LogP: Towards a Realistic Model of Parallel Computation *

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

Computer Science Division,
University of California, Berkeley

Abstract

A vast body of theoretical research has focused either on overly simplistic models of parallel computation, notably the PRAM, or overly specific models that have few representatives in the real world. Both kinds of models encourage exploitation of formal loopholes, rather than rewarding development of techniques that yield performance across a range of current and future parallel machines. This paper offers a new parallel machine model, called LogP, that reflects the critical technology trends underlying parallel computers. It is intended to serve as a basis for developing fast, portable parallel algorithms and to offer guidelines to machine designers. Such a model must strike a balance between detail and simplicity in order to reveal important bottlenecks without making analysis of interesting problems intractable. The model is based on four parameters that specify abstractly the computing bandwidth, the communication bandwidth, the communication delay, and the efficiency of coupling communication and computation. Portable parallel algorithms typically adapt to the machine configuration, in terms of these parameters. The utility of the model is demonstrated through examples that are implemented on the CM-5.

Keywords: massively parallel processors, parallel models, complexity analysis, parallel algorithms, PRAM

1 Introduction

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, *i.e.*, 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 can be classified as impractical in that they exploit artificial factors not present in any reasonable machine, such as zero communication delay or infinite bandwidth. Others can be classified as overly specialized, in that they are tailored to the idiosyncrasies of a single machine, such as a particular interconnect topology. The most widely used parallel model, the PRAM[13], is unrealistic because it assumes that all processors work synchronously and that interprocessor communication is free. Surprisingly fast algorithms can be developed by exploiting these loopholes, but in many cases the algorithms perform poorly under more realistic assumptions[30]. Several variations on the PRAM have attempted to identify restrictions that would make it more practical while preserving much of its simplicity [1, 2, 14, 19, 24, 25]. The bulk-synchronous parallel model (BSP) developed by Valiant[32] attempts to bridge theory and practice

* A version of this report appears in the Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, May 1993, San Diego, CA.

† Also affiliated with International Computer Science Institute, Berkeley.

with a more radical departure from the PRAM. It allows processors to work asynchronously and models latency and limited bandwidth, yet requires few machine parameters as long as a certain programming methodology is followed. We used the BSP as a 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.

Historically, it has been difficult to develop a reasonable abstraction of parallel machines because the machines exhibited such a diversity of structure. However, technological factors are now forcing a convergence towards systems formed by a collection of essentially complete computers connected by a communication network (Figure 1). This convergence is reflected in our LogP model which addresses significant common issues while suppressing machine specific ones such as network topology and routing algorithm. The LogP model characterizes a parallel machine by the number of processors(P), the communication bandwidth(g), the communication delay(L), and the communication overhead(o). In our approach, a good algorithm embodies a strategy for adapting to different machines, in terms of these parameters.

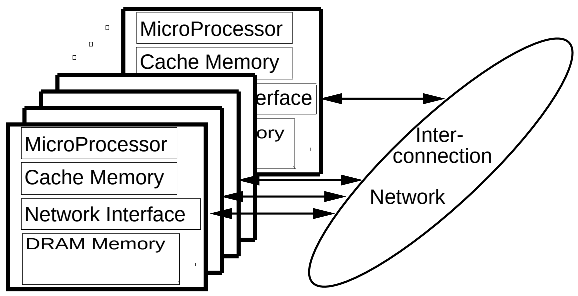

Figure 1: *This organization characterizes most massively parallel processors (MPPs). Current commercial examples include the Intel iPSC, Delta and Paragon, Thinking Machines CM-5, Ncube, Cray T3D, and Transputer-based MPPs such as the Meiko Computing Surface or the Parsytec GC. This structure describes essentially all of the current "research machines" as well.*

We believe that the common hardware organization described in Figure 1 will dominate commercial MPPs at least for the rest of this decade, for reasons discussed in Section 2 of this paper. In Section 3 we develop the LogP model, which captures the important characteristics of this organization. Section 4 puts the model to work, discussing the process of algorithm design in the context of the model and presenting examples that show the importance of the various communication aspects. Implementation of these algorithms on the CM-5 provides preliminary data towards validating the model. Section 5 presents communication networks in more detail and examines how closely our model corresponds to reality on current machines. Finally, Section 6 compares our model to various existing parallel models, and summarizes why the parameters making up our model are necessary. It also addresses several concerns that might arise regarding the utility of this model as a basis for further study.

2 Technological Motivations

The possibility of achieving revolutionary levels of performance has led parallel machine designers to explore a variety of exotic machine structures and implementation technologies over the past thirty years. Generally, these machines have performed certain operations very well and others very poorly, frustrating attempts to formulate a simple abstract model of their performance characteristics. However, technological

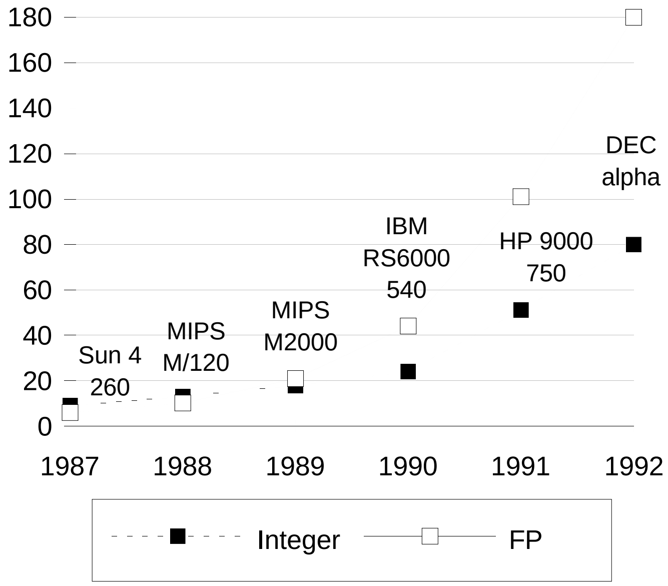

Figure 2: *Performance of state-of-the-art microprocessors over time. Performance is approximately number of times faster than the VAX-11/780. The floating point SPEC benchmarks improved at about 97% per year since 1987, and integer SPEC benchmarks improved at about 54% per year.*

factors are forcing a 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. This convergence is likely to accelerate in the future as physically small computers dominate more of the computing market. Variations on this structure will involve clustering of localized collections of processors and the details of the interface between the processor and the communication network. The key technological justifications for this outlook are discussed below.

Microprocessor performance is advancing at a rate of 50 to 100% per year[17], as indicated by Figure 2. This tremendous evolution comes at an equally astounding cost: estimates of the cost of developing the recent MIPS R4000 are 30 engineers for three years, requiring about $30 million to develop the chip, another $10 million to fabricate it, and one million hours of computer time for simulations[15]. This cost is borne by the extremely large market for commodity uniprocessors. To remain viable, parallel machines must be on the same technology growth curve, with the added degree of freedom being the number of processors in the system. The effort needed to reach such high levels of performance combined with the relatively low cost of purchasing such microprocessors led Intel, Thinking Machines, Meiko, Convex, IBM and even Cray Research to use off-the-shelf microprocessors in their new parallel machines[5]. The technological opportunities suggest that parallel machines in the 1990s and beyond are much more likely to aim at thousands of 64-bit, off-the-shelf processors than at a million custom 1-bit processors.

Memory capacity is increasing at a rate comparable to the increase in capacity of DRAM chips: quadrupling in size every three years[16]. Today's personal computers typically use 8 MB of memory and workstations use about 32 MB. By the turn of the century the same number of DRAM chips will offer 64 times the capacity of current machines. The access time falls very slowly with each generation of DRAMs, so sophisticated cache structures will be required in commodity uniprocessors to bridge the difference between processor cycle times and memory access times. Cache-like structures may be incorporated into the memory chips themselves, as in emerging RAM-bus and synchronous DRAM technology[17]. Multiprocessors will need to incorporate state-of-the-art memory systems to remain competitive.

