An Evaluation of Current High-Performance Networks

Christian Bell† Dan Bonachea* Yannick Cote† Jason Duell† Paul Hargrove†
Parry Husbands† Costin Iancu† Michael Welcome† Katherine Yelick*†

†Computational Research Division, Lawrence Berkeley National Laboratory
*Computer Science Division, University of California at Berkeley

upc@lbl.gov http://upc.lbl.gov/

Abstract

High-end supercomputers are increasingly built out of commodity components, and lack tight integration between the processor and network. This often results in inefficiencies in the communication subsystem, such as high software overheads and/or message latencies. In this paper we use a set of microbenchmarks to quantify the cost of this commoditization, measuring software overhead, latency, and bandwidth on five contemporary supercomputing networks. We compare the performance of the ubiquitous MPI layer to that of lower-level communication layers, and quantify the advantages of the latter for small message performance. We also provide data on the potential for various communication-related optimizations, such as overlapping communication with computation or other communication. Finally, we determine the minimum size needed for a message to be considered 'large' (i.e., bandwidth-bound) on these platforms, and provide historical data on the software overheads of a number of supercomputers over the past decade.

1. Introduction

Over the past ten years, improvements in communication latency and the software overhead of communication on high-end machines have lagged far behind the exponential increases in processor performance. One reason for this trend is that most current large-scale parallel machines are constructed as clusters of workstations or personal computers: the commodity PCI buses commonly found in such systems were not designed to support parallel computation, and provide a lower level of hardware integration between the CPU/memory and networking subsystems than was characteristic of the custom-designed parallel machines built in the early 90's. Also, the emergence of MPI as the dominant parallel programming model has focused attention on message passing performance, with large message bandwidth being the most common measure of network quality. Vendors have accordingly tuned and optimized their systems for bandwidth, while software overhead and message latency have generally received less attention.

At the same time, however, there is ongoing interest in the scientific application community in using irregular data structures and communication patterns. To improve solution time, accuracy, and/or memory usage, developers often move from dense matrices to sparse ones, from structured meshes to unstructured ones, and from static algorithms to ones that adapt in space or time. These algorithms naturally involve communication of small amounts of data in a demand-driven style, such as retrieving ghost nodes in an unstructured mesh, filling in boundaries in an adaptive rectangular mesh, or sending events in an event-driven simulation. Bulk-synchronous programming models with two-sided communication can be used for these algorithms, but at a significant cost in programming complexity, since small messages are packed into large ones and point-to-point synchronization is replaced by global synchronization.

In this paper we evaluate both small and large message performance on five contemporary supercomputing networks. Using the LogP [3] model (and its extension for large messages, LogGP [1]) as a starting point for our tests and analysis, we offer a number of contributions:

- We describe a set of network benchmarks for measuring bandwidth, latency, and software overhead, which we have implemented over a wide variety of network APIs, including MPI [17], VIPL [12], SHMEM [6], LAPI [13], E-registers [6], and GM [7].
- We provide data from these benchmarks for both small and large message performance on many of the supercomputer networks in use today, and compare the performance of MPI to that of lower-level network APIs.
- Using our results, we examine various application speedups that can be achieved via network-related optimizations, such as overlapping computation with

System	Network	Bus to NIC & bandwidth	One-sided, reliable network hardware	CPU type	Non-MPI network APIs benchmarked
Cray T3E	Proprietary	Proprietary (330 MB/sec)	Yes	450 MHz Alpha	SHMEM, E-registers
IBM RS/6000 SP	SP Switch 2	GXX bus (2 GB/sec)	No	375 MHz Power 3+	LAPI
Compaq Alphaserver ES45	Quadrics	PCI 64/66 (532 MB/sec)	Yes	667 MHz Alpha	SHMEM
IBM Netfinity cluster	Myrinet 2000	PCI 32/66 (266 MB/sec)	Yes	866 MHz Pentium III	GM
PC cluster	Gigabit Ethernet (SysKonnect)	PCI 64/66 (532 MB/sec)	No	1.2 GHz Pentium III	VIPL

Table 1. Systems Evaluated

communication, pipelining messages, and the use of message packing.

- We provide a historical portrait of the trends in small message performance over the past 10 years.

Our interest in these issues grows out of our work implementing compiler and runtime support for global address space (GAS) languages, such as Unified Parallel C [22], Titanium [23], and Co-Array Fortran [2]. These languages combine the convenience of a shared memory style of programming with the control over performance of message passing. Programmers have full control over how their data is laid out across processors, and can access this data via standard mechanisms such as pointer dereferences, array indexing, or *memcpy*-style bulk copy calls. This allows serial programs (including irregular, adaptive programs that are hard to program using explicit, two-sided messaging APIs) to be parallelized incrementally (with the caveat that initial versions will rely heavily on small message performance). Compilers for GAS languages should be able to minimize some of the costs associated with the use of small messages: as we show in this paper, latency can be hidden by overlapping communication with computation or other communication. It may also be possible to perform a certain amount of message aggregation automatically. Application developers can then assess whether (and to what degree) they wish to hand-code further optimizations at the source code level, using standard techniques like message packing. The tradeoffs that this GAS model presents between programming complexity and performance will vary between applications, and will also depend on network performance. Our work in this paper is a first step toward understanding the likely performance of GAS languages for various applications on different supercomputing architectures, and the methods by which they may be optimized.

