# BSP vs LogP[1]

Gianfranco Bilardi[2,3] Kieran T. Herley[4] Andrea Pietracaprina[2]

Geppino Pucci[2] Paul Spirakis[5]

## Abstract

A quantitative comparison of the BSP and LogP models of parallel computation is developed. We concentrate on a variant of LogP that disallows the so-called stalling behavior, although issues surrounding the stalling phenomenon are also explored. Very efficient cross simulations between the two models are derived, showing their substantial equivalence for algorithmic design guided by asymptotic analysis. It is also shown that the two models can be implemented with similar performance on most point-to-point networks. In conclusion, within the limits of our analysis that is mainly of an asymptotic nature, BSP and (stall-free) LogP can be viewed as closely related variants within the bandwidth-latency framework for modeling parallel computation. BSP seems somewhat preferable due to its greater simplicity and portability, and slightly greater power. LogP lends itself more naturally to multiuser mode.

**Key words:** Models of Computation, Parallel Computation, Bridging Models, Portability, BSP Model, LogP Model.

## 1 Introduction

Widespread use of parallel computers crucially depends on the availability of a model of computation simple enough to provide a convenient basis for software development, accurate enough to enable realistic performance predictions, yet general enough that software be portable with good performance across a wide range of architectures. The formulation of a *bridging model* that bal-

[1]This research was supported in part by the ESPRIT III Basic Research Programme of the EC under contract No. 9072 (Project GEPPCOM). A preliminary version of this paper appeared in *Proc. of the 8th ACM Symposium on Parallel Algorithms and Architectures*, Padova, I, pages 25–32, June 1996.

[2]Dipartimento di Elettronica e Informatica, Università di Padova, I-35131 Padova, Italy

[3]Department of Electrical Engineering and Computer Science, University of Illinois at Chicago, Chicago, IL 60607 USA

[4]Department of Computer Science, University College Cork, Cork, Ireland

[5]Computer Technology Institute, Patras, Greece

ances these conflicting requirements has proved a difficult task, a fact amply demonstrated by the proliferation of models in the literature over the years.

The BSP [1] and the LogP [2, 3] models have been proposed in this context and have attracted considerable attention (see [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] for BSP and [16, 17, 18, 10, 19] for LogP). In both models the communication capabilities of the machine are summarized by a few parameters that broadly capture bandwidth and latency properties. In BSP, the fundamental primitives are global barrier synchronization and the routing of arbitrary message sets. LogP lacks explicit synchronization and imposes a more constrained message-passing style which aims at keeping the load of the underlying communication network below a specified capacity limit. Intuitively, BSP ought to offer a more convenient abstraction for algorithm design and programming, while LogP ought to provide better control of machine resources.

While it is part of the folklore that BSP and LogP bear a marked resemblance to each other, to our knowledge, a quantitative and systematic comparison between the two models had not yet been undertaken. In this paper, we begin such a comparison. Technically, a clarification is in order at the outset, to put our discussion and results in the proper perspective. The original definition of LogP includes an operating regime, called *stalling*, which occurs when some processors become hot spots, i.e., the number of messages addressed to them exceeds a certain threshold. In our comparison with BSP we will mostly focus on a stall-free version of LogP. There are two main reasons for this. Firstly, the intended use of LogP appears to be that of writing stall-free programs and secondly, the behaviour dictated by the model in stalling situations appears hard to realize on real machines. However, the issue of stalling is a rather subtle one and, as we shall indicate, deserves further investigation, which might lead to valuable insights on the bandwidth-latency paradigm of parallel computing.

Our main objectives and results in comparing BSP and *stall-free* LogP are the following:

- **To characterize the distance between the two models**

  Quantitatively, we measure the distance between the models by the slowdown incurred by one in simulating the other. We show that, when the bandwidth and latency parameters have the same value in both models, BSP can simulate LogP with constant slowdown and LogP can simulate BSP with at most logarithmic slowdown (indeed constant slowdown for a wide range of the relevant parameters)[1].

  The small value of the slowdown in both directions indicates that, at least from the point of view of asymptotically efficient algorithmic design, the two models are substantially equivalent. The (slightly) greater power of BSP and its greater simplicity with respect to LogP appear to be points in its favor.

- **To compare the performance of the two models on the same platform**

  The greater power of BSP with respect to LogP could, concievably, hide a potential loss of performance when implementing BSP over LogP on the same hardware platform. This could outweigh the advantages provided by BSP's more convenient programming abstraction.

  We investigate this issue for hardware platforms that can be modeled as point-to-point networks. For several such networks, well-known routing results show that similar values can be achieved for the bandwidth and latency parameters in both models. The indication is that, asymptotically, no substantial price is paid for the convenience of the BSP abstraction.

In summary, BSP and stall-free LogP can be viewed as closely related variants within the bandwidth-latency framework for modeling parallel computation. BSP seems somewhat preferable due to greater simplicity and portability, and slightly greater power.

[1] Recently Ramachandran *et al.* [20], as part of a broader exploration of the interrelationships between LogP, BSP and the QSM shared-memory model [21], have observed that our simulation of stall-free LogP on BSP can be immediately made *work-preserving* while maintaining the same slowdown.

These indications cannot be regarded as definitive. First, constant factors that are disregarded in the kind of asymptotic analysis presented here are clearly of considerable significance when it comes to the performance of practical applications on real machines. Second, commercial multiprocessors are not necessarily accurately modeled by point-to-point networks. However, we hope that the methodology proposed here can be refined to afford a more accurate comparison between BSP and LogP as well as between other models of computation.

The rest of the paper is organized as follows. Section 2 provides a formal definition of the models. Regarding LogP, we clarify some aspects of its definition. In particular, we propose a precise characterization of the behaviour of stalling, while trying to remain faithful to the informal description of it given in [3]. We then discuss some possible implications of stalling on the use of LogP as a programming model, and on its realizability on actual machines.

Section 3 deals with the problem of simulating LogP on BSP. We begin by showing how a stall-free LogP computation can be executed on the BSP model. The slowdown is constant under the assumption that the bandwidth and latency parameters have the same value in both models. (This assumption is explored further in Section 5.) The proposed simulation is technically simple and clearly exposes the fact that, due to the upper limit to message transmission time, LogP is really a loosely synchronous model rather than an asynchronous one.