Since the parallel machine nodes are very similar to the core of a workstation, the cost of a node is comparable to the cost of a workstation. As the most expensive supercomputer costs less than 25 M$ for the processors and memory, and since the price of workstations have remained at about 5-10 K$, the largest parallel machines will have a few thousand nodes. This economic observation is valid today, with no vendor producing a system with more than two thousand nodes.[1]

Summarizing, we can expect that the nodes of parallel machines of the 1990s will be capable of computing hundreds of Mflops and capable of storing hundreds of megabytes. The number of such nodes will not scale into the millions, *so parallel algorithms will 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. While new media offer much higher network bandwidth, their realizable performance is limited by the interface between the network and the node. Currently, communication bandwidth through that interface lags far behind internal processor memory bandwidth. The lack of attention paid to the network interface in current microprocessors also means that substantial time is lost on each communication operation, regardless of programming style. Although the interface is 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 interconnection topology: the networks of new commercial machines are typically different from their predecessors and different from each other. Operating in the presence of network faults is becoming extremely important as parallel machines go into production use, which suggests that the physical interconnect on a single system will vary over time to avoid broken components. Finally, adaptive routing techniques are becoming increasingly practical. Thus, attempting to exploit a specific network topology is likely to yield algorithms that are not very robust in practice. An abstract view of the latency and bandwidth properties of the network provides a framework for adapting algorithms to the target machine configuration.

No single programming methodology is becoming clearly dominant: shared-memory, message-passing, and data parallel styles all have significant popularity. Thus, the computational model should apply regardless of programming style. The technological factors discussed above make this goal tractable as most recent parallel machines support a range of programming styles using roughly similar hardware mechanisms[33].

The essential message is clear: technological forces are leading to massively parallel machines constructed from at most a few thousand nodes, each containing a powerful processor and substantial memory, interconnected by networks with limited bandwidth and significant latency. This renders both PRAM and network models inappropriate as a foundation for algorithm development since they do not accurately predict performance of programs on real computers. Our conclusion is that a new model which captures the technological reality more faithfully is needed.

3 LogP Model

Starting from the technological motivations discussed in the previous section, 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:

[1] Mainstream workstations may contain multiple processors in the future, perhaps on a single chip. Current trends would indicate that large parallel machines would comprise a few thousand of these multiprocessor nodes.

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 module to its target 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. We assume unit time for local operations and call it a cycle.

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.

The parameters L, o and g are measured as multiples of the processor cycle. The model is *asynchronous*, *i.e.*, processors work asynchronously and the latency experienced by any message is unpredictable, but is bounded above by L in the absence of stalls. Because of variations in latency, the 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 size* (a simple extension deals with longer messages).

In analyzing an algorithm, the key metrics are the maximum time and the maximum space 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.[2]

3.1 Discussion of parameters

This particular 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 "the knee of the curve" in that additional detail would seek to capture phenomena of modest impact while dropping parameters would encourage algorithmic techniques that are not well supported in practice.

We have resisted the temptation to provide a more detailed model of the individual processors, such as cache size, and rely 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 tuning, but if the issues raised by the level of detail embodied in LogP are not addressed, it would seem that the algorithm design is incomplete.

Fortunately, the parameters are not equally important in all situations; often it is possible to ignore one or more parameters and work with a simpler model. 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

[2]There are certain anomalous situations in which reducing the latency of certain messages actually increases the running time of an algorithm. These arise primarily when the computational schedule is based on the order of message arrival, rather than the information contained in the message.

architectures improve to a point where o can be eliminated, but today this seems premature. More specific rationale for the particular parameters and their role is provided in the remainder of the paper.

3.2 Discouraged loopholes and rewarded techniques

The LogP 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. Although the EREW PRAM penalizes data access contention at the word level, it does not penalize contention at the module level.

The technique of multithreading is often suggested as a way of masking latency. This technique assigns to each physical processor the task of simulating several virtual processors; thus, 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 L/g virtual processors. Under LogP, multithreading represents a convenient technique which simplifies analysis, as long as these constraints are met, rather than a fundamental requirement[27, 32].

On the other hand, LogP encourages techniques that work well in practice, such as coordinating the assignment of work with data placement, so as to reduce the communication bandwidth requirement and the frequency of remote references. The model also encourages the careful scheduling of computation 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.

Although the model is stated in terms of primitive message events, we do not assume that algorithms must be described in terms of explicit message passing operations, such as send and receive. Shared memory models are implemented on distributed memory machines through an implicit exchange of messages[22]. Under LogP, reading a remote location requires time $2L + 4o$. Prefetch operations, which initiate a read and continue, can be issued every g cycles and cost $2o$ units of processing time. Some recent machines migrate locations to local caches when they are referenced; this would be addressed in algorithm analysis by adjusting which references are remote.

3.3 Broadcast and Summation

As a concrete illustration of the role of various parameters of the model, we sketch optimal algorithms for two simple problems: broadcast and summation. The solutions are quite different from those on the PRAM.

First, we consider the problem of broadcasting a single datum from one processor to $P - 1$ others. The main idea is simple: all processors that have received the datum transmit it as quickly as possible, while ensuring that no processor receives more than one message. The source of the broadcast begins transmitting the datum at time 0. The first datum enters the network at time o, takes L cycles to arrive at the destination, and is received by the node at time $L + 2o$. Meanwhile, the source will have initiated transmission to other processors at time $g, 2g, \ldots$, assuming $g \geq o$, each of which acts as the root of a smaller broadcast tree. As indicated in Figure 3, the optimal broadcast tree for p processors is unbalanced[3] with the fan-out at each node determined by the relative values of L, o, and g. Observe that the processor overhead of successive transmissions overlaps the delivery of previous messages. Nodes may experience idle cycles at the end of the algorithm while the last few messages are in transit.

To obtain an optimal algorithm for the summation of n input values we first consider how to sum as many values as possible within a fixed amount of time T. This produces the communication and computation

[3] A special case of this algorithm with $o = 0$ and $g = 1$ appears in [4].

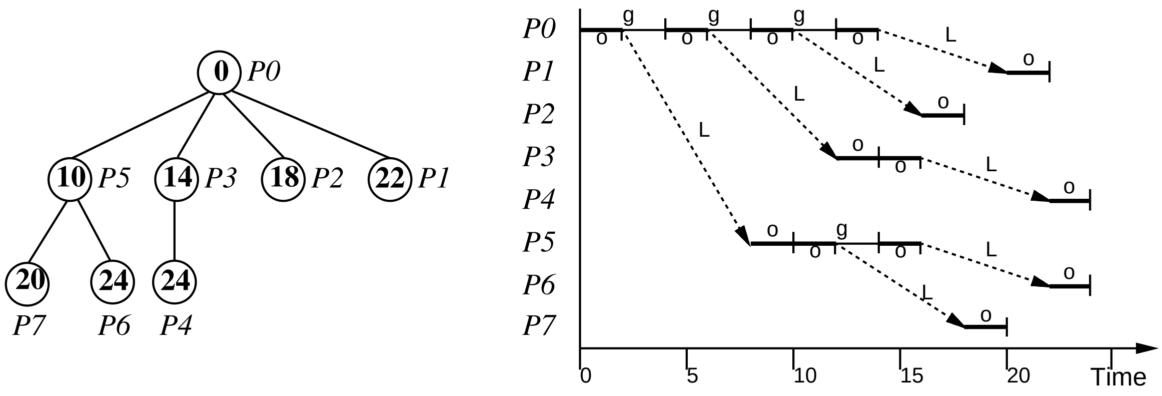

Figure 3: *Optimal broadcast tree for $P = 8, L = 6, g = 4, o = 2$ (left) and the activity of each processor over time (right). The number shown for each node is the time at which it has received the datum and can begin sending it on. The last value is received at time 24.*

schedule for the summation problem. The pattern of communication among the processors again forms a tree; in fact, the tree has the same shape as an optimal broadcast tree[20]. Each processor has the task of summing a set of the elements and then (except for the root processor) transmitting the result to its parent. The elements to be summed by a processor consist of original inputs stored in its memory, together with partial results received from its children in the communication tree. To specify the algorithm, we first determine the optimal schedule of communication events and then determine the distribution of the initial inputs.

If $T \leq L+2o$, the optimal solution is to sum $T+1$ values on a single processor, since there is not sufficient time to receive data from another processor. Otherwise, the last step performed by the root processor (at time $T-1$) is to add a value it has computed locally to a value it just received from another processor. The remote processor must have sent the value at time $T-1-L-2o$, and we assume recursively that it forms the root of an optimal summation tree with this time bound. The local value must have been produced at time $T-1-o$. Since the root can receive a message every g cycles, its children in the communication tree should complete their summations at times $T-(2o+L+1), T-(2o+L+1+g), T-(2o+L+1+2g), \ldots$. The root performs $g-o-1$ additions of local input values between messages, as well as the local additions before it receives its first message. This communication schedule must be modified by the following consideration: since a processor invests o cycles in receiving a partial sum from a child, all transmitted partial sums must represent at least o additions. Based on this schedule, it is straight-forward to determine the set of input values initially assigned to each processor and the computation schedule. Notice that the inputs are not equally distributed over processors. (The algorithm is easily extended to handle the limitation of p processors by pruning the communication tree.)