While our motivation is the performance of GAS languages, the information in this paper should also be of general interest to parallel application developers, particularly those who wish to know how large their messages need to be in order for their communication costs to be primarily bandwidth-driven. We also hope this work may influence benchmarking trends to focus more attention on software overhead and latency, and encourage vendors to minimize them in future systems. As our data shows, the current level of support for these parameters in the marketplace is quite varied, and their historical trends are not nearly as encouraging as those for most other aspects of computer performance.

2. Systems evaluated

The systems and networks measured for this paper include many of those in production use in parallel computing today, including the Cray T3E, the IBM SP, Quadrics, Myrinet 2000, and Gigabit Ethernet (GigE). Table 1 gives a high-level summary of the platforms tested, and some of their key attributes.

The machines range from custom-designed, tightly integrated systems like the T3E to loosely coupled commodity PCs connected via network cards running on industry standard PCI buses. Reflecting the predominant trend in the supercomputing marketplace, most of the systems follow the cluster of workstations approach. The T3E is the major exception, with a network interconnect designed around custom 'E-registers' integrated into the memory controller. The IBM SP uses a proprietary bus to its network card that is faster than the industry-standard PCI bus, but otherwise resembles the remaining systems in its high-level design.

All of the systems provide one or more user APIs for performing one-sided remote memory accesses, in which the remote CPU does not need to be explicitly programmed by the user to handle remote 'get' or 'put' requests. However, not all of the systems support this natively in the network hardware. Those that do mainly use an RDMA (Remote Direct Memory Access) approach, in which the CPU sends the parameters for a transfer (including its length and remote memory address) to the network card, which then handles the transfer. Again, the T3E is an exception in that it transfers data remotely via its fixed-length E-registers. In both the RDMA and T3E approaches the remote CPU is not involved, as the network hardware entirely handles servicing the memory request at the remote end. The systems that lack hardware support for remote memory operations instead service requests in software at the network library layer: while the programmer does not need to insert receive calls, the remote CPU is still involved. One of the networks, Myrinet 2000, currently performs remote 'put' operations in hardware but implements remote 'gets' in software: for this paper we only benchmark 'puts', so it is placed in the hardware supported category.

The networks also differ in whether their hardware provides reliable delivery of messages (this includes networks where firmware is used on the NIC to provide reliability). Those without such support implement it in software, again at the network library layer. All of the systems that provide hardware support for remote memory accesses also provide reliable delivery in hardware as well, and vice versa, so these features are perfectly correlated in our sample.

All the networks measured support kernel bypass: user programs are allowed to directly access the network hardware to avoid the cost of performing one or more system calls for each message. The Gigabit Ethernet cluster accomplishes this by using a brand of card (manufactured by SysKonnect) that has MVIA [18] drivers supporting the VIPL API, which avoids the system calls incurred by more commonly used IP-based protocols like TCP and UDP. The latency and overhead numbers reported here are thus likely to be better than those of most cluster systems running Gigabit Ethernet (furthermore, our SysKonnect machines were directly connected together, and so our numbers also do not reflect the delay of going through a GigE hub or switch).

3. Performance parameters

The parameters of our performance evaluation are based on those of the LogP [3] model, which is a well-established approach to modeling small message performance, and its extension to capture large message performance, LogGP [1]. Our model is quite close to LogGP, but contains certain differences in order to better reflect the observed behaviors of the networks tested. The parameters of our version are:

EEL End-to-end latency, i.e., the total time for a message, from the beginning of the send function call to the receipt of the data on the remote end.

$\mathbf{o}_s$ Send overhead, defined as the amount of time during a message send that the sending CPU is busy with transmission-related activity, and is thus unavailable for other work.

$\mathbf{o}_r$ Receive overhead. The same as send overhead, but for the receiving side of a message transmission.

g The gap, defined as the average time between messages of minimum size during a large sequence of message transmissions. Inverting this parameter gives the maximum number of messages that can be sent during a given interval.

G Additional gap per byte as messages increase in size. The reciprocal of this parameter is the effective maximum bandwidth of the network.

P The number of processors.

Most of our parameters are either identical to or only trivially different from those in the LogP/LogGP models. We have split their single software overhead parameter into separate send and receive components, since these are typically quite different on networks with hardware support for servicing remote memory operations, in which case CPU receive overhead is close to zero (one of the original LogP authors makes this same extension to the model in [4]). Our gap parameters are identical to those in LogGP. The P parameter is also identical, but since this paper is not concerned with communication patterns whose performance depends on number of processors (such as tree-based broadcasts, etc.), it does not appear in the remainder of this paper.

We make a more substantive departure from the original LogP/LogGP models in our choice to measure total end-to-end latency (EEL) instead of the traditional LogP latency term L, which refers only to the transport latency of the network hardware. The LogP/LogGP models assume that the send overhead, transport latency, and receive overhead components of a message transmission are performed serially, so that the $EEL = o_s + L + o_r$ (or, in LogGP, $EEL = o_s + L + o_r + G * bytes$). This assumption, depicted in Figure 1, is untenable on some of the networks examined in this paper, at least when nonblocking messaging functions are used (since nonblocking functions are crucial to any parallel program that wishes to overlap computation with communication, we have used them throughout our benchmarks). The call on the initiator's side to asynchronously complete a send is likely to overlap with the transport of the message, and/or with the call on the receiver's side to asynchronously begin the receive, as shown in Figure 2. On several networks, when MPI is used, we observe that the combined time for the send and receive overheads exceeds EEL, so the derived value for L is negative. It is not clear how to measure L if one relaxes the serialization assumption and allows for overlap in the model. Accordingly, the latency measurements in this paper use only EEL, with no attempt made to measure the traditional LogP network transport latency parameter.

