On stalling in LogP*

(Extended Abstract)

Gianfranco Bilardi[1,2], Kieran T. Herley[3], Andrea Pietracaprina[1], and Geppino Pucci[1]

[1] Dipartimento di Elettronica e Informatica, Università di Padova, Padova, Italy.
{bilardi,andrea,geppo}@artemide.dei.unipd.it
[2] T.J. Watson Research Center, IBM, Yorktown Heights, NY 10598, USA.
[3] Department of Computer Science, University College Cork, Cork, Ireland.
k.herley@cs.ucc.ie

Abstract. We investigate the issue of stalling in the LogP model. In particular, we introduce a novel quantitative characterization of stalling, referred to as δ-*stalling*, which intuitively captures the realistic assumption that once the network's capacity constraint is violated, it takes some time (at most δ) for this information to propagate to the processors involved. We prove a lower bound that shows that LogP under δ-stalling is strictly more powerful than the stall-free version of the model where only strictly stall-free computations are permitted. On the other hand, we show that δ-stalling LogP with $\delta = L$ can be simulated with at most logarithmic slowdown by a BSP machine with similar bandwidth and latency values, thus extending the equivalence (up to logarithmic factors) between stall-free LogP and BSP argued in [1] to the more powerful L-stalling LogP.

1 Introduction

Over the last decade considerable attention has been devoted to the formulation of a suitable computational model that supports the development of efficient and portable parallel software. The widely-studied BSP [6] and LogP [2] models were conceived to provide a convenient framework for the design of algorithms, coupled with a simple yet accurate cost model, to allow algorithms to be ported across a wide range of machine architectures with good performance. Both models view a parallel computer as a set of p processors with local memory that exchange messages through a communication medium whose performance is essentially characterized by two key parameters: *bandwidth* (g for BSP and G for LogP) and *latency* (ℓ for BSP and L for LogP).

A distinctive feature of LogP is that it embodies a *network capacity constraint* stipulating that at any time the total number of messages in transit towards any specific destination should not exceed the threshold $\lceil L/G \rceil$. If this constraint is respected, then every message is guaranteed to arrive within L steps of its submission time. If, however, a processor attempts to submit a message with destination d whose injection into the network would violate the constraint, then the processor is forced to stall until the delivery of some outstanding messages brings the traffic for d below the $\lceil L/G \rceil$ threshold. It seems clear that the intention of the original LogP proposal [2] was strongly to

* This research was supported, in part, by the Italian CNR, and by MURST under Project *Algorithms for Large Data Sets: Science and Engineering*.

encourage the development of stall-free programs. Indeed, the delays incurred in the presence of stalling were not formally quantified within the model, making the performance of stalling programs an issue difficult to assess with any precision. At the same time, adhering strictly to the stall-free mode might make algorithm design artificially complex, e.g., in situations involving randomization where stalling is unlikely but not impossible. Hence, ruling out stalling altogether might not be desirable.

The relation between BSP and LogP has been investigated in [1], where it is shown that the two models can simulate one another efficiently, under the reasonable assumption that both exhibit comparable values for their respective bandwidth and latency parameters. These results were obtained under a precise specification of stalling behaviour, that attempted to be faithful to the original formulation of the model. Interestingly, however, while the simulation of stall-free LogP programs on the BSP machine can be accomplished with constant slowdown, the simulation of stalling programs incurs a higher slowdown. This difference appears also in subsequent results of [5], where work-preserving simulations are considered. Should stalling programs turn out inherently to require a larger slowdown, it would be an indication that stalling adds power to the LogP model, in contrast with the objective of discouraging its use.

The definition of stalling proposed in [1] states that at each step the network accepts submitted messages up to the capacity threshold for each destination, forcing a processor to stall *immediately* upon submitting a message that exceeds the network capacity, and subsequently awakening the processor *immediately* when its message can be injected without violating the capacity constraint. Although consistent with the informal descriptions given in [2], the above definition of stalling implies the somewhat unrealistic assumption that the network is able to detect and react to the occurrence of a capacity constraint violation *instantaneously*. More realistically, some time lag is necessary between the submission of a message and the onset of stalling, to allow information to propagate through the network.