The computation schedule for our summation algorithm can also be represented as a tree with a node for each computation step. Figure 4 shows the communication schedule for the processors and the computational schedule for a processor and two of its children. Each node is labeled with the time at which the step completes, the wavy edges represent partial results transmitted between processors, and the square boxes represent original inputs. The initial work for each processor is represented by a linear chain of input-summing nodes. Unless the processor is a leaf of the communication tree, it then repeatedly receives a value, adds it to its partial sum and performs a chain of $g-o-1$ input-summing nodes. Observe that local computations overlap the delivery of incoming messages and the processor reception overhead begins as soon as the message arrives.

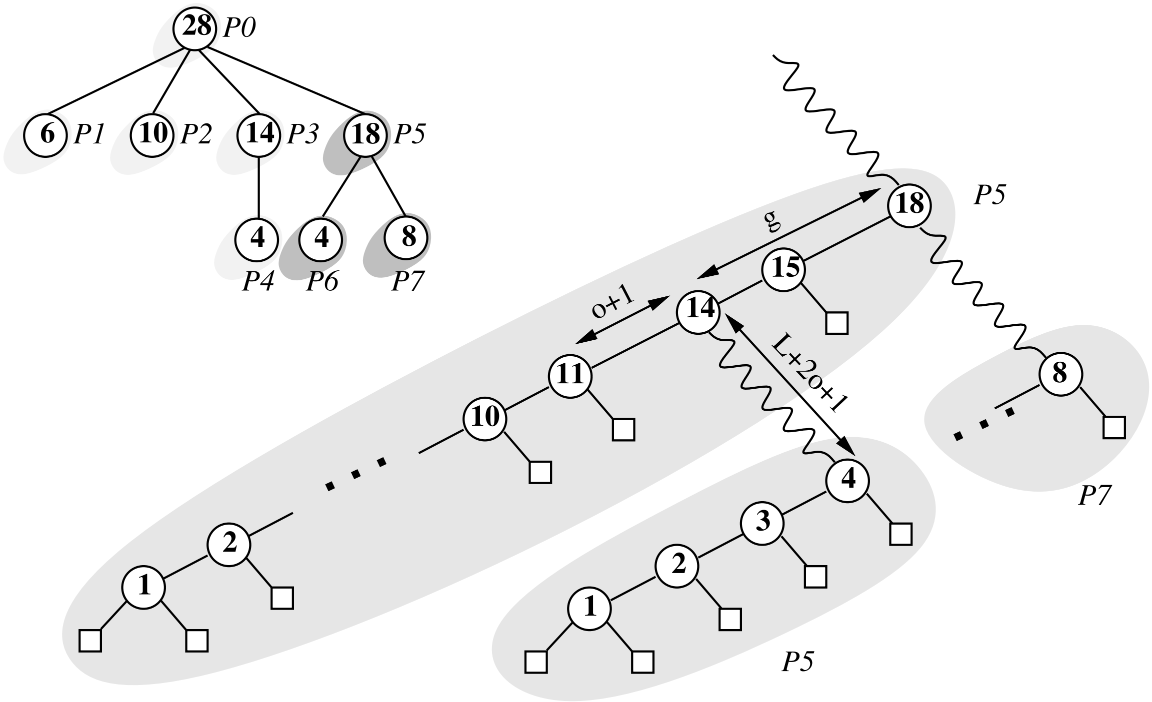

Figure 4: *Communication tree for optimal summing (left) and computation schedule for a subset of processors (right) for* $T = 28, P = 8, L = 5, g = 4, o = 2$.

4 Algorithm Design

In the previous section, we stepped through the design of optimal algorithms for extremely simple problems and explained the parameters of our model. We now consider more typical parallel processing applications and show how the use of the LogP model leads to efficient parallel algorithms in practice. In particular, we observe that efficient parallel algorithms must pay attention to both computational aspects (such as the total amount of work done and load balance across processors) and communication aspects (such as remote reference frequency and the communication schedule). Thus, a good algorithm should co-ordinate work assignment with data placement, provide a balanced communication schedule, and overlap communication with processing.

4.1 Fast Fourier Transform

Our first example, the fast Fourier transform, illustrates these ideas in a concrete setting. We discuss the key aspects of the algorithm and then an implementation that achieves near peak performance on the Thinking Machines CM-5. We focus on the "butterfly" algorithm [9] for the discrete FFT problem, most easily described in terms of its computation graph. The n-input (n a power of 2) butterfly is a directed acyclic graph with $n(\log n + 1)$ nodes viewed as n rows of $(\log n + 1)$ columns each. For $0 \le r < n$ and $0 \le c < \log n$, the node (r, c) has directed edges to nodes $(r, c+1)$ and $(\bar{r}_c, c+1)$ where $\bar{r}_c$ is obtained by complementing the $(c+1)$-th most significant bit in the binary representation of r. Figure 5 shows an 8-input butterfly.

The nodes in column 0 are the problem inputs and those in column $\log n$ represent the outputs of the computation. (The outputs are in bit-reverse order, so for some applications an additional rearrangement

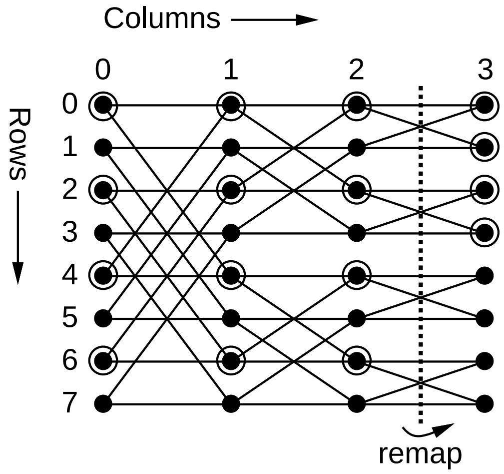

Figure 5: *An 8-input butterfly with $P = 2$. Nodes assigned to processor 0 under the hybrid layout are circled.*

step is required.) Each non-input node represents a complex operation, which we assume takes one unit of time. Implementing the algorithm on a parallel computer corresponds to laying out the nodes of the butterfly on its processors; the layout determines the computational and communication schedules, much as in the simple examples above.

4.1.1 Data placement and work assignment

There is a vast body of work on this structure as an interconnection topology, as well as on efficient embeddings of the butterfly on hypercubes, shuffle-exchange networks, etc. This has led many researchers to feel that algorithms must be designed to match the interconnection topology of the target machine. In real machines, however, the n data inputs and the $n \log n$ computation nodes must be laid out across P processors and typically $P << n$. The nature of this layout, and the fact that each processor holds many data elements has a profound effect on the communication structure, as shown below.

A natural layout is to assign the first row of the butterfly to the first processor, the second row to the second processor and so on. We refer to this as the *cyclic* layout. Under this layout, the first $\log \frac{n}{P}$ columns of computation require only local data, whereas the last $\log P$ columns require a remote reference for each node. An alternative layout is to place the first $\frac{n}{P}$ rows on the first processor, the next $\frac{n}{P}$ rows on the second processor, and so on. With this *blocked* layout, each of the nodes in the first $\log P$ columns requires a remote datum for its computation, while the last $\log \frac{n}{P}$ columns require only local data. Under either layout, each processor spends $\frac{n}{P} \log n$ time computing and $(g\frac{n}{P} + L) \log P$ time communicating, assuming $g \geq 2o$.

Since the initial computation of the cyclic layout and the final computation of the blocked layout are completely local, one is led to consider *hybrid* layouts that are cyclic on the first $\log P$ columns and blocked on the last $\log P$. Indeed, switching from cyclic to blocked layout at any column between the $\log P$-th and the $\log \frac{n}{P}$-th (assuming that $n > P^2$) leads to an algorithm which has a single "all-to-all" communication step between two entirely local computation phases. Figure 5 highlights the node assignment for processor 0 for an 8-input FFT with $P = 2$ under the hybrid layout; remapping occurs between columns 2 and 3.

The computational time for the hybrid layout is the same as that for the simpler layouts, but the communication time is lower by a factor of $\log P$: each processor sends $\frac{n}{P^2}$ messages to every other, requiring only $g(\frac{n}{P} - \frac{n}{P^2}) + L$ time. The total time is within a factor of $(1 + \frac{g}{\log n})$ of optimal, showing that this layout has the potential for near-perfect speedup on large problem instances.

