9	7.0
22.9	23.1	46.0	4.9	30	29	3.0	5.5	30	30.4	2.9	9.6
52.9	52.9	106.0	5.4	55	52	2.9	5.5	55	55.9	3.0	15.5
77.9	76.5	151.0	5.3	80	76	2.9	5.5	80	80.4	2.9	21.6
102.9	103.0	205.9	6.0	105	99	3.0	5.5	105	105.5	3.0	27.7

Table 2: **Calibration Summary.** *This table demonstrates the calibration of desired LogP parameter values versus measured values. The table also shows that the LogP parameters can be varied independent of one another.*

byte message size). The calibrated values are used in the analysis done below.

4 Applications and Baseline Characteristics

With a methodology in place for varying communication characteristics, we now characterize the architectural requirements of the applications in our study. To ensure that the data is not overly influenced by startup characteristics, the applications must use reasonably large data sets. Given the experimental space we wish to explore, it is not practical to choose data sets taking hours to complete; however, an effort was made to choose realistic data sets for each of the applications. We used the following criteria to characterize applications in our benchmark suite and to ensure that the applications demonstrate a wide range of architectural requirements:

- **Message Frequency:** The more communication intensive the application, the more we would expect its performance to be affected by the machine's communication performance. For applications that use short messages, the most important factor is the message frequency, or equivalently the average interval between messages. However, the behavior may be influenced by the burstiness of communication and the balance in traffic between processors.
- **Write or Read Based:** Applications that read remote data and wait for the result are more likely to be sensitive to latency than applications that mostly write remote data. The latter are likely to be more sensitive to bandwidth. However, dependences that cause waiting can appear in applications in many forms.
- **Short or Long Messages:** The Active Message layer used for this study provides two types of messages, short packets and bulk transfers. Applications that use bulk messages may have high data bandwidth requirements, even though message initiations are infrequent.
- **Synchronization**: Applications can be bulk synchronous or task queue based. Tightly synchronized applications are likely to be dependent on network round trip times, and so may be very sensitive to latency. Task queue applications may tolerate latency, but may be sensitive to overhead. A task queue based application attempts to overlap message operations with local computation from a task queue. An increase in overhead decreases the available overlap between the communication and local computation.
- **Communication Balance**: Balance is simply the ratio of the maximum number of messages sent per processor to the average number of messages sent per processor. It is difficult to predict the influence of network performance on applications with a relatively large communication imbalance since varying LogP parameters may exacerbate or may actually alleviate the imbalance.

4.1 Benchmark Suite

Table 3 summarizes the programs we chose for our benchmark suite as run on both a 16 and a 32 node cluster. Most applications are well parallelized when scaled from 16 to 32 processors. Each application is discussed briefly below.

- **Radix Sort:** sorts a large collection of 32-bit keys spread over the processors, and is thoroughly analyzed in [16]. It progresses as two iterations of three phases. First, each processor determines the local rank for one digit of its keys. Second, the global rank of each key is calculated from local histograms. Finally, each processor uses the global histogram to distribute the keys to the proper location. For our input set of one million keys per processor on 32 processors the application spends 98% of its time in the communication phases.

Program	Description	Input Set	16 node Time (sec)	32 node Time (sec)
Radix	Integer radix sort	16 Million 32-bit keys	13.66	7.76
EM3D(write)	Electro-magnetic wave propagation	80000 Nodes, 40% remote, degree 20, 100 steps	88.59	37.98
EM3D(read)	Electro-magnetic wave propagation	80000 Nodes, 40% remote, degree 20, 100 steps	230.0	114.0
Sample	Integer sample sort	32 Million 32-bit keys	24.65	13.23
Barnes	Hierarchical N-Body simulation	1 Million Bodies	77.89	43.24
P-Ray	Ray Tracer	1 Million pixel image 16390 objects	23.47	17.91
Murφ	Protocol Verification	SCI protocol, 2 procs, 1 line, 1 memory each	67.68	35.33
Connect	Connected Components	4 Million nodes 2-D mesh, 30% connected	2.29	1.17
NOW-sort	Disk-to-Disk Sort	32 Million 100-byte records	127.2	56.87
Radb	Bulk version of Radix sort	16 Million 32-bit keys	6.96	3.73

Table 3: **Applications and Data Sets.** *This table describes our applications, the input set, the application's communication pattern, and the base run time.*

The communication density plot of Figure 4a is useful in understanding the communication behavior of this application. The darkness of cell i, j indicates the fraction of messages sent from processor i to processor j. The dark line off the diagonal reflects the global histogram phase, where the ranks are accumulated across processors in a kind of pipelined cyclic shift. The grey background is the global distribution phase. Overall, the communication is frequent, write-based and balanced.

- **EM3D:** EM3D [10] is the kernel of an application that models propagation of electromagnetic waves through objects in three dimensions. It first spreads an irregular bipartite graph over all processors. During each time-step, changes in the electric field are calculated as a linear function of the neighboring magnetic field values and vice versa. We use two complementary versions of EM3D, one write-based and the other read-based. Both versions contain relatively short computation steps. The write-based EM3D uses pipelined writes to propagate updates by augmenting the graph with special boundary nodes. EM3D(write) represents a large class of bulk synchronous applications, alternating between local computation and global communication phases. The read version uses simple blocking reads to pull update information locally and does not need to create special boundary nodes. The locality of connectivity in the graph for both versions is indicated by the dark swath in Figures 4b and 4c.
- **Sample Sort:** is a probabilistic algorithm which sorts a large collection of 32-bit keys by first choosing $p - 1$ "good" splitter values and broadcasting them to all processors. Every processor distributes its keys to the proper destination processor, based on the splitter values, and finally, a local radix sort is performed on the received keys. An interesting aspect of this application is the potential for unbalanced all-to-all communication as each processor potentially receives a different number of keys. This is reflected in the vertical bars in Figure 4d. For our input size, the local sort time is dominated by the distribution of keys to their proper destinations. For our input of 16 million keys Sample sort spends 85% of the time in the two communication phases.
- **Barnes:** Our implementation of this hierarchical N-Body force calculation is similar to the version in the SPLASH benchmark suite [41]. However, the main data structure, a spatial oct-tree, is replicated in software rather than hardware. Each timestep consists of two phases, a tree construction phase and an interaction phase among the simulated bodies. Updates of the oct-tree are synchronized through blocking locks. During the interaction phase, the processors cache oct-tree nodes owned by remote processors in a software man-