In this paper we delve further into the issue of stalling in LogP along the following directions:

- We generalize the definition of stalling, by introducing the notion of δ-*stalling*. Intuitively, δ captures the time lag between the submission of a message by a processor which violates the capacity constraint, and the time that the processor "realizes" that it must stall. (A similar time lag affects the "unstalling" process.) The extreme case of $\delta = 1$ essentially corresponds to the stalling interpretation given in [1]. While remaining close to the spirit of the original LogP , δ-stalling LogP has the potential of reflecting more closely the behaviour of actual platforms, without introducing further complications in the design and analysis of algorithms.
- We prove that allowing for stalling in a LogP program enhances the computational power of the model. In particular, we prove a lower bound which separates δ-stalling LogP from stall-free LogP computations by a non-constant factor.
- We devise an algorithm to simulate δ-stalling LogP programs in BSP, which achieves at most logarithmic slowdown under the realistic assumption $\delta = L$. This result, combined with those in [1], extends the equivalence (up to logarithmic factors) between LogP and BSP to L-stalling computations.

The rest of the paper is organized as follows. In Section 2 the definitions of BSP and LogP are reviewed and the new δ-stalling rule is introduced. In Section 3 a lower bound is shown that separates δ-stalling LogP from stall-free LogP computations. In Section 4 the simulation of δ-stalling LogP in BSP is presented.

2 The models

Both the BSP [6] and the LogP [2] models can be defined in terms of a virtual machine consisting of p serial processors with unique identifiers. Each processor i, $0 \leq i < p$, has direct and exclusive access to a private memory and has a local clock. All clocks run at the same speed. The processors interact through a communication medium, typically a network, which supports the routing of messages. In the case of BSP, the communication medium also supports global barrier synchronization. The distinctive features of the two models are discussed below. In the rest of this section we will use P_i^{B} and P_i^{L} to denote, respectively, the i-th BSP processor and the i-th LogP processor, with $0 \leq i < p$.

BSP A BSP machine operates by performing a sequence of *supersteps*, where in a superstep each processor may perform local operations, send messages to other processors and read messages previously delivered by the network. The superstep is concluded by a barrier synchronization which informs the processors that all local computations are completed and that every message sent during the superstep has reached its intended destination. The model prescribes that the next superstep may commence only after completion of the previous barrier synchronization, and that the messages generated and transmitted during a superstep are available at the destinations only at the start of the next superstep. The performance of the network is captured by a *bandwidth* parameter g and a *latency* parameter ℓ. The running time of a superstep is expressed in terms g and ℓ as $T_{superstep} = w + gh + \ell$, where w is the maximum number of local operations performed by any processor and h the maximum number of messages sent or received by any processor during the superstep. The overall time of a BSP computation is simply the sum of the times of its constituent supersteps.

LogP In a LogP machine, at each time step, a processor can be either *operational* or *stalling*. If it is operational, then it can perform one of the following types of operations: execute an operation on locally held data (*compute*); submit a message to the network destined to another processor (*submit*); receive a message previously delivered by the network (*receive*). A LogP program specifies the sequence of operations to be performed by each processor.

As in BSP, the behaviour of the network is modeled by a bandwidth parameter G (called *gap* in [2]) and a *latency* parameter L with the following meaning. At least G time steps must elapse between consecutive submit or receive operations performed by the same processor. If, at the time that a message is submitted, the total number of messages in transit (i.e., submitted to the network but not yet delivered) for that destination is at most $\lceil L/G \rceil$, then the message is guaranteed to be delivered within L steps. If, however, the number of messages in transit exceeds $\lceil L/G \rceil$, then, due to congestion,

the message may take longer to reach its destination, and the submitting processor may *stall* for some time before continuing its operations. The quantity $\lceil L/G \rceil$ is referred to as the network's *capacity constraint*. Note that message delays are unpredictable, hence different executions of a LogP program are possible. If no stalling occurs, then every message arrives in at most L time steps after its submission.