4.1.2 Communication schedule

The algorithm presented so far is incomplete because it does not specify the *communication schedule* (the order in which messages are sent and received) that achieves the stated time bound. Our algorithm is a special case of the "layered" FFT algorithm proposed in [25] and adapted for the BSP model[32]. These earlier models do not emphasize the communication schedule: [25] has no bandwidth limitations and hence no contention, whereas [32] places the scheduling burden on the router which is assumed to be capable of routing any balanced pattern in the desired amount of time.

A naive schedule would have each processor send data starting with its first row and ending with its last row. Notice, that all processors first send data to processor 0, then all to processor 1, and so on. All but L/g processors will stall on the first send and then one will send to processor 0 every g cycles. A better schedule is obtained by staggering the starting rows such that no contention occurs: processor i starts with its $\frac{in}{P^2}$-th row, proceeds to the last row, and wraps around.

4.1.3 Implementation of the FFT algorithm

To verify the prediction of the analysis, we implemented the hybrid algorithm on a CM-5 multiprocessor and measured the performance of the three phases of the algorithm: (I) computation with cyclic layout, (II) data remapping, and (III) computation with blocked layout. The CM-5 is a massively parallel MIMD computer based on the Sparc processor. Each node consists of a 33 Mhz Sparc RISC processor chip-set (including FPU, MMU and 64 KByte direct-mapped write-through cache), 8 MBytes of local DRAM memory and a network interface. The nodes are interconnected in two identical disjoint incomplete fat trees, and a broadcast/scan/prefix control network.[4] Figure 6 demonstrates the importance of the communication schedule: the three curves show the computation time and the communication times for the two communication schedules. With the naive schedule, the remap takes more than 1.5 times as long as the computation, whereas with staggering it takes only $\frac{1}{7}$th as long.

The two computation phases involve purely local operations and are standard FFTs. Figure 7 shows the computation rate over a range of FFT sizes expressed in Mflops/processor. For comparison, a CM-5's Sparc node achieves roughly 3.2 MFLOPS on the Linpack benchmark. This example provides a convenient comparison of the relative importance of cache effects, which we have chosen to ignore, and communication balance, which other models ignore. The drop in performance for the local FFT from 2.8 Mflops to 2.2 Mflops occurs when the size of the local FFTs exceeds cache capacity. (For large FFTs, the cyclic phase involving one large FFT suffers more cache interference than the blocked phase which solves many small FFTs.) The implementation could be refined to reduce the cache effects, but the improvement would be small compared to the speedup associated with improving the communication schedule.

4.1.4 Quantitative analysis

The discussion so far suggests how the model may be used in a qualitative sense to guide parallel algorithm design. The following shows how the model can be used in a more quantitative manner to predict the execution time of an implementation of the algorithm. From the computational performance in Figure 7 we can calibrate the "cycle time" for the FFT as the time for the set of complex multiply-adds of the butterfly primitive. At an average of 2.2 Mflops and 10 floating-point operations per butterfly, a cycle corresponds to $4.5\mu s$, or 150 clock ticks (we use *cycles* to refer to the time unit in the model and *ticks* to refer to the 33 Mhz hardware clock). In previous experiments on the CM-5[33] we have determined that $o \approx 2\mu s$ (0.44 cycles, 56 ticks) and, on an unloaded network, $L \approx 6\mu s$ (1.3 cycles, 200 ticks). Furthermore, the

[4] The implementation does not use the vector accelerators which are not available at the time of writing.

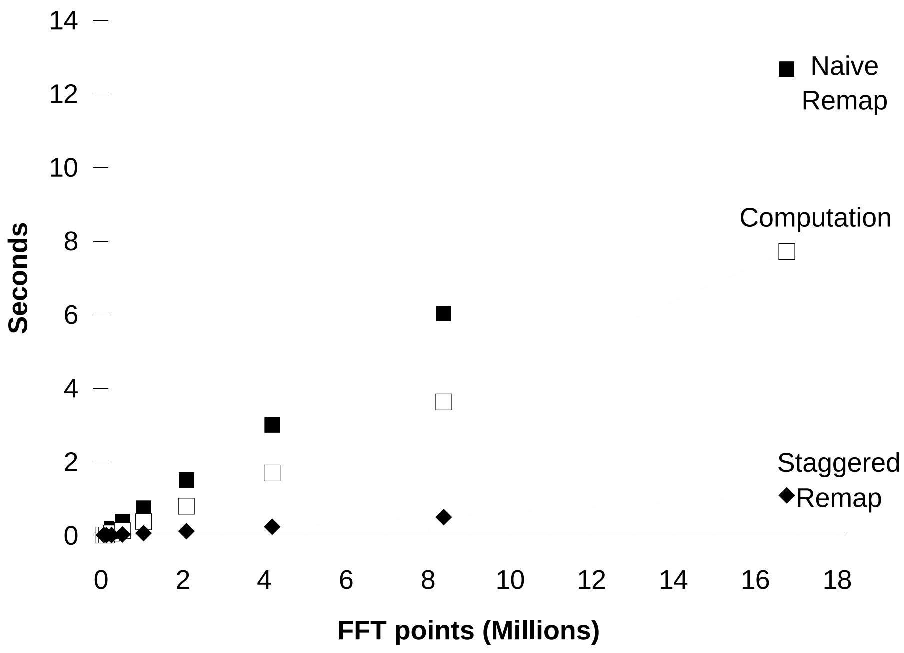

Figure 6: *Execution times for FFTs of various sizes on a 128 processor CM-5. The* compute *curve represents the time spent computing locally. The* bad remap *curve shows the time spent remapping the data from a cyclic layout to a blocked layout if a naive communication schedule is used. The* good remap *curve shows the time for the same remapping, but with a contention-free communication schedule, which is an order of magnitude faster. The X axis scale refers to the entire FFT size.*

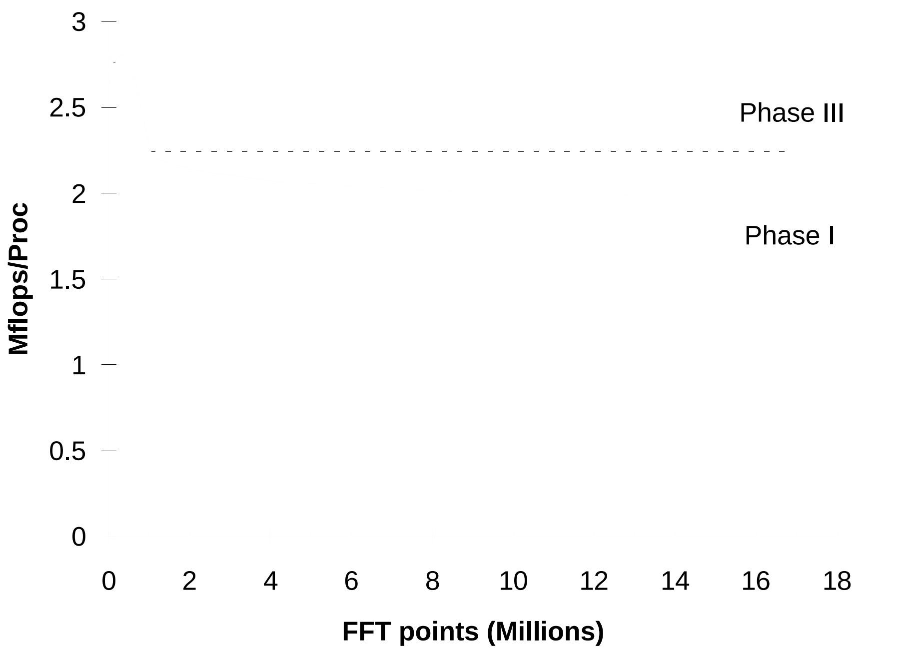

Figure 7: *Per processor computation rates for the two computation phases of the FFT in Mflops (millions of floating-point operations per second).*

bisection bandwidth[5] is 5MB/s per processor for messages of 16 bytes of data and 4 bytes of address, so we take g to be $4\mu s$(0.44 cycles, 56 ticks). In addition there is roughly $1\mu s(0.22\ cycles, 28\ ticks)$ of local computation per data point to load/store values to/from memory. Analysis of the staggered remap phase predicts the communication time is $\frac{n}{P}\max(1\mu s + 2o, g) + L$. For these parameter values, the transmission rate is limited by processing time and communication overhead, rather than bandwidth. The remap phase is predicted to increase rapidly to an asymptotic rate of 3.2MB/s. The observed performance is roughly 2MB/s for this phase, nearly half of the available network bandwidth.

