DISTRIBUTED SYSTEMS

Growth of distributed systems has attained unstoppable momentum. If we better understood how to think about, analyze, and design distributed systems, we could direct their implementation with more confidence.

LEONARD KLEINROCK

DISTRIBUTED SYSTEMS IN NATURE

How did the killer bees find their way up to North America? By what mechanism does a colony of ants carry out its complex tasks? What guides and controls a flock of birds or a school of fish? The answers to these questions involve examples of loosely coupled systems that achieve a common goal with distributed control.

Throughout nature we find an enormous amount of processing taking place at the level of the individual organism (be it an ant, a sparrow, or a human), and we have only begun to comprehend how processing and memory functions operate, especially in the human species. How does a human perform the acts of perception, cognition, decision making, and motor control? This processing occurs in a fraction of a second, using natural processing elements that are orders of magnitude slower than our current computer processing elements [8].

We do know that the brain is organized and structured very differently from our present computing machines. In human beings (i.e., in their internal neural systems) and in groups of organisms, nature has been extremely successful in implementing distributed systems that are far more clever and impressive than any computing machine humans have yet devised. We have succeeded in manufacturing highly complex devices capable of high-speed computation and massive accurate memory, but we have not yet gained sufficient understanding of distributed systems—our systems are still highly constrained and rigid in their construction and behavior. The gap between natural and man-made systems is huge, and we have a long way to go before we bridge the gap in understanding and implementation (see Figure 1, pp. 1202–1203).

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-82-C0064.

ACM/IEEE-CS Joint Issue © 1985 ACM 0001-0782/85/1100-1200 75¢

WHY SHOULD WE STUDY DISTRIBUTED SYSTEMS?

Currently we are experiencing the effects of the confluence of powerful forces in information technology. By far, the most significant effect is the host of revolutionary changes that have been brought about by the integrated chip—especially in the form of VLSI and the resulting enormous improvements in processing, storage, and communications. At the same time, we are experiencing a frightening backlog in software-application development while the user community is clamoring for unprecedented power in processing, communications, storage, and applications. Fortunately, we have the potential for this power—if only we could figure out how to put all the pieces together!

Distributed systems have come into existence in our industrial society in some very natural ways. For example, we have seen the emergence of a large number of distributed databases—systems that have evolved because the source of the data is not centralized and where there is a local need for frequent and immediate access to the locally generated data (e.g., the employee database at a branch office of a nationwide organization) in addition to a global need to view the entire database. Situations such as these require us to place some processing power at the many distributed locations for collecting, preprocessing, and accessing data. On-line transaction processing is an application that may contain a local component as well as a distributed-processing component, and the current proliferation of desktop personal computers is a manifestation of distributed-processing power. Indeed, if we measure processing power in MIPS (millions of instructions per second), we note that the number of installed MIPS in personal computers is an order of magnitude greater than the number installed in mainframes. However, most of those PC MIPS lie idle most of the time. Imagine what a terrific distributed-processing system we could fire up with that unused power! When data and processing are distributed, we are obliged to provide communications to link the resources. Thus we are led into the use of packet networks, satellite networks, internets, cellular and packet radio networks, metropolitan area networks, and local area networks.

Distributed systems can provide the necessary power to meet the growing demands of the user community. We are demanding capability faster than the advances in devices alone can supply, and to meet these demands we will have to rely on innovative computing architectures such as parallel-processing systems. These large distributed databases, along with distributed-processing and distributed-communication networks, have given rise to some very complex distributed-system structures, and it is essential that we learn how to think about them properly (see Figure 2, pp. 1204–1205).

ARCHITECTURE AND ALGORITHMS

The world of applications has an insatiable need for computing power. A good mathematician can easily consume any finite computing capability by posing a combinatoric problem whose computational complexity grows exponentially with a variable of the problem (e.g., the enumeration of all graphs with N nodes). The ways in which we push back this "power wall" involve both hardware and software solutions. Typically, the methods for speeding up the computation include the following:

- faster devices (a physics and engineering problem),
- architectures that permit concurrent processing (a system design problem),
- optimizing compilers for detecting concurrency (a software-engineering problem),
- algorithms for specification of concurrency (a language problem), and
- more expressive models of computation (an analytic problem).

Characterizing the Architecture

There are many ways of classifying machine architectures—too many, in fact. The following classification was selected for the purposes of this article.

We begin with the purely serial uniprocessor in which a single instruction stream operates on a single data stream (SISD). These systems are "centralized" at the global level, but really do contain many elements of a distributed system at the lower levels, for example, at the level of communications on the VLSI chips themselves.

Next is the vector machine, in which a single instruction stream operates on a multiple data stream (SIMD). These include array processors (e.g., systolic arrays) and pipeline processors.

The third consists of multiple processors that, collectively, can process multiple instruction streams on multiple data streams (MIMD). The form of multiprocessing that takes place when multiple processors cooperate closely to process tasks from the same job is referred to as parallel processing. On the other hand, the term distributed processing is applied to the form of multiprocessing that takes place when the multiple processors cooperate loosely and process separate jobs.

Vector machines and multiprocessing systems all provide some form of concurrency. The effect of this concurrency on system performance is important and is therefore a very active area of research (see Figure 3, p. 1206).

Since the onslaught of the VLSI revolution, a number of machine architectures have been implemented in an attempt to provide the supercomputing power toward which concurrent processing tempts us [5]. Two excellent recent summaries of some of these projects are offered by Hwang [9] and by Schneck et al. [17]. There you will find the Butterfly machine, the Cosmic Cube, various kinds of tree machines, the Cedar project, the Sisal language, the Connection machine, and others whose names are intriguingly close to Mother Nature's systems.

Characterizing the Algorithm

The major goal in characterizing the algorithm is to identify and exploit its inherent parallelism (i.e., potential for concurrency). The levels of resolution at which we can attempt to find this parallelism are listed below in decreasing order of granularity [16]:

- job execution,
- task execution,
- process execution,
- instruction execution,
- register transfer, and
- logic device.