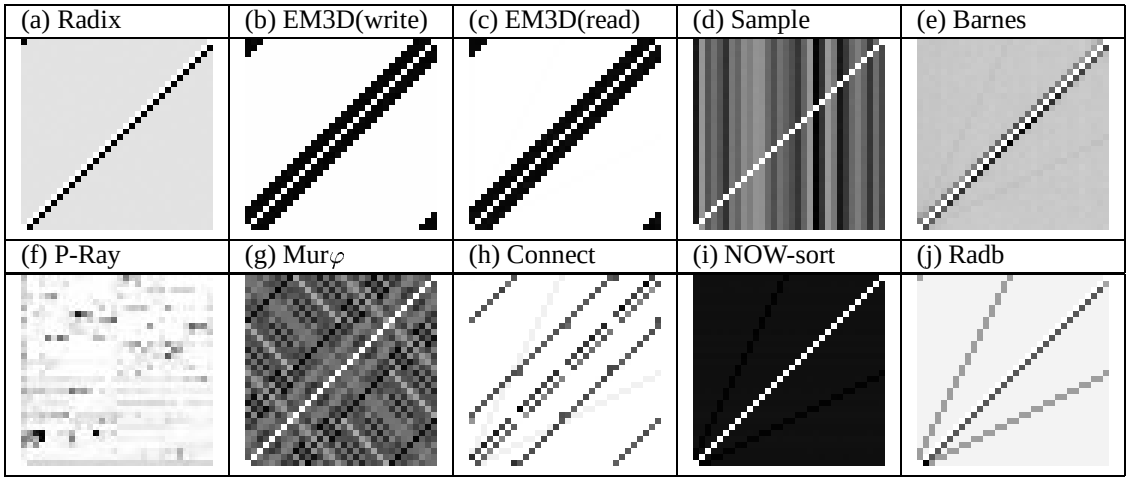

Figure 4: **Communication Balance.** *This figure demonstrates the communication balance between each of the 32 processors for our 10 applications. The greyscale for each pixel represents a message count. Each application is individually scaled from white, representing zero messages, to black, representing the maximum message count per processor as shown in Table 4. The y-coordinate tracks the message sender and the x-coordinate tracks the receiver.*

aged cache. Communication is generally balanced, as the solid grey square shows in Figure 4e.

- **P-Ray:** This scene passing ray tracing program distributes a read-only spatial oct-tree over all processors. The processors evenly divide ownership of objects in the scene. When a processor needs access to an object stored on a remote processor, the object is cached locally in a fixed sized software-managed cache. Communication thus consists entirely of blocking read operations; the frequency of such operations is a function on the scene complexity and the software caching algorithm. The dark spots in Figure 4f indicate the presence of "hot" objects which are visible from multiple points in the scene.
- **Parallel Murφ:** In this parallel version of a popular protocol verification tool [15, **?**], the exponential space of all reachable protocol states are explored to catch protocol bugs. Each processor maintains a work queue of unexplored states. A hash function maps states to "owning" processors. When a new state is discovered, it is sent to the proper processor. On reception of a state description, a processor first checks if the state has been reached before. If the state is new, the processor adds it to the work queue to be validated against an assertion list.
- **Connected Components:** First, a graph is spread across all processors [30]. Each processor then performs a connected components on its local subgraph to collapse portions of its components into representative nodes. Next, the graph is globally adjusted to point remote edges (crossing processor boundaries) at the respective representative nodes. Finally, a global phase successively merges components between neighboring processors. The communication to computation ratio is determined by the size of the graph.
- **NOW-sort:** The version of NOW-sort used in this study sorts records from disk-to-disk in two passes [**?**]. The sort is highly tuned, setting a the MinuteSort world record in 1997. The sorting algorithm contains two phases. In the first phase, each processor reads the records from disk and sends them to the final destination processor. The perfectly balanced nature of the communication of phase 1 is shown by the solid black square in Figure 4i. The sort uses one-way Active Messages directly, sending bulk messages at the rate the records can be read from disk. Phase 2 of the algorithm consists of entirely local disk operations. Unlike the other applications, NOW-sort performs a large amount of I/O, so can overlap communication overhead with disk accesses.
- **Radb:** This version of the radix sort [2] was restructured to use bulk messages. After the the global histogram phase, all keys are sent to their destination processor in one bulk message. Depending on network characteristics, use of these bulk messages can speed up the performance of the sort relative to the standard radix sort.

4.2 Characteristics

As summarized in Table 3, the applications represent a broad spectrum of problem domains and communication/computation characteristics. To quantify the differences among our target applications, we instrumented our communication layer to record baseline char-

Program	Avg. Msg./ Proc	Max Msg./ Proc	Msg./ Proc/ms	Msg. Interval (μs)	Barrier Interval (ms)	Percent Bulk Msg.	Percent Reads	Bulk Msg. KB/s	Small Msg. KB/s
Radix	1,278,399	1,279,018	164.76	6.1	408	0.01%	0.00%	26.7	4,612.9
EM3D(write)	4,737,955	4,765,319	124.76	8.0	122	0.00%	0.00%	0.6	3,493.2
EM3D(read)	8,253,885	8,316,063	72.39	13.8	369	0.00%	97.07%	0.0	2,026.9
Sample	1,015,894	1,294,967	76.76	13.0	1,203	0.00%	0.00%	0.0	2,149.2
Barnes	819,067	852,564	18.94	52.8	279	23.25%	20.57%	110.4	407.1
P-Ray	114,682	278,556	6.40	156.2	1,120	47.85%	96.49%	358.5	93.5
Connect	6,399	6,724	5.45	183.5	47	0.06%	67.42%	0.0	152.5
Murφ	166,161	168,657	4.70	212.6	11,778	49.99%	0.00%	3,876.6	65.8
NOW-sort	69,574	69,813	1.22	817.4	1,834	49.82%	0.00%	3,125.1	17.2
Radb	4,372	5,010	1.17	852.7	25	34.73%	0.04%	33.6	21.4