We then consider possible extensions of the simulation to stalling LogP computations, which exhibits a higher slowdown. It is not clear whether the higher slowdown is inherently required for simulating such anomalous computations, or whether it can be reduced by means of more sophisticated simulation techniques.

Section 4 presents both deterministic and randomized schemes to simulate BSP on LogP with a slowdown that is at most logarithmic in the number of processors, but becomes constant for wide ranges of parameter values. It is rather obvious that the BSP simulation must contain a

LogP algorithm for barrier synchronization. Perhaps less obvious is that, in order to comply with the LogP capacity constraint, the simulation must also embody a technique to decompose a set of messages into smaller sets where fewer messages can be sent or received by any given processor. This decomposition is achieved by a careful combination of known techniques. The results are mainly of an asymptotic nature and should eventually be refined to yield better estimates of constant factors involved.

In Section 5, we show that, for several well-known point-to-point topologies, the BSP abstraction can be supported nearly as efficiently as the LogP abstraction, to a higher degree than implied by the simulation results of the previous sections. Point-to-point networks have been chosen here as the basis for the analysis because they do provide an accurate model for the communication capabilities of some multiprocessors and because a large body of network routing results is available in the literature. It would be interesting to derive similar results for other types of architectures.

Finally, Section 6 concludes with a number of remarks derived from our analysis and directions for further research.

## 2 The Models

Both the BSP [1] and the LogP [3] models can be defined in terms of a virtual machine consisting of $p$ serial processors with unique identifiers $0, 1, ..., p-1$. Each processor has direct and exclusive access to a private memory bank and has a local clock. All clocks run at the same speed. The processors interact through a communication medium which supports the exchange 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.

## 2.1 BSP

A BSP machine operates by performing a sequence of *supersteps*. Conceptually, each superstep consists of three consecutive phases: a *local computation phase*, a *global communication phase*, and a *barrier synchronization*. Each processor can be thought of as being equipped with an *output pool*, into which outgoing messages are inserted, and an *input pool*, from which incoming messages are extracted. During the local computation phase, a processor may extract messages from its input pool, perform operations involving data held locally, and insert messages into its output pool. During the communication phase, every message held in the output pool of a processor is transferred to the input pool of its destination processor. The previous contents of the input pools, if any, are discarded. The superstep is concluded by a barrier synchronization, which informs the processors that all local computations are completed and that every message has reached its intended destination. The model prescribes that the next superstep may commence only after completion of the 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 running time of a superstep is expressed in terms of two parameters $g$ and $\ell$ as

$$T_{superstep} = w + gh + \ell, \tag{1}$$

where $w$ is the maximum number of local operations performed by any processor and $h$ is the maximum number of messages sent or received by any processor. The overall time of a BSP computation is simply the sum of the times of its constituent supersteps.

Intuitively, Relation (1) can be interpreted as follows. The time unit is chosen to be the duration of a local operation. For sufficiently large sets of messages ($h \gg \ell/g$), the communication medium delivers $p$ messages every $g$ units of time, so that $1/g$ can be viewed as measuring the available

bandwidth per processor. Parameter $\ell$ must be an upper bound on the time required for global barrier synchronization ($w = 0, h = 0$). Moreover, $g + \ell$ must be an upper bound on the time needed to route any partial permutation ($w = 0, h = 1$), and therefore on the latency of a message in the absence of other messages.

An interesting property of the model is that the same BSP program will run and give the same results, regardless of the values of parameters $g$ and $\ell$. Thus, these parameters, while they certainly influence the performance of a program, do not affect its correctness. This is clearly a desirable property when it comes to portability.

A drawback of the model is that all synchronizations are essentially global so that, for instance, two programs cannot run independently on two disjoint sets of processors. This is an obstacle for multiuser modes of operation.

## 2.2 LogP

In a LogP machine, at each time step, a processor can be either *operational* or *stalling*. When operational, a processor can do one of the following: (a) execute an operation on locally held data, (b) receive a message, (c) submit a message destined to another processor to the communication medium.

Conceptually, for each processor there is an output register where the processor puts any message to be *submitted* to the communication medium. The preparation of a message for submission requires $o$ time units, where $o$ is referred to as the *overhead* parameter. Once submitted, a message is *accepted* by the communication medium, possibly after some time has elapsed, and eventually delivered to its destination. Between the submission and acceptance of a message, the sending processor is assumed to be stalling. When a submitted message is accepted, the submitting processor reverts to the operational state.

Upon arrival, a message is promptly removed from the communication medium and buffered in some input buffer associated with the receiving processor. However, the actual *acquisition* of the incoming message by the processor may occur at a later time and requires overhead time $o$.

The behavior of the communication medium is modeled by two parameters, $G$ (*gap*) and $L$ (*latency*), which characterize its routing performance [2]. Specifically, the model prescribes that at least $G$ time steps must elapse between consecutive submissions or consecutive acquisitions by the same processor. Although the exact delivery time of a message is unpredictable, the model guarantees that the message arrives at its destination at most $L$ time steps after its acceptance. However, in order to capture network capacity limitations, the model requires that at any time there are no more than $\lceil L/G \rceil$ messages in transit for the same destination (*capacity constraint*). According to the proposers, parameter $G$ is the reciprocal of the maximum message injection (hence reception) rate per processor that the communication medium can sustain, while parameter $L$ provides an upper bound on network latency when the system is operating within capacity.

If accepting all the messages submitted at a given time does not violate the capacity constraint, then all the submitted messages are immediately accepted by the network. Otherwise, the acceptance of some of the messages will be delayed, until congestion clears, leaving the processors whose submitted messages have not yet been accepted in a stalling state. The specific mechanism by which messages are treated when the capacity constraint is violated is only informally described in the original paper ([3], p. 81). Here, we propose a characterization of such a mechanism which, while more formally stated, is faithful to the original one.

*Stalling Rule:* At a given time $t$, let $\lceil L/G \rceil - s$ be the number of messages in transit destined for processor $i$ that have been accepted but not yet delivered, and let $k$ be the number of submitted messages for processor $i$ yet to be accepted. Then, $\min\{k, s\}$ of these messages are accepted from

[2] The notation $G$ is adopted here rather than the customary $g$ to avoid confusion with the corresponding BSP parameter.

the output registers.

While the above stalling rule determines exactly the number of messages accepted at each time for each destination, it leaves the order in which messages are accepted completely unspecified. As this aspect is not mentioned in [3], we assume that any order is possible.

