Fast Measurement of LogP Parameters for Message Passing Platforms

Thilo Kielmann, Henri E. Bal, and Kees Verstoep

Department of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
kielmann@cs.vu.nl bal@cs.vu.nl versto@cs.vu.nl

Abstract. Performance modeling is important for implementing efficient parallel applications and runtime systems. The LogP model captures the relevant aspects of message passing in distributed-memory architectures. In this paper we describe an efficient method that measures LogP parameters for a given message passing platform. Measurements are performed for messages of different sizes, as covered by the *parameterized LogP* model, a slight extension of LogP and LogGP. To minimize both intrusiveness and completion time of the measurement, we propose a procedure that sends as few messages as possible. An implementation of this procedure, called the *MPI LogP benchmark*, is available from our WWW site.

1 Introduction

Performance modeling is important for implementing efficient parallel applications and runtime systems. For example, *application-level schedulers* (AppLeS) [2] aim to minimize application runtime based on application-specific performance models (e.g., for completion times of given subtasks) which are parameterized by dynamic resource performance characteristics of CPUs and networks. An AppLeS may, for example, determine suitable data distributions and task assignments based on the knowledge of message transfer times and computation completion times.

Another example for the use of performance models is our MagPIe library [8, 9] which optimizes MPI's collective communication. Based on a model for the completion times of message sending and receiving, it optimizes communication graphs (e.g., for broadcast and scatter) and finds suitable segment sizes for splitting large messages in order to minimize collective completion time.

The LogP model [4] captures the relevant aspects of message passing in distributed-memory systems. It defines the number of *processors* P, the network *latency* L, and the time (*overhead*) o a processor spends sending or receiving a message. In addition, it defines the *gap* g as the minimum time interval between consecutive message transmissions or receptions at a processor, which is the reciprocal value of achievable end-to-end bandwidth. Because LogP is intended for short messages, o and g are constant. The LogGP model extends LogP to also cover long messages [1]. It adds a parameter G for modeling the *gap per byte* for long messages, which are typically handled more efficiently. Other variants of LogP have also been proposed where the overhead at the sender and the receiver side is treated separately as o_s and o_r, and where some parameters depend on the message size [5, 7, 8].

For practical use of LogP, the actual parameters of a parallel computing platform have to be measured. Inside a supercomputer or workstation cluster, the network performance characteristics remain constant, except for possible changes in system software. In this case, the respective LogP parameters may be measured offline, and measurement efficiency hardly matters. Our MagPIe library, however, targets multiple clusters connected via wide-area networks. In this context, off-line measurements are not feasible due to two reasons, so measurement efficiency is very important. First, intrusiveness on other ongoing communication has to be kept as small as possible. Second, the performance of wide-area networks may change during application runtime [11], causing measurements also to be performed regularly.

The main problem with measurement efficiency is how to accurately measure the *gap* parameter. The measurement methods described in [5, 7] measure the *gap* by sending large sequences of messages in order to saturate the communication links in which case the link capacity (as expressed via the gap) can be observed. This measurement procedure has two drawbacks. It is highly intrusive and may disturb other ongoing communication. Also, it is time consuming when measuring long messages, especially when the network has high latency and/or low bandwidth, as is the case with wide-area connections as targeted by MagPIe.

In this paper, we present a procedure that measures LogP parameters without saturating the network with long messages. Only for empty messages (with zero bytes of data), the gap has to be determined by saturating the network. This can be achieved in reasonable time even across wide area links. For all other message sizes, simple message roundtrips (and the gap for empty messages) are sufficient to determine the corresponding LogP parameters. In the remainder of the paper, we briefly clarify the LogP variant we use (*parameterized LogP* [8]), then we describe our measurement procedure and compare our measurements with results obtained by saturation-based measurements.

2 Parameterized LogP

The *parameterized LogP* model defines five parameters, in analogy to LogP. P is the number of processors. L is the end-to-end latency from process to process, combining all contributing factors such as copying data to and from network interfaces and the transfer over the physical network. $o_s(m)$, $o_r(m)$, and $g(m)$ are send overhead, receive overhead, and gap. They are defined as functions of the message size m. $o_s(m)$ and $o_r(m)$ are the times the CPUs on both sides are busy sending and receiving a message of size m. For sufficiently long messages, receiving may already start while the sender is still busy, so o_s and o_r may overlap. The gap $g(m)$ is the minimum time interval between consecutive message transmissions or receptions. It is the reciprocal value of the end-to-end bandwidth from process to process for messages of a given size m. Like L, $g(m)$ covers all contributing factors. From $g(m)$ covering $o_s(m)$ and $o_r(m)$, follows $g(m) \geq o_s(m)$ and $g(m) \geq o_r(m)$. A network N is characterized as $N = (L, o_s, o_r, g, P)$.

