On stalling in LogP [⋆]

Gianfranco Bilardi [a,b] Kieran Herley [c]
Andrea Pietracaprina [a] Geppino Pucci [a]

[a] *Dip. di Ingegneria dell'Informazione, Università di Padova, Padova, Italy*
[b] *T.J. Watson Research Center, IBM, Yorktown Heights, NY, USA*
[c] *Dept. of Computer Science, University College Cork, Cork, Ireland*

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 [1,7] to the more powerful L-stalling LogP.

Key words: BSP, LogP, Parallel Computation, Bridging Models, Stalling.

1 Introduction

Over the past decade considerable attention has been devoted to the formulation of a suitable computational model that supports the development

[⋆] This research was supported, in part, by MIUR of Italy under Project *ALINWEB* and by Grant CPDA033838 of the University of Padova. A preliminary version of this work was presented at the *Workshop on Advances in Parallel and Distributed Computational Models*, Cancun, MEX, May 2000.

Email addresses: `gianfranco.bilardi@unipd.it` (Gianfranco Bilardi), `k.herley@cs.ucc.ie` (Kieran Herley), `andrea.pietracaprina@unipd.it` (Andrea Pietracaprina), `geppino.pucci@unipd.it` (Geppino Pucci).

of efficient and portable parallel software. The widely-studied BSP [2] and LogP [3] 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 [3] was to encourage strongly 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 [4]. 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 relationship between BSP and LogP has been investigated in [5,1], 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. Indeed, this is a subtle point, which originally escaped the attention of the authors of [5], who claimed that their techniques for stall-free simulations would extend to stalling programs with the same slowdown. As reported in [1], however, Vijaya Ramachandran pointed out [6] that there is no such straightforward extension. The difference between stalling and stall-free programs is also stressed in [7], in the context of work-preserving simulations. Should stalling programs turn out inherently to require a larger slowdown, it would be an indication that stalling adds power to the LogP model, conflicting 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 [3], 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 nonconstant factor.
- We devise an algorithm to simulate δ-stalling LogP programs in BSP, which achieves at most logarithmic slowdown under the realistic assumption $\delta = L$, and $O((L/G)\log p)$ slowdown for the extreme case of $\delta = 1$. The former result, combined with those in [1], extends the equivalence (up to logarithmic factors) between LogP and BSP to L-stalling computations. The latter result appears to be in contrast with [7, Theorem 3.9], where it is stated that any *step-by-step* simulation of a stalling LogP program on BSP can have arbitrarily large slowdown. However, the authors of [7] base their claim on a very rigid subdivision of the LogP program into BSP supersteps, which artificially restricts the class of possible simulation strategies.

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

2 The models

Both the BSP [2] and the LogP [3] 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 of 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 [3]) 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 destination processor. However, the actual acquisition of the incoming message by the processor, through a receive operation, may occur at a later time. We also assume that a receive instruction is non-blocking, in the sense that the receiving processor proceeds with its execution if no message is available for acquisition at the time the instruction is issued. Clearly, it is up to the programmer to implement a busy-waiting cycle in case the processor cannot proceed correctly without acquiring the message.

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 [3] 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 for each destination messages up to the threshold dictated by the capacity constraint, possibly blocking the messages exceeding this threshold 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 violating that constraint.

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 (unblocking) a processor *instantaneously* in case a capacity constraint violation is detected (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 integer-valued 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) Suppose P_i^{L} is signalled to stall at t' and let $\bar{t}$ denote the latest time step when a message submitted by P_i^{L} during steps $[t, t')$ arrives at its destination. Then, the processor reverts to operational state at some time t'', with $\bar{t} < t'' \leq \bar{t} + \delta$. (Note that if $t' > t''$ no stalling takes place.)

Intuitively, parameter δ represents an upper bound on 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.

In [1] a restricted version of LogP has 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. The following theorem shows that, at least for the case $G = L$, allowing for δ-stalling in LogP makes the model strictly more powerful than SF-LogP.

Theorem 1 *There exists a problem for which the best SF-LogP algorithm is at least an $\Omega\left(\sqrt{\log p}\right)$-factor slower than a simple algorithm running on δ-LogP, for any $\delta \geq 1$.*

PROOF. Consider the *2-Compaction (2C)* problem [8], defined on a shared-memory machine as the problem of compacting the only two nonzero components of an input vector of size p at the front of the vector. The 2C problem can be naturally rephrased on LogP by having the entries of the input vector distributed uniformly among the p processors. Clearly on δ-LogP the 2-compaction problem can be solved deterministically in $O(L)$ time, for any $\delta \geq 1$, by simply letting each processor holding a nonzero entry transmit its identity and the entry value to both processors with index 0 and 1. Clearly, such a strategy would be illegal for SF-LogP, since for $G = L$ stalling may occur due to violation of the capacity constraint $\lceil L/G \rceil = 1$.