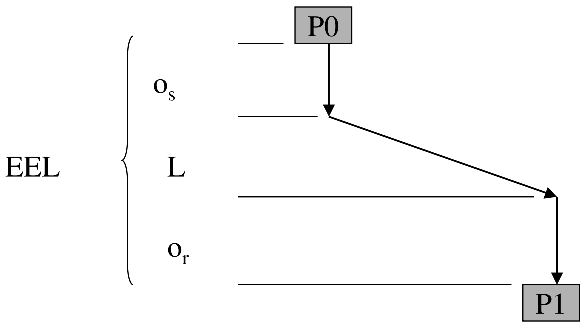

Figure 1. Traditional LogP model

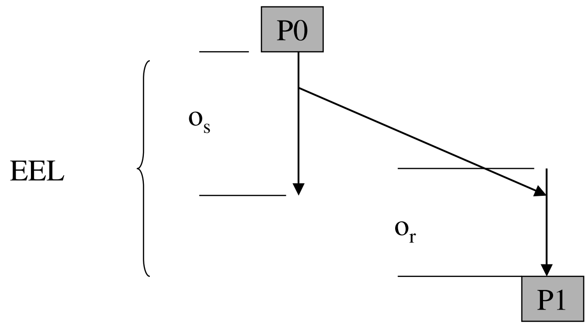

Figure 2. Observed behavior on several current networks

4. Benchmarks used

A set of three benchmarks was used to capture our parameters. On each system we ran at least two versions of these benchmarks: a common, portable MPI version, and one or more versions written using the lower-level network API(s) exposed on the machine. The APIs tested for each machine are listed in Table 1. Though the low-level benchmarks all used the same messaging logic, they did not share a common code base, so each implementation was free to adjust its logic to maximize performance for the given API within the parameters of the benchmark's semantics.

Tests were run over 10,000 messages, to amortize the cost of timer calls. Certain systems generated results that were noisy, with outliers skewing strongly on the side of longer message timings. Since we are interested in the behavior of these networks under optimal conditions, we used the minimum observed timings over 10 runs or more on each system. In practice these minimums tended to be close to the median on all but the noisiest networks (the high variance of such networks generally appeared to be the result of process contention caused by very relaxed scheduling of parallel jobs). When tests called for a minimum sized message, one eight bytes in size was used, rather than one of zero bytes, to avoid having the network APIs potentially optimize away messages. For the reasons discussed in the previous section, nonblocking 'put' operations were used for all tests.

We now discuss the logic and design of the three benchmarks, and how they were used in combination to measure our parameters.

4.1. Ping-pong test

This test sends a message of minimal size (8 bytes) to a remote processor, which receives it and replies with another such message. The first processor blocks for the reply, and then initiates the back-and-forth cycle again. This is repeated 10,000 times. The resulting total time, divided by the number of iterations, gives the average round trip time for a minimal message. Dividing that in half gives the average time taken to get from a send call on one processor to the end of the receive call on its remote counterpart, i.e., the end-to-end latency (EEL) of the network.

4.2. Flood test

The flood test measures how often messages of a given size can be injected into the network in a sustained fashion. The results provide the g and G parameters, and by implication the network bandwidth. The flood test was run over message sizes ranging from 8 bytes to 128 kilobytes, using powers of 2. We hypothesized that some networks might perform better when given multiple messages at once: depending on the network hardware and software stack, handling groups of messages can reduce locking and various other sources of overhead. To measure this, we ran the flood test with various 'queue depths.' The basic idea of a queue depth is that the benchmark program attempts to keep a certain number of sends outstanding at any given time by using non-blocking send calls: the number chosen is the queue depth q. To achieve this, the benchmark initiates q sends, waits for $q/2$ of these messages to complete locally, then issues another $q/2$ new sends. This pattern of waiting for $q/2$ completions, then issuing as many new ones, is done in a loop until all messages are issued. The last q worth of outstanding sends is then completed. Finally, the sender waits for a single reply from the receiver, to guarantee that all the messages it sent have actually completed at the other end. This final message is necessary since most non-blocking network APIs only guarantee upon completion that the buffer used to send the message is available for reuse, not that the message has been received on the other end of the network connection. For some APIs (such as the SHMEM API on the T3E and Quadrics) a 'quiet' function is available which provides a similar guarantee that all messages have been delivered. Regardless of which method is used to guarantee delivery, for sufficiently large N this final reply or quiet call is amortized away, and dividing the total time by the number of messages provides the gap parameter g when minimum sized messages are used. The per-byte gap cost G can then be calculated by taking the difference between g and the average times for larger message sizes.

4.3. CPU overlap test

The CPU overlap test determines the amount of software overhead involved in sending and receiving messages. The code is identical to the flood test with a message size of 8 bytes and a queue depth of one, except that an increasing amount of computation is gradually inserted into the program. This computation is placed between the calls that initiate and complete a non-blocking send operation. Sufficiently small amounts of computation will not make a difference in the benchmark's timing for most networks, since the call to complete the send would have blocked anyway. As the amount of computation per message is increased, eventually the sum of the send overhead (o_s) and the computation exceeds the gap g, and the total running time for the benchmark is increased. By taking the maximum amount of CPU time that can be added without increasing the benchmark time, and subtracting it from g, the send overhead (o_s) can be inferred. For interfaces where the send overhead is so large as to determine the gap, even a trivial amount of inserted computation increases the benchmark's time, and so $o_s = g$. Receive side overhead o_r is measured in a similar manner by inserting computation in between the start and completion of non-blocking receive calls. For APIs where no receive calls are used on the remote side, this works differently: the computation is done in isolation, followed by a 'quiet' call to guarantee all messages have been received.