To illustrate how the parameters are used, we introduce $s(m)$ and $r(m)$, the times for sending and receiving a message of size m when both sender and receiver simultaneously start their operations. $s(m) = g(m)$ is the time at which the sender is ready to send the next message. Whenever the network itself is the transmission bottleneck, $o_s(m) < g(m)$, and the sender may continue computing after $o_s(m)$ time. But because $g(m)$ models the time a message "occupies" the network, the next message cannot be sent before $g(m)$. $r(m) = L + g(m)$ is the time at which the receiver has received the message. The latency L can be seen as the time it takes for the first bit of a message to travel from sender to receiver. The message gap adds the time after the first bit has been received until the last bit of the message has been received. Figure 1 (left) illustrates this modeling. When a sender transmits several messages in a row, the latency will contribute only once to the receiver completion time but the gap values of all messages sum up. This can be expressed as $r(m_1, m_2, \ldots, m_n) = L + g(m_1) + g(m_2) + \ldots + g(m_n)$.

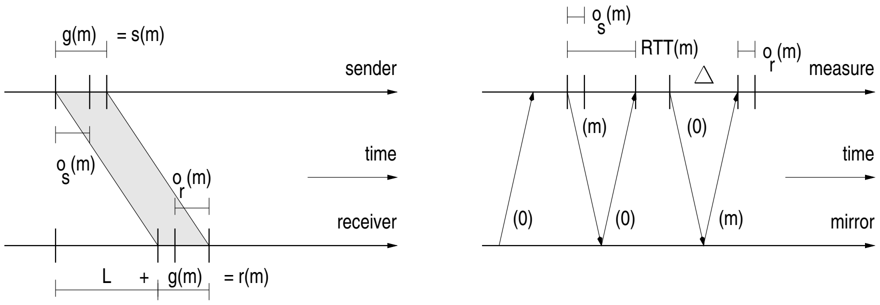

Fig. 1. Message transmission as modeled by parameterized LogP (left); fast measurement procedure (right)

For completeness, we show that *parameterized LogP* subsumes the original models LogP and LogGP. In Table 1, LogGP's parameters are expressed in terms of *parameterized LogP*. We use 1 byte as the size for short messages; any other reasonable "short" size may as well be used instead. Note that neither LogP nor LogGP distinguishes between o_s and o_r. For short messages, they use $r = o + L + o$ to relate the L parameter to receiver completion time which gives L a slightly different meaning compared to *parameterized LogP*. We use this equation to derive LogP's L from our own parameters.

3 Fast parameter measurement

Previous LogP micro benchmarks [5, 7] measure the gap values by saturating the link for each message size. Our method has to use saturation only for obtaining $g(0)$. As we use $g(0)$ for deriving other values, we measure it first. We measure the time RTT_n for a roundtrip consisting of n messages sent in a row by *measure*, and a single, empty reply message sent back by *mirror*. The procedure starts with $n = 10$. The number of messages n is doubled until the gap per message changes only by $\epsilon = 1\%$. At this point, saturation is assumed to be reached. We take the time measured for sending the so-far largest number of messages (without reply) as $n \cdot g(0)$. We start with a small number of messages in a row in order to speed up the measurement. So we have to ensure that the messages are sufficiently many such that the roundtrip time is dominated by bandwidth rather than latency. Therefore, we also keep doubling n until the inequality $RTT_1 < \epsilon \cdot RTT_n$ holds. By waiting for a reply we enforce that the messages are really sent to *mirror* instead of just being buffered locally.