Table 4: **Communication Summary**. *For a 32 processor configuration, the table shows run times, average number of messages sent per processor, and the maximum number of messages sent by any processor. Also shown is the message frequency expressed in the average number of messages per processor per millisecond, the average message interval in microseconds,the average barrier interval, the percentage of the messages using the Active Message bulk transfer mechanism, the percentage of total messages which are read requests or replies, the average bandwidth per processor for bulk messages, and the average bandwidth per processor for small messages.*

acteristics for each program (with unmodified LogGP parameters) on 32 nodes. Table 4 shows the average number of messages, maximum number of messages per node (as an indication of communication imbalance), the message frequency expressed in the average number of messages per processor per millisecond, the average message interval in microseconds, and the average interval between barriers as a measure of how often processors synchronize. Table 4 also shows the percentage of the messages using the Active Message bulk transfer mechanism, the percentage of the total messages which are a read request or reply, the average bandwidth per processor for bulk messages, and the average bandwidth per processor for small messages. Note that the reported bandwidth is for bytes transmitted through the communication layer as opposed to bandwidth delivered to the application.

Table 4 shows that the communication frequency of our applications varies by more than two orders of magnitude, and yet none of them are "embarrassingly parallel." This disparity suggests that it is quite difficult to talk about typical communication behavior or sensitivity. Most of the applications have balanced communication overall, whereas others (Sample, P-Ray) have significant imbalances. Barnes and EM3D(write) are bulk synchronous applications employing barriers relatively frequently. Barnes, Murφ, P-Ray, Radb and NOW-sort utilize bulk messages while the other applications send only short messages. Finally, EM3D(read), Barnes, P-Ray, and Connect do mostly reads, while the other applications are entirely write based. Applications doing reads are likely to be dependent on network round trip times, and thus sensitive to latency, while write based applications are more likely to be tolerant of network latency. Most of the applications demonstrate regular communication patterns. However, Connect and P-Ray are more irregular and contain a number of hot spots. While these applications do not constitute a workload, their architectural requirements vary across large classes of parallel applications.

5 Sensitivity to Network Performance

Given our methodology and application characterization, we now quantify the effect of varying LogGP parameters on our application suite. To this end, we independently vary each of the parameters in turn to observe application slowdown. For each parameter, we attempt to explain any observed slowdowns based on application characteristics described in the last section. Using this intuition, we develop models to predict application slowdown.

5.1 Overhead

Figure 5(b) plots application slowdown as a function of added overhead measured in microseconds for our applications run on 32 nodes. The extreme left portion of the x-axis represents runs on our cluster. As overhead is increased, the system becomes similar to a switched LAN implementation. Currently, 100 μs of overhead with latency and gap values similar to our network is approximately characteristic of TCP/IP protocol stacks [24, 25, 39]. At this extreme, applications slow down from 2x to over 50x. Clearly, efforts to reduce cluster communication overhead have been successful. Further, all but one of our applications demonstrate a linear dependence to overhead, suggesting that further reduction in overhead will continue to yield improved performance. Qualitatively, the four applications with the highest communication frequency, Radix, Sample, and both EM3D read and write, display the highest sensitivity to overhead. Barnes is the only application which

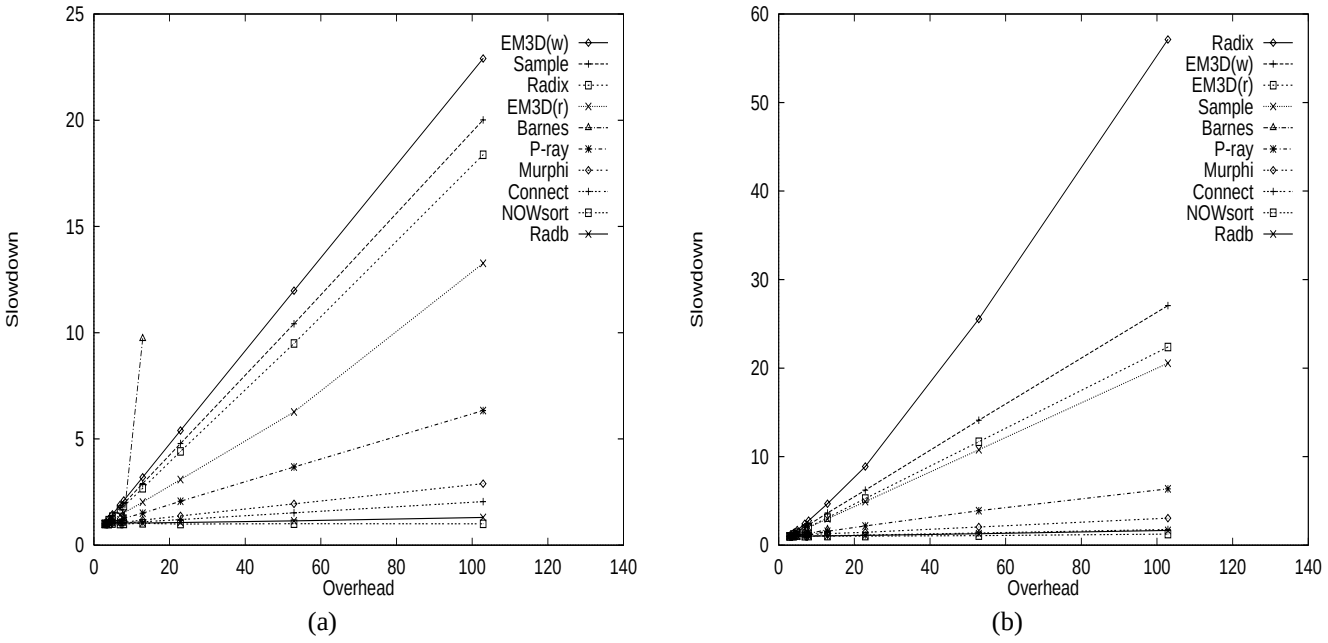

Figure 5: **Sensitivity to Overhead for 16 and 32 Nodes.** *This figure plots application slowdown as a function of overhead in microseconds. Slowdown is relative to application performance with our system's baseline LogGP parameters. Measurements for the graph on the left were taken on 16 nodes, while measurements for the graph on the right were taken on 32 nodes with a fixed input size.*

demonstrates a non-linear dependence to overhead. Instrumentation of Barnes on 16 nodes revealed that as overhead is increased, lock contention causes the program to go into livelock. With zero added overhead, the average number of failed lock attempts per processor is 2000 per timestep. At 13 μs of overhead, the number of failed lock attempts per processor per timestep skyrockets to over 1 million. This implementation of Barnes does not complete for overhead values greater than 13 μs on 16 nodes and 7 μs on 32 nodes.