5. Results

5.1. Comparing small message latency

The total height of the bars in Figure 3 gives the end-to-end latency (EEL) of the networks we examined for an 8-byte send. The EEL is a good indication of how an application will perform when it blocks for the completion of a small message, such as during the fetch of a single value for immediate use. However, since our tests were written using nonblocking send APIs (as the overlapping operations we next discuss require them), a program using blocking functions—which typically incur less software overhead than nonblocking APIs—might see a slightly lower EEL than reported here. For uniformity we report only times for 'puts', even for APIs which also support remote 'gets'.

As our results clearly show, the end-to-end latency of current high-performance networks varies considerably, with a factor of ten separating the smallest and largest observed values. These differences are not clearly correlated to system bandwidth, nor to the tightness of integration between the CPU/memory system and the network interface: for instance, the IBM SP has much higher latency than the other systems despite having the fastest bus between the memory and the network controller.

On most machines, the performance of the more specialized layers is better than MPI. This is most pronounced on the Quadrics system. The exception is the IBM machine, where LAPI is somewhat slower than MPI. This appears to be due to higher locking overhead: we report data for a single-threaded MPI library, but LAPI currently has only a thread-safe implementation (if the benchmark is run on the SP with a thread-safe version of the MPI library, its latency is higher than LAPI's).

5.2. Potential for overlapping computation

When optimizing communication, a standard technique is to overlap the communication with computation. Figure 3 shows the potential for latency hiding by superimposing the send and receive overhead on the end-to-end latency (EEL). As discussed in Section 3, several of the APIs show a combined send and receive overhead that is higher than the EEL, thus violating the assumptions of the traditional LogP model.

These results show the potential benefits of overlapping communication with computation. For example, while the EEL of Myrinet/GM is 8.5 usec compared to 1.7 usec on Quadrics/SHMEM, the two become much closer when considering only send side overhead, which is 0.7 usec on Myrinet and 0.4 usec on Quadrics. With $8.5 - 0.7 = 7.8$ usec of potential overlap time on Myrinet/GM, and $1.7 - 0.4 = 1.3$ usec of Quadrics/SHMEM, overlapping has much higher potential on Myrinet/GM.

The results also help explain the EEL differences between MPI and the other network layers. MPI implementations generally incur a significant software overhead, and this tends to be reflected in higher latencies. This is not surprising, given that most of the MPI implementations tested here are built on top of the other network APIs, and can thus only add additional costs (the one MPI implementation that beats the alternative API's latency is the IBM SP's, which is not built on top of LAPI, and which benefits from lower locking overheads, as described above). However, networks which depart from the classic LogP assumption of the serialization of overhead costs can display much lower latencies than LogP would predict. For example, the T3E's relatively low MPI latency is mainly due to the overlap of its send and receive overheads: if these were executed serially, its EEL would be over 12 microseconds. One side effect of this lower latency is that very little computational overlap is possible during messaging. On some networks, particularly Myrinet, the cost of using MPI instead of a lower-level API is thus a 'hidden' one: latencies are similar, but the opportunity for computational overlap is greatly reduced.

Another trend highlighted by the data is the significant improvement that hardware support for one-sided remote