An $\Omega\left(\sqrt{\log p}\right)$ lower bound for 2C was proved in [8] for the EREW-PRAM model with unbounded local computation, where in a single step each processor can first read a cell from the shared memory, then perform an unbounded amount of local computation, and finally write a value to the shared memory. Concurrent read and write accesses to the same shared cell are disallowed. This result yields an $\Omega\left(L\sqrt{\log p}\right)$ lower bound for 2C on SF-LogP by observing that when $G = L$, any T-step computation of a p-processor SF-LogP can be easily simulated in $O(\max\{1, T/L\})$ steps on a p-processor EREW-PRAM with unbounded local computation. In the simulation, the i-th PRAM processor simulates cycles of L consecutive steps of the i-th SF-LogP processor, reading the single incoming message for the processor and writing the single outgoing message generated by the processor (since $\lceil L/G \rceil = 1$) into a shared vector of p components used for communications.

It must be remarked that the proof of the above result heavily 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.

3 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 simulates *cycles* of $C = \max\{G, \delta\} \leq L$ consecutive time steps (including possible stalling steps) of a legal execution of the LogP program. For $0 \leq i < p$, processor P_i^{B} simulates the behaviour of processor P_i^{L} using its own local memory to store the contents of P_i^{L}'s local memory. In order to simplify bookkeeping operations, we assume that L is an integral multiple of the cycle time C, and that the particular LogP execution chosen for simulation is one where all messages are delivered by the network to their destinations at cycle boundaries. The analysis will show that such a legal execution exists. Note that in the chosen execution more the one message may be delivered to one processor at the same time[1] . However, our simulation can be adapted easily to the case where C does not divide L and messages destined to the same processor cannot be delivered at the same time step.

[1] In fact, although the definition of LogP requires that at least G time steps elapse between consecutive receive operations issued by a processor, it does not put any constraint on the actual message delivery by the network.

Consider the simulation of an arbitrary LogP program. Throughout the simulation, each processor P_i^{B} maintains two integer variables t_i and w_i, and a program counter ρ_i. Variable t_i represents the simulation clock and always indicates the next time step of the simulation process. The variable ρ_i acts a "program counter": at any time, ρ_i indicates to the next instruction to be executed by P_i^{L} in the computation being simulated. The role of w_i is to keep track of whether or not P_i^{L} is stalled or operational and, in the case of the former, when it should be re-awakened. Specifically, P_i^{L} is stalling in the time interval $[t_i, w_i - 1]$ if $w_i > t_i$, and is operational at step t_i otherwise. Initially, both t_i and w_i are set to 0.

In order to correctly simulate LogP communication, each processor P_i^{B} maintains three queues in its local memory: $Q_{in}(i)$, $Q_{out}(i)$, and $U(i)$. In the simulation of a cycle, $Q_{in}(i)$ stores all those messages that have been already delivered but not yet received by P_i^{L} (*i.e.* not yet subject of a receive operation); $Q_{out}(i)$ accumulates all messages submitted by P_i^{L} during the cycle; finally, $U(i)$ stores those messages submitted by P_i^{L} in previous cycles and still in transit to their destinations during the current cycle.

We now describe the details of the simulation of the k-th cycle, $k \geq 0$, which comprises time steps $C \cdot k, C \cdot k + 1, \ldots C \cdot (k+1) - 1$. Note that at the beginning of the cycle's simulation we have that $t_i = C \cdot k$.

(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 P_i^{L}'s program, updates ρ_i accordingly and sets $t_i = C \cdot (k+1)$. A submit is simulated by inserting the message into $Q_{out}(i)$, and a receive is simulated by extracting a message from $Q_{in}(i)$.

(2) All messages in $\bigcup_i (Q_{out}(i) \cup U(i))$ 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 $C \cdot (k + \lceil r/(1 + \lceil C/G \rceil) \rceil)$ (i.e., the message will be delivered at the beginning of the $(\lceil r/(1 + \lceil C/G \rceil) \rceil)$-th next cycle).

(4) Each message to be delivered in cycle $k+1$ is sent to its destination processor that stores it in queue Q_{in}, while all other messages are sent back to their source processors, which store them in queue U.