To determine the effect of scaling the number of processors on sensitivity to overhead, we executed our applications on 16 nodes with fixed inputs. Figure 5(a) plots the resulting slowdown as a function of overhead for runs on 16 nodes. With the exception of Radix, the applications demonstrate almost identical sensitivity to overhead on 16 processors as they did on 32 processors, slowing down by between a factor of between 2 and 25. Recall that Radix contains a phase to construct a global histogram. The number of messages used to construct the histogram is a function of the radix and the number of processors, not the number of keys. For a constant number of keys, the relative number of messages per processor increases as processors are added. Radix thus becomes more sensitive to overhead as the number of processors is increased for a fixed input size. In addition, the difference in sensitivities between 16 and 32 nodes is exacerbated by a serial phase in program, which is described below.

To develop insight into our experimental results, we develop a simple analytical model of application sensitivity to added overhead. The model is based on the fact that added overhead is incurred each time a processor sends or receive a message. Thus, given an processor's base runtime, r_{orig}, the added overhead, Δo, and m, the number of communication events for each processor, we expect runtime, r_{pred}, to be:

$$r_{pred} = r_{orig} + 2m\Delta o$$

The factor of two arises because, for Split-C programs, all communication events are one of a request/response pair. For each request sent, the processor will incur an overhead penalty receiving the corresponding response in addition to the overhead for the sent request. If the processor is sending a response, it must have incurred an overhead penalty when it received the request message.

Given this model for the overhead sensitivity of individual processors, we extrapolate to predicting overall application runtime by making the following simplifying assumptions. First, applications run at the speed of the slowest processor, and second, the slowest processor is the processor that sends the most messages. Thus, by replacing m in the equation with the maximum number of messages sent by a processor from Table 4, we derive a simple model for predicting application sensitivity to added overhead as a function of the

o μs	Radix		EM3D(write)		EM3D(read)		Sample		Barnes	
	measure	*predict*	*measure*	*predict*	*measure*	*predict*	*measure*	*predict*	*measure*	*predict*
2.9	7.8	7.8	38	38	114	114	13.2	13.2	43.2	43.2
3.9	10.5	10.3	48.1	47.5	138.7	130.7	16.1	15.8	50.1	44.9
4.9	13.2	12.9	58.1	57.0	161.6	147.3	18.7	18.4	56.3	51.8
6.9	18.7	18.0	77.4	76.1	208.8	180.5	23.8	23.6	76.1	60.3
7.9	21.5	20.5	87.4	85.6	232.9	197.2	26.5	26.2	N/A	N/A
13	36.3	33.3	138.5	133.3	354.4	280.3	39.3	39.1	N/A	N/A
23	68.9	58.9	236.2	228.6	600.1	446.7	65.2	65.0	N/A	N/A
53	198.2	135.7	535.9	514.5	1332.5	945.6	142.7	142.7	N/A	N/A
103	443.2	263.6	1027.8	991.0	2551.7	1777.2	272.1	272.2	N/A	N/A

o μs	P-Ray		Murφ		Connect		NOW-sort		Radb	
	measure	*predict*	*measure*	*predict*	*measure*	*predict*	*measure*	*predict*	*measure*	*predict*
2.9	17.9	17.9	35.3	35.3	1.17	1.17	56.9	56.9	3.73	3.73
3.9	19.0	18.5	37.1	35.7	1.19	1.18	56.7	57.0	3.77	3.74
4.9	19.6	19.0	37.7	36.0	1.20	1.19	61.2	57.1	3.77	3.75
6.9	22.0	20.1	41.8	36.7	1.23	1.20	57.9	57.4	3.82	3.77
7.9	20.8	20.7	41.9	37.0	1.24	1.21	58.3	57.6	3.83	3.78
13	28.2	23.5	46.2	38.7	1.31	1.25	58.1	58.3	3.93	3.83
23	39.0	29.1	51.2	42.1	1.44	1.34	58.3	59.7	4.10	3.93
53	69.7	45.8	72.6	52.2	1.85	1.61	61.7	63.9	4.81	4.23
103	114.0	73.6	107.8	69.1	2.52	2.08	71.1	70.8	6.19	4.73

Table 5: **Predicted vs. Measured Run Times Varying Overhead.** *This table demonstrates how well our model for sensitivity to overhead predicts observed slowdown for the 32 node runs. For each application, the column labeled $measure$ is the measured runtime, while the column labeled $predict$ is the runtime predicted by our model. For frequently communicating applications such as Radix, EM3D(write), and Sample, the model accurately predicts measured runtimes.*

maximum number of messages sent by any processor during execution.

Table 5 describes how well this model predicts application performance when compared to measured runtimes. For two applications which communicate frequently, Sample, and EM3D(write), our model accurately predicts actual application slowdown . For a number of other applications, most notably Radix, P-Ray and Murφ, the sensitivity to overhead was actually stronger than our model's prediction; the model consistently under-predicts the run time. To explain this phenomenon, we turn back to the assumptions in our model.

Recall in the simple overhead model, we assume that P_m, the processor which sent the most messages, m, is the slowest processor. In addition, we implicitly assume all work in the program is perfectly parallelizable. It is this second assumption which leads to under-predicted run times. If a processor, P_n, serializes the program in a phase n messages long, when we increase o by Δo, then the serial phase will add to the overall run time by $n\Delta o$. However, the simple model does not capture this "serialization effect" when $P_m \neq P_n$.