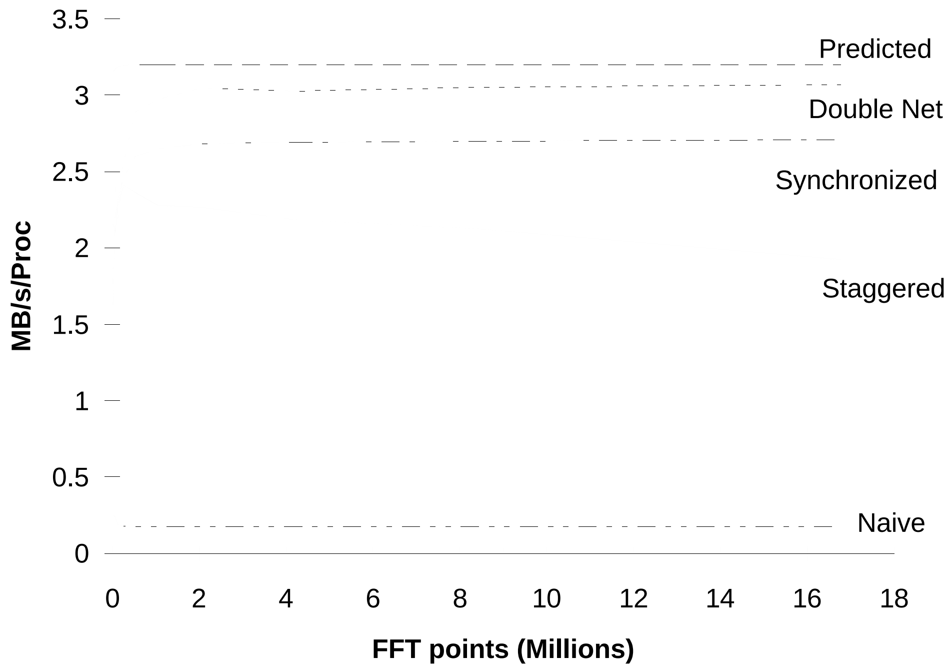

Figure 8: *Predicted and measured communication rates expressed in Mbytes/second per processor for the staggered communication schedule. The staggered schedule is theoretically contention-free, but the asynchronous execution of the processors causes some contention in practice. The* synchronized *schedule performs a barrier synchronization periodically (using a special hardware barrier). The* double net *schedule uses both data networks, doubling the available network bandwidth.*

The analysis does not predict the gradual performance drop for large FFTs. In reality, processors execute asynchronously due to cache effects, network collisions, etc. It appears that they gradually drift out of sync during the remap phase, disturbing the communication schedule. To reduce this effect we added a barrier synchronizing all processors after every $\frac{n}{P^2}$ messages.[6] Figure 8 shows that this eliminates the performance drop.

We can test the effect of reducing g by improving the implementation to use both fat-tree networks present in the machine, thereby doubling the available network bandwidth. The result shown in Figure 8 is that the performance increases by only 15% because the network interface overhead (o) and the loop processing dominate.

This detailed quantitative analysis of the implementation shows that the hybrid-layout FFT algorithm is nearly optimal on the CM-5. The computation phases are purely local and the communication phase is

[5]The bisection bandwidth is the minimum bandwidth through any cut of the network that separates the set of processors into halves.

[6]For simplicity, the implementation uses the hardware barrier available on the CM-5. The same effect could have been achieved using explicit acknowledgement messages.

overhead-limited, thus the processors are 100% busy all the time (ignoring the insignificant L at the end of the communication phase). Performance improvements in the implementation are certainly possible, but without affecting the algorithm itself.

4.1.5 Overlapping communication with computation

In future machines we expect architectural innovations in the processor-network interface to significantly reduce the value of o with respect to g. Algorithms for such machines could try to overlap communication with computation in order to mask communication time, as in the optimal summation example. If o is small compared to g, each processor idles for $g - 2o$ cycles between successive transmissions during the remap phase. The remap can be merged into the computation phases, as in the optimal algorithms[28]. The initial portion of the remap is interleaved with the pre-remap computation, while the final portions can be interleaved with the post-remap computation. Unless g is extremely large, this eliminates idling of processors during remap.

4.2 Other examples

We now discuss three other problems that have been carefully studied on parallel machines and show how the LogP model motivates the development of efficient algorithms for them. Here we provide only a qualitative assessment of the key design issues.

4.2.1 LU Decomposition

Linear algebra primitives offer a dramatic example of the importance of careful development of high performance parallel algorithms. The widely used Linpack benchmark achieves greater than 10 GFLOPS on recent parallel machines. In this section we examine LU decomposition, the core of the Linpack benchmark, to show that the key ideas employed in high performance linear algebra routines surface easily when the algorithm is examined in terms of our model.

In LU decomposition using Gaussian elimination, an $n \times n$ non-singular matrix A is reduced in $n - 1$ elimination steps to a unit-diagonal lower triangular matrix L and an upper triangular matrix U such that $PA = LU$ for some permutation matrix P. Since L and U are constructed by overwriting A, we will refer only to the matrix A, with $A^{(k)}$ denoting the matrix A at the start of step k. In the k-th elimination step, the k-th row and column of $A^{(k)}$ are replaced by the k-th column of L and the k-th row of U. This involves *partial pivoting* to determine the pivot, *i.e.*, the element in column k (below the diagonal) of largest absolute value, *swapping* the k-th and pivot rows of $A^{(k)}$, and *scaling* of the k-th column by dividing it by the pivot. Thereafter, the $(n-k) \times (n-k)$ lower right square submatrix of $A^{(k)}$ is *updated* to $A^{(k+1)}$:

$$A_{ij}^{(k+1)} = A_{ij}^{(k)} - L_{ik}U_{kj},\; i, j = k+1, \ldots, n$$

The row permutations that result from pivoting are carried along as part of the final result. The parallelism in this algorithm is trivial: at step k all $(n-k)^2$ scalar updates are independent. The pivoting, swapping and scaling steps could be parallelized with appropriate data layout.

To obtain a fast algorithm, we first focus on the ratio of communication to computation. Observe that regardless of the data layout, the processor responsible for updating $A_{ij}^{(k)}$ must obtain L_{ik} and U_{kj}. A bad data layout might require each processor to obtain the entire pivot row and the entire multiplier column. Thus, step k would require $2(n-k)g + L$ time for communication [7] and $2(n-k)^2/P$ time for computation.

[7] We are assuming here that the $2(n-k)$ elements of the pivot row and multiplier column are distributed equally among processors and are communicated by an efficient all-to-all broadcast.

The communication can be reduced by a factor of 2 by choosing a *column* layout in which n/P columns of A are allocated to each processor. For this layout, only the multipliers need be broadcast since pivot row elements are used only for updates of elements in the same column. A more dramatic reduction in communication cost can be had by a *grid* layout in which each processor is responsible for updating a $(n-k)/\sqrt{P} \times (n-k)/\sqrt{P}$ submatrix of $A^{(k)}$. This requires each processor to receive only $2(n-k)/\sqrt{P}$ values, a gain of $\sqrt{P}$ in the communication ratio. Some of this advantage will be foregone due to the communication requirements of pivoting and scaling down a column that is shared by many processors. However, this cost is asymptotically negligible in comparison to the communication cost for update. [8]

Our specification of the grid layout is incomplete since there are many ways to distribute A among P processors so that each receives a submatrix of A determined by a set of $n/\sqrt{P}$ rows and columns. The two extreme cases are *blocked* and *cyclic* allocation in each dimension. In the former case, the rows and columns assigned to a processor are contiguous in A while in the latter they are maximally scattered ($\sqrt{P}$ apart). It should be clear that blocked grid layout leads to severe load imbalance: by the time the algorithm completes $n/\sqrt{P}$ elimination steps, $2\sqrt{P}$ processors would be idle and only one processor is active for the last $n/\sqrt{P}$ elimination steps. In contrast, the scattered layout allows all P processors to stay active for all but the last $\sqrt{P}$ steps. It is heartening to note that the fastest Linpack benchmark programs actually employ a scattered grid layout, a scheme whose benefits are obvious from our model.[9]

4.2.2 Sort

In general, most sorting algorithms have communication patterns which are data-dependent although some, such as bitonic sort, do exhibit highly structured oblivious patterns. However, since processors handle large subproblems, sort algorithms can be designed with a basic structure of alternating phases of local computation and general communication. For example, column sort consists of a series of local sorts and remap steps, similar to our FFT algorithm. An interesting recent algorithm, called splitter sort[7], follows this compute-remap-compute pattern even more closely. A fast global step identifies $P-1$ values that split the data into P almost equal chunks. The data is remapped using the splitters and then each processor performs a local sort.