Clearly, as we drop down the list to finer granularity, we expose more and more parallelism, but we also increase the complexity of scheduling these tiny objects to the processors and of providing the communications among so many objects (the problem of interprocess communication—IPC). As was stated earlier, if we operate at the top level (i.e., at the job level), then we think of the system as a distributed-processing system; if we operate at the task or process level, we have a parallel-processing system; if we operate at the instruction level, we have the vector machine and the array processor.

Regardless of the level at which we operate, it behooves us to create a "model" of the algorithm or, if you will, of the computation we are processing [10]. A very common model is the graph model of computation, which is normally used at the task or process level (another common modeling method is the use of Petri Nets). In this model, the nodes represent the tasks (or processes), and the directed edges represent the dependencies among the tasks, thereby displaying the partial ordering of the tasks and the parallelism that can be exploited (see Figure 4, p. 1207).

However, the problem of finding the parallelism in the lines of code that represent the algorithm still remains, and there is an ongoing effort to simplify (and even automate) this task by developing parallel programming languages for implementing these algorithms (e.g., Ada®, concurrent Pascal).

Matching the Architecture to the Algorithm

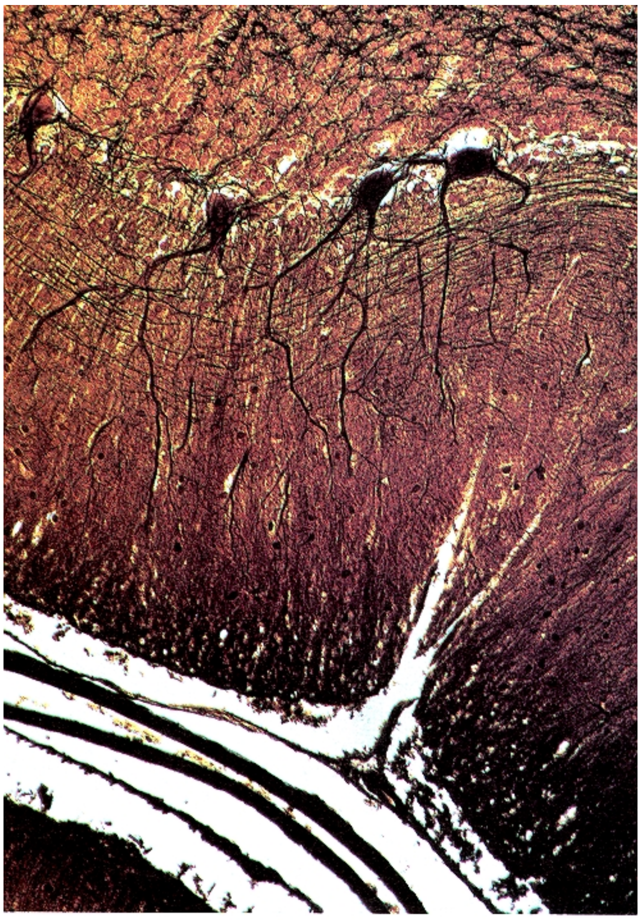
The performance of a distributed system depends strongly on how well the architecture and the algorithm are matched. For example, a highly parallel algorithm will perform well on a highly parallel architecture; a distributed system requiring lots of interprocessor communication will perform poorly if the communication bandwidth is too narrow. This matching problem becomes fierce and crucial when we attempt to coordinate an exponentially growing number of processors requiring an exponentially growing amount of interprocessor communication. The apparent solution to such an unmanageable problem is one that is self-organizing.

If we choose to use the graph model discussed, we are faced with a number of architecture/algorithm problems, namely, partitioning, scheduling, memory access, interprocess communication, and synchronization. The partitioning problem refers to decisions regarding the level of granularity and the choices involving which objects should be grouped into the same node of the task graph. The scheduling problem refers to the assignment of processors and memory modules to nodes of the computation graph. In general, this is an NP-complete problem (tough as nails to do optimally). The memory-access problem refers to the mechanism that allows processors to communicate with the various memory modules; usually, either shared-memory or message-passing schemes are used. The interprocessor-communication problem refers to the nature of the communication paths and connections that are available to provide processors access to the memory modules and to other processors; this may take the form of an interconnection network in a parallel-processing system, a local area network in a local distributed-processing system or shared data system or shared peripheral system, or a packet-switched, value-added, long-haul network in a nationwide distributed system. Synchronization refers to the requirement that no node in the graph model can begin execution until all of its predecessor nodes have completed their execution.

The use of broadcast or multicast communication opens up a number of interesting alternatives for communication. Local area networks take exquisite advantage of these communication modes. Algorithms that require tight coupling (i.e., lots of IPC) need not only large bandwidths (which, for example, could be provided by fiber-optic channels), but also low latency. Specifically, the speed of light introduces a 15,000-microsecond latency delay for a communication that must travel from coast-to-coast across the United States.