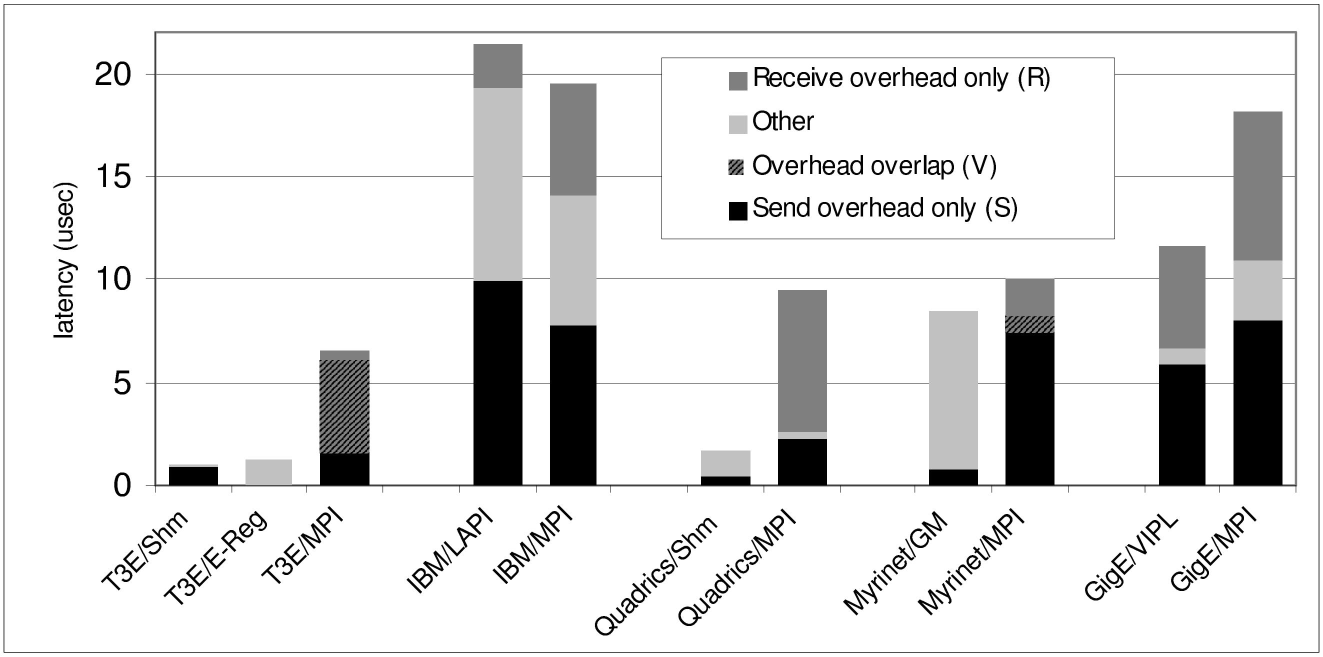

Figure 3. Send and receive software overheads (o_s and o_r) superimposed on end-to-end latency (EEL). For MPI on the T3E and Myrinet, the sum of the overheads is greater then EEL, and so $o_s = S + V$ and $o_r = R + V$. For the other configurations $o_s = S$ and $o_r = R$.

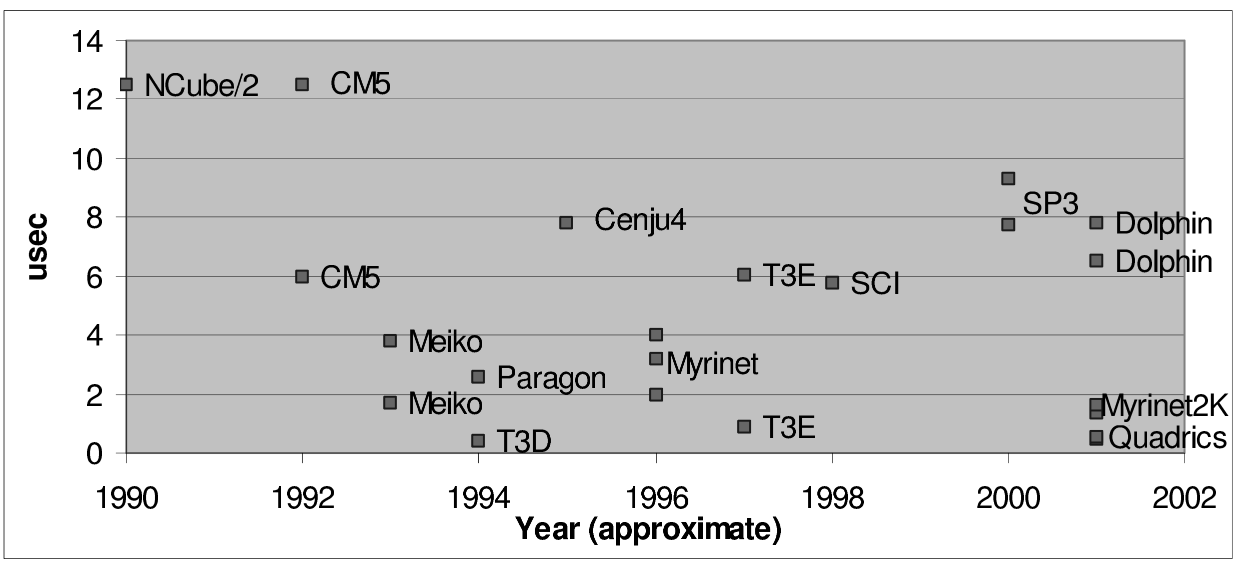

Figure 4. Software send overhead for small messages on selected supercomputers since 1990. For systems with two data points, the larger figure is for MPI and the smaller for a lower-level API provided by the system.

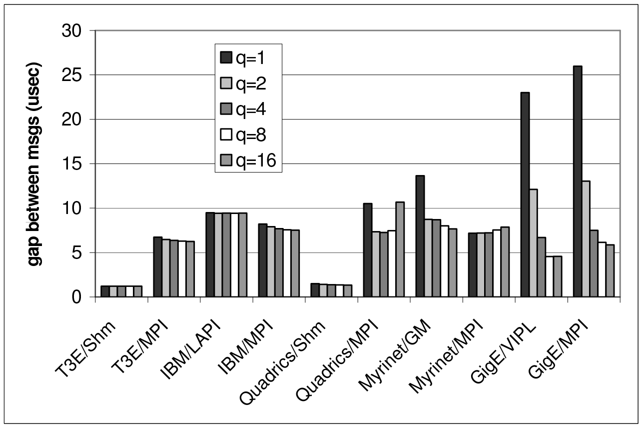

Figure 5. Effect of clustering 8-byte sends on message gap

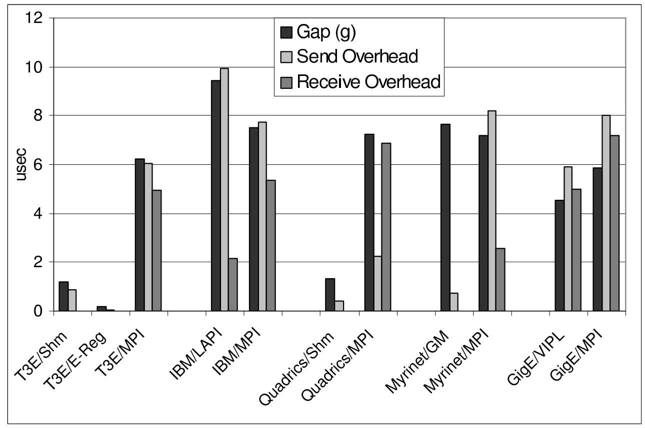

Figure 6. Gap and overheads for 8-byte messages

memory operations tends to make for both latency time and software overhead. In particular, the receive overhead for one-sided APIs on networks with hardware support for remote operations (the T3E, Quadrics, and Myrinet) is effectively zero. For an application in which processors are acting equally as senders and receivers, the amount of computational overlap possible is given by $EEL - (o_s + o_r)$, and so machines with zero receive overhead have a large advantage.