4.2.3 Connected Components

Generally, efficient PRAM algorithms for the connected components problem have the property that the data associated with a small number of graph nodes is required for updating the data structures at all other nodes in the later stages of computation. For example, in [29] each component is represented by one node in the component and processors “owning" such nodes are the target of increasing numbers of “pointer-jumping" queries as the algorithm progresses. This leads to high contention, which the CRCW PRAM ignores, but LogP makes apparent.

We consider a randomized PRAM algorithm given in [31] and adapt it to the LogP model. By carefully analyzing various subroutines and performing several local optimizations, we are able to show that the

[8] We remark also that pipelining successive elimination steps appears easier to organize with column layout than with grid layout: we could schedule the broadcast of multipliers during the k-th step so that the processor responsible for the $(k+1)$-st column receives them early, allowing it to initiate the $(k+1)$-st elimination step while the update for the previous step is still under way.

[9] The variations that we have described above do not change the basic algorithm which is built around a rank-1 update operation on a matrix. In blocked LU decomposition, the elimination step involves operations on sub-matrices. Instead of dividing by the pivot element, the inverse of the pivot sub-matrix is computed and used to compute the multiplier submatrices. Similarly, the elimination step involves multiplication of sub-matrices or rank-r updates where r is the side of the sub-matrices. Blocked decomposition has been found to outperform scalar decomposition on several machines. (See, for example, [12].) The main reason for this is the extensive use of Level 3 BLAS (which are based on matrix-matrix operations and re-use cache contents optimally) in the blocked decomposition algorithm.

severe contention problem of naive implementations can be considerably mitigated and that for sufficiently dense graphs our connected components algorithm is compute-bound. For details of the analysis and the implementation, see [31].

5 Matching the Model to Real Machines

The LogP model abstracts the communication network into three parameters. 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 by L/g. 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. From a purely theoretical viewpoint, it might seem better to define L to be a simple function of P. However, from a hardware design viewpoint, it might seem important to specify the topology, routing algorithm, and other properties. In this section, we consider how well our "middle ground" model reflects the characteristics of real machines.

5.1 Average distance

A significant segment of the parallel computing literature assumes that the number of network links traversed by a message, or *distance*, is the primary component of the communication time. This suggests that the topological structure of the network is critical and that an important indicator of the quality of the network is the average distance between nodes. The following table shows the asymptotic average distance and the evaluation of the formula for the practical case, $P = 1,024$.

Network	Ave. Distance	$P = 1,024$
Hypercube	$\frac{\log p}{2}$	5
Butterfly	$\log p$	10
4^{deg} Fat Tree	$4 \log_4 p - 2/3$	9.33
3D Torus	$\frac{3}{4} p^{1/3}$	7.5
3D Mesh	$p^{1/3}$	10
2D Torus	$\frac{1}{2} p^{1/2}$	16
2D Mesh	$\frac{2}{3} p^{1/2}$	21

For configurations of practical interest the difference between topologies is a factor of two, except for very primitive networks, such as a 2D mesh or torus. Even there, the difference is only a factor of four. Moreover, this difference makes only a small contribution to total message transmission time, as discussed below.

5.2 Unloaded communication time

In a real machine, transmission of an M-bit long message in an unloaded or lightly loaded network has four parts. First, there is the *send overhead*; *i.e.*, the time that 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. The time for the last bit to cross the network to the destination node is Hr, where H is the distance of the route and r is the delay through each intermediate node. Finally, there is the *receive overhead*, *i.e.*, 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.

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

Table 1 indicates that, for current machines, message communication time through a lightly loaded network is dominated by the send and receive overheads, and thus is relatively insensitive to network structure. Furthermore, networks with a larger diameter typically have wider links (larger w), smaller routing delays (r), and a faster cycle time because all the physical wires are short. All these factors reduce the transmission time. Thus, modeling the communication latency as an arbitrary constant, in the absence of contention, is far more accurate than simple rules, such as $L = \log P$. In determining LogP parameters for a given machine, it appears reasonable to choose $o = \frac{T_{snd}+T_{rcv}}{2}$, $L = Hr + \lceil \frac{M}{w} \rceil$, where H is the maximum distance of a route and M is the fixed message size being used, and g to be M divided by the per processor bisection bandwidth.

Machine	Network	Cycle ns	w bits	$T_{snd} + T_{rcv}$ cycles	r cycles	avg. H (1024 Proc.)	T(M=160) (1024 Proc.)
nCUBE/2	Hypercube	25	1	6400	40	5	6760
CM-5	Fattree	25	4	3600	8	9.3	3714
Dash[22]	Torus	30	16	30	2	6.8	53
J-Machine[11]	3d Mesh	31	8	16	2	12.1	60
Monsoon[26]	Butterfly	20	16	10	2	5	30
nCUBE/2 (AM)	Hypercube	25	1	1000	40	5	1360
CM-5 (AM)	Fattree	25	4	132	8	9.3	246

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

The send and receive overheads in Table 1 warrant some explanation. The very large overheads for the commercial message passing machines (nCUBE/2 and CM-5) reflect the standard communication layer from the vendor. Large overheads such as these have led many to conclude that large messages are essential. For the nCUBE/2, the bulk of this cost is due to buffer management and copying associated with the asynchronous send/receive communication model[33]. This is more properly viewed as part of the computational work of an algorithm using that style of communication. For the CM-5, the bulk of the cost is due to the protocol associated with the synchronous send/receive, which involves a pair of messages before transmitting the first data element. This protocol is easily modeled in terms of our parameters as $3(L + 2o) + ng$, where n is the number of words sent. The final two rows show the inherent hardware overheads of these machines, as revealed by the Active Message layer[33].

The table also includes three research machines that have focused on optimizing the processor network interface: Dash[22] for a shared memory model, J-machine[11] for a message driven model, and Monsoon[26] for a dataflow model. Although a significant improvement over the commercial machines, the overhead is still a significant fraction of the communication time.

5.3 Saturation

In a real machine the latency experienced by a message tends to increase as a function of the load, i.e., the rate of message initiation, because more messages are in the network competing for resources. Studies such as [10] show that there is typically a saturation point at which the latency increases sharply; below the saturation point the latency is fairly insensitive to the load. This characteristic is captured by the capacity constraint in LogP.

5.4 Long messages

The LogP model does not give special treatment to long messages, yet some machines have special hardware (*e.g.*, a DMA device connected to the network interface) to support sending long messages. The processor overhead for setting up that device is paid once and a part of sending and receiving long messages can be overlapped with computation. This is tantamount to providing two processors on each node, one to handle messages and one to do the computation. Our basic model assumes that each node consists only of one processor that is also responsible for sending and receiving messages. Therefore the overhead o is paid for each word (or small number of words). Providing a separate network processor to deliver or receive long messages can at best double the performance of each node. This can simply be modeled as two processors at each node.

5.5 Specialized hardware support

Some machines provide special hardware to perform a broadcast, scan, or global synchronization. In LogP, processors must explicitly send messages to perform these operations.[10] However, the hardware versions of these operations are typically limited in functionality; for example, they may only work with integers, not floating-point numbers. They may not work for only a subset of the machine.

The most common global operation observed in developing algorithms for the LogP model is a barrier, as in the FFT example above. As discussed below, some parallel models assume this to be a primitive operation. It is simpler to support in hardware than global data operations. However, there is not yet sufficient evidence that it will be widely available. One virtue of having barriers available as a primitive is that analysis is simplified by assuming the processors exit the barrier in synchrony.

5.6 Contention free communication patterns

By abstracting the internal structure of the network into a few performance parameters, the model cannot distinguish between "good" permutations and "bad" permutations. Various network interconnection topologies are known to have specific contention-free routing patterns. Repeated transmissions within this pattern can utilize essentially the full bandwidth, whereas other communication patterns will saturate intermediate routers in the network. These patterns are highly specific, often depending on the routing algorithm, and amount of buffering in each router, as well as the interconnection topology. We feel this is an important message to architects of parallel computers. If the interconnection topology of a machine or the routing algorithm performs well on only a very restricted class of communication patterns, it will only be usable to a very specific set of algorithms. Even so, for every network there are certain communication patterns where performance will be degraded due to internal contention among routing paths. The goal of the hardware designer should be to make these the exceptional case rather than the frequently occurring case. A possible extension of the LogP model to reflect network performance on various communication patterns would be to provide multiple g's, where the one appropriate to the particular communication pattern is used in the analysis.