Another consideration in matching architectures to algorithms is the balance and trade-off among communication, processing, and storage. We have all seen sys-

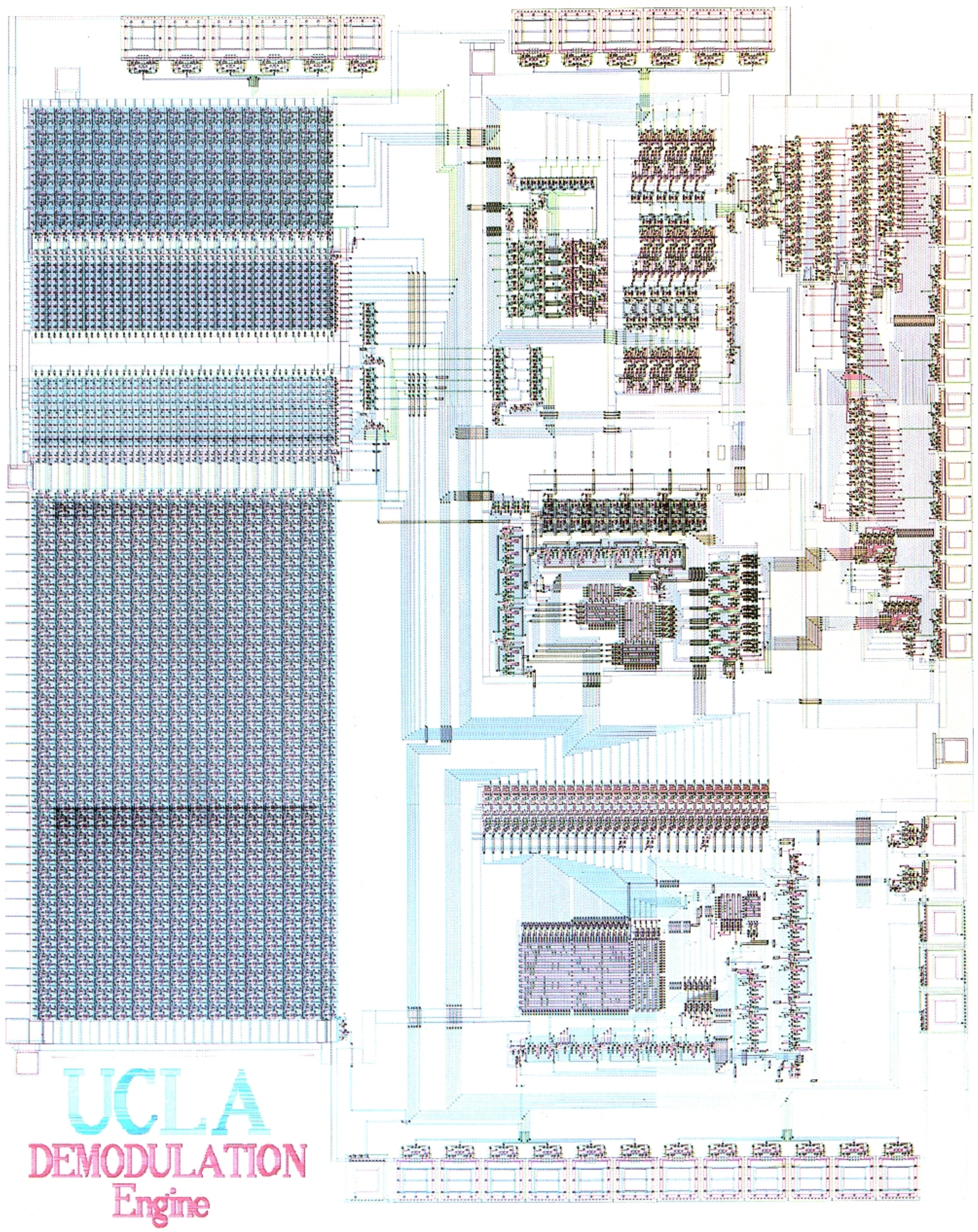
Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).



(a)

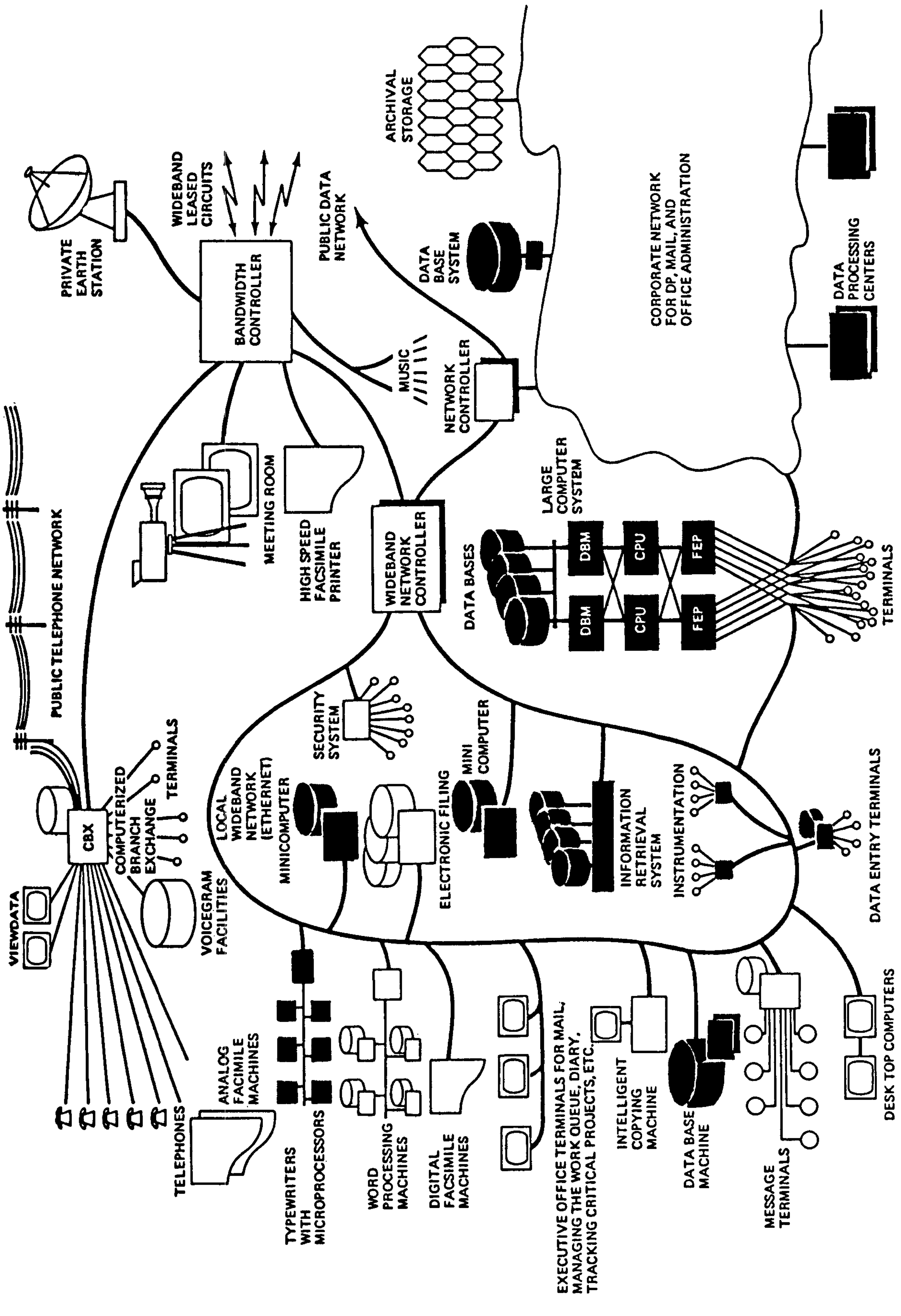
There is an amazing contrast between the neural structure of the human brain (a) and the architecture of today's VLSI chips (b). The brain is massively parallel, densely (and weirdly) connected with leaky transmission paths, highly fault tolerant, self-repairing, adaptive, noisy, and probably nondeterministic. Man-made computers are highly constrained, precisely (and often symmetrically) laid out with carefully isolated wires, not very fault tolerant, largely serial and centralized, deterministic, minimally adaptive, and hardly self-repairing. (Photo (a) is the courtesy of Peter Arnold, Inc.)

