LogP

A Practic Parallel

OUR GOAL IS TO DEVELOP A MODEL OF PARALLEL COMPUTATION THAT WILL serve as a basis for the design and analysis of fast, portable parallel algorithms, such as algorithms that can be implemented effectively on a wide variety of current and future parallel machines. If we look at the body of parallel algorithms developed under current parallel models, many are impractical because they exploit artificial factors not present in any reasonable machine, such as zero communication delay or infinite bandwidth. Others are overly specialized because they are tailored to the idiosyncrasies of a single machine, such as a particular interconnect topology. To improve this situation, we need to design algorithms using a model that is realistic enough to capture the most important performance factors in real machines, yet abstract enough to be generally useful and to keep the algorithm analysis tractable. Ideally, producing a better algorithm under the model should yield a better program in practice.

The Parallel Random Access Machine (PRAM) [8] is the most popular model for representing and analyzing the complexity of parallel algorithms. A

A new parallel machine model reflects the critical technology trends underlying parallel computers

A L MODEL *of* COMPUTATION

David E. Culler, Richard M. Karp, David Patterson, Abhijit Sahay, Eunice E. Santos, Klaus Erik Schauser, Ramesh Subramonian, and Thorsten von Eicken

PRAM consists of a collection of processors which compute synchronously in parallel and communicate with a global random access memory. The PRAM model is useful for gross classification of algorithms and problems, but it ignores important performance bottlenecks in modern parallel machines because it assumes a single shared memory in which each processor can access any memory cell in unit time. Surprisingly fast algorithms can be developed by exploiting such loopholes, but these algorithms usually perform poorly under more realistic assumptions [18]. Several variations on the PRAM impose restrictions to make it more practical while attempting to preserve much of its simplicity. These variations address memory contention [11, 14], asynchrony [9,] latency [15], bandwidth [1], and memory hierarchy [3, 10].

An important class of models at the opposite extreme are network models,

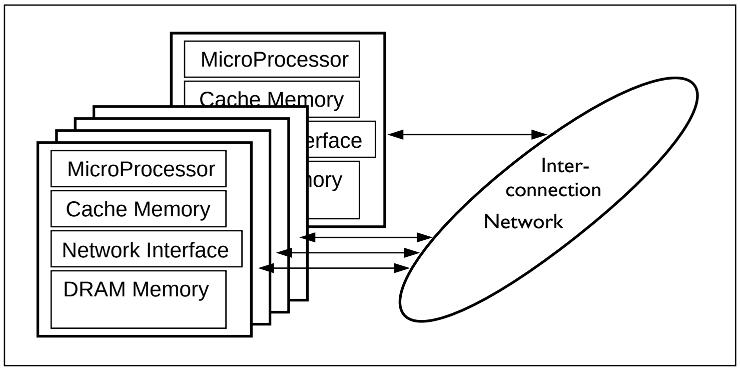

Figure 1. This organization characterizes most massively parallel processors (MPPs). Current commercial examples include the Intel Paragon, Thinking Machines CM-5, IBM SP-2, Ncube, Cray T3D, and Meiko CS-2. This structure describes essentially all of the current "research machines" as well.

in which communication is only allowed between directly connected processors. Other communication is explicitly forwarded through intermediate nodes [13]. In each step the nodes can communicate with their nearest neighbors and operate on local data. Many algorithms have been created which are perfectly matched to the structure of a particular network, including parallel prefix and non-commutative summing on a tree, physical simulations and numerical solvers for partial differential equations on a mesh, and FFT and bitonic sorting on a butterfly. However, these elegant algorithms lack robustness, as they usually do not map with equal efficiency onto interconnection structures different from those for which they were designed.

The perfect correspondence between algorithm and network usually requires a number of processors on the order of the number of data items in the input. In the more typical case where there are many data items per processor, the pattern of communication is less dependent on the network. Where problems have a local, regular communication pattern, such as stencil calculation on a grid, it is easy to arrange the data so that only a diminishing fraction of the communication is external to the processor. Basically, the interprocessor communication diminishes as the surface to volume ratio, so with large enough problem sizes, the cost of communication becomes trivial. Finally, most current networks allow messages to cut through intermediate nodes without disturbing the processor. This is much faster than explicit forwarding, and reduces the dependence of algorithm performance on the details of network topology.