5.3. Potential for overlapping communication

Another strategy for optimization is to overlap communication with more communication. Rather than filling idle CPU time with computation, as in the previous section, it can be used to send additional messages. Figure 5 shows the gap (g) between 8-byte messages from our flood test, which pushes multiple messages into the network as quickly as possible. On each machine, we vary the queue depth (q) parameter, which indicates the number of messages that are simultaneously awaiting transmission.

As predicted, for some networks and APIs, clustering messages results in greater efficiency, as shown by the reduction in the gap as q is increased. This is most notable on Quadrics/MPI, Myrinet/GM, and both GigE layers. On these machines, communication overlap is especially valuable, as it not only uses up otherwise idle CPU time, but also reduces the amount of idle time that needs to be filled.

5.4. Combining both types of overlap

On some systems, it may be possible to combine both forms of overlap, first initiating as large a group of clustered messages as is possible, then filling any remaining time with computation. The degree to which this strategy is possible in practice is difficult to determine, as it may be largely limited by an application's structure: the number of messages available for clustering and the amount of computation that can be performed before the messages are completed will vary with a program's semantics. Also, the size of the messages that are used may change the relative sizes of the communication gap and software overhead. Nevertheless, we can get a rough idea of which systems are likely to have extra CPU time available for computation during clustering from Figure 6, which compares the gap between 8-byte messages with the send and receive software overhead per message. The chart shows the best g obtained from all queue depths tested on each machine, while the overheads were measured only with $q = 1$. As a result, on certain platforms (the SP, GigE, and Myrinet/MPI) the send overhead appears to be higher than the gap: presumably clustering lowers the send overhead on these systems, much as it does the gap, but we have not measured this. Despite this complication, it is clear from the chart that certain platforms are more likely to have CPU time left over for computation during clustering after the overhead of message traffic is subtracted from the gap. Myrinet/GM, Quadrics/SHMEM, and T3E/SHMEM are the clearest cases of this. Since these APIs require no CPU activity to receive a message, they are likely to support computational overlap even when an application is simulaneously sending and receiving clusters of messages.

The cumulative effect of Figures 5 and 6 suggests that using lower-level, one-sided network APIs is advantageous for parallel applications which frequently send small messages: the latencies and software overheads incurred by each message tend to be considerably lower than when MPI is used. Furthermore, these APIs generally provide more potential for reducing the cost of communication via overlapping either communication or computation. While the ideal strategy is probably still to coalesce small messages whenever possible, this can be difficult in applications

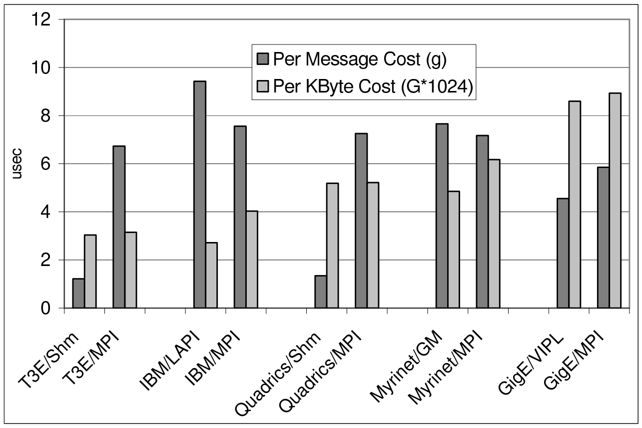

Figure 7. Per-message gap g vs. large message per-byte gap G

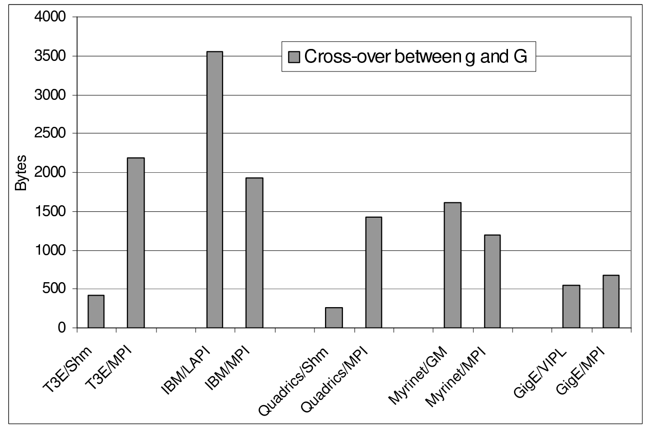

Figure 8. Minimum size for 'large' (bandwidth-bound) messages

where each processor is communicating with many targets. The ability to optimize communication through overlap is very promising in the context of optimizing GAS compilers, since some of these languages allow applications programmers to explicitly indicate that overlapping or reordering accesses is legal [22].

5.5. Large message performance

A third type of communication optimization is to aggregate small messages into large ones. Estimating the cost of such a transformation involves a detailed understanding of factors (such as memory system performance) which are beyond the scope of this paper. Another important factor is the size of messages in the application, before and after packing. In the absence of such information, we can still compute an upper bound on the speedup possible from packing, by calculating the maximum bandwidth achievable for the aggregated message size. This is given by the cost per byte (G) in LogGP.