FIGURE 1. Natural and Man-Made Architectures



(b)

FIGURE 1. Natural and Man-Made Architectures


PRIVATE EARTH STATION
WIDEBAND LEASED CIRCUITS
BANDWIDTH CONTROLLER
PUBLIC DATA NETWORK
ARCHIVAL STORAGE
DATA BASE SYSTEM
CORPORATE NETWORK FOR DP, MAIL, AND OFFICE ADMINISTRATION
DATA PROCESSING CENTERS
MUSIC
NETWORK CONTROLLER
PUBLIC TELEPHONE NETWORK
MEETING ROOM
HIGH SPEED FACSIMILE PRINTER
WIDEBAND NETWORK CONTROLLER
LARGE COMPUTER SYSTEM
DATA BASES
DBM
CPU
FEP
TERMINALS
VIEWDATA
CBX
COMPUTERIZED BRANCH EXCHANGE
TERMINALS
VOICEGRAM FACILITIES
LOCAL WIDEBAND NETWORK (ETHERNET)
SECURITY SYSTEM
MINICOMPUTER
ELECTRONIC FILING
MINI COMPUTER
INFORMATION RETRIEVAL SYSTEM
INSTRUMENTATION
DATA ENTRY TERMINALS
TELEPHONES
ANALOG FACIMILE MACHINES
TYPEWRITERS WITH MICROPROCESSORS
WORD PROCESSING MACHINES
DIGITAL FACSIMILE MACHINES
EXECUTIVE OFFICE TERMINALS FOR MAIL, MANAGING THE WORK QUEUE, DIARY, TRACKING CRITICAL PROJECTS, ETC.
INTELLIGENT COPYING MACHINE
DATA BASE MACHINE
MESSAGE TERMINALS
DESK TOP COMPUTERS

tems where one of these resources can be exchanged for others. For example, if we do some preprocessing in the form of data compression prior to transmission, we can cut down on the communication load (trade processing for communication). If we store a list of computational results, we can cut down on the need to recompute the elements of the list each time we need the same entry (trade storage for processing). Similarly, if we store data from a previous communication, we need merely transmit the data address or name of the previous message rather than the message itself (trade storage for communication). Selecting the appropriate mix in a given problem setting is an important issue.

Distributed algorithms operating in a distributed network environment (e.g., a packet-switched network) pose the possibility that network failures may cause the network to temporarily be partitioned into two (or more) isolated subnetworks. In such a case, detection and recovery mechanisms must be introduced (see Figure 5, p. 1207).

Lastly, it should be mentioned that very little is known about characterizing those properties of an algorithm that cause it to perform well or poorly in a distributed environment.

PERFORMANCE AND BEHAVIOR

We do know some things about the way distributed systems behave, precious few though they may be. The most interesting thing about them is that they come to us from research in very different fields of study. Unfortunately, the collection of results (of which the following is a sample) is just that—a collection, with no fundamental models or theory behind it.

We begin by considering closely coupled systems, in particular, parallel-processing systems. One of the most compelling applications of parallel processing is in the area of scientific computing, where the speed of the world's largest uniprocessors is hopelessly inadequate to handle the computational complexity required for many of these problems [3]. Of course, the idea is that, as we apply more parallel processors to the computational job, the time to complete that job will drop in proportion to the number of (identical) processors, P. The "speedup" factor, denoted by S, is a common measure of performance for parallel-processing systems and is defined as the time required to complete the job using P processors, divided into the time required to complete the job using one of these processors. S may also be interpreted as the average number of busy processors, that is, the concurrency. The best we can achieve is for S to grow directly with P; that is,

$$S \leq P.$$

Thus, in general, $1 \leq S \leq P$. In the early days of parallel processing, Minsky [15] conjectured a depressingly pessimistic form for the typical speedup; namely,

$$S = \log P.$$

Often that kind of poor performance is indeed observed. Fortunately, however, experience has shown that things need not be that bad. For example, we can achieve $S = 0.3P$ for certain programs by carefully extracting the parallelism in Fortran DO loops [14]. However, Amdahl has pointed out a serious limitation to the practical improvements one can achieve with parallel processing (the same argument applies to the improvements available with vector machines) [1]. He argues that, if a fraction, f, of a computation must be done serially, then the fastest that S can grow with P is

$$S_{\max} = \frac{P}{fP + 1 - f}.$$

We see that, for $f = 1$ (everything must be serial), $S_{\max} = 1$; for $f = 0$ (everything in parallel), $S_{\max} = P$.