(5) For $0 \leq i < p$, if one of the messages in queue $U(i)$ was assigned rank $r > \lceil L/G \rceil$ in Step 3, then

   (a) w_i is set to the maximum delivery time of all messages in $U(i)$;

   (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 violated the capacity constraint are "undone" and ρ_i 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.

We first show the correctness of the simulation by proving that the BSP processors do indeed mimic the steps of a legal execution of the LogP program. Consider a message m submitted by P_i^{L} at time t during the k-th cycle, and destined to P_j^{L}. As before, we let $c_j(t)$ denote the total number of messages destined to P_j^{L} submitted at or before time step t, which are still in transit at the beginning of the step.

Lemma 2 *In the execution being simulated, m is arrives within step $t + L$, if $c_j(t) \leq \lceil L/G \rceil$, and within step $t + Gc_j(t) + L$, otherwise.*

PROOF. The lemma easily follows from the fact that the algorithm schedules m to arrive at time step $t_m \leq t + C\lceil c_j(t)/(1 + \lceil C/G \rceil) \rceil$, and from the assumption that L is an integral multiple of C.

Lemma 3 *Let $r > \lceil L/G \rceil$ be the rank assigned to m. Then, in the execution* *being simulated, P_i^{L} starts stalling within δ steps from the submission of m at step t, and reverts to operational state within δ steps of the arrival of the last of its messages submitted prior to the onset of stalling.*

PROOF. If $\delta < G$ then, as a consequence of Step 5(b), P_i^{L} is forced to stall immediately after submission of m. If $\delta \geq G$ then the stalling will start at the beginning of the next cycle, which is at most $C = \delta$ time steps from the submission of m. In both cases, the setting of w_i done in Step 5(a) ensures that the processor reverts to operational state immediately after the last message submitted before stalling arrives at its destination.

The following theorem is an immediate consequence of the above lemmas

Theorem 4 *For any given LogP program, the above algorithm simulates one of its legal executions.*

The next theorem establishes the performance of the simulation algorithm.

Theorem 5 *For any δ, $1 \leq \delta \leq L$, the above algorithm correctly simulates a cycle of $C = \max\{G, \delta\}$ arbitrary δ-LogP steps in time*

$$O\left(C + \frac{L}{G}\log p + \left(\frac{L}{G}g + \ell\right)\frac{\log p}{\log(L/G)} + \ell\frac{\log p}{\log(\ell/g)}\right).$$

PROOF. Consider the simulation of an arbitrary cycle. Steps 1 and 5.(b) involve $O(C)$ local computation altogether. It is easy to see that the sorting of $\bigcup_i (Q_{out}(i) \cup U(i))$ performed in Step 2 involves $O((L/G))$ messages per processor. Indeed, for $0 \leq i < p$, $|Q_{out}(i)| = O(C/G)$, while $|U(i)| = O(L/G)$, since in $U(i)$ there can be at most $\lceil L/G \rceil$ messages of rank $r \leq \lceil L/G \rceil$ (all these messages are delivered within L time steps of their submission) and at most $\lceil \delta/G \rceil$ messages of rank $r > \lceil L/G \rceil$ (the processor begins stalling within δ times from the submission of the oldest among such messages). Therefore, by employing the BSP sorting algorithm of [9], Step 2 takes time $O\left((L/G)\log p + ((L/G)\, g + \ell)\log p/\log(L/G)\right)$. Step 3 requires a segmented prefix computation, which can be accomplished in time $O\left(L/G + \ell \log p/\log(\ell/g)\right)$ using standard techniques [2]. Finally, Step 4 entails the routing of an $O\left((L/G)\right)$-relation, and Step 5.(a) requires $O\left((L/G)\right)$ local computation. The theorem follows by adding up the contributions of the individual steps.

The following corollary follows immediately.

Corollary 6 *When $\ell = \Theta(L)$, $g = \Theta(G)$ an arbitrary δ-LogP program can be simulated in BSP with slowdown $O\left((L/G)\log p/\min\{G, 1+\log(L/G)\}\right)$, if $\delta = 1$, and with slowdown $O\left(\log p/\min\{G, 1+\log(L/G)\}\right)$, 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.

4 Conclusions

Previous work had exposed a puzzling feature of the stalling regime in the LogP model of computation. On the one hand, the model was meant to encourage the design of programs that, by complying with the capacity constraint, avoid saturating the network. On the other hand, attempts to give a precise definition of stalling behaviour indicated that stalling could in fact be a source of computational power, which might make it attractive to program designers while at the same time making efficient implementations of the LogP abstraction on realistic platforms more difficult to achieve.