The bulk-synchronous parallel model (BSP) developed by Valiant [19] attempts to bridge theory and practice with a more radical variant of the PRAM that captures key performance bottlenecks in a simple fashion by imposing restrictions on the programming model. The key idea is that the program executes as a series of *supersteps.* In a superstep each processor performs local computation and initiates a limited number of messages. Supersteps are separated by global barriers, and all messages from one superstep are received before the next begins. The superstep must be sufficiently long so that the time to route all the messages in a step is at most a fixed fraction of the duration of the step. Thus, if the algorithm designer can arrange to overlap communication with enough independent computation in each step, the communication latency and bandwidth can be ignored (up to a constant factor).

The BSP was a very encouraging starting point in our search for a parallel model that would be realistic, yet simple enough to be used to design algorithms that work predictably well over a wide range of machines. The model should allow the algorithm designer to address key performance issues without specifying unnecessary detail. It should allow machine designers to give a concise performance summary of their machine against which algorithms can be evaluated.

The other source of encouragement was the apparent architectural convergence in the field. Historically, it has been difficult to develop a reasonable abstraction of parallel machines because the machines exhibited such a diversity of structure represented by the radically different hardware organizations, including MIMD and SIMD machines, vector processors, systolic arrays, dataflow, shared memory, and message passing machines. However, technological factors have brought convergence towards systems with a familiar appearance—a collection of essentially complete computers, each consisting of a microprocessor, cache memory, and sizable DRAM memory, connected by a robust communication network (Figure 1). Variations on this structure involve localized clusters of processors and the specifics of the interface between the processor and the communication network.

The key drivers of this convergence are the phenomenal increase of microprocessor performance and memory capacity, as well as the equally astounding cost of developing these highly integrated circuits. Microprocessor performance is advancing at a rate of 50 to 100% per year, while memory capacity is quadrupling every three years. Their large development cost is borne by the extremely large market for commodity uniprocessors. To remain viable, parallel machines must ride the same technology growth curve, with the added degree of freedom being the number of processors in the system. This has led Intel, Thinking Machines, Meiko, Convex, HP, IBM and Cray Research to use off-the-shelf microprocessors or even full workstation nodes in their latest parallel machines. The technological opportunities suggest that future parallel machines are much more likely to aim at hundreds or thousands of 64-bit, off-the-shelf processors than at a million custom 1-bit processors. Thus, parallel algorithms need to be developed under the assumption of a large number of data elements per processor. This has significant impact on the kinds of algorithms that are effective in practice.

Network technology is advancing as well, but it is not driven by the same volume market forces as microprocessors and memory. Currently, communication bandwidth lags far behind internal processor memory bandwidth and the time to move data across the network is far greater than the time to move data between chips on a node. Moreover, the realizable performance is limited by the interface between the network and the node, which consumes processing cycles just getting data into and out of the network. Although network interfaces are improving, processors are improving in performance even faster, so we must assume that high latency and overhead of communication, as well as limited bandwidth, will continue to be problems.

There appears to be no consensus emerging on the interconnection topology: The networks of new commercial machines are typically different from their predecessors and different from one another. In addition, most production parallel machines can operate in the presence of network faults and allow the operating system to assign programs to collections of nodes. Thus, the physical interconnect underlying a program may vary even on a single machine. Attempting to exploit a specific network topology is likely to yield algorithms that are not very robust in practice.

The convergence of parallel architectures is reflected in our LogP model that addresses significant common issues while suppressing machine specific ones, such as network topology and routing algorithm. The model characterizes a parallel machine by a small set of parameters. In our approach, a good algorithm embodies a strategy for adapting to different machines in terms of these parameters.

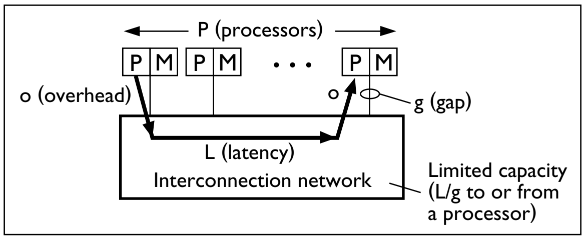

Figure 2. The LogP model describes an abstract machine configuration in terms of four performance parameters: L, the latency experienced in each communication event; o, the overhead experienced by the sending and receiving processors for each communication event; g, the gap between successive sends or successive receives by a processor; and P, the number of processor/memory modules.