The actual amount of parallelism (i.e., S) achieved in a parallel-processing system is a quantity that we would like to be able to compute. S is a strong function of the structure of the computational graph of the jobs being processed. I, with one of my students [2], have been able to calculate S exactly as a simple function of the graph model. Specifically, we consider a parallel-processing system with P processors and with an arrival rate of λ jobs per second. We assume the collection of jobs can be modeled with an arbitrary computation graph with an average of N tasks per job, each task requiring an average of $\bar{x}$ seconds. Then it can be shown that

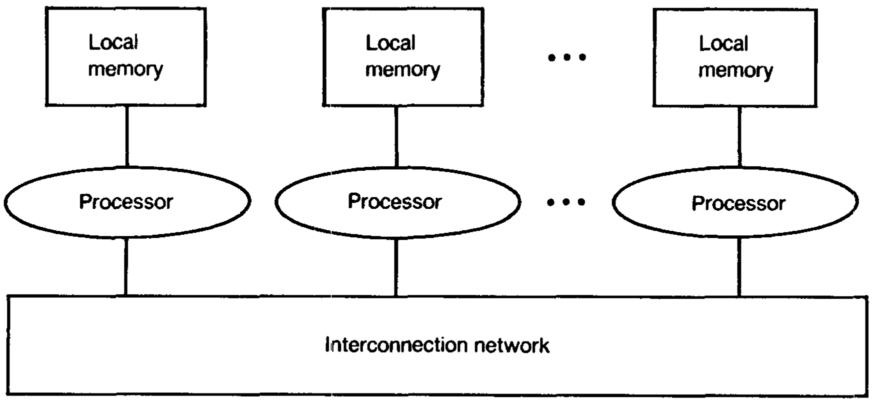
$$S = \begin{cases} \lambda N\bar{x} & \text{for} \quad \lambda N\bar{x} \leq P \\ P & \text{for} \quad \lambda N\bar{x} \geq P. \end{cases}$$

This is a very general result; in some special cases, the distribution of the number of busy processors can be found as well.

So far, we have given ourselves the luxury of increasing the system's computational capacity as we have added more processors to the system. Let us now consider adding more processors, but in a fashion that maintains a constant total system capacity (i.e., a constant system throughput in jobs completed per second). This will allow us to see the effect of *distributing* the computation for a job over many smaller processors. The particular structure we are considering is the regular series-parallel structure shown in Figure 6 (p. 1208), where we have taken a total processing capacity of C MIPS and divided it equally into mn processors, each of C/mn MIPS. On entering the system, a job selects

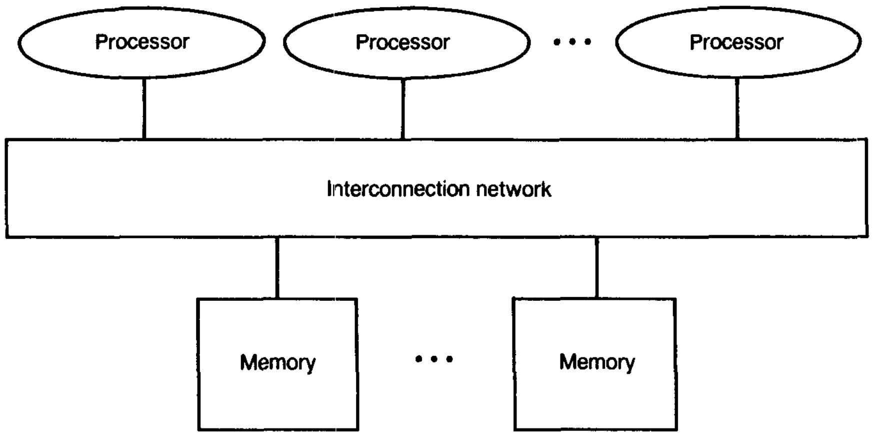
FIGURE 2. A Complex Distributed System (left)

Humans have created some unbelievably complex distributed systems. The fact that they work at all is amazing, given that we have not yet uncovered the basic principles determining their behavior. (From Martin, J. *Design and Strategy for Distributed Data Processing.* Prentice-Hall, Englewood Cliffs, N.J., 1981.)

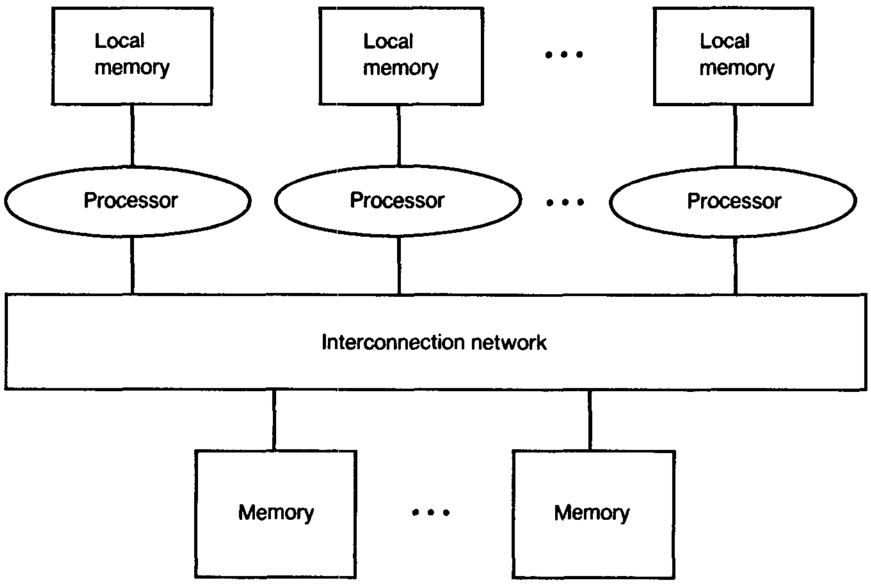


(a) Message-Passing Architecture: Local Memory

Locality of memory reference, bandwidth of communication, processor overhead, and cost are key issues determining the appropriate architecture for a given application.