We observe that there are two sources of nondeterminism in LogP: (i) the delay between acceptance and delivery of a message by the network, and (ii) the delay between submission and acceptance of a message when the destination is congested. As a consequence, even for a fixed input, a given LogP program admits a multitude of different executions. A program is deemed to be correct if it computes the required input-output map under all admissible executions [3]. The class of admissible executions varies with the value of the parameters $L$, $G$, even for fixed $p$, with undesirable implications for the portability of the code. This aspect of the model might benefit from further investigation.

Another property of LogP which is worth observing is that, if two programs run on disjoint sets of processors, then their executions do not interfere. This is a desirable property, as it nicely supports partitioning of the computation into independent subcomputations, as well as multiuser modes of operation.

An intriguing aspect of the LogP model is the stalling regime. Stalling ought to capture the degradation in performance observed in real networks [22] when traffic exceeds a certain threshold ([2], p. 9). Correspondingly, the model discourages the development of stalling programs; indeed, it is clear from the literature on LogP algorithms that programmers are expected to make every effort to ensure that their programs are stall-free and to allow stalling only as a last resort. However, whether the specific definition of stalling given in [3] and formalized above completely achieves these intended goals is not clear, due to the following considerations.

[3] A similar definition of correctness is given in [3], page 81, with apparent reference to stall-free executions only. The present definition covers also stalling executions.

According to the stalling rule, when the capacity constraint for a given processor is exceeded, the latency of individual messages does grow, but the delivery rate at the hot spot is the highest possible given the bandwidth limitation (one message every $G$ steps). Therefore, if sending the messages to that processor is the core of the given computation, stalling might provide an efficient way of accomplishing the task, in spite of the loss of CPU cycles incurred by the stalling processors. In such situations, the LogP performance model would actually encourage the use of stalling.

It remains to be seen whether the potential performance advantage deriving from stalling can be effectively exploited in some application, and what the cost of enforcing the stalling rule on a real platform might be. If taken literally, the stalling rule implies that, for every processor, a count of the messages in transit destined to that processor is maintained, with immediate effects when the count exceeds the capacity constraint. It appears hard to implement this behavior, within the required performance bounds, especially in networks with high latency ($L >> G$).

For the reasons discussed above, in the present paper, we focus mostly on *stall-free* programs, defined as those programs for which all admissible executions are stall-free. However, the above discussion also shows that stalling deserves further attention.

Next, we discuss some constraints that can be reasonably assumed for the LogP parameters and can be summarized as follows:

$$\max\{2, o\} \leq G \leq L.$$

- $G \geq o$. Since a processor is always forced to spend $o$ time steps for preparing/acquiring a message, without loss of generality, we can assume $G \geq o$.

- $G \geq 2$. The choice of $\lceil L/G \rceil$ as the capacity threshold may pose some (probably unintended) modeling problems when $G = 1$, hence $\lceil L/G \rceil = L$. Consider a situation where, at time $t = 0$,

$L$ processors simultaneously send a message destined to the same processor. According to the model, no stalling occurs, hence all these $L$ messages are guaranteed to be delivered by time $t = L$. Since the communication medium can deliver only one message at a time to a single destination, one message must arrive for each time $t = 1, 2, \ldots, L$. The implication is that, for any processor $j$ and every $L$-tuple of processors $i_1, i_2, \ldots, i_L$, there is one $i_h$ from which $j$ can be reached in one time step. This is clearly a strong performance requirement hard to support on a real machine. In contrast, if $G$ were such that $\lceil L/G \rceil \leq \lceil L/2 \rceil$ such an extreme scenario would not occur since each message in transit to a processor may be realistically delivered in at least $\Theta(L)$ time steps.

- $G \leq L$. This assumption is made to permit bounded size for the input buffers of each processor. In fact, assume that $G > L$, and consider a program where processor $i \in \{0, 1\}$ sends a message to processor 2 at time $\max\{G, 2L\}k + Li$, for $k = 0, 1, \ldots$. At any time, there is only $\lceil L/G \rceil = 1$ message in transit in the medium, therefore the capacity constraint is always satisfied and no stalling occurs. However, messages are delivered to processor 2 and stored in its input buffers at a rate strictly greater than $1/G$ (in fact, the rate is $1/L > 1/G$ when $G < 2L$, and is exactly $2/G$ when $G \geq 2L$). However, messages can be actually acquired by the processor only at the lower rate $1/G$, thus requiring unbounded buffer space.

# 3 Simulation of LogP on BSP

In this section, we study the simulation of LogP programs on BSP. The key result is the following theorem, which applies to stall-free programs. Some issues arising from stalling are also discussed.

**Theorem 1** *A stall-free LogP program can be simulated in BSP with slowdown $O(1 + g/G + \ell/L)$. When $\ell = \Theta(L)$ and $g = \Theta(G)$, the slowdown becomes constant.*

The simulation consists of a sequence of BSP supersteps, each of which simulates a *cycle* of $L/2$ consecutive instructions of the given stall-free LogP program. (For simplicity, we assume $L$ even. Minor modifications are required to handle the case of $L$ odd.)

In a superstep, the $i$-th BSP processor $B_i$ executes the instructions specified by the program for the $i$-th LogP processor $L_i$ in the corresponding cycle, using its own local memory to store the contents of $L_i$'s local memory. Message submissions are simulated by insertions in the output pool, while the actual transmission of the messages takes place during the communication phase at the end of the superstep. As for message acquisitions, at the beginning of the superstep, each processor transfers the messages that were sent to it in the preceding superstep into a local FIFO queue and simulates the actual acquisitions by extractions from the queue. Therefore, all messages submitted in a cycle arrive at their destination in the subsequent cycle.

The correctness of the simulation follows from the existence of an execution of the LogP program consistent with such a delivery schedule. In fact, in a cycle of $L/2$ consecutive steps, no more than $\lceil L/G \rceil \leq L/2$ messages for the same processor are submitted, since all these messages could be still in transit at the end of the cycle and the program is stall-free. Therefore, it is possible to select distinct arrival times for all the messages within the next cycle, so that the delivery time for each message is at most $L$.

Finally, each superstep of the simulation involves the routing of an $h$-relation where $h \leq \lceil L/G \rceil$, hence the overall simulation time of a cycle is $O(L + g\lceil L/G \rceil + \ell)$. Considering that a cycle corresponds to a segment of the LogP computation of duration $L/2$, the slowdown stated in Theorem 1 is established.