LogP Model

Starting from the technological motivations previously discussed, together with programming experience and examination of popular theoretical models, we have developed a model of a distributed-memory multiprocessor in which processors communicate by point-to-point messages. The model specifies the performance characteristics of the interconnection network, but does not describe the structure of the network.

The main parameters of the model are the following (illustrated in Figure 2):

L: An upper bound on the *latency,* or delay, incurred in communicating a message containing a word (or small number of words) from its source processor/memory module to its target processor/memory module.

o: The *overhead,* defined as the length of time that a processor is engaged in the transmission or reception of each message. During this time, the processor cannot perform other operations.

g: The *gap,* defined as the minimum time interval between consecutive message transmissions or consecutive message receptions at a processor. The reciprocal of *g* corresponds to the available per-processor communication bandwidth.

P: The number of *processor*/memory modules.

The parameters *L, o,* and *g* are typically measured as multiples of the processor cycle time. The model is asynchronous, in that processors work asynchronously and the latency experienced by any message is unpredictable, but is bound above by *L* in the absence of stalls. Because of variations in latency, messages directed to a given target module may not arrive in the same order as they are sent. The basic model assumes that all messages are of a small fixed size. Furthermore, it is assumed that the network has a finite capacity, such that at most $\lceil L/g \rceil$ messages can be in transit from any processor or to any processor at any time. If a processor attempts to transmit a message that would exceed this limit, it stalls until the message can be sent without exceeding the capacity limit.

In analyzing an algorithm, the key metrics are the maximum time and the maximum amount of storage used by any processor. In order to be considered correct, an algorithm must produce correct results under all interleavings of messages consistent with the upper bound of *L* on latency. However, in estimating the running time of an algorithm, we assume that each message incurs a latency of *L*.

LogP models communication but does not attempt to model local computation. We have resisted the temptation to provide a more detailed model of the individual processors taking into account factors such as cache size or pipeline structure, and rely instead on the existing body of knowledge in implementing fast sequential algorithms on modern uniprocessor systems to fill the gap. An implementation of a good parallel algorithm on a specific machine will surely require a degree of local tuning.

There is a concern that LogP has too many parameters, which makes analysis of interesting algorithms difficult. Fortunately, the parameters are not equally important in all situations; often it is possible to ignore one or more parameters without seriously weakening the analysis. For example, in algorithms that communicate data infrequently, it is reasonable to ignore the bandwidth and capacity limits. In some algorithms, messages are sent in long streams which are pipelined through the network, so that message transmission time is dominated by the inter-message gaps, and the latency may be disregarded. In some machines the overhead dominates the gap, so g can be eliminated. One convenient approximation technique is to increase o to be as large as g, so g can be ignored. This is conservative by at most a factor of two. We hope that parallel architectures improve to a point where o can be eliminated, but today this seems premature.

Our choice of parameters represents a compromise between faithfully capturing the execution characteristics of real machines and providing a reasonable framework for algorithm design and analysis. No small set of parameters can describe all machines completely. On the other hand, analysis of interesting algorithms is difficult with a large set of parameters. We believe that LogP represents a good compromise in that additional detail would seek to capture phenomena of only modest impact, while dropping parameters would encourage algorithmic techniques that are not well supported in practice.

LogP ***encourages techniques that work well in practice, such as coordinating the assignment of work with data placement, so as to reduce the amount of communication.***

Discouraged Loopholes and Rewarded Techniques

By including L, o, and g, the model eliminates a variety of loopholes that other models permit. For example, many PRAM algorithms are excessively fine-grained, since there is no penalty for interprocessor communication. Since the PRAM model assumes that each memory cell is independently accessible, it neglects the issue of contention caused by concurrent access to different cells within the same memory module. The PRAM model also assumes, unrealistically, that the processors operate completely synchronously. There are many variations on the basic PRAM model which address one or more of these problems, namely memory contention, asynchrony, latency, and bandwidth. LogP addresses all of these issues.