A more important result of the of the serialization effect is that it reduces speedup as a function of overhead, i.e. speedup gets worse the greater the overhead. Thus, parallel efficiency will decrease as overhead increases for any applications which have a serial portion. Notice how for Radix, parallel decreases as a function of overhead when scaled from 16 to 32 nodes.

Radix sort demonstrates a dramatic example of the serialization effect. The sensitivity to overhead for Radix on 32 processors is over double that of 16 processors. When overhead rises to 100μs, the slowdown differential between 16 and 32 processors is a factor of three. The global histogram phase contains a serialization proportional to the radix and number of processors [16]. In the unmodified case, the phase accounts for 20% of the overall execution time on 32 processors. When the overhead is set to 100μs, this phase accounts for 60% of the overall execution time. However, on 16 processors with 100μs of overhead, the histogram phase takes only 16% of the total time.

5.2 Gap

We next measure application sensitivity to gap. Figure 6 plots application slowdown as a function of added gap in microseconds.

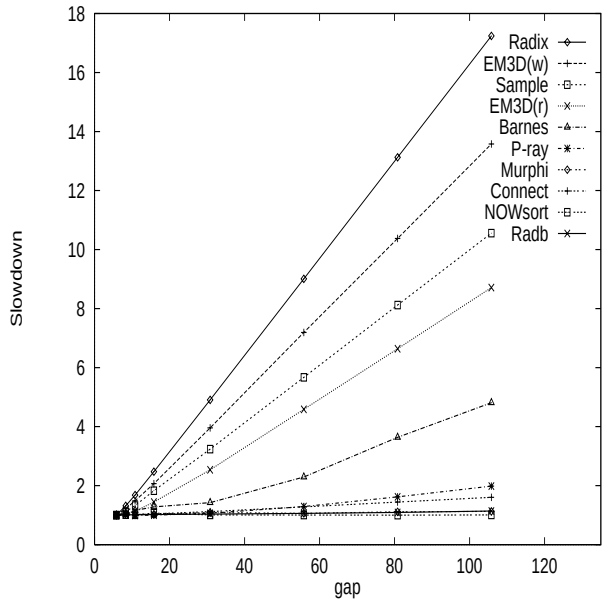

Figure 6: **Sensitivity to gap.** *This figure plots slowdown as a function of gap in microseconds.*

The programs demonstrate widely varying reactions to gap, ranging from being unaffected by 100 μs of gap to slowing down by a factor of 16. The qualitative difference between application sensitivity to gap and sensitivity to overhead can be explained by the fact that sensitivity to gap is incurred by the program only on the portion of the messages where the application attempts to send at a rate greater than the gap. The rest of the messages are not sent quickly enough to be affected by the gap. Thus, infrequently communicating applications can potentially ignore gap entirely, while overhead is incurred independent of message frequency. The four applications with the highest communication frequency, Radix, EM3D(write) and read, and Sample, suffer the largest slowdowns from added gap. The other applications are much more tolerant to gap, slowing down by no more than a factor of 4 even in the presence of 100 μs of added gap.

Developing a model for application sensitivity to gap presents more difficulties than developing the model for sensitivity to overhead. A processor is not affected unless it attempts to send messages more frequently than the gap. At this point, the processor must stall for a period waiting for the network interface to become available. Without more precise knowledge of inter-message intervals, the model for gap sensitivity depends on assumptions made about these intervals. At one extreme, the uniform model assumes that all messages are sent at the application's average message interval, I, from Table 4. In this case, the predicted runtime, $r_{pred}^{(u)}$, can be predicted as a function of total gap, g, the average message interval, I, the base runtime, r_{base}, and the maximum number of messages sent by any node, m:

$$r_{pred}^{(u)} = \begin{cases} r_{base} + m(g - I) & \text{if } g > I \\ r_{base} & \text{otherwise} \end{cases}$$

At the other extreme, the burst model assumes that all messages are sent in discreet communication phases where the application attempts to send as fast as the communication layer will allow. Under this model, the added gap, Δg, is incurred for each communication event. This second model again assumes that the applications runs at the speed of the processor sending m messages, the maximum number of messages per processor from Figure 4, and would predict runtime, $r_{pred}^{(b)}$, as:

$$r_{pred}^{(b)} = r_{base} + m\Delta g$$

Application communication patterns determine which of the two models more closely predicts actual application runtime. The uniform model predicts that applications ignore increased gap until reaching a threshold equaling the application's average message interval. At this threshold, the applications should slowdown linearly with increasing gap. The burst model predicts a linear slowdown independent of average message interval. Given the linear dependence to gap demonstrated by the applications in Figure 6, we believe that for our applications, the burst model more accurately predicts application behavior. Table 6 depicts how well the burst model predicts actual program slowdown. As anticipated, the model over predicts sensitivity to gap since not all messages are sent in bursts. The model works best for heavily communicating applications, as a larger percentage of their messages are slowed by gap.

The two models considered demonstrate a range of possible application behavior. Applications communicating at very regular intervals would follow the uniform model, while applications communicating in discreet phases would track the burst model.