In the preliminary conference version of this paper [23], it was unjustifiably claimed that the simulation of Theorem 1 could be extended to arbitrary LogP programs while maintaining the same slowdown. Instead, as pointed out by Ramachandran [24, 20], such an extension does not appear

straightforward, and may in fact not be possible.

In the above simulation, for a cycle where stalling occurs, the upper bound $h = O(L/G)$ no longer holds for the corresponding BSP superstep, possibly leading to a superstep time considerably larger than in the stall-free case. Performance can be improved if messages are suitably preprocessed before being sent. In fact, standard sorting [25] and prefix [4] techniques can be used to assign messages an order of network acceptance consistent with the stalling rule. Along these lines, an $O(((\ell + g)/G) \log p)$ slowdown can be obtained, which is still not negligible. Whether this bound can be improved upon remains to be seen. Interestingly, due to Theorem 1, a non-trivial lower bound on the slowdown of any BSP simulation of LogP stalling programs would also apply to any stall-free LogP simulation of LogP stalling programs, and would therefore indicate that stalling adds computational power to LogP.

# 4 Simulation of BSP on LogP

We now consider the reverse problem of simulating an arbitrary BSP program in the LogP model. First, we develop deterministic, stall-free simulations. Then, we explore the potential of randomization; here, we allow stalling to occur, but only with polynomially small probability, so that the expected simulation time is essentially determined by the stall-free part of the execution and is not very sensitive to the specific way by which stalling is resolved.

**Theorem 2** *Any BSP superstep involving at most $w$ local operations per processor and the routing of an $h$-relation can be simulated in stall-free LogP with worst-case time*

$$O(w + (Gh + L)S(L, G, p, h)),$$

*where $S(L, G, p, h) = O(1)$ for $h = \Omega(p^\epsilon + L \log p)$ and $S(L, G, p, h) = O(\log p)$ otherwise.*

When $G = \Theta(g)$ and $L = \Theta(\ell)$, $S(L, G, p, h)$ is an upper bound to the slowdown of the simulation. An explicit expression for $S$ is derived in this section.

The simulation of a BSP superstep where each processor executes at most $w$ local operations and the generated messages form an $h$-relation has the following general structure. First, for $1 \leq i \leq p$, the $i$-th LogP processor $L_i$ executes the local computation of the $i$-th BSP processor $B_i$, buffering all generated messages in its local memory. Second, $L_i$ joins a synchronization activity which will end after all the processors have completed their local computation. Third, a LogP routing algorithm is invoked to send all the messages generated in the superstep to their destinations, while also making each processor aware of termination, so that no further synchronization is needed before starting the next superstep.

The simulation time for the superstep can then be expressed as $T_{superstep} = w + T_{\text{synch}} + T_{\text{rout}}(h)$, where $T_{\text{synch}}$ is the duration of the synchronization activity, measured from the moment when the last LogP processor joins the activity, and $T_{\text{rout}}(h)$ is the time to deliver all messages. The result stated in Theorem 2 follows from bounds for $T_{\text{synch}}$ and $T_{\text{rout}}(h)$ which are derived in the next subsections.

### 4.1 Synchronization

We base processor synchronization in LogP on the *Combine-and-Broadcast* (CB) primitive which, given an associative operator $op$ and input values $x_0, x_2, \ldots, x_{p-1}$, initially held by distinct processors, returns $op(x_0, x_2, \ldots, x_{p-1})$ to all processors.

A simple algorithm for CB consists of an ascend and a descend phase on a complete $\max\{2, \lceil L/G \rceil\}$-ary tree with $p$ nodes, which are associated with the processors. At the beginning of the algorithm, a leaf processor just sends its local input to its parent. An internal node waits until it receives a value from each of its children, then combines these values with its local one and forwards the result to

its parent. Eventually, the root computes the final result and starts a descending broadcast phase. When $\lceil L/G \rceil \geq 2$, the algorithm clearly complies with the LogP capacity constraint, since no more than $\lceil L/G \rceil$ messages can be in transit to the same processor at any time. When $\lceil L/G \rceil = 1$, the tree is binary, and we additionally constrain transmissions to the father to occur only at times which are even multiples of $L$ for left children and odd multiples of $L$ for right children.

Let $T_{\mathrm{CB}}$ denote the running time of the CB algorithm. We have:

$$T_{\mathrm{CB}} \leq 3(L+o)\frac{\log p}{\log(1+\lceil L/G \rceil)} = O\left(L\frac{\log p}{\log(1+\lceil L/G \rceil)}\right).$$

The above algorithm is optimal for the CB problem to within constant factors, as an immediate consequence of the following proposition.

**Proposition 1** *Any stall-free LogP algorithm for CB with OR as the associative operation requires time*

$$\Omega\left(L\frac{\log p}{\log(1+\lceil L/G \rceil)}\right).$$

*Proof:* Using the simulation strategy developed in Section 3, we can transform any $T$-time LogP stall-free algorithm for CB into a $\Theta(T+L)$-time algorithm for the problem of computing the Boolean OR of $p$ bits on a $p$-processor BSP machine with parameters $g = G$ and $\ell = L$, when the bits are initially distributed evenly among the first $\lceil p/\lceil L/G \rceil \rceil$ BSP processors, and each superstep routes $h$-relations with $h \leq \lceil L/G \rceil$. Then, the proof follows from the lower bound for this last problem developed by Goodrich in [25]. □

A different optimal tree-based algorithm for CB appears in [17, 26], where the running time, however, is not explicitly expressed as a function of $p$, $G$ and $L$.

The synchronization needed to simulate a BSP superstep is implemented as follows. Upon completion of its own local activity, each LogP processor enters a Boolean 1 as input to a CB

computation with Boolean AND as the associative operator. The activity terminates when CB returns 1 to all processors. It is easy to see that the CB algorithm described above works correctly even if the processors join the computation at different times. In this case, $T_{CB}$ represents the time to completion measured from the joining time of latest processor. We have:

**Proposition 2** *The synchronization used to simulate a BSP superstep in LogP can be performed, without stalling, in time*

$$T_{\text{synch}} = O\left(L\frac{\log p}{\log(1+\lceil L/G\rceil)}\right).$$

### 4.2 Deterministic routing of $h$-relations

A major impediment to be overcome in realizing arbitrary $h$-relations in LogP is the capacity constraint. For $h \leq \lceil L/G\rceil$, an $h$-relation can be routed in worst-case time $2o + G(h-1) + L \leq 4L$ by having each processor send its messages, one every $G$ steps. For larger $h$, this simple-minded strategy could lead to the violation of the capacity constraint. Hence, a mechanism to decompose the $h$-relation into subrelations of degree at most $\lceil L/G\rceil$ is required.