In this paper, we have provided further evidence that stalling, in the form made precise in [1], enhances the computational power of LogP. However, we have also shown how the newly proposed mechanism of δ-stalling yields a version of LogP where stalling has a considerably more limited power and whose implementation on realistic platforms ought to be considerably more efficient.

References

[1] G. Bilardi, K. Herley, A. Pietracaprina, G. Pucci, P. Spirakis, BSP vs LogP, Algoritmica 24 (1999) 405–422, special Issue on Coarse Grained Parallel Algorithms.

[2] L. Valiant, A bridging model for parallel computation, Communications of the ACM 33 (8) (1990) 103–111.

[3] D. Culler, R. Karp, D. Patterson, A. Sahay, E. Santos, K. Schauser, R. Subramonian, T. Eicken, LogP: A practical model of parallel computation, Communications of the ACM 39 (11) (1996) 78–85.

[4] D. Culler, A. Dusseau, R. Martin, K. Shauser, Fast parallel sorting under LogP: from theory to practice, in: Proc. of the Workshop on Portability and Performance for Parallel Processors, Southampton, UK, 1993, pp. 18–29.

[5] G. Bilardi, K. Herley, A. Pietracaprina, G. Pucci, P. Spirakis, BSP vs LogP, in: Proc. of the 8th ACM Symp. on Parallel Algorithms and Architectures, Padova, Italy, 1996, pp. 25–32.

[6] V. Ramachandran, Personal communication, June 1998.

[7] V. Ramachandran, B. Grayson, M. Dahlin, Emulations between QSM, BSP and LogP: A framework for general-purpose parallel algorithm design, Journal on Parallel and Distributed Computing 63 (2003) 1175–1192.

[8] P. MacKenzie, Lower bounds for randomized exclusive write PRAMs, Theory of Computing Systems 30 (6) (1997) 599–626.

[9] M. Goodrich, Communication-efficient parallel sorting, in: Proc. of the 28th ACM Symp. on Theory of Computing, Philadelphia, Pennsylvania USA, 1996, pp. 247–256.

Gianfranco Bilardi received the Laurea (1978) (*summa cum laude*) in Electrical Engineering from the University of Padova and the Master (1982) and the PhD (1985) degrees, both in Electrical Engineering, from the University of Illinois at Urbana-Champaign. From 1984 to 1990, he was an assistant professor of Computer Science at Cornell University, Ithaca, New York. In 1990, he joined the Department of Information Engineering at the University of Padova, Italy, as a Professor of Computer Engineering. His research interests lie in the area of parallel and VLSI computing. He is the author of more than 70 publications in international journals and conferences. Prof. Bilardi is a member of the ACM and a senior member of the IEEE.

Kieran T. Herley received his BS (1982) and MS (1983) both in Computer Science from University College Cork, Ireland. Further studies at Cornell University lead to an MS (1986) and a PhD (1990) in Computer Science. Since 1990, he has been a Lecturer in the Department of Computer Science of University College Cork, Ireland. Dr. Herley's research interests include the design and analysis of parallel algorithms and parallel computational models. He is a member of ACM and EATCS.

Andrea Pietracaprina received the Laurea (1987) in Computer Science (*summa cum laude*) from the University of Pisa, Italy and the MS (1991) and PhD (1994) both in Computer Science from the University of Illinois at Urbana-Champaign, USA. From 1994 to 1998 he was an Assistant Professor at the Department of Pure and Applied Mathematics of the same university. Since 1998, he has been with the Department of Information Engineering of the University of Padova, where he is currently a Professor of Computer Science. Prof. Pietracaprina's main research interests lie in the fields of parallel computation, network routing, graph theory, and combinatorial optimization. He is the author of about 40 papers in international journals and conferences. Prof. Pietracaprina is a member of ACM and IEEE.

Geppino Pucci received the Laurea (1987) (*summa cum laude*) and the PhD (1993) degrees both in Computer Science from the University of Pisa, Italy. From 1988 to 1990 he was a Research Associate at the Computing Laboratory of the University of Newcastle-upon-Tyne, UK. In 1993, he was at the International Computer Science Institute, Berkeley, USA, as a postdoctoral fellow. Since 1993 he has been with the Department of Information Engineering of the University of Padova, Italy, where he is currently a Professor of Computer Science. His research interests include design and analysis of parallel algorithms, theory of computation, probabilistic modeling, and algorithm engineering for problems in computational sciences. On these subjects, he has authored over 50 papers in international journals and conferences. Prof. Pucci is a member of ACM and IEEE.