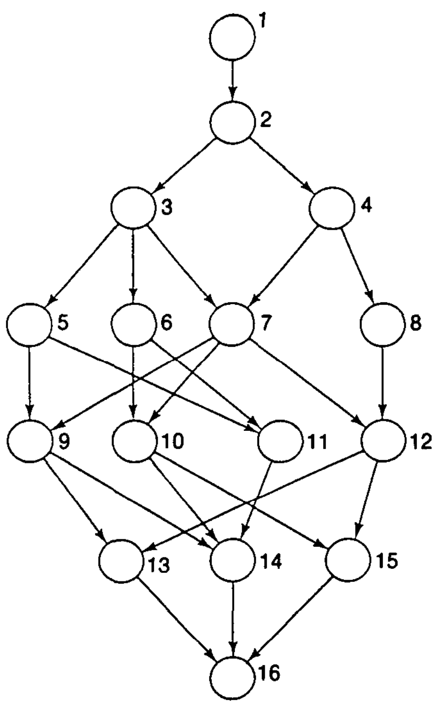


(b) Shared-Memory Architecture



(c) Hybrid Architecture

FIGURE 3. Architectures for Connecting Processors and Memory



The graph model of computation is an extremely useful model for displaying the parallelism inherent in an algorithm (i.e., a job). The entire graph represents the computational tasks associated with that job, the nodes represent the tasks themselves, and the directed arcs, which define a partial ordering of the nodes, represent the sequence in which the tasks must be performed.

FIGURE 4. Graph Model of Computation

(equally likely) any one of the m series branches down which it will travel. It will receive $1/n$ of its total processing needs at each of the n series-connected processors. The key result for this system [13] is that the mean response time for jobs in this series-parallel pipeline system is mn times as large as it would have been had the jobs been processed by a single processor of C MIPS! There are some statistical assumptions behind this result, but the message is clear—distributed processing of this kind is terrible. Why, then, is everyone talking about the advantages of distributed processing? The answer must be that a large number of *small* processors (e.g., microprocessors) with an aggregate capacity of C MIPS is less expensive than a large uniprocessor of the same total capacity. It can be shown that the series-parallel system *will* have the same response time as the uniprocessor if the aggregate capacity of the series-parallel system has K times the capacity of the uniprocessor where

$$K = mn - \rho(mn - 1)$$

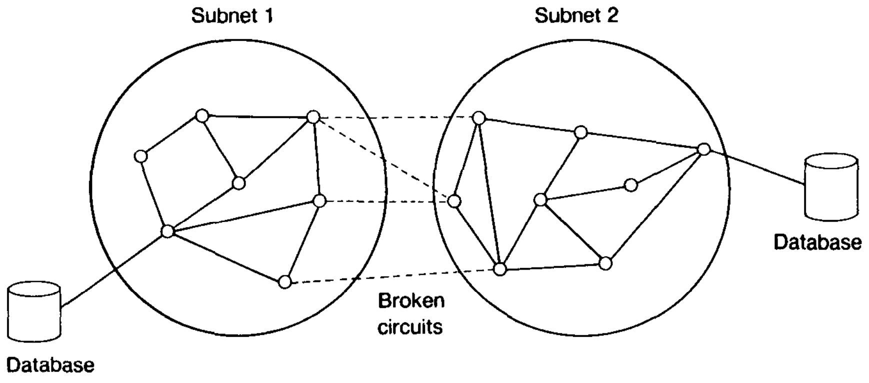
and where ρ is the utilization factor for each processor; namely, ρ = arrival rate of jobs times the average service time per job for a processor. This says that, for light loads ($\rho \ll 1$), $K = mn$, whereas, for heavy loads ($\rho \rightarrow 1$), $K = 1$. Is it the case that smaller machines are mn times less expensive than larger machines (so that we can purchase mn times the capacity at the same total price, as is needed in the light-load case)? To answer this question, recall a law that was empirically observed by Grosch more than three decades ago. Grosch's law [7] states that the capacity of a computer is related to its cost, which we denote by D (dollars) through the following equation:

$$C = JD^2$$

where J is a constant. This law may be rewritten as

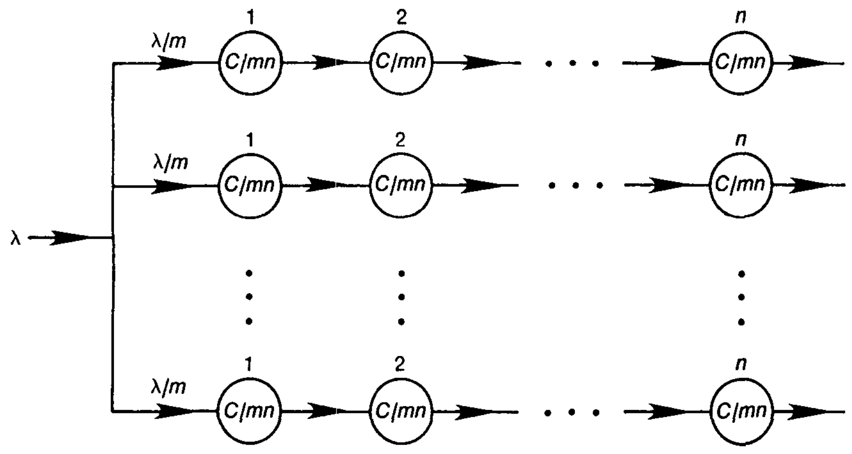
$$\frac{D}{C} = \frac{1}{\sqrt{JC}}.$$

Grosch tells us that the economics are *exactly the reverse* of what we need to break even with distributed processing! He says that larger machines are cheaper per MIPS. If Grosch is correct today, then why are microprocessors selling like hotcakes? A more recent look at the economics explains why. Ein-Dor [4] shows that, if we consider all computers at the same time, Grosch's law is clearly not true, as seen in Figure 7 (p. 1208). In this figure we see that microcomputers are a good buy.