Figure 7 shows the two different gap values for each network: the inverse throughput (g) for small messages, which gives a lower bound on the per-message cost, and the inverse bandwidth for large messages (G), which gives a lower bound on the per byte cost. These two values differ by about three orders of magnitude, so we show G in microseconds per kilobyte rather than per byte.

One would expect the inverse bandwidth (G) to be determined primarily by the hardware, rather than varying with the communication layer. On most machines this is true, but both the IBM and Myrinet show some loss of bandwidth (increased G) for MPI relative to LAPI and GM, respectively. Ignoring these differences, the machines are roughly sorted by their G value, with the highest bandwidth machine being the T3E, and the lowest being GigE. There is little correlation between G and the per-message gap g, which varies between 1 and 10 usecs.

One common question in performance analysis is whether a given application is more sensitive to the latency or the bandwidth of the network. This is the same as asking whether the average message sent by the application is 'large' (bandwidth-bound), or 'small' (bound by fixed per-message costs). Within a context where one is clustering messages, the relevant fixed cost per message is the base gap g between small messages rather than the latency. Assuming that the cost during pipelining of a message of size M is roughly $g + M * G$, we can determine the size at which the two terms have equal weight, i.e., where $M = g/G$. Any message larger than g/G will be dominated by the bandwidth term ($M * G$), and can thus be considered 'large.'

Figure 8 shows that the minimum size for a large message varies greatly in the networks under consideration: the difference is a factor of six from the smallest to the largest crossover size.

6. Historical trends in performance

As our benchmarks show, the level of support for small message performance is quite disparate among contemporary high-performance network architectures. Figure 4, which shows software overheads for small message 'puts' on various systems over the last decade, demonstrates that this variability is part of a historical pattern.

While most parameters in computing—from CPU speed to memory latency to hard disk seek time—show at least gradual improvement over time, software overheads and message latencies for network transmissions have exhibited a comparatively amorphous behavior whose trend is generally toward worse performance, even by absolute measures. While these parameters are unlikely to ever keep pace with the exponential improvements in processor performance, we hope that they can at least exhibit a positive trend in the future, and that support for them may become more consistent across vendors. The likely replacement in commodity PCs of the aging PCI bus standard in favor of newer technologies like Infiniband [10], PCI-X [20], and HyperTransport [8] may be a positive development, for instance, as the PCI bus is currently a known bottleneck for many of the networks examined here.

7. Related work

Historically, performance assessment of communication networks has been performed using two different methodologies. One school of thought is primarily interested in round-trip latency and large message bandwidth as indicators of network performance. Dongarra et al. [5] measure latency and bandwidth for a large class of multiprocessor systems: Convex, Cray, IBM, Intel, KSR, Meiko, nCUBE, NEC, SGI and TMC using a ping-pong benchmark. Luecke et al. [14] evaluate the communication performance of Linux and NT clusters, the Cray Origin 2000, IBM SP and Cray-T3E. More recently, Petrini et al. [21] examine the performance of Quadrics networks using uni- and bi-directional ping benchmarks.

A different school of thought adopts a more detailed model of the network performance. In 1993, Culler et al. [3] introduced the LogP performance model of parallel computation. Their model is built upon the realization that modern parallel systems are essentially comprised of complete computers connected by a communication fabric. Culler et al. [4] measured the model's parameters for the Intel Paragon, Meiko CS-2 and Myrinet using an approach similar to our flood and CPU overlap benchmarks. Ianello et al. [9] measure the same parameters for an implementation of Fast Messages [19] running on Myrinet. Further research has extended the LogP model to take into account other factors that influence application performance, and tailor the model for different communication layers (e.g., MPI) and architectures. Alexandrov et al. extend it with G (LogGP [1]) to capture the cost of large messages. Moritz and Frank further extend LogGP (LoGPC [16]) to take into account the effects of message pipelining and network contention. Al-Tawil and Moritz [15] use LogGP to analyze the behavior of MPI under the different send protocols mandated by the standard. Ino et al. tailor the LogGP model (LogGPS [11]) to account for the synchronization costs hidden in MPI implementations. Using LogGPS, they are able to determine the threshold where an MPI implementation switches from an asynchronous to a synchronous protocol.

Our approach in this study of network performance occupies a middle ground between these methodologies. We find the parameters of the LogP model to be very useful in understanding program performance and also in guiding program optimizations. However, the increased hardware and software complexity of modern systems results in an observable behavior that does not fit the original model. In particular, the assumption that the one-way message latency is given by the sum $L + o_s + o_r$ does not hold for some systems (T3E/MPI and Myrinet/MPI). This is the justification for our choice of the EEL parameter as a better indication of network performance.

8. Future work

There are many directions in which the work in this paper should be extended. The current benchmarks test only message 'put' operations; gathering information on the performance of 'get' operations has obvious relevance for operations like prefetching of data. We measure software overhead only for unclustered 8-byte messages, and it would be interesting to know the extent to which these overheads increase with message size (the answers might be different for networks with hardware support for remote transfers versus those without it), and/or decrease with clustering (i.e., with the q parameter in our flood test). Also, we have run our tests only on 2-node configurations: some of our results might vary when more nodes are used (in particular, it would be interesting to see if the benefits of clustering still apply when messages have different target nodes). Our tests have all been performed either on uniprocessor machines, or on SMPs with only one processor in use. Given that most production cluster architectures use SMP nodes, it would be a useful contribution to measure the effect that contention for the network and other resources has on our parameters. As mentioned previously, our estimates for the maximum speedup that can be attained from message packing are based only on the bandwidth figures for the various networks. A more realistic assessment of the likely speedups attainable would need to measure the cost of packing and unpacking aggregated messages, which add significant costs to this strategy. Finally, research needs to be conducted into how the metrics presented here can be used by compilers for global address space languages to perform optimizations of programs.