Although any specific parallel computer will, of course, have a fixed number of processors, PRAM algorithms often allow the number of concurrently executing tasks to grow as a function of the size of the input. The rationale offered is that these tasks can be assigned to the physical processors, with each processor apportioning its time among the tasks assigned to it. This technique of multithreading is a convenient way of masking latency [16, 19]. Since each physical processor simulates several virtual processors, computation does not have to be suspended during the processing of a remote request by one of the virtual processors. In practice, this technique is limited by the available communication bandwidth and by the overhead involved in context switching. We do not model context switching overhead, but capture the other constraints realistically through the parameters o and g. Moreover the capacity constraint allows multithreading to be employed only up to a limit of $\lceil L/g \rceil$ virtual processors. Under LogP, multithreading represents a convenient technique which simplifies analysis as long as these constraints are met, rather than a fundamental requirement.

Most importantly, LogP encourages techniques that work well in practice, such as coordinating the assignment of work with data placement, so as to reduce the amount of communication. The model also encourages the careful scheduling of communication and overlapping of computation with communication within the limits imposed by network capacity. The limitation on network capacity also encourages balanced communication patterns in which no processor is flooded with incoming messages.

Algorithm Design

In evaluating the utility of the LogP model, we considered three different criteria:

- Do the solutions to basic theoretical problems differ in interesting ways under LogP from those under traditional models?
- For practical applications, does designing against the performance characteristics of the model lead to qualitatively good solutions?
- Is it possible to accurately predict the performance of the implementation of algorithms on real machines?

A number of studies have been conducted to answer these questions. We include a brief summary here; the interested reader is encouraged to consult the references for a more complete treatment.

The original LogP paper [6] considers the problems of optimal broadcast and summation to illustrate use of the model. The traditional solution to these problems is a simple, balanced tree. Under LogP the optimal broadcast (or summation) tree is unbalanced with the fan-out of each node determined by the relative values of *L, o,* and *g.* (Nodes that start later must have fewer children, since it takes time to communicate with each one.) Accounting for the three aspects of communication cost encourages the algorithm designer to schedule communication along with the computation. Further techniques for scheduling communication are presented in [12] for several communication problems.

The qualitative value of the model has been demonstrated on problems such as FFT, sorting, connected components and the solution of systems of linear equations. The Cooley/Tukey FFT algorithm has a butterfly communication pattern on an element-by-element basis, and is often cited as one of the reasons that machines should use an interconnection network with a topology closely related to the butterfly. However, when the number of data elements is much larger than the number of processors, the assignment of data elements to processors affects the inherent communication in the algorithm. LogP encourages the designer to work with the data layout as part of the algorithm design. Rather than using a strict blocked or cyclic data layout throughout the FFT algorithm, using a hybrid data layout reduces the amount of communication. In addition, the FFT problem illustrates the practical importance of balancing the communication among processors. Measurements on a Thinking Machines CM-5 show that a balanced communication schedule is an order of magnitude faster than naive schedules for remapping the data between layouts [6].

In designing triangular solvers, we were able to use LogP not only as a model to predict algorithm performance but also as a tool for deriving lower bounds on running time. A significant result was that standard blocked and blocked-cyclic layouts do not yield optimal parallel algorithms. Results for this problem are given in detail in [17]. Several other important problems have also been studied in detail under the LogP model. In [6] we examine LU decomposition, the core of the LINPACK benchmark.

The use of the LogP model to predict running time for algorithm implementations is illustrated in [7] for four sorting algorithms on a wide range of problem sizes and machine sizes. We found that utilizing LogP led to significant insights on how to design a more efficient sorting algorithm and provided quite accurate predictions on running time. For example, Figure 3 shows the predicted and measured times over a wide range of problem sizes and machine configurations for Histo-Radix Sort (a parallel version of radix sort [4]), on uniformly distributed keys. In order to place the data on a common scale, we divide the time by the total number of keys per processor (*N/P*) and present this normalized time in microseconds (*μs*) per key per processor. For small values of *N/P*, our measurements are only 9% higher than the prediction; for large values of *N/P*, our measurements are 37% higher. Ignoring the capacity constraint, we predict the number of processors has a negligible impact on the execution time per key, but measurements show the execution time increases slowly with the number of processors due to a slight increase in time for distribution. LogP helped identify deficiencies in the implementation which once corrected resulted in large performance improvements.