5.3 Latency

Traditionally, most attempts at improving network performance have focused on improving the network latency. Further, perceived dependencies on network latencies have led programmers to design their applications to hide network latency. Figure 7 plots application slowdown as a function of latency added to each message. Perhaps surprisingly, most of applications are fairly insensitive to added latency. The applications demonstrate a qualitatively different ordering of sensitivity to latency than to overhead and gap. Further, for all but one of the applications the sensitivity does not appear to be strongly correlated with the read frequency or barrier interval, the operations most likely to demonstrate the strongest sensitivity to latency.

g μs	Radix		EM3D(write)		EM3D(read)		Sample		Barnes	
	measure	*predict*	*measure*	*predict*	*measure*	*predict*	*measure*	*predict*	*measure*	*predict*
5.8	7.8	7.8	38	38	114	114	13.2	13.2	43.2	43.2
8	10.2	11	46.1	49.9	119	134.8	14.8	16.5	44.1	45.4
10	13	14.2	56.5	61.8	129.7	155.6	17.5	19.7	50.2	47.5
15	19.2	20.5	78.5	85.6	164.7	197.2	24.2	26.2	55.3	51.8
30	38.1	39.7	150.3	157.1	289.3	321.9	42.9	45.6	61.6	64.6
55	69.9	71.7	273.1	276.2	523	529.8	75.1	78	99.1	85.9
80	101.9	103.7	394	395.4	756.9	737.7	107.5	110.4	157.3	107.2
105	133.8	135.7	515.6	514.5	993.1	945.6	139.7	142.7	207.9	128.5

g μs	P-Ray		Murφ		Connect		NOW-sort		Radb	
	measure	*predict*	*measure*	*predict*	*measure*	*predict*	*measure*	*predict*	*measure*	*predict*
5.8	17.9	17.9	35.3	35.3	1.17	1.17	56.9	56.9	3.73	3.73
8	18.1	18.6	37.4	35.8	1.19	1.18	57.9	57.0	3.77	3.74
10	17.8	19.3	36.1	36.2	1.21	1.19	57.6	57.2	3.78	3.75
15	17.9	20.7	36.2	37.0	1.24	1.23	60.9	57.6	3.80	3.78
30	19.1	24.9	38.4	39.5	1.34	1.32	57.3	58.6	3.86	3.85
55	23.2	31.8	37.5	43.8	1.51	1.50	57.2	60.4	3.96	3.98
80	29.0	38.8	39.3	48.0	1.68	1.69	56.9	62.1	4.08	4.10
105	35.5	45.8	39.9	52.2	1.85	1.88	57.4	63.9	4.25	4.23

Table 6: **Predicted vs. Measured Run Times Varying gap.** *This table demonstrates how well the burst model for sensitivity to gap predicts observed slowdown. For each application, the column labeled $measure$ is the measured runtime, while the column labeled $predict$ is the runtime predicted by our model.*

The sensitivity of EM3D(read), Barnes, P-Ray, and Connect to latency results from these applications' high frequency of read operations (see Figure 4). Read operations require network round-trips, making them the most sensitive to added latency. However, for all but one of these applications, the observed slowdowns are modest (at most a factor of four in the worst-case) even at the latencies of store-and-forward networks (100 μs).

EM3D(read) performs a large number of blocking reads; it represents a "worst-case" application from a latency perspective because it does nothing to tolerate latency. It is also the only application for which a simple model of latency is accurate. Interestingly, for equal amounts of added "work" per message (100 μs of latency and 50 μs of overhead), the simple latency model for EM3D(read) is quite accurate yet the simple overhead model under predicts the run time.

The applications which do not employ read operations largely ignore added latency. The small decrease in performance at the tail of the slowdown curves is caused by the increase in gap associated with large latencies as the Active Message flow control mechanism limits the network capacity (see Table 2).

5.4 Bulk Gap

Only applications attempting to send large amounts of data in bursts should be affected by reductions in bulk transfer bandwidth. Note that we do not slow down transmission of small messages, but rather add a delay corresponding to the size of the message for each bulk message. Further, applications should tolerate decreases in available bulk bandwidth until the bandwidth dips below the application's requirements at any point during its execution.

Figure 8 plots application slowdown as a function of the maximum available bulk transfer bandwidth. Overall, the applications in our suite do not display strong sensitivity to bandwidth. No application slows by more than a factor of three even when bulk bandwidth is reduced to 1 MB/s. Further, all of the applications, including Radb, which moves all of its data in a single burst using bulk messages, do not display sensitivity until bulk bandwidth is reduced to 15 MB/s. Surprisingly, the NOW-sort is also insensitive to reduced bandwidth. This version of the NOW-sort uses two disks per node. Each disk can deliver 5.5 MB/s of bandwidth [**?**], and during the communication phase a single disk is used for reading and the other for writing. As Figure 8 shows, NOW-sort is disk limited. Until the network bandwidth drops below that of a single disk, NOW-sort is unaffected by decreased bandwidth.

5.5 Summary

Varying the LogGP parameters for our cluster of workstations and benchmark suite lead to a number of interesting results. Applica-

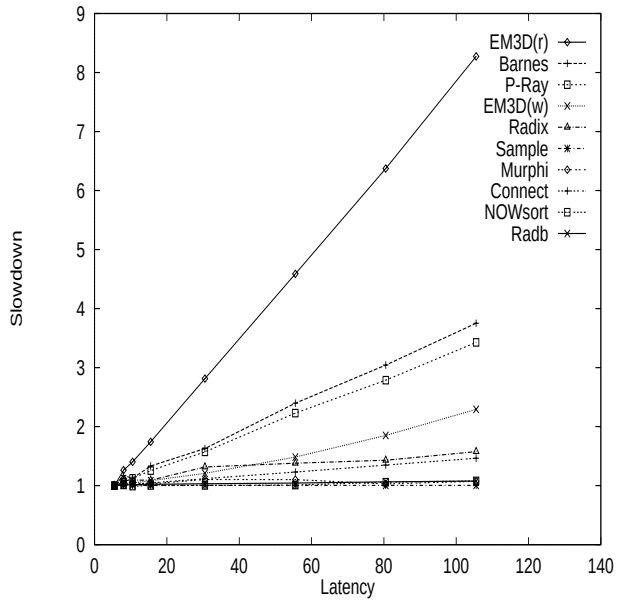

Figure 7: **Sensitivity to Latency.** *This figure plots slowdown as a function of latency in microseconds.*