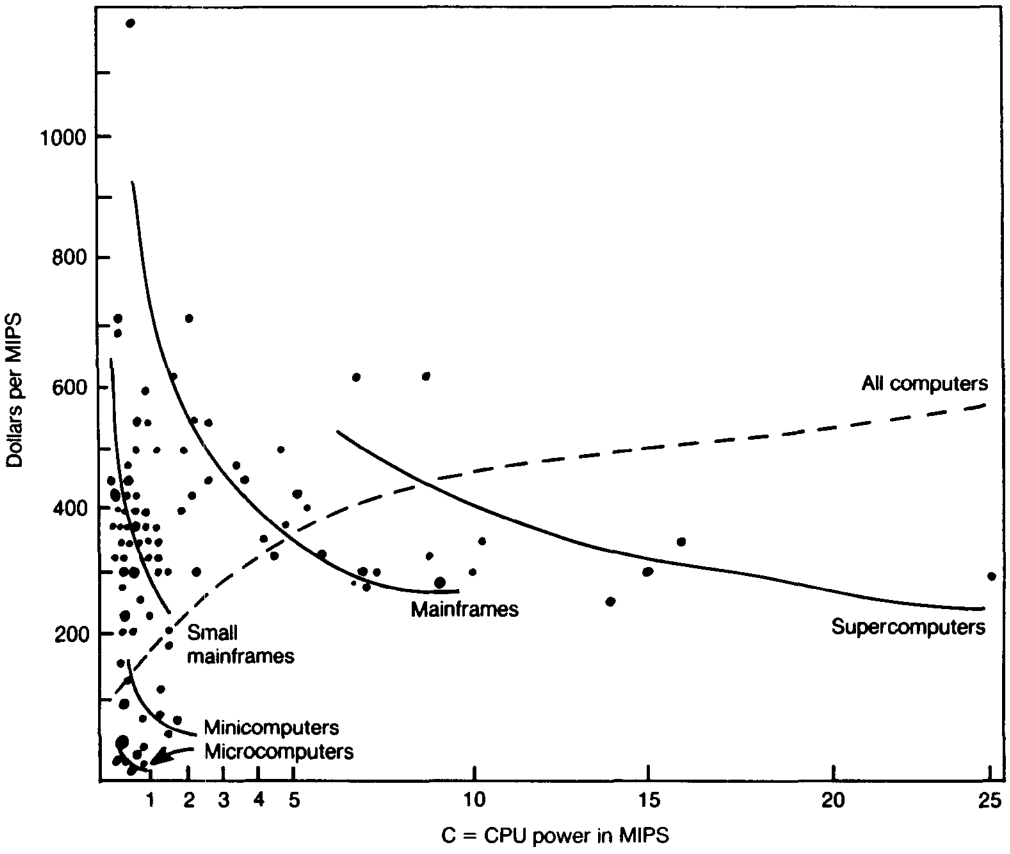
Network failures can create two separated subnetworks that cannot communicate until the failure is repaired. Maintaining consistency of databases in such a situation is a key issue in distributed-systems design.

FIGURE 5. A Partitioned Network



When a constant amount of processing capacity (C) is distributed into *mn* equal (and smaller) processors in a network such as this, the response time increases by a factor of *mn*. How can one justify a distributed system in the face of this degradation?

FIGURE 6. A Symmetrical Distributed-Processing Network



The cost per MIPS seems to rise with the number of MIPS when we examine all computers in a single group. However, when we separate them into families, we find that the opposite is true, thus confirming Grosch's law. Figure taken from Ein-Dor, P. Grosch's law re-revisited: CPU power and the cost of computation. *Commun. ACM 28*, 2 (Feb. 1985), 142–151.

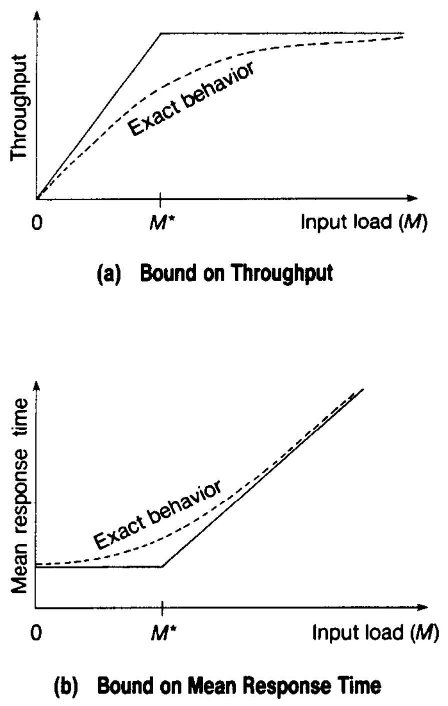
FIGURE 7. Economics of Computer Power

However, as Ein-Dor points out, Grosch's law is still true today if we consider *families* of computers. Each family has a decreasing cost per unit of capacity as capacity is increased. Ein-Dor goes on to make the observation that, if one needs a certain number of MIPS, then one should purchase computers from the smallest family that can currently supply that many MIPS. Furthermore, once in the family, it pays to purchase the biggest member machine in that family (as predicted by Grosch).