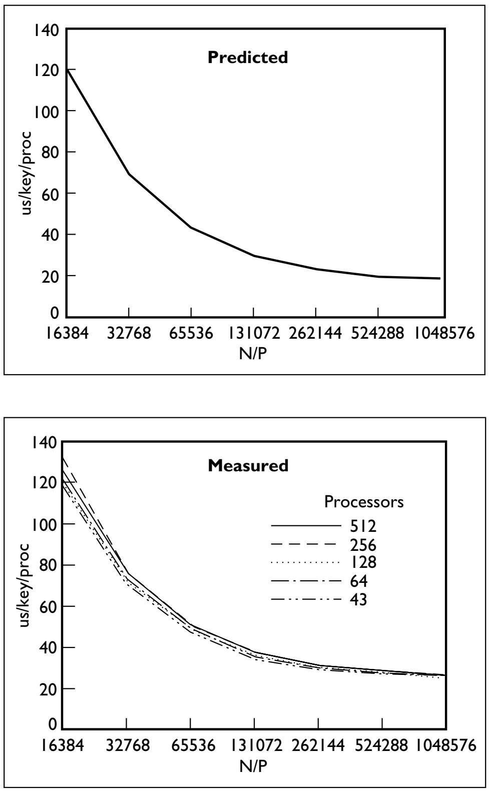

Figure 3. Predicted and measured execution time per key of radix sort on 32 to 512 processors of the CM-5 (from [4]).

Matching the Model to Real Machines

The LogP model abstracts the communication network into three parameters (illustrated in Figure 2). When the interconnection network is operating within its capacity, the time to transmit a small message will be $2o + L$: an overhead of o at the sender and the receiver, and a latency of L within the network. The available bandwidth per processor is determined by g and the network capacity, $\lceil L/g \rceil$. In essence, the network is treated as a pipeline of depth L with initiation rate g and a processor overhead of o on each end. Here, we indicate how to determine appropriate values for the LogP parameters for any given machine. We note there are other relevent factors that need to be discussed such as saturation, long messages, specialized hardware support and communication patterns. These topics are discussed in detail in [2, 6].

In a real machine, transmission of an *M*-bit long

Machine	Network	Cycle ns	w bite	$T_{snd} + T_{rcv}$ cycles	r cycles	avg. H (1024 Proc.)	L (M = 160) (1024 Proc.)
nCUBE/2	Hypercube	25	1	6400	40	5	6760
TMC CM-5	4-ary Fat-tree	25	4	3600	8	9.3	3714
IBM SP-2	Banyan	25	8	2100	5	9.3	1560
Meiko CS-2	4-ary Fat-tree	14	8	2700	20	9.3	3050
Intel Paragon	2d Mesh	7	16	4300	7-10	21	4450
Cray T3D	3d Torus	7	16	35	3	10	145
Dash	2d Torus	30	16	30	2	6.8	53
J-Machine	3d Mesh	31	8	16	2	12.1	60
Monsoon	Butterfly	20	16	10	2	5	30
nCUBE/2 (AM)	Hypercube	25	1	1000	40	5	1360
CM-5 (AM)	4-ary Fat-tree	25	4	132	8	9.3	246
Meiko CS-2 (AM)	4-ary Fat-tree	14	8	230	20	9.3	570
Intel Paragon (AM)	2d Mesh	7	16	540	7-10	21	750

Table 1. Network timing parameters for a one-way message without contention on several current commercial and research multiprocessors. The final rows refer to the active message layer, which uses the commercial hardware, but reduces the interface overhead.

message in an unloaded or lightly loaded network has four parts. First, the *send overhead* is the time the processor is busy interfacing to the network before the first bit of data is placed onto the network. The message is transmitted into the network channel a few bits at a time, determined by the channel width w. Thus, the time to get the last bit of an M-bit message into the network is $\lceil M/w \rceil$ cycles. Most modern parallel machines employ some form of *cut-through routing*, so the time for the last bit to cross the network to the destination node is Hr, where H is the distance of the route (number of hops) and r is the delay through each intermediate routing node, independent of the message size. Finally, the *receive overhead* is the time from the arrival of the last bit until the receiving processor can do something useful with the message. In summary, the total message communication time for an M-bit message across H hops is given by the following.

$$T(M, H) = T_{snd} + \lceil \tfrac{M}{w} \rceil + Hr + T_{rcv}$$

Table 1 lists values for various parallel machines. The send and receive overheads for conventional message passing libraries include a considerable amount of computational work for protocol processing and buffer management. The overheads with Active Message (AM) libraries at the bottom of the table give a better indication of the hardware capability.