Table 1. LogGP's parameters expressed in terms of *parameterized LogP*

LogP/LogGP		parameterized LogP
L	=	$L + g(1) - o_s(1) - o_r(1)$
o	=	$(o_s(1) + o_r(1))/2$
g	=	$g(1)$
G	=	$g(m)/m$, for a sufficiently large m
P	=	P

All other parameters can be determined by the procedure shown in Fig. 1 (right). It starts with a synchronization message by which the so-called *mirror* process indicates being ready. For each size m, two message roundtrips are necessary from *measure* to *mirror* and back. (We use $RTT(m) = RTT_1(m)$.) In the first roundtrip, *measure* sends an m-bytes message and in turn receives a zero-bytes message. We measure the time for just sending and for the complete roundtrip. The send time directly yields $o_s(m)$. $g(m)$ and L can be determined by solving the equations for $RTT(0)$ and $RTT(m)$, according to the timing breakdown in Fig. 1 (left):

$$RTT(0) = 2(L + g(0)) \qquad RTT(m) = L + g(m) + L + g(0)$$
$$g(m) = RTT(m) - RTT(0) + g(0) \qquad L = (RTT(0) - 2g(0))/2$$

In the second roundtrip, *measure* sends a zero-bytes message, waits for $\Delta > RTT(m)$ time, and then receives a m-bytes message. Measuring the receive operation now yields $o_r(m)$, because after waiting $\Delta > RTT(m)$ time, the message from *mirror* is available at *measure* immediately, without further waiting.

For each message size, the roundtrip tests are initially run a small number of times. As long as the variance of measurements is too high, we successively increase the amount of roundtrips. We keep adding roundtrips until the average error is less than ϵ, or until an upper bound on the total number of iterations is reached (60 for small messages, 15 for large messages).

Initially, measurements are performed for all sizes $m = 2^k$ with $k \in [0, k_m]$. The value of k_m has to be chosen big enough to cover any non-linearity caused by the tested software layer. In our experiments, we used $k_m = 18$ to cover all changes in send modes of the assessed MPI implementation (MPICH).

After measuring the initial set of message sizes, we check whether the gap per byte $(g(m)/m)$ has stabilized for large m. If this is not the case, sending larger messages may achieve lower gaps (and hence higher throughput). So k_m is incremented and the next message size is tested. This process is performed until $g(2^{k_m})$ is close (within ϵ) to the value linearly extrapolated from $g(2^{k_m-2})$ and $g(2^{k_m-1})$.

So far, the "interesting" range of message sizes has been determined. Finally, possible non-linear behavior remains to be detected. For any size m_k, we check whether the measured values for $o_s(m_k)$, $o_r(m_k)$, and $g(m_k)$ are consistent with the corresponding, predicted values for size m_k, extrapolated from the measurements of the previous two (smaller) message sizes, m_{k-1} and m_{k-2}. If the difference is larger than ϵ, we do new measurements for $m = (m_{k-1}+m_k)/2$, and repeat halving the intervals until either the extrapolation matches the measurements, or until $m_k - m_{k-1} \leq \max(32 \text{ bytes}, \epsilon \cdot m_k)$.

3.1 Limitations of the method

Except for measuring $g(0)$, all parameters are derived from pairs of single messages sent between the *measure* and *mirror* processes. The correctness of timing these messages relies on the independence of the message pairs from each other: the time it takes to send a message from *measure* to *mirror* and back must always be the same, whether or not other messages have been exchanged before. Whenever *measure* issues several messages in a row, sending is slowed down to the rate at which the message pipeline is drained. This exactly is the effect used to measure $g(0)$. For all other measurements, we avoid this effect by always sending messages in pairs from *measure* to *mirror* and back. Before *measure* may send the next message, it first has to receive from *mirror*. This procedure enforces that pipelines will always be drained between individual message pairs, assuming that message headers carry "piggybacked" flow control information that resets senders to their initial state after each message roundtrip. This assumption may fail for communication protocols which update their flow control information in a more lazy fashion. So far, we found our assumption to be reasonable, as it works both with TCP and with our user-level Myrinet control software LFC [3].

In some cases, our measurements reveal values for the receive overhead such that $o_r(m) > g(m)$ which seems to contradict *parameterized LogP*. This phenomenon is caused by different behavior of the receive operation depending on whether the incoming message is *expected* to arrive. Messages are expected to arrive whenever the application called a matching receive operation before the message actually arrives at the receiving host. The treatment of expected messages may be more efficient because unexpected messages, for example, may have to be copied to a separate receive buffer, before they can later be delivered to the application. In our measurement procedure, $o_r(m)$ is measured with unexpected messages whereas $g(m)$ is measured while receiving expected messages. Whenever $o_r(m) > g(m)$, $g(m)$ gives an upper bound for processing expected messages. With synchronous receive operations, this measurement setup is unavoidable, because otherwise the measured receive overhead cannot be separated from the time waiting for the message to arrive. (With our MPI-based implementation, we can also measure the receive overhead of expected messages for the asynchronous receive operation, *MPI_Irecv*, in combination with *MPI_Wait*.)