[10] It is an unintended use of the model to synchronize implicitly without sending messages, *e.g.* by relying on an upper bound on the communication latency L.

6 Other Models

Our model was motivated by the observation of a convergence in the hardware organization of general purpose massively parallel computers. Parallel systems of the next decade will consist of up to a few thousand powerful processor/memory nodes connected by a fast communication network. In this section we briefly review some of the existing computational models and explain why they fail to fully capture the essential features of the coming generation of machines.

6.1 PRAM models

The PRAM[13] is the most popular model for representing and analyzing the complexity of parallel algorithms. The PRAM model is simple and very useful for a gross classification of algorithms, but it does not reveal important performance bottlenecks in distributed memory machines because it assumes a single shared memory in which each processor can access any cell in unit time. In effect, the PRAM assumes that interprocessor communication has infinite bandwidth, zero latency, and zero overhead ($g = 0, L = 0, o = 0$). Thus, the PRAM model does not discourage the design of algorithms with an excessive amount of interprocessor communication. Since the PRAM model assumes that each memory cell is independently accessible, it also neglects the issue of contention caused by concurrent access to different cells within the same memory module. The EREW PRAM deals only with contention for a single memory location. The PRAM model also assumes unrealistically that the processors operate completely synchronously.

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 for this is that these tasks can be assigned to the physical processors, with each processor apportioning its time among the tasks assigned to it. However, the PRAM model does not charge for the high level of message traffic and context switching overhead that such simulations require, nor does it encourage the algorithm designer to exploit the assignment. Similar criticisms apply to the extensions of the PRAM model considered below.

It has been suggested that the PRAM can serve as a good model for expressing the logical structure of parallel algorithms, and that implementation of these algorithms can be achieved by general-purpose simulations of the PRAM on distributed-memory machines[27]. However, these simulations require powerful interconnection networks, and, even then, may be unacceptably slow, especially when network bandwidth and processor overhead for sending and receiving messages are properly accounted.

6.2 Extensions of the PRAM model

There are many variations on the basic PRAM model which address one or more of the problems discussed above, namely memory contention, asynchrony, latency and bandwidth.

Memory Contention: The *Module Parallel Computer* [19, 24] differs from the PRAM by assuming that the memory is divided into modules, each of which can process one access request at a time. This model is suitable for handling memory contention at the module level, but does not address issues of bandwidth and network capacity.

Asynchrony: Gibbons[14] proposed the *Phase PRAM*, an extension of the PRAM in which computation is divided into "phases." All processors run asynchronously within a phase, and synchronize explicitly at the end of each phase. This model avoids the unrealistic assumption that processors can remain in synchrony without the use of explicit synchronization primitives, despite uncertainties due to fluctuations in execution times, cache misses and operating system calls. To balance the cost of synchronization with the time spent computing, Gibbons proposes to have a single processor of a phase PRAM simulate several PRAM processors. Other proposals for asynchrony include[8, 18, 21].

Latency: The *delay model* of Papadimitriou and Yannakakis[25] accounts for communication latency, *i.e.* it realizes there is a delay between the time some information is produced at a processor and the time it can be used by another. A different model that also addresses communication latency is the *Block Parallel Random Access Machine* (BPRAM) in which block transfers are allowed[1].

Bandwidth: A model that deals with communication bandwidth is the *Local-Memory Parallel Random Access Machine* (LPRAM)[2]. This is a CREW PRAM in which each processor is provided with an unlimited amount of local memory and where accesses to global memory are more expensive. An asynchronous variant which differs in that it allows more than one outstanding memory request has been studied in [23].

Memory Hierarchy: Whereas PRAM extends the RAM model by allowing unit time access to any location from any processor and LogP essentially views each node as in the RAM, the *Parallel Memory Hierarchy model* (PMH)[3] rejects the RAM as a basis and views the memory as a hierarchy of storage modules. Each module is characterized by its size and the time to transfer a block to the adjoining modules. A multiprocessor is represented by a tree of modules with processors at the leaves. This model is based on the observation that the techniques used for tuning code for the memory hierarchy are similar to those for developing parallel algorithms.

Other primitive parallel operations: The scan-model[6] is an EREW PRAM model extended with unit-time scan operations (data independent prefix operations), *i.e.*, it assumes that certain scan operations can be executed as fast as parallel memory references. For integer scan operations this is approximately the case on the CM-2 and CM-5.

The observation of the deficiences of the PRAM led Snyder to conclude it was unrealizable and to develop the *Candidate Type Architecture* (CTA) as an alternative[30]. The CTA is a finite set of sequential computers connected in a fixed, bounded degree graph. The CTA is essentially a formal description of a parallel machine; it requires the communication cost of an algorithm to be analyzed for each interconnection network used. The BSP model presented below abstracts the structure of the machine, so the analysis of algorithms is based on a few performance parameters. This facilitates the development of portable algorithms.

6.3 Bulk Synchronous Parallel Model

Valiant's Bulk Synchronous Parallel (BSP) model is closely aligned with our goals, as it seeks to bridge the gap between theoretical work and practical machines[32]. In the BSP model a distributed-memory multiprocessor is described in terms of three elements:

1. *processor/memory modules*,
2. an *interconnection network*, and
3. a *synchronizer* which performs barrier synchronization.

A computation consists of a sequence of *supersteps*. During a superstep each processor performs local computation, and receives and sends messages, subject to the following constraints: the local computation may depend only on data present in the local memory of the processor at the beginning of the superstep, and a processor may send at most h messages and receive at most h messages in a superstep. Such a communication pattern is called a h-relation.

Although the BSP model was one of the inspirations for our work, we have several concerns about it at a detailed level.

- In the BSP model, the length of a superstep must be sufficient to accommodate an arbitrary h-relation (and hence the most unfavorable one possible). Our model enables communication to be scheduled more precisely, permitting the exploitation of more favorable communication patterns, such as those in which not all processors send or receive as many as h messages. Valiant briefly mentions such refinements as varying h dynamically or synchronizing different subsets of processors independently, but does not describe how such refinements might be implemented.

- In the BSP model, the messages sent at the beginning of a superstep can only be used in the next superstep, even if the length of the superstep is longer than the latency. In our model processors work asynchronously, and a processor can use a message as soon as it arrives.

- The BSP model assumes special hardware support to synchronize all processors at the end of a superstep. The synchronization hardware needed by the BSP may not be available on many parallel machines, especially in the generality where multiple arbitrary subsets of the machine can synchronize. In our model all synchronization is done by messages, admittedly at higher cost than if synchronization hardware were available.

- Valiant proposes a programming environment in which algorithms are designed for the PRAM model assuming an unlimited number of processors, and then implemented by simulating a number of PRAM processors on each BSP processor. He gives a theoretical analysis showing that the simulation will be optimal, up to constant factors, provided that the *parallel slackness*, *i.e.* the number of PRAM processors per BSP processor, is sufficiently large. However, the constant factors in this simulation may be large, and would be even larger if the cost of context switching were fully counted. For example, in a real processor, switching from one process to another would require resetting not only the registers, but also parts of the cache.

6.4 Network models

In a network model, communication is only allowed between directly connected processors; other communication is explicitly forwarded through intermediate nodes. 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. Examples are parallel prefix and non-commutative summing (tree), physical simulations and numerical solvers for partial differential equations (mesh), FFT and bitonic sorting (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. Most current networks allow messages to “cut through” intermediate nodes without disturbing the processor; this is much faster than explicit forwarding.

The perfect correspondence between algorithm and network usually requires a number of processors equal to 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. Wherever problems have a local, regular communication pattern, such as stencil calculation on a grid, it is easy to lay the data out so that only a diminishing fraction of the communication is external to the processor. Basically, the interprocessor communication diminishes like the surface to volume ratio and with large enough problem sizes, the cost of communication becomes trivial. This is also the case for some complex communication patterns, such as the butterfly, as illustrated by the FFT example in Section 4.

We conclude that the design of portable algorithms can best be carried out in a model such as LogP, in which detailed assumptions about the interconnection network are avoided.

6.5 Potential Concerns

The examples of the previous sections illustrate several ways in which our model aids algorithm design. As with any new proposal, however, there will naturally be concerns regarding its utility as a basis for further study. Here we attempt to anticipate some of these concerns and describe our initial efforts to address them.