In determining LogP parameters for a given machine, it appears reasonable to choose $o = \frac{T_{snd} + T_{rcv}}{2}$ and $L = Hr + \lceil \frac{M}{w} \rceil$, where H is the maximum distance of a route and M is the fixed message size being used. The gap g is at least M divided by the per processor bisection bandwidth. However, in many designs the limiting factor is the processing rate in the network interface hardware itself. A methodology for determining the LogP parameters empirically for Active Messages is developed in [5]. Analogous techniques could be developed for message passing and shared memory programming models.

Summary

Our search for a machine-independent model for parallel computation is motivated by technological trends which are driving the high-end computer industry toward massively parallel machines constructed from nodes containing powerful processors and substantial memory, interconnected by networks with limited bandwidth and significant latency.

The LogP model attempts to capture the important bottlenecks of such parallel computers with a small number of parameters: The latency (L), overhead (o), bandwidth (g) of communication, and the number of processors (P). We believe the model is sufficiently detailed to reflect the major practical issues in parallel algorithm design, yet simple enough to support detailed algorithmic analysis. At the same time, the model avoids specifying the programming style or the communication protocol, being equally applicable to shared-memory, message passing, and data parallel paradigms.

As with any new proposal, there will naturally be concerns regarding its utility as a basis for further study. From our discussions and from indepth exploration of several parallel processing applications on LogP, we have observed the following:

1. Algorithms may adapt their computation and communication structures in response to each of the parameters of the model.
2. In specific situations some parameters become insignificant and one can work with a simplified model.
3. Adjusting data placement and scheduling communication are important techniques for improving algorithms.
4. LogP can be used not only for algorithm design and analysis but also as a model to determine lower bounds on parallel running time.
5. The LogP model is extremely valuable in guiding algorithm design. Discrepancies between predicted and measured execution time often highlighted deficiencies in implementation which violated constraints specified by the model. Correcting these resulted in a large performance improvement and led to a close match between predicted and measured execution time.

We believe the LogP model opens several avenues of research. It potentially provides a concise summary of the performance characteristics of current and future machines. This will require refining the process of parameter determination and evaluating a large number of machines. Such a summary can focus the efforts of machine designers toward architectural improvements that can be measured in terms of these parameters. For example, a machine with large gap *g* is only effective for algorithms with a large ratio of computation to communication. In effect, the model defines a four dimensional parameter space of potential machines. The product line offered by a particular vendor may be identified with a curve in this space, characterizing the system scalability. It will be important to evaluate the complexity of a wide variety of algorithms in terms of the model and to evaluate the predictive capabilities of the model. The model provides a new framework for classifying algorithms and identifying which are most attractive in various regions of the machine parameter space. We hope this will stimulate the development of new parallel algorithms and the examination of the fundamental requirements of various problems within the LogP framework.

Acknowledgments
Several people provided helpful comments on earlier drafts of this article, including Larry Carter, Dave Douglas, Jeanne Ferrante, Seth Copen Goldstein, Anoop Gupta, John Hennessy, Tom Leighton, Charles Leiserson, Lesley Matheson, Rishiyur Nikhil, Abhiram Ranade, Luigi Semenzato, Larry Snyder, Burton Smith, Guy Steele, Robert Targan, Leslie Valiant, and the anonymous reviewers. Special thanks to Andrea Dusseau for the implementation and analysis of the parallel radix sort algorithm.