The measurement procedure described above assumes that network links are symmetrical, such that sending from *measure* to *mirror* has the same parameters as for the reverse direction. However, this assumption may not always be true. On wide area networks, for example, the achievable bandwidth (the gap) and/or the network latency may be different in both directions, due to possibly asymmetric routing behavior or link speed. Furthermore, if the machines running the *measure* and *mirror* processes are different (like a fast and a slow workstation), then also the overhead for sending and receiving may depend on the direction in which the message is sent. In such cases, the parameters o_s, o_r, and g may be measured by performing our procedure twice, while switching the roles of *measure* and *mirror* in between. Asymmetric latency can only be measured by sending a message with a timestamp t_s, and letting the receiver derive the latency from $t_r - t_s$, where t_r is the receive time. This requires clock synchronization between sender and receiver. Without external clock synchronization (like using GPS receivers or specialized software like the *network time protocol*, NTP), clocks can only be synchronized up to a granularity of the roundtrip time between two hosts [10], which is useless for measuring network latency. Unfortunately, as we can not generally assume the clocks of (possibly widely) distributed hosts to be tightly synchronized, we can not measure asymmetric network latencies within our measurement framework.

4 Result evaluation

We implemented the measurement procedure on our experimentation platform called the DAS system, which consists of four cluster computers. Each cluster contains Pentium Pros that are connected by Myrinet. The clusters are located at four Dutch universities and are connected by dedicated 6 Mbit/s ATM networks. (The system is more fully described on *http://www.cs.vu.nl/das/*.)

For the measurements presented in Fig. 2, we have used our MPI message passing system (described in [8, 9]) which can send messages inside clusters over Myrinet and between clusters over the ATM links, using TCP. We implemented the procedure as an MPI application, called the *MPI LogP benchmark*. We measured the LogP parameters for MPI_Send and MPI_Recv as described above, except for $g(m)$ which was measured with our fast method, and by the link saturation method [5, 7]. The graphs in Fig. 2 show o_s (for comparison) and g, as measured by both methods. In general, on both networks, the curves for g are rather close to each other, confirming the efficacy of our method. There is a general trend that the new, fast method measures slightly larger gaps. This can partially be explained by the systematic error of the saturation method which has to be stopped heuristically based on the increase rate ϵ of the measured gap values, causing part of the gap being missed. However, there is a region (64 byte—1 Kbyte over TCP, and 128 byte—4 Kbyte over Myrinet) where the saturation method measures significantly less (up to 50%) than the fast method. We could attribute the majority of this effect to a cache sensitivity of the *mirror* process which has better data locality with the saturation-based method as it does not send messages while draining the link. So, cache misses occur with somewhat larger messages, compared to the fast, roundtrip-based measurement.

Table 2 provides a breakdown of the measurement completion times shown in Fig. 2 for measuring $g(0)$, o_s/o_r (with implicit $g(m > 0)$), and $g(m > 0)$ (with saturation) over both networks. With our fast measurement procedure, only the first two measurements are necessary, yielding a performance gain of a factor of 10 over Myrinet, and a factor of 17 over the TCP link.

5 Conclusions

We presented a new, fast micro benchmark for measuring LogP parameters for messages of various sizes. We used the *parameterized LogP* [8] performance model. The