1. Does the model reveal any interesting theoretical structure in problems?
2. The model is much more complex than the PRAM. Is it tractable to analyze non-trivial algorithms?
3. Many algorithms have elegant communication structures. Is it reasonable to ignore this?
4. Similarly, there are algorithms with trivial communication patterns, such as local exchanges on a grid. Is it reasonable to treat everything as the general case?
5. Many interconnection topologies have been well-studied and their properties – both good and bad – are well understood. What price do we pay for ignoring topology?

The first two concerns are general algorithmic concerns and the last three are more specific to the handling of communication issues in our model. We address the algorithmic concerns first.

6.5.1 Algorithmic Concerns

The first of these concerns is partially addressed by the simple examples discussed in Section 3. The problem of how best to sum n numbers results in a formulation that is distinct from that obtained on the PRAM or various network models. We believe that the greater fidelity of the model in addressing communication issues in a manner that is independent of interconnection topology brings out an interesting structure in a variety of problems.

The second concern is perhaps the more significant: can we do reasonable analysis with so many factors? The first remark to be made on this issue is that not all the factors are equally relevant in all situations and one can often work with a simplified model. In many algorithms, messages are sent in long streams that can be pipelined, so that communication time is dominated by the inter-message gaps and L can be ignored. Algorithms that exhibit very small amounts of communication can be analyzed ignoring g and o. If overlapping communication with computation is not an issue, o need not be considered a separate parameter at all. We have presented analyses of several algorithms that illustrate these ideas.

6.6 Communication Concerns

As we discussed in the comparison to network models, the elegant communication pattern that some algorithms exhibit on an element-by-element basis disappears when we have a large number of elements allocated to each processor. For a communication structure to be meaningful, it must exist when data is mapped onto large processor nodes. We note, however, that the benefits of the structured communication pattern are not entirely foregone in this mapping: the structure guides us in choosing an effective element-to-processor allocation. In fact, with appropriate data layout the communication pattern for many algorithms is seen to be built around a small set of communication primitives such as broadcast, reduction or permutation. This was illustrated by the FFT example in Section 4, where we observed that once layout is addressed, the basic communication pattern is simply a mapping from a cyclic layout to a blocked one. Very similar observations apply to parallel sorting, summation, LU decomposition, matrix multiplication and other examples. For a simple communication pattern such as a grid, the LogP model motivates one to group together a large number of neighboring grid points onto the same processor so that the computation to communication ratio is large. The precise juxtaposition of processors is not as important.

7 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 models in current use do not accurately reflect the performance characteristics of such machines and, thus are an incomplete framework for development of practical algorithms.

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.

Optimal algorithms for broadcast and summation demonstrate how an algorithm may adapt its computational structure and the communication schedule in response to each of the parameters of the model. Realistic examples, notably the FFT, demonstrated how the presence of a large number of data elements per processor influences the inherent communication pattern. This suggests that adjusting the data placement is an important technique for improving algorithms, much like changes in the representation of data structures. Furthermore, these examples illustrate the importance of balanced communication, which is scarcely addressed in other models. These observations are borne out by implementation on the CM-5 multiprocessor. Finally, alternative parallel models are surveyed.

We believe that 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.

Acknowledgements

Several people provided helpful comments on earlier drafts of this paper, 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 Tarjan, Leslie Valiant, and the anonymous reviewers.

Computational support was provided by the NSF Infrastructure Grant number CDA-8722788. David Culler is supported by an NSF Presidential Faculty Fellowship CCR-9253705 and LLNL Grant UCB-ERL-92/172. Klaus Erik Schauser is supported by an IBM Graduate Fellowship. Richard Karp and Abhijit Sahay are supported by NSF/DARPA Grant CCR-9005448. Eunice Santos is supported by a DOD-NDSEG Graduate Fellowship. Ramesh Subramonian is supported by Lockheed Palo Alto Research Laboratories under RDD 506. Thorsten von Eicken is supported by the Semiconductor Research Corporation.

References

[1] A. Aggarwal, A. K. Chandra, and M. Snir. On Communication Latency in PRAM Computation. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21. ACM, June 1989.

[2] A. Aggarwal, A. K. Chandra, and M. Snir. Communication Complexity of PRAMs. In *Theoretical Computer Science*, pages 3–28, March 1990.

[3] B. Alpern, L. Carter, E. Feig, and T. Selker. The Uniform Memory Hierarchy Model of Computation. *Algorithmica*, 1993. (to appear).

[4] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 11–22, June 1992.

[5] G. Bell. Ultracomputers: A Teraflop Before Its Time. *Communications of the Association for Computing Machinery*, 35(8):26–47, August 1992.

[6] G. E. Blelloch. Scans as Primitive Parallel Operations. In *Proceedings of International Conference on Parallel Processing*, pages 355–362, 1987.

[7] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceedings of the Symposium on Parallel Architectures and Algorithms*, 1990.

[8] R. Cole and O. Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *Proceedings of the Symposium on Parallel Architectures and Algorithms*, pages 169–178, 1989.

[9] J. M. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comp*, 19:297–301, 1965.

[10] W. J. Dally. Performance Analysis of k-ary n-cube Interconnection Networks. *IEEE Transaction on Computers*, 39(6):775–785, June 1990.

[11] W. Dally et al. The J-Machine: A Fine-Grain Concurrent Computer. In *IFIP Congress*, 1989.

[12] J. J. Dongarra, R. van de Geijn, and D. W. Walker. A Look at Scalable Dense Linear Algebra Libraries. In J. Saltz, editor, *Proceedings of the 1992 Scalable High Performance Computing Conference*. IEEE Press, 1992.

[13] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the 10th Annual Symposium on Theory of Computing*, pages 114–118, 1978.

[14] P. B. Gibbons. A More Practical PRAM Model. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168. ACM, 1989.

[15] J. L. Hennessy. MIPS R4000 Overview. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[16] J. L. Hennessy and D. A. Patterson. *Computer Architecture — A Quantitative Approach*. Morgan Kaufmann, 1990.

[17] IEEE. *Symposium Record — Hot Chips IV*, August 1992.

[18] P. Kanellakis and A. Shvartsman. Efficient parallel algorithms can be made robust. In *Proceedings of the 8th Symposium on Principles of Distributed Computing*, pages 211–221, 1989.

[19] R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient PRAM Simulation on a Distributed Memory Machine. In *Proceedings of the Twenty-Fourth Annual ACM Symposium of the Theory of Computing*, pages 318–326, May 1992.

[20] R. M. Karp, A. Sahay, E. Santos, and K. E. Schauser. Optimal Broadcast and Summation in the LogP Model. Technical Report UCB/CSD 92/721, UC Berkeley, 1992.

[21] Z. M. Kedem, K. V. Palem, and P. G. Spirakis. Efficient robust parallel computations. In *Proceedings of the 22nd Annual Symposium on Theory of Computing*, pages 138–148, 1990.

[22] D. Lenoski et al. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[23] C. U. Martel and A. Raghunathan. Asynchronous PRAMs with memory latency. Technical report, University of California, Davis, Division of Computer Science, 1991.

[24] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.

[25] C. H. Papadimitriou and M. Yannakakis. Towards an Architecture-Independent Analysis of Parallel Algorithms. In *Proceedings of the Twentieth Annual ACM Symposium of the Theory of Computing*, pages 510–513. ACM, 1988.

[26] G. M. Papadopoulos and D. E. Culler. Monsoon: an Explicit Token-Store Architecture. In *Proc. of the 17th Annual Int. Symp. on Comp. Arch.*, Seattle, Washington, May 1990.

[27] A. G. Ranade. How to emulate shared memory. In *Proceedings of the 28th IEEE Annual Symposium on Foundations of Computer Science*, pages 185–194, 1987.

[28] A. Sahay. Hiding Communication Costs in Bandwidth-Limited Parallel FFT Computation. Technical Report UCB/CSD 92/722, UC Berkeley, 1992.

[29] Y. Shiloach and U. Vishkin. An $O(\log n)$ Parallel Conectivity Algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[30] L. Snyder. Type Architectures, Shared Memory, and the Corollary of Modest Potential. In *Ann. Rev. Comput. Sci.*, pages 289–317. Annual Reviews Inc., 1986.

[31] R. Subramonian. The influence of limited bandwidth on algorithm design and implementation. In *Dartmouth Institute for Advanced Graduate Studies in Parallel Computation (DAGS/PC)*, June 1992.

[32] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the Association for Computing Machinery*, 33(8):103–11, August 1990.

[33] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.