References

1. Aggarwal, A., Chandra, A. K., and Snir, M. Communication complexity of PRAMs. In *Theoretical Computer Science* (March 1990).
2. Alexander, A., Ionescu, M., Schauser, K., and Scheiman, C. LogP: Incorporating long messages into the LogP model. In *The 7th Annual Symposium on Parallel Algorithms and Architectures.* July 1995.
3. Alpern, B., Carter, L., Feig, E., and Selker, T. The uniform memory hierarchy model of computation. *Algorithmica* (1993).
4. Blelloch, G. E., Leiserson, C. E., Maggs, B. M., Plaxton, C. G., Smith, S. J., and Zagha, M. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the Symposium on Parallel Architectures and Algorithms.* 1990.
5. Culler, D., Liu, L. T., Martin, R., and Yoshikawa, C. Assessing fast network interfaces. *IEEE Micro 16*, 1 (Feb. 1996).
6. Culler, D. E., Karp, R. M., Patterson, D. A., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. LogP: Towards a realistic model of parallel computation In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* May 1993.
7. Dusseau, A., Culler, D., Schauser, K., and Martin, R. Fast parallel sorting under LogP: Experiences with CM-5. *IEEE Trans. Parallel and Distrib. Syst. 7,* 8. (1996).
8. Fortune, S., and Wyllie, J. Parallelism in random access machines. In *Proceedings of the 10th Annual Symposium on Theory of Computing.* 1978.
9. Gibbons, P. B. A more practical PRAM model. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures.* ACM, 1989.
10. Heywood, T. and Ranka, S. A practical hierarchical model of parallel computation. *J. Parallel and Distrib. Comput. 16*, 3 (Nov. 1992).
11. Karp, R. M., Luby, M. and Meyer auf der Heide, F. Efficient PRAM simulation on a distributed memory machine. In *Proceedings of the 24th Annual ACM Symposium of the Theory of Computing* (May 1992).
12. Karp, R. M., Sahay, A., Santos, E. and Schauser, K. E. Optimal broadcast and summation in the LogP model. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures.* ACM, 1993.
13. Leighton, F. T. *Introduction to Parallel Algorithms and Architectures: Arrays-Trees-Hypercubes.* Morgan Kaufmann, New York, 1992.
14. Mehlhorn, K. and Vishkin, U. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica 21.* (1984).
15. Papadimitriou, C. H. and Yannakakis, M. Towards an architecture-independent analysis of parallel algorithms. In *Proceedings of the 20th Annual ACM Symposium of the Theory of Computing.* ACM, 1988.
16. Ranade, A. G. How to emulate shared memory. In *Proceedings of the 28th IEEE Annual Symposium on Foundations of Computer Science.* 1987.
17. Santos, E. E. Solving triangular linear systems in parallel using substitution. In *Proceedings of the 7th Annual IEEE Symposium on Parallel and Distributed Processing.* 1995.
18. Snyder, L. Type architectures, shared memory, and the corollary of modest potential. In *Ann. Rev. Comput. Sci.* Annual Reviews Inc., 1986, pp 289–317.
19. Valiant, L. G. A bridging model for parallel computation. *Commun. ACM 33*, 8 (Aug. 1990).

DAVID CULLER (culler@cs.berkeley.edu) is an associate professor of computer science and engineering at the University of California, Berkeley.

RICHARD M. KARP (karp@cs. washington.edu) is Professor of computer science and engineering and Adjunct Professor of Molecular Biotechnology at the University of Washington in Seattle.

DAVID PATTERSON (pattrsn@cs.berkeley.edu) is Professor of computer science and engineering at the University of California, Berkeley.

ABHIJIT SAHAY (sahay@sphinx.com) is Systems R&D Engineer at Iris Financial Engineering and Systems, Inc.

EUNICE E. SANTOS (santos@eecs.lehigh.edu) is an assistant professor in the electrical engineering and computer science department at Lehigh University.

KLAUS ERIK SCHAUSER (schauser@cs.ucsb.edu) is an assistant professor at the computer science department at the University of California, Sanat Barbara.

RAMESH SUBRAMONIAN (subramon@dipl.rdd. lmsc.lockheed.com) works for the Lockheed Corp.

THORSTEN von EICKEN (tve@cs. cornell.edu) is an assistant professor in the computer science department of Cornell University.

Computational support was provided by the NSF Infrastructure Grant number CDA-8722788.