Upon arrival, a message is promptly removed from the network and buffered in some input buffer associated with the receiving processor. However, the actual acquisition of the incoming message by the processor, through a receive operation, may occur at a later time. LogP also introduces an *overhead* parameter o to represent both the time required to prepare a message for submission and the time required to unpack the message after it has been received. Throughout the paper we will assume that $\max\{2, o\} \leq G \leq L \leq p$. The reader is referred to [1] for a justification of this assumption.

2.1 LogP's stalling behaviour

The original definition of the LogP model in [2] provides only a qualitative description of the stalling behaviour and does not specify precisely how the performance of a program is affected by stalling. In [1], the following rigorous characterization of stalling was proposed. At each step the network accepts messages up to saturation, for each destination, of the capacity limit, possibly blocking the messages exceeding such a limit at the senders. From a processor's perspective, the attempt to submit a message violating the capacity constraint results in immediate stalling, and the stalling lasts until the message can be accepted by the network without capacity violation.

The above characterization of stalling, although consistent with the intentions of the model's proposers, relies on the somewhat unrealistic assumption that the network is able to monitor at each step the number of messages in transit for each destination, blocking (resp., unblocking) a processor *instantaneously* in case a capacity constraint violation is detected (resp., ends). In reality, the stall/unstall information would require some time to propagate through the network and reach the intended processors. Below we propose an alternative, yet rigorous, definition of stalling, which respects the spirit of LogP while modelling the behaviour of real machines more accurately.

Let $1 \leq \delta \leq L$ be an integral parameter. Suppose that at time step t processor P_i^{L} submits a message m destined to P_j^{L}, and let $c_j(t)$ denote the total number of messages destined to P_j^{L} which have been submitted up to (and including) step t and are still in transit at the beginning of this step. If $c_j(t) \leq \lceil L/G \rceil$, then m reaches its destination at some step t_m, with $t < t_m \leq t + L$. If, instead, $c_j(t) > \lceil L/G \rceil$ (i.e., the capacity constraint is violated), the following happens:

1. Message m reaches its destination at some step t_m, with $t < t_m \leq t + Gc_j(t) + L$.
2. P_i^{L} may be signalled to stall at some time step t', with $t < t' \leq t + \delta$. Until step t' the processor continues its normal operations.
3. Let $\bar{t}$ denote the latest time step when a message that caused P_i^{L} to stall during steps $[t, t')$ arrives at its destination. Then, the processor is signalled to revert to operational state at some time t'', with $\bar{t} < t'' \leq \bar{t} + \delta$. (Note that if $t' > \bar{t} + \delta$ no stalling takes place.)

Intuitively, parameter δ represents an upper bound to the time the network takes to inform a processor that one of the messages it submitted violated the capacity constraint, or that it may revert to operational state as the result of a decreased load in the network.

We refer to the LogP model under the above stalling rule as *δ-stalling LogP*, or *δ-LogP* for short. A *legal execution* of a δ-LogP program is one where message delivery times and stalling periods are consistent with the model's specifications and with the above rule.[1]

In [1] a restricted version of LogP has also been considered, which regards as correct only those programs whose executions never violate the capacity constraint, that is, programs where processors never stall. We refer to such a restricted version of the model as *stall-free LogP*, or *SF-LogP* for short.

3 Separation between δ-stalling LogP and stall-free LogP

In this section, we demonstrate that allowing for δ-stalling in LogP makes the model strictly more powerful than SF-LogP. We prove our claim by exhibiting a simple problem Π such that *any* SF-LogP algorithm for Π requires time which is asymptotically higher than the time attained by a simple δ-LogP algorithm for Π.

Let Π be the problem of *2-compaction* [4]. On a shared memory machine, the problem is defined as follows: given a vector $\boldsymbol{x} = (x_0, x_1, \ldots x_{p-1})$ of p integer components with at most two nonzero values x_{i_0} and x_{i_1}, $i_0 < i_1$, compact the nonzero values at the front of the array. On LogP, we recast the problem as follows. Vector $\boldsymbol{x}$ is initially distributed among the processors so that P_i^{L} holds x_i, for $0 \leq i < p$. The problem simply requires to make (i_0, x_{i_0}) and (i_1, x_{i_1}) known, respectively, to P_0^{L} and P_1^{L}.