By Hall's Theorem [27], any $h$-relation can be decomposed into disjoint 1-relations and, therefore, be routed off-line in optimal $2o + G(h-1) + L$ time in LogP. Off-line techniques may indeed be useful when the $h$-relation is input independent, hence, known before the program is run.

In general, however, the $h$-relation becomes known only at run-time and the required decomposition must be performed on-line. Next, we describe a protocol for routing $h$-relations, which decomposes the relation by standard sorting techniques. We let $r$ (resp., $s$) denote the maximum number of messages sent (resp., received) by any processor, whence $h = \max\{r, s\}$.

1. Compute $r$ and broadcast it to every processor. Then, make the number of messages held by the processor exactly $r$, by forming in each processor a suitable number of dummy messages with nominal destination $p$.

2. Sort all messages by destination and provide each message with its *rank* in the sorted sequence.

3. Compute $s$ and broadcast it to every processor. (Ignore the dummy messages when calculating this quantity.)

4. For each $i$ such that $0 \leq i < h = \max\{r, s\}$, execute a routing cycle delivering all (non-dummy) messages whose rank taken modulo $h$ is $i$.

Both Step 1 and Step 3 can be executed by means of CB in time $r + T_{\mathrm{CB}}$. Also, it is easy to see that the $h$ cycles of Step 4 can be pipelined with a period of $G$ steps without violating the capacity constraint; hence, Step 4 takes optimal time $2o + G(h-1) + L$. Overall, the $h$-relation is routed in time

$$T_{\mathrm{rout}}(h) \leq 2T_{\mathrm{CB}} + T_{\mathrm{sort}}(r, p) + 2o + (G+2)h + L, \tag{2}$$

where $T_{\mathrm{sort}}(r, p)$ denotes the time required for sorting $rp$ keys in the range $[0, p]$ evenly distributed among the processors. Upper bounds on $T_{\mathrm{sort}}(r, p)$ are given below.

**Sorting** In what follows we describe two LogP sorting schemes. The first scheme is based on the *AKS network* [28] for sorting $p$ messages, extended to the case of $r$ messages per processor through standard techniques. The second scheme is based on the *Cubesort* algorithm [29]. The former scheme turns out to be more efficient for small values of $r$, while the second yields better performance for large values of $r$.

The AKS network can be regarded as a directed graph with $p$ nodes connected by $K = O(\log p)$ sets of edges, with each set realizing a matching among the $p$ nodes. Consider the case $r = 1$. The sorting algorithm runs in $K$ comparison steps. In the $i$-th step, the endpoints of each edge of the $i$-th matching exchange their keys and select the minimum or the maximum according to their position with respect to the orientation of the edge. When $r > 1$, first a local sorting is performed

within each processor, and then the algorithm proceeds as before, replacing each compare-swap step by a merge-split step among sorted sequences of $r$ messages [30].

The above algorithm can be easily implemented in LogP, since the message transmissions required at each step are known in advance and can be decomposed into a sequence of $r$ 1-relations that are routed in time $2o + G(r-1) + L$. On LogP, the running time of the algorithm is

$$T_{\mathrm{AKS}}(r,p) = O((Gr + L)\log p),$$

since the cost of the message transmissions dominates the cost of the initial sorting step (which, for $r$ keys in the range $[0,p]$, is $O(r \log p)$) and the cost of the $O(\log p)$ local merge-splits steps ($O(r)$ per step).

Cubesort consists of $O(25^{\log^* pr - \log^* r}(\log pr/\log(r+1))^2)$ rounds, where each round partitions the $pr$ keys into groups of size at most $r$ and sorts the groups in parallel. In LogP, the algorithm can be implemented by letting each round be preceded by a suitable data redistribution so that the subsequent group sortings can be done locally within the processors. Since keys are in the range $[0,p]$, each local sorting can be performed in time $T_{\mathrm{seq-sort}}(r) = r\min\{\log r, \lceil \log p/\log r \rceil\}$ by using Radixsort. Note that when $r = p^\epsilon$, for any positive constant $\epsilon$, we get $T_{\mathrm{seq-sort}}(r) = O(r)$. Each data redistribution involves an $r$-relation which is known in advance and can therefore be decomposed into $r$ 1-relations, routed in time $2o + G(r-1) + L$. Thus, the running time of the algorithm in LogP is

$$T_{\mathrm{CS}}(r,p) = O\left(25^{\log^* pr - \log^* r}\left(\frac{\log pr}{\log(r+1)}\right)^2 (T_{\mathrm{seq-sort}}(r) + Gr + L)\right).$$

Note that for $r \leq 2^{\sqrt{\log p}}$, the AKS-based sorting scheme outperforms the Cubesort-based one. In contrast, when $r = p^\epsilon$, for any positive constant $\epsilon$, the running time of the latter sorting becomes

$T_{\mathrm{CS}}(r,p) = O(Gr+L)$, which is clearly optimal and improves upon $T_{\mathrm{AKS}}(r,p)$ by a factor $O(\log p)$. Additional material on sorting in the LogP model can be found in [16, 10].

By adding the contributions to the simulation of a BSP superstep due to local computation, synchronization (see Proposition 2), and sorting plus routing (this subsection), we obtain:

$$T_{superstep} = O\left(w + (Gh+L)S(L,G,p,h)\right),$$

which yields Theorem 2 with

$$S(L,G,p,h) = \frac{L\log p}{(Gh+L)\log(1+\lceil L/G\rceil)} + \min\left\{\log p, \left(\frac{\log ph}{\log(h+1)}\right)^2 \left\lceil \frac{T_{\mathrm{seq-sort}}(h)}{Gh+L}\right\rceil\right\}.$$

(Note that the term $25^{\log^* ph - \log^* h}$ does not appear in the slowdown, since this term becomes constant when $h$ is large enough that sorting via Cubesort is preferable to sorting via AKS.) In all cases, $S(L,G,p,h) = O(\log p)$. Moreover, $S(L,G,p,h) = O(1)$ for $h$ sufficiently large (e.g., $h = \Omega(p^\epsilon + L\log p)$, for constant $\epsilon$).