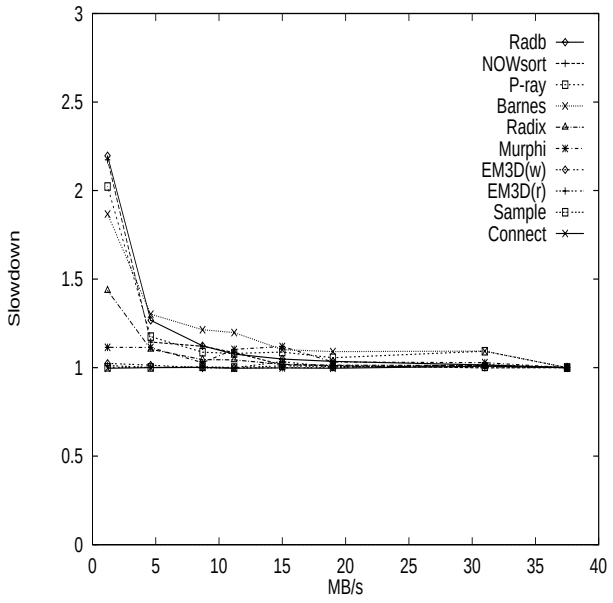

Figure 8: **Sensitivity to Bulk Gap.** *This figure plots slowdown as a function of maximum available network bandwidth.*

tions displayed the strongest sensitivity to network overhead, slowing down by as much as a factor of 50 when overhead is increased to roughly 100 μs. Even lightly communicating processes suffer a factor of 3-5 slowdown when the overhead is increased to values comparable to many existing LAN communication stacks. Frequently communicating applications also display strong sensitivity to gap suggesting that the communication phases are bursty and limited by the rate at which messages can be injected into the network. For both overhead and gap, a simple model is able to predict sensitivity to the parameters for most of our applications. Perhaps most interesting is the fact that all the applications display a linear dependence to both overhead and gap. This relationship suggest that continued improvement in these areas should result in a corresponding improvement in application performance (limited by Amdahl's Law). In contrast, if the network performance were "good enough" for the applications, (i.e. some other part of the system was the bottleneck), then we should observe a region were the application did not slow down as network performance decreased.

The effect of added latency and bulk gap is qualitatively different from the effect of added overhead and gap. Further, the effects are harder to predict because they are more dependent on application structure. For example, applications which do not perform synchronization or read operations (both of which require round trip network messages) can largely ignore added latency. For our measured applications, the sensitivity to overhead and gap is much stronger than sensitivity to latency and available bandwidth. Thus, efforts in improving overhead and gap will result in the largest performance improvements across a wide class of applications demonstrating diverse architectural requirements.

Finally, this study demonstrates an interesting tradeoff between processor performance and communication performance. For many parallel applications, relatively small improvements in network overhead and gap can result in a factor of two performance improvement. This result suggests that in some cases, rather than making a significant investment to double a machine's processing capacity, the investment may be better directed toward improving the performance of the communication system.

6 Related Work

The Flash team recently conducted a study with very similar goals under simulation [22]. This study focuses on understanding the performance requirements for a communication controller for a cache-coherent distributed memory machine. It is difficult to make a direct quantitative comparison with their results because of differences in the simulated machine, the application sets, the small problem sizes used for simulation, and the metrics used. Still, a qualitative comparison is useful. The Flash study introduces a concept of occupancy, the cycles spent in the communication controller on each end

of a message event. In their study, the message request and response is free from the processor's viewpoint. From the processor's viewpoint, occupancy adds to both the round-trip time and also slows the rate at which requests can be serviced, i.e., occupancy is part of our latency as well as gap. Their study also shows that many applications are surprisingly sensitive to occupancy. The Flash hardware prototype should provide an ideal testbed to perform a systematic *in situ* study like the one reported here in the DSM context.

The Wind Tunnel team has explored a number of cooperative shared memory design points relative to the Tempest interface through simulation and prototyping [27, 35], focused primarily on protocols. In principle, the wind tunnel provides enough power to systematically determine application sensitivity to the LogGP parameters within a given protocol, although the small local memory of the underlying CM5 may limit the study to small data sets.

Cypher [13] described the characteristics of a set of substantial message passing applications also showing that application behavior varies widely. The applications were developed in the context of fairly heavy weight message passing libraries and are more heavily biased to bulk transfers. The average message intervals are much larger than what our applications exhibit. Calculations on the data shows that actual data transfer rate is quite low. However, it is difficult to predict from the reported data how the performance would be affected with changes in the underlying message layer.

7 Conclusions

We have developed a simple empirical method for exploring the sensitivity of parallel applications to various aspects of communication performance. Using a high performance cluster as a baseline, we are able to increase each of the four communication parameters of the LogGP model — latency, overhead, message bandwidth and byte bandwidth — independently, and we have observed the effects on a broad range of applications. In general, we see that efforts to improve communication performance in a cluster architecture beyond what would be expected of a well-designed LAN are well rewarded. Applications are most sensitive to overhead, and some are hyper-sensitive to overhead in that the execution time increases with overhead at a faster rate than the message frequency would predict. Many applications are also sensitive to message rate or to message transfer bandwidth, but the effect is less pronounced than with overhead. The linear response to increased gap suggests that communication tends to be very bursty, rather than spaced at even intervals. Applications are least sensitive to network latency, the effects are uncorrelated with message frequency, and they appear complex to predict. Overall, the considered applications are surprisingly tolerant to latency.

The results suggest that there is considerable additional gain to be obtained by further reducing the communication overheads in clusters. Ongoing architectural efforts which integrate the network interface closer to the processor, either by placing the network interface onto the memory bus [8, 9, 35], into the memory controller [26], or by minimizing the number of stores and loads required to launch a communication event will continue to improve application performance.

Acknowledgments

We would like to thank Bob Horst for motivating this study. We would also like to thanks those people who provided us valuable assistance understanding the applications: Steve von Worley for P-Ray, Remzi Arpaci-Dusseau and Andrea Arpaci-Dusseau for Now-sort, Eric Anderson for Barnes, Ulrich Stern for Murφ, Chris Scheiman for Bulk Radix and Steve Lumetta for Connect and EM3D.

References

[1] Anant Agarwal, Ricardo Binchini, David Chaiken, Kik Johnson, David Kranz, John Kubiatowicz, Ben Hong Lim, Ken Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 2–13, May 1995.

[2] Albert Alexandrov, Mihai Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: Incorporating Long Messages into the LogP model - One step closer towards a realistic model for parallel computation. In *7th Annual Symposium on Parallel Algorithms and Architectures*, May 1995.