On δ-LogP the 2-compaction problem can be solved by the following simple deterministic algorithm in $O(L)$ time, for any $\delta \geq 1$: each processor that holds a 1 transmits its identity and its input value first to P_0^{L} and then to P_1^{L}. Observe that if $G = L$ such a strategy is illegal for SF-LogP, since it generates a violation of the capacity constraint (since, in this case, $\lceil L/G \rceil = 1$). The following theorem shows that, indeed, for $G = L$, 2-compaction cannot be solved on SF-LogP in $O(L)$ time, thus providing a separation between SF-LogP and δ-LogP.

Theorem 1. *For any constant ϵ, $0 < \epsilon < 1$, solving 2-compaction with probability greater than $(1+\epsilon)/2$ on SF-LogP with $G = L$ requires $\Omega\left(L\sqrt{\log n}\right)$ steps.*

Proof (Sketch). In [4] it is proved that solving 2-compaction with probability greater than $(1+\epsilon)/2$ on the EREW-PRAM requires $\Omega\left(\sqrt{\log n}\right)$ steps, even if each processor is allowed to perform an unbounded amount of local computation per step. The theorem follows by showing that when $G = L$, any T-step computation of a p-processor SF-LogP can be simulated in $O\left(\lceil T/L \rceil\right)$ steps on a p-processor EREW-PRAM with unbounded local computation. (Details of the simulation will be provided in the full version of the paper.)

[1] Note that the characterization of stalling proposed in [1] corresponds to the one given above with $\delta = 1$, except that in [1] a processor reverts to the operational state as soon as the capacity constraint violation ends, which may happen before the message causing the violation reaches its destination.

It must be remarked that the above theorem relies on the assumption $G = L$. We leave the extension of the lower bound to arbitrary values of G and L as an interesting open problem.

4 Simulation of LogP on BSP

This section shows how to simulate δ-LogP programs efficiently on BSP. The strategy is similar in spirit to the one devised in [1] for the simulation of SF-LogP programs, however it features a more careful scheduling of interprocessor communication in order to correctly implement the stalling rule.

The algorithm is organized in *cycles*, where in a cycle P_i^{B} simulates $C = \max\{G, \delta\} \leq L$ consecutive steps (including possible stalling steps) of processor P_i^{L}, using its own local memory to store the contents of P_i^{L}'s local memory, for $0 \leq i < p$. In order to simplify bookkeeping operations, the algorithm simulates a particular legal execution of the LogP program where all messages reach their destinations at cycle boundaries. (From what follows it will be clear that such a legal execution exists.)

Each processor P_i^{B} has a program counter ρ that at any time indicates the next instruction to be simulated in the P_i^{L}'s program. It also maintains in its local memory a pool for outgoing messages $Q_{out}(i)$, a FIFO queue for incoming messages $Q_{in}(i)$ (both initially empty), and two integer variables t_i and w_i. Variable t_i represents the clock and always indicates the next time step to be simulated, while w_i is employed in case of stalling to indicate when P_i^{L} reverts to the operational state. Specifically, P_i^{L} is stalling in the time interval $[t_i, w_i - 1]$, hence it is operational at step t_i, if $w_i \leq t_i$. Initially both t_i and w_i are set to 0. The undelivered messages causing processors to stall are retained in a global pool S, which is evenly distributed among the processors.

We now outline the simulation of the k-th cycle, $k \geq 0$, which comprises time steps $C \cdot k, C \cdot k + 1, \ldots C \cdot (k + 1) - 1$. At the beginning of the cycle's simulation we have that $t_i = C \cdot k$ and $Q_{in}(i)$ contains all messages delivered by the network to P_i^{L} at the beginning of step $C \cdot k$, for $0 \leq i < p$. Also, S contains messages that have been submitted in previous cycles and that will reach their destination at later cycles, that is, at time steps $C \cdot k'$ with $k' > k$. The simulation of the k-th cycle proceeds as follows.