### 4.3 Randomized routing of $h$-relations

While the slowdown of simulations based on sorting is asymptotically small, it remains substantial for practical purposes. In part, this is a reflection of the inherent difficulty of decomposing a general $h$-relation on-line. However, in many cases, the problem can be eased if some properties of the $h$-relation are known in advance.

In this subsection, we show that if the degree $h$ of the relation is known in advance to each processor, then routing can be accomplished in asymptotically optimal time by resorting to simple randomized procedures. Specifically, we consider the case $h > \lceil L/G\rceil$, since otherwise the routing is trivially accomplished in $O(L)$ steps, and establish the following result.

**Theorem 3** *Let $\lceil L/G \rceil \leq h \leq p$ be known in advance by the processors, and let $\lceil L/G \rceil \geq c_1 \log p$, for some constant $c_1 > 0$. For any constant $c_2 > 0$, any $h$-relation can be realized in the LogP model, without stalling, in time $\alpha Gh$, with probability at least $1 - p^{-c_2}$, where $\alpha = 4e2^{(c_2+3)/c_1}$.*

To implement the $h$-relation, the following protocol is executed in each LogP processor.

1. Group the messages in $R$ *batches* by randomly and independently assigning an integer uniformly distributed between 1 and $R$ to each message.

2. Execute $R$ rounds, each of $2(L + o)$ steps. In Round $r$, $1 \leq r \leq R$, transmit up to $\lceil L/G \rceil$ messages belonging to batch $r$, one transmission every $G$ steps.

3. Transmit all remaining messages, one transmission every $G$ steps.

We now show that, for a suitable value of $R$, with high probability (i) all messages are transmitted in Step 2, and (ii) the capacity constraint is never violated (i.e., no stalling occurs).

Let $X_r(j)$ and $Y_r(j)$ denote, respectively, the number of messages originating from and destined to the $j$-th LogP processor in Round $r$, for $1 \leq r \leq R$. For simplicity, we consider the case where each processor is source/destination of exactly $h$ messages, since this is clearly a worst case for our analysis. Then, both $X_r(j)$ and $Y_r(j)$ are the sum of $h$ independent Bernoulli variables which take value 1 with probability $1/R$ (and value 0 with probability $1 - 1/R$). Let us choose

$$R = (1 + \beta)\frac{h}{\lceil L/G \rceil},$$

for some constant $\beta \geq 1$, which implies an expected value for $X_r(j)$ and $Y_r(j)$ of $h/R = \lceil L/G \rceil/(1+\beta)$. By applying the well-known Chernoff bound [31], we get

$$\text{Prob}(X_r(j) > \lceil L/G \rceil) = \text{Prob}(Y_r(j) > \lceil L/G \rceil) \leq \left(\frac{e^\beta}{(1+\beta)^{1+\beta}}\right)^{\frac{\lceil L/G \rceil}{1+\beta}}$$

where $X_r(j) > \lceil L/G \rceil$ implies that some message in Round $r$ is not sent by the $j$-th processor, and $Y_r(j) > \lceil L/G \rceil$ implies that in Round $r$ there is (potential) stalling due to violation of the capacity constraint by messages destined to the $j$-th processor. Thus, the probability that no stalling occurs and that no processor has messages to transmit in Step 3, is at least

$$1 - 2Rp \left( \frac{e^{\beta}}{(1+\beta)^{1+\beta}} \right)^{\frac{\lceil L/G \rceil}{1+\beta}} \geq 1 - p^{c_2},$$

where the last inequality is obtained by making use of the relations $R, h \leq p$ and $\lceil L/G \rceil \geq c_1 \log p$, and letting $\beta = e2^{(c_2+3)/c_1} - 1$. As a consequence, the time bound of the proposed protocol is $2(L+o)R \leq 4LR = 4(1+\beta)Gh = \alpha Gh$ with probability at least $1 - p^{-c_2}$, as stated.

For the randomized protocol illustrated above, a nonstalling execution can be guaranteed only with high probability. However, according to the specification of stalling given in Section 2, even in the presence of such an event, an $h$-relation is completed in time $O(Gh^2)$, which provides a worst-case upper bound to the running time of the protocol. The key observation is that the total time spent by a processor $L_i$ stalling while trying to send a message to processor $L_j$ is at most $Gh$ since, while a hot-spot, $L_j$ receives one message every $G$ steps, and there are at most $h$ messages for it to receive. Since there are at most $h$ different destinations for $L_i$'s messages, the total time spent stalling is $O(Gh^2)$.

By employing the randomized protocol to perform $h$-relations when $h < p$, in conjunction with the deterministic strategy when $h \geq p$, it can be easily shown that the communication phases of any sequence of $T$ BSP supersteps, where the $i$-th superstep requires the routing of an $h_i$-relation, can be simulated by LogP in time $O(G \sum_{i=1}^{T} h_i)$ with high probability, provided that the $h_i$'s are known and that $\lceil L/G \rceil = \Omega(\log p)$. It should be observed that this result is rather insensitive to the specific form of the adopted stalling rule, since it holds under the reasonable assumption that

the stalling rule guarantees a polynomially bounded time to route an $h$-relation.

Consequently, a $p$-processor machine supporting LogP with parameters $L$ and $G$, with $\lceil L/G \rceil = \Omega(\log p)$, is able to simulate programs written for a BSP machine with parameters $g = G$ and $\ell = L$ with constant slowdown with high probability, as long as the degree of the $h$-relation in each superstep is known by the processors in advance and is large enough to hide the extra cost due to barrier synchronization (namely, $h = \Omega((L/G) \log p / \log(1 + \lceil L/G \rceil))$). Note that the randomized simulation widens the range of optimality of the deterministic simulation considerably. It would be interesting to explore good randomized strategies for small values of $\lceil L/G \rceil$, which are not covered by Theorem 3, perhaps by adapting some of the randomized routing strategies for bandwidth-latency models proposed in the literature (e.g., [32, 4, 19]).