9. Conclusion

Using a variant of the LogGP model, we constructed a set of microbenchmarks to measure different aspects of network performance. The results provided a comparison of today's networks across factors such as latency, bandwidth, and software overheads. Hardware support for network reliability and servicing of remote memory requests were found to be key factors in attaining better small message performance. One-sided APIs were generally found to have superior latencies and software overheads compared with two-sided MPI implementations on the same hardware. Some of the assumptions of the LogP/LogGP model turned out to be untenable on existing MPI implementations.

We examined the relative cost of different programming strategies for an application that has a fixed volume of communication to perform. Using as our starting point a naïve model in which each read and write is done as a blocking remote access of a single 8-byte word, we quantified the relative benefits of three standard optimization techniques: overlapping communication with computation, overlapping communication with other communication (pipelining), and aggregating small messages into large ones. Our results suggest that compiler-based communication scheduling may be needed for optimal performance, especially for programs that require the use of fine-grained, asynchronous communication: some of the trade-offs between optimizations are highly dependent on the particular system in use.

10. Acknowledgments

This work was supported in part by the Department of Energy under contracts DE-FC03-01ER25509 and DE-AC03-76SF00098, by the National Science Foundation under ACI-9619020 and EIA-9802069, and by the Department of Defense. The information presented here does not necessarily reflect the position or the policy of the United States Government and no official endorsement should be inferred.

We wish to thank Oak Ridge National Laboratory for the use of their Alpha-based Quadrics system ('falcon'), and the National Energy Research Scientific Computing Center (NERSC) for the use of their T3E ('mcurie'), IBM SP ('seaborg'), and Myrinet 2000 cluster ('alvarez'). The 'Millennium' PC cluster at U.C. Berkeley was also used for some preliminary Myrinet development. Thanks also go to David Addison and Ashley Pittman of Quadrics, who provided very useful assistance at various points in our benchmark development.

References

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[2] Co-Array Fortran technical specification. http://www.co-array.org/caf_def.htm.

[3] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

[4] D. E. Culler, L. T. Liu, R. P. Martin, and C. Yoshikawa. LogP performance assessment of fast network interfaces. *IEEE Micro*, February 1996.

[5] J. J. Dongarra and T. Dunigan. Message-passing performance of various computers. *Concurrency: Practice and Experience*, 9(10):915–926, 1997.

[6] Cray T3E C and C++ optimization guide. http://www.cray.com/craydoc/manuals.

[7] GM reference manual. http://www.myri.com/scs/GM/doc/refman.pdf.

[8] HyperTransport I/O Link specification, version 1.04. http://www.hypertransport.org/docs/Htlink1_04.pdf.

[9] G. Iannello, M. Lauria, and S. Mercolino. LogP performance characterization of Fast Messages atop Myrinet. In *Proceedings of 6th Euromicro Workshop on Parallel and Distributed Processing*, 1998.

[10] Infiniband specification. http://www.infinibandta.org/specs.

[11] F. Ino, N. Fujimoto, and K. Hagihara. LogGPS: Modeling message-passing protocols in high-level communication libraries. In *IPSJ Transactions on High Performance Computing Systems*, volume 42, 2001.

[12] Intel Virtual Interface Architecture developer's guide. http://developer.intel.com/design/servers/vi/developer/ia_imp_guide.htm.

[13] Using the LAPI. http://www.research.ibm.com/actc/Opt_Lib/LAPI_Using.htm.

[14] G. Luecke, B. Raffin, and J. Coyle. Comparing the communication performance and scalability of a Linux and an NT cluster of PCs, a SGI Origin 2000, an IBM SP and a Cray T3E-600. In *Proceedings of IEEE Computer Society International Workshop on Cluster Computing*, 1999.

[15] C. A. Moritz, K. Al-Tawil, and B. Fraguela. Performance comparison of MPI on MPP and workstation clusters. In *Proceedings of the 10th ISCA International Conference on Parallel and Distributed Computer Systems*, 1997.

[16] C. A. Moritz and M. Frank. LoGPC: Modeling network contention in message-passing programs. In *Measurement and Modeling of Computer Systems*, pages 254–263, 1998.

[17] The Message Passing Interface standard. http://www-unix.mcs.anl.gov/mpi/.

[18] M-VIA: A high performance modular VIA for Linux. http://www.nersc.gov/research/FTG/via.

[19] S. Pakin, M. Lauria, and A. Chien. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, December 1995.

[20] PCI-X 2.0 specification. http://www.pcisig.com/specifications/pci_x_20.

[21] F. Petrini, A. Hoisie, W. Feng, and R. Graham. Performance evaluation of the Quadrics interconnection network. In *Workshop on Communication Architecture for Clusters*, 2001.

[22] UPC language specification, version 1.0. http://upc.gwu.edu.

[23] K. Yelick, L. Semenzato, G. Pike, et al. Titanium: A high-performance Java dialect. ACM 1998 Workshop on Java for High-Performance Network Computing, Februrary 1998.