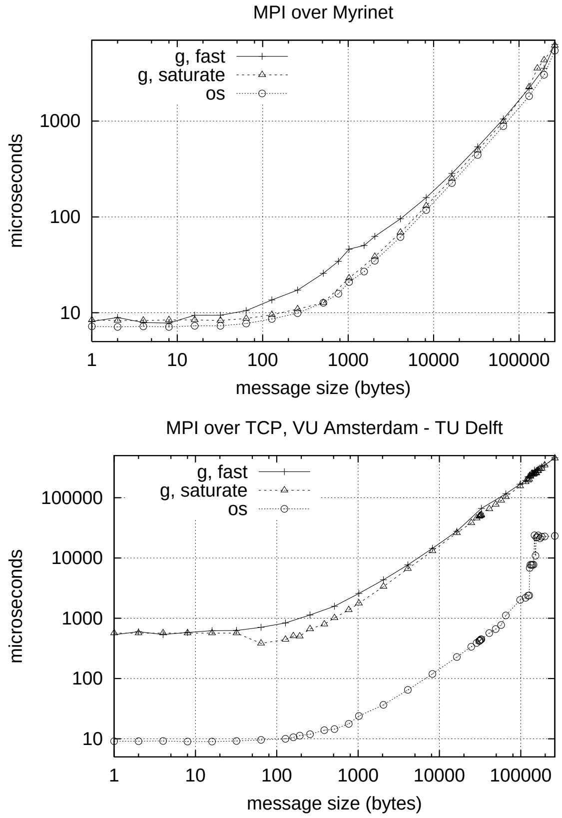

Fig. 2. Measured send overhead and gap; over Myrinet (top) and over TCP (bottom)

major improvement of our measurement procedure is that the minimal *gap* between two messages can be observed without saturating the network for each message size. Furthermore, our procedure adapts itself to the network characteristics in order to measure parameters for all relevant message sizes.

We implemented the new measurement procedure, called the *MPI LogP benchmark*, for our MPI platform and verified on two different networks that it gets the same results as a saturation-based measurement. The improvements in measurement time are significant. However, the time needed for a full measurement with various message sizes still takes too long to be performed during application runtime. As our ultimate goal is to enable applications to react to changing WAN conditions, we will need to restrict the

Table 2. Breakdown of measurement completion times (seconds)

	Myrinet	TCP
$g(0)$	0.05	12.3
o_s/o_r (with implicit $g(m>0)$)	0.16	102.7
$g(m>0)$ (with saturation)	1.96	2018.7

measurements to only a few message sizes and extrapolate the others by a technique like the one in [6]. The *MPI LogP benchmark* is available from

http://www.cs.vu.nl/albatross/

Acknowledgements

This work is supported in part by a USF grant from the Vrije Universiteit. The wide-area DAS system is an initiative of the Advanced School for Computing and Imaging (ASCI). We thank Rutger Hofman for his contributions to this research. We thank John Romein for keeping the DAS in good shape, and Cees de Laat (University of Utrecht) for getting the wide area links of the DAS up and running.

References

1. A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model — One Step Closer Towards a Realistic Model for Parallel Computation. In *Proc. Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 95–105, Santa Barbara, CA, July 1995.
2. F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-Level Scheduling on Distributed Heterogeneous Networks. In *Proc. Supercomputing'96*, Nov. 1996. Online at http://www.supercomp.org/sc96/proceedings/.
3. R. Bhoedjang, T. Rühl, and H. Bal. User-Level Network Interface Protocols. *IEEE Computer*, 31(11):53–60, 1998.
4. D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–12, San Diego, CA, May 1993.
5. D. E. Culler, L. T. Liu, R. P. Martin, and C. O. Yoshikawa. Assessing Fast Network Interfaces. *IEEE Micro*, 16(1):35–43, Feb. 1996.
6. M. Faerman, A. Su, R. Wolski, and F. Berman. Adaptive Performance Prodiction for Distributed Data-Intensive Applications. In *Supercomputing'99*, Nov. 1999. Online at http://www.supercomp.org/sc99/proceedings/.
7. G. Iannello, M. Lauria, and S. Mercolino. Cross–Platform Analysis of Fast Messages for Myrinet. In *Proc. Workshop CANPC'98*, number 1362 in Lecture Notes in Computer Science, pages 217–231, Las Vegas, Nevada, January 1998. Springer.
8. T. Kielmann, H. E. Bal, and S. Gorlatch. Bandwidth-efficient Collective Communication for Clustered Wide Area Systems. In *Proc. International Parallel and Distributed Processing Symposium (IPDPS 2000)*, Cancun, Mexico, May 2000.
9. T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. MAGPIE: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 131–140, Atlanta, GA, May 1999.
10. V. Paxson. On Calibrating Measurements of Packet Transit Times. In *Proc. SIGMETRICS'98/PERFORMANCE'98*, pages 11–21, Madison, Wisconsin, June 1998.
11. R. Wolski. Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service. In *Proc. High-Performance Distributed Computing (HPDC-6)*, pages 316–325, Portland, OR, Aug. 1997. The network weather service is at http://nws.npaci.edu/.