## 5 BSP vs LogP on Processor Networks

The preceding sections show that the simulation of stall-free LogP on BSP is considerably simpler than the simulation of BSP on stall-free LogP. Moreover, under the assumption that $G = \Omega(g)$ and $L = \Omega(\ell)$, the former exhibits a smaller slowdown, which reinforces the intuition that BSP provides a more powerful abstraction than stall-free LogP does. However, there is no guarantee that the parameters obtained by a direct implementation of the two models on the same machine satisfy the above relations. Indeed, a key ingredient needed to implement a BSP or a LogP abstraction on a machine is an algorithm for routing $h$-relations. The algorithm has to support arbitrary values of $h$ for the BSP implementation, and only $\lceil L/G \rceil$-relations for stall-free LogP. It is thus conceivable that the restriction to small-degree relations yield faster routing algorithms, and therefore smaller values of $G$ and $L$ for stall-free LogP, compared to the corresponding BSP parameters. In this section we show that this is not the case for a wide range of machines.

In general, Theorem 1 implies that any machine that supports BSP with parameters $g$ and $\ell$

| Topology | $\gamma(p)$ | $\delta(p)$ | Reference |
|---|---|---|---|
| $d$-dim Array $d = O(1)$ | $p^{1/d}$ | $p^{1/d}$ | [34] |
| Hypercube (multi-port) | 1 | $\log p$ | [32] |
| Hypercube (single-port) | $\log p$ | $\log p$ | [32] |
| Butterfly, CCC, Shuffle-Exchange | $\log p$ | $\log p$ | [32] |
| Pruned Butterfly Mesh-of-Trees | $\sqrt{p}$ | $\log p$ | [35] |

Table 1: Bandwidth and latency parameters of prominent topologies.

also supports stall-free LogP with parameters $G = \Theta(g)$ and $L = \Theta(\ell + g)$. Conversely, Theorem 2 implies that any machine that supports LogP with parameters $G$ and $L$ also supports BSP with parameters $g = \Theta(G \cdot S(L, G, p, h)) = O(G \log p)$ and $\ell = \Theta(L \log p)$. However, tighter relations between the parameters may result when the best direct implementations of the two models are considered for a specific machine. We next examine this issue for machines that can be accurately modeled by suitable networks of processors with local memory.

For many prominent interconnections, algorithms are known that route $h$-relations, for *arbitrary* $h$, in *optimal* time $\Theta(\gamma(p)h + \delta(p))$, where $\delta(p)$ denotes the network diameter and $\gamma(p)$ a bandwidth-related network parameter (e.g., $\gamma(p) = O(p/b(p))$ where $b(p)$ is the bisection width [33]). Table 1 indicates the asymptotic values of $\gamma(p)$ and $\delta(p)$ for a number of such interconnections.

Any implementation of BSP on any of these networks requires $g = \Omega(\gamma(p))$ and $\ell = \Omega(\delta(p))$. Moreover, there is an implementation that matches these lower bounds for both $g$ and $\ell$, where the value of $\ell$ stems from the fact that on any processor network barrier synchronization can always be implemented in time proportional to the diameter. Thus, the choice $g^\star = \Theta(\gamma(p))$ and $\ell^\star = \Theta(\delta(p))$ represents the best attainable parameter setting for BSP implementations for these interconnections.

As for stall-free LogP, the definition of the model requires that any $\lceil L/G \rceil$-relation be routed in time $L$, which implies $L \geq \lceil L/G \rceil \gamma(p) + \delta(p)$. Therefore, it follows that $L = \Omega(\gamma(p) + \delta(p))$ and $G = \Omega(\gamma(p))$. When combined with the observation relating to Theorem 1 stated above, this suggests that the choice $L^\star = \Theta(\gamma(p) + \delta(p))$ and $G^\star = \Omega(\gamma(p))$ represents the best attainable parameter setting for LogP implementations for these interconnections.

We can summarize the above discussion as follows:

**Observation 1** *For most processor networks in the literature,* $G^\star = \Theta(g^\star)$ *and* $L^\star = \Theta(\ell^\star + g^\star)$*, where* $G^\star, L^\star$ *and* $g^\star, \ell^\star$ *represent the best attainable parameters for stall-free LogP and BSP implementations, respectively.*

# 6 Conclusions

The arguments developed in the previous sections suggest that BSP and stall-free LogP exhibit comparable power when regarded as computational models for the design of algorithms. Namely, we have provided asymptotically efficient cross-simulations between the two models and argued that both abstractions can be implemented on most prominent point-to-point interconnections with comparable values of their respective bandwidth and latency parameters.

When considering ease of use, the BSP abstraction, with its facility for handling arbitrary $h$-relations, provides a more convenient framework for algorithm design than LogP, which forces the programmer to cast algorithms in a way that respects the capacity constraint. While many algorithms can quite naturally be expressed as stall-free LogP programs without undue difficulty, there are others that appear to require considerable ingenuity to be formulated within the LogP framework. This difficulty is evident in the LogP literature: for example, the simple parallel implementation of Radixsort in [16] involves relations that may violate the capacity constraint and whose cost cannot be estimated reliably under those circumstances.

With respect to portability, an important question is how would a change of the machine parameters affect a program. In BSP, such a change will impact performance, but not alter correctness. In LogP, the change might turn correct programs into incorrect ones, or stall-free programs into stalling ones, although the extent to which these undesirable phenomena occur needs investigation.

With respect to partitionability of the system into subsystems running independent computations, BSP's global synchronization might induce unnecessary complications, whereas LogP leads to natural solutions.

Although it is recognized that the LogP model is more descriptive than BSP and may therefore provide more accurate performance predictions, our findings suggest that the loss in accuracy incurred by choosing BSP over LogP is relatively minor in comparison with the advantages provided by the former's more convenient programming abstraction. This conclusion, however, cannot be considered definitive. In fact, it should be remarked that the BSP simulation on LogP is not straightforward and its slowdown, even when constant, may be significant in practice. A better control of constant factors is needed to confirm the results of our analysis, which is mainly of asymptotic nature. Further light needs to be shed on the issue of supporting the two abstractions on parallel platforms by considering other architectures that do not fall within the point-to-point framework. Finally, a more systematic study ought to be devoted to the issue of stalling, possibly leading to interesting developments within the LogP approach to bandwidth-latency models of parallel computation.

**Acknowledgments** We gratefully acknowledge Vijaya Ramachandran for pointing out the incorrect claim about the generalization of Theorem 1 to LogP programs that might stall, which appeared in the earlier versions of this paper, and for suggesting a more careful examination of the issue of stalling in LogP. We also would like to thank the SPAA'96 program committee for the

extensive and valuable feedback given on the conference version of this paper.