1. For $0 \leq i < p$, if $w_i < C \cdot (k+1)$ then P_i^{B} simulates the next $x = C \cdot (k+1) - \max\{t_i, w_i\}$ instructions in the P_i^{L}'s program. A submit is simulated by inserting the message into $Q_{out}(i)$, and a receive is simulated by extracting a message from $Q_{in}(i)$. The processor also increments ρ by x and sets $t_i = C \cdot (k + 1)$.
2. All messages in $\bigcup_i Q_{out}(i)$ together with those in S are sorted by destination and, within each destination group, by time of submission.
3. Within each destination group, messages are ranked and a message with rank r is assigned delivery time $C \cdot (k + \lceil r/\lceil L/G \rceil \rceil)$ (i.e., the message will be delivered at the beginning of the $(\lceil r/\lceil L/G \rceil \rceil)$-th next cycle).
4. Each message to be delivered at cycle $k + 1$ is placed in the appropriate $Q_{in}(i)$ queue (that of its destination), while all other messages are placed in S.
   Comment: Note that S contains only those messages for which a violation of the capacity constraint occurred.
5. For $0 \leq i < p$, if one of the messages submitted by P_i^{L} is currently in S then
   (a) w_i is set to the maximum delivery time of P_i^{L}'s messages in S;

(b) If $\delta < G$, then all operations performed by P_i^{L} in the simulated cycle subsequent to the submission of the first message that ended up in S are "undone" and ρ is adjusted accordingly.
Comment: Note that when $\delta < G$ processor P_i^{L} submits only one message in the cycle, hence the operations to be undone do not involve submits and their undoing is straightforward.

6. Messages in S are evenly redistributed among the processors.

Theorem 2. *For any δ, $1 \leq \delta \leq L$, the above algorithm correctly simulates a cycle of $C = \max\{G, \delta\}$ arbitrary LogP steps in time*

$$O\left(C\left(1+\log p\left(\frac{1}{G}+\frac{g/G}{1+\log(C/G)}+\frac{\ell/C}{1+\log\min\{C/G,\ell/g\}}\right)\right)\right) \quad .$$

Proof (Sketch). Consider of the simulation of an arbitrary cycle. The proof of correctness, which will be provided in the full version of the paper, entails showing that the operations performed by the BSP processors in the above simulation algorithm do indeed mimic the computation of their LogP counterparts in a legal execution of the cycle. As for the running time, Steps 1 and 5.(b) involve $O(C)$ local computation. Step 2 involves the sorting of $O((C/G)p)$ messages, since $|Q_{out}(i)| = O(C/G)$, for $0 \leq i < p$, and there can be no more than $\lceil \delta/G \rceil = O(C/G)$ messages in S sent by the same (stalling) processor. Finally, the remaining steps are dominated by the cost of prefix operations performed on evenly distributed input sets of size $O((C/G)p)$ and by the routing of $O(C/G)$-relations. The stated running time then follows by employing results in [3, 6].

The following corollary is immediately established.

Corollary 1. *When $\ell = \Theta(L)$, $g = \Theta(G)$ an arbitrary LogP program can be simulated in BSP with slowdown $O((L/G)\log p)$, if $\delta = 1$, and with slowdown $O(\log p/\min\{G, 1+\log(L/G)\})$, if $\delta = \Theta(L)$.*

The corollary, combined with the results in [1], shows that LogP, under the reasonable L-stalling rule, and BSP can simulate each other with at most logarithmic slowdown when featuring similar bandwidth and latency parameters.

References

1. G. Bilardi, K.T. Herley, A. Pietracaprina, G. Pucci and P. Spirakis. BSP vs. LogP. *Algorithmica*, 24:405–422, 1999.
2. D.E. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T.V. Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, November 1996.
3. M.T. Goodrich. Communication-Efficient Parallel Sorting. In *Proc. of the 28th ACM Symp. on Theory of Computing*, pages 247–256, Philadelphia PA, 1996.
4. P.D. MacKenzie. Lower bounds for randomized exclusive write PRAMs. *Theorey of Computing Systems*, 30(6):599–626, 1997.
5. V. Ramachandran, B. Grayson, and M. Dahlin. Emulations between QSM, BSP and LogP: a framework for general-purpose parallel algorithm design. TR98-22, Dept. of CS, Univ. of Texas at Austin, November 1998. (Summary in *Proc. of ACM-SIAM SODA*, 1999.)
6. L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.