[3] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW Team. A Case for NOW (Networks of Workstations). *IEEE Micro*, February 1995.

[4] Remzi H. Arpaci, David E. Culler, Arvind Krishnamurthy, Steve Steinberg, and Kathy Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.

[5] E. Barton, J. Crownie, and M. McLaren. Message Passing on the Meiko CS-2. In *Parallel Computing*, volume 20, pages 497–507, April 1994.

[6] M.A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E.W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.

[7] S. Borkar. Supporting Systolic and Memory Communication in iWarp. In *The 17th Annual International Symposium on Computer Architecture*, pages 70–81, Seattle, WA, USA, May 1990.

[8] John B. Carter, Al Davis, Ravindra Kuramkote, Chen-Chi Kuo, Leigh B. Stoller, and Mark Swanson. Avalanche: A communication and memory architecture for scalable parallel computing. Technical Report UUCS-95-022, University of Utah, Software—Practice and Experience 1995.

[9] D. Chiou, B.S. Ang, Arvind, M.J. Beckerle, G.A. Boughton, R. Greiner, J.E. Hicks, and J.C. Hoe. StarT-NG: Delivering Seamless Parallel Computing. In *EURO-PAR'95 Conference*, August 1995.

[10] D. E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, 1993.

[11] David E Culler, Richard M. Karp, David A. Patterson, A. Sahay, Klaus E. Schauser, Eric Santos, R. Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 262–273, 1993.

[12] David E. Culler, Lok Tin Liu, Richard P. Martin, and Chad O. Yoshikawa. Assessing Fast Network Interfaces. In *IEEE Micro*, volume 16, pages 35–43, February 1996.

[13] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Atchitectural Requirements of Parallel Scientific Applications with Explict Communication. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 2–13, May 1993.

[14] W. J. Dally, J. S. Keen, and M. D. Noakes. The J-Machine Architecture and Evaluation. In *COMPCON*, pages 183–188, February 1993.

[15] D.L. Dill, A. Drexler, A.J. Hu, and C.H Yang. Protocol Verification as a Hardware Design Aid. In *International Conference on Computer Design: VLSI in Computers and Processors*, 1992.

[16] Andrea C. Dusseau, David E. Culler, Klaus E. Schauser, and Richard P. Martin. Fast Parallel Sorting Under LogP: Experience with the CM-5. In *IEEE Transactions on Parallel and Distributed Systems*, volume 7, pages 791–805, 1996.

[17] S. Frank, H. Burkhard II, and J. Rthnie. The KSR 1: Bridging the Gap Between Shared Memory and MPPs. In *COMPCON*, pages 285–294, February 1993.

[18] Richard B. Gillett. Memory Channel Network for PCI. In *IEEE Micro*, volume 16, pages 12–18, February 1996.

[19] V. G. Grafe and J. E. Hoch. The Epsilon-2 Hybrid Dataflow Architecture. In *COMPCON*, pages 88–93, March 1990.

[20] J. R. Gurd, C. C. Kerkham, and I. Watson. The Machester Prototype Dataflow Computer. In *Communications of the ACM*, volume 28, pages 34–52, January 1985.

[21] Mark Heinrich, Jeffrey Kuskin, David Ofelt, John Heinlein, Joel Baxter, Jaswinder Pal Singh, Richard Simoni, Kourosh Gharachorloo, David Nakahira, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274–285, October 1994.

[22] Chris Holt, Mark Heinrich, Jaswinder Pal Singh, Edward Rothberg, and John Hennessy. The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors. Technical Report CSL-TR-95-660, Stanford University, January 1995.

[23] E. D. Brooks III, B. C. Gorda, K. H. Warren, and T.S. Welcome. *BBN TC2000 Architecture and Programming Models*.

[24] Jonathan Kay and Joseph Pasquale. The Importance of Non-Data-Touching Overheads in TCP/IP. In *Proceedings of the 1993 SIGCOMM*, pages 259–268, San Francisco, CA, September 1993.

[25] Kimberly Keeton, David A. Patterson, and Thomas E. Anderson. LogP Quantified: The Case for Low-Overhead Local Area Networks. In *Hot Interconnects III*, Stanford University, Stanford, CA, August 1995.

[26] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[27] Alvin R. Lebeck and David A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[28] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D.S. Wells, M. C. Wond, S. Yang, and R. Zak. The Network Atchitecture of the CM-5. In *Symposium on Prallel and Distributed Algorithms*, pages 272–285, June 1992.

[29] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Implementation and Performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 92–103, May 1992.

[30] Steven S. Lumetta, Arvind Krishnamurthy, and David E. Culler. Towards Modeling the Performance of a Fast Connected Components Algorithm on Parallel Machines. In *Proceedings of Supercomputing '95*, 1995.

[31] Richard P. Martin. HPAM: An Active Message Layer for a Network of Workstations. In *Proceedings of the 2nd Hot Interconnects Conference*, July 1994.

[32] Scott Pakin, Mario Lauria, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Supercomputing '95*, San Diego, California, 1995.

[33] Greg M. Papadopoulos and David E. Culler. Monsoon: An Explicit Token-Store Architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 82–91, May 1990.

[34] Paul Pierce and Greg Regnier. The Paragon Implementation of the NX Message Passing Interface. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 184–190, May 1994.

[35] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 325–336, April 1994.

[36] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete Computer Simulation: The SimOS Approach. In *IEEE Parallel and Distributed Technology*, Fall 1995.

[37] S. Sakai, Y. Yamaguchi, K. Hiraki, Y. Kodama, and T. Yuba. An Architecture of a Dataflow Single Chip Processor. In *Proc. of the 16th Annual Int. Symp. on Comp. Arch.*, pages 46–53, Jerusalem, Israel, June 1989.

[38] Steven L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[39] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the Fifteenth SOSP*, pages 40–53, Copper Mountain, CO, December 1995.

[40] Thorsten von Eicken, David E. Culler, Steh C. Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, May 1992.

[41] Steven Cameron Woo, Moriwoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, June 1995.