## References

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

[2] D.E. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos R. Subramonian, and T.V. Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of Fourth ACM SIGPLAN Symposium on principles and Practice of Parallel Programming*, pages 1–12, May 1993.

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

[4] A.V. Gerbessiotis and L.G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, 1994.

[5] W.F. McColl. BSP programming. In K.M. Chandy G.E. Blelloch and S. Jagannathan, editors, *DIMACS Series in Discrete Mathematics*, pages 21–35. American Mathematical Society, 1994.

[6] W.F. McColl. General purpose parallel computing. In A.M. Gibbons and P. Spirakis, editors, *Proc. of the 1991 ALCOM Spring School on Parallel Computation*, pages 337–391, Warwick, UK, 1994. Cambridge University Press.

[7] W.F. McColl. Scalable computing. In J. Van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, LNCS 1000, pages 46–61. Springer-Verlag, 1995.

[8] G.E. Blelloch, P.B. Gibbons, Y. Matias, and M. Zagha. Accounting for memory bank contention and delay in high-bandwidth multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 8(9):943–958, September 1997.

[9] T. Cheatham, A. Fahmy, D. Stefanescu, and L. Valiant. Bulk Synchronous Parallel Computing — A Paradigm for Transportable Software. In *Proc. of the 28th Annual Hawaii Conference on System Sciences. Volume II: Software Technology*, pages 268–275. IEEE Computer Society Press, January 1995.

[10] M. Adler, J.W. Byers, and R.M. Karp. Parallel sorting with limited bandwidth. In *Proc. of the 7th ACM Symp. on Parallel Algorithms and Architectures*, pages 129–136, S. Barbara, CA, USA, July 1995.

[11] A. Bäumker, W. Dittrich, and F. Meyer auf der Heide. Truly efficient parallel algorithms: $c$-optimal multisearch for and extension of the BSP model. In *Proc. of the 3rd European Symposium on Algorithms*, LNCS 979, pages 17–30, 1995.

[12] B.H.H. Juurlink and H.A.G. Wijshoff. A quantitative comparison of paralle computation models. In *Proc. of the 8th ACM Symp. on Parallel Algorithms and Architectures*, pages 13–24, June 1996.

[13] P. De la Torre and C.P. Kruskal. Submachine locality in the bulk synchronous setting. In *Proc. of EUROPAR 96*, LNCS 1124, pages 352–358, August 1996.

[14] M. Goudreau, J.M.D. Hill, W. McColl, S. Rao, D.C. Stefanescu, T. Suel, and T. Tsantilas. A proposal for the BSP worldwide standard library. Technical report, Oxford University Computing Laboratory, Wolfson Building, Parks Rd., Oxford OX1 3QD, UK, 1996.

[15] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas. Towards efficiency and portability: Programming with the BSP model. In *Proc. of the 8th ACM Symp. on Parallel Algorithms and Architectures*, pages 1–12, June 1996.

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

[17] R. Karp, A. Sahay, E.E. Santos, and K.E. Shauser. Optimal broadcast and summation in the LogP model. In *Proc. of the 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 142–153, Velen, Germany, 1993.

[18] A. Alexandrov, M.F. Ionescu, K.E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model. In *Proc. of the 7th ACM Symp. on Parallel Algorithms and Architectures*, pages 95–105, Santa Barbara, CA, July 1995.

[19] M. Adler, J.W. Byers, and R.M. Karp. Scheduling parallel communication: The $h$-relation problem. In *Proc. of the 20th International Symp. on Mathematical Foundations of Computer Science*, LNCS 969, pages 1–20, 1995.

[20] V. Ramachandran, B. Grayson, and M. Dahlin. Emulations between QSM, BSP and LogP: a framework for general-purpose parallel algorithm design. To appear in *Proc. of the 10th ACM-SIAM Symp. on Discrete Algorithms*, Baltimore, MD, January 1999.

[21] P. B. Gibbons and Y. Matias and V. Ramachandran. Can a shared-memory model serve as a bridging model for parallel computation? In *Proc. of the 9th ACM Symp. on Parallel Algorithms and Architectures*, pages 72–83, Newport, RI, June 1997.

[22] W.J. Dally. Performance analysis of $k$-ary $n$-cube interconnection networks. *IEEE Trans. on Computers*, 39(6):775–785, June 1990.

[23] G. Bilardi, K.T. Herley, A. Pietracaprina, G. Pucci and P. Spirakis. BSP vs LogP. In *Proc. of the 8th ACM Symposium on Parallel Algorithms and Architectures*, pages 25–32, June 1996.

[24] V. Ramachandran. Personal Communication, June 1998.

[25] M.T. Goodrich. Communication-efficient parallel sorting. In *Proc. of the 28th ACM Symp. on Theory of Computing*, pages 247–256, Philadelphia, Pennsylvania USA, May 1996.

[26] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. In *Proc. of the 4th ACM Symposium on Parallel Algorithms and Architectures*, pages 13–22, June 1992.

[27] B. Bollobàs. *Graph theory : an introductory course*. Springer-Verlag, New York, NY, 1979.

[28] M. Ajtai, J. Komlòs, and E. Szemerèdi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, 1983.

[29] R. Cypher and J.L.C. Sanz. Cubesort: a parallel algorithm for sorting $n$ data items with $s$-sorters. *Journal of Algorithms*, 13:211–234, 1992.

[30] D.E. Knuth. *The Art of Computer Programming*, volume 3 : *Sorting and Searching*. Addison Wesley, Reading, MA, 1973.

[31] T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Information Processing Letters*, 33(6):305–308, February 1990.

[32] L.G. Valiant. General purpose parallel computing. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, chapter 18, pages 944–996. Elsevier, NL, 1990.

[33] C.D. Thompson. *A complexity theory for VLSI.* PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, Aug. 1980. Tech. Rep. CMU-CS-80-140.

[34] J.F. Sibeyn and M. Kaufmann. Deterministic 1-$k$ routing on meshes, with application to hot-potato worm-hole routing. In *Proc. of the 11th Symp. on Theoretical Aspects of Computer Science*, LNCS 775, pages 237–248, 1994.

[35] P. Bay and G. Bilardi. Deterministic on-line routing on area-universal networks. *Journal of the ACM*, 42(3):614–640, May 1995.