Now that we have discussed the performance of parallel-processing systems for some special cases, let us generalize the ways in which jobs pass through a multiprocessor system, and analyze the system throughput and response time. Indeed, we bound these key system-performance measures in the following way: Suppose we have a population of M customers competing for the resources of the system. Assume that customers generate jobs to be processed by some of the system's resources, that the way in which these jobs bounce around among the resources is specified in a probabilistic fashion, and that the mean response time of this system is T seconds. When a customer's job leaves the system, that customer then begins to generate another job request for the system, where the average time to generate the request is t_o seconds. Of interest is the mean response time, T, and the system throughput γ as a function of the other system parameters. Although we have been extremely general in the system description, we can nevertheless place an excellent upper bound on the system throughput and an excellent lower bound on the mean response time as shown in Figure 8. In this figure, the quantity M^* is defined as the ratio of the mean cycle time $T_o + t_o$ to the mean time x_o required on the critical resource in a cycle; T_o is the mean response time when $M = 1$, and the critical resource is that system resource that is most heavily loaded [11].

To find the exact behavior (shown in dashed lines in the figure) rather than the bounds, one must be much more explicit about the distributions of the service time required by jobs at each resource in the system as well as the queueing discipline at each. Using the bounds or the exact results, the effect of parameter changes on the system behavior can be seen. For example, one can examine the accuracy of the common rule of thumb that suggests that the proper mix of microprocessor speed, memory size, and communication bandwidth is in the proportion 1 MIPS, 1 Mbyte, and 1 Mbit per second; some suggest that we will soon see a 10, 10, 10 mix instead of the 1, 1, 1 mix. Of course the correct answer to this question depends on the total system configuration.

Once we evaluate the throughput and mean response time for a system, we usually want to find the relationship between the two, which typically has the well-known shape (shown in Figure 9, p. 1210) that clearly demonstrates the trade-off between them—a low delay implies a small throughput and vice versa.



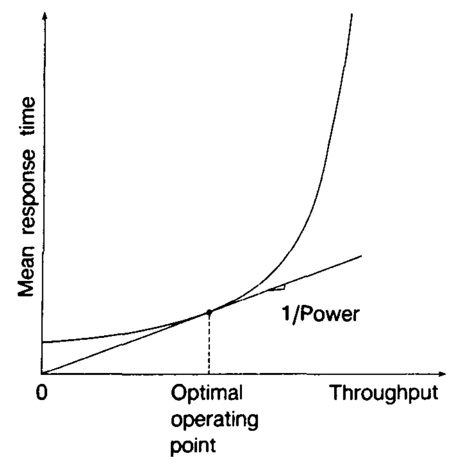
Excellent bounds on throughput (a) and mean response time (b) as a function of the number of users (or any measure of the input load) are easily obtained for a very large class of distributed systems. The exact behavior can be derived for more restricted systems and demonstrates the excellence of the bounds.

FIGURE 8. Bounds on Throughput and Response Time

We are immediately compelled to inquire about the location of the "optimal" operating point for a system. The answer depends on how much you hate delay versus how much you love throughput. One way to quantify this love–hate choice is to define a quantity known as "power" (denoted by P), which is defined as

$$P = \frac{\gamma}{T}.$$

The operating point that optimizes (i.e., maximizes) the power (large throughput and small delay) is located at that throughput where a straight line (of minimum slope) out of the origin touches the throughput-delay profile (usually tangentially); such a tangent and operating point are shown in Figure 9. This result holds for all profiles and all flow-control functions (see below). Moreover, for a large class of queueing curves, this optimal operating point implies that the system should be loaded in such a way that each resource has, on the *average, exactly one job* to work on [12].



The delay-throughput relationship, an example of the key profile in systems performance evaluation, clearly shows the trade-off between the two. In general, you cannot get a small delay and a large throughput at the same time. We can, however, maximize "power," which is the ratio of throughput to delay, in order to define the natural point for a system.

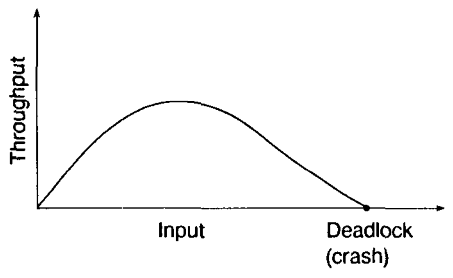
FIGURE 9. The Key System Profile

Unfortunately, there are some distributed systems that do not have the nice relationship shown in Figure 8a where the throughput rises asymptotically to its maximum value as the "input" is increased. Often we find the behavior depicted in Figure 10 where the throughput reaches a peak and then declines as the input increases further, possibly dropping to zero, in which case we say that the system has crashed. Such behavior has been observed in paged virtual-memory systems (thrashing), in computer networks (deadlocks and degradations), and in automobile traffic flow (bumper-to-bumper traffic). Here again, one must find a method for controlling the input (i.e., setting the system operating point) so as to achieve optimal or near-optimal performance (somewhere near the peak of the curve in Figure 10).

"Flow control" is the name associated with this operation, and it can be implemented in a centralized or a distributed fashion in distributed systems with the latter being the more challenging design problem [6]. One example of distributed control is the dynamic routing procedure found in many of today's packet-switching networks where no single switching node is responsible for the network routing. Instead, all nodes participate in the selection of network routes in a distributed fashion. A great deal of research is currently under way to evaluate the performance of other distributed algorithms in networks and distributed systems. Examples are the distributed election of a leader, distributed rules for traversing all the links of a network, and distributed rules for controlling access to a database.

Another large class of distributed-control algorithms has to do with sharing a common communication channel among a number of devices in a distributed fashion [19]. If the channel is a broadcast channel (also known as a one-hop channel), then the analytic and design problem is fairly manageable and a number of popular local area network algorithms for media access control have been studied and implemented. Examples here include CSMA/CD (carrier-sense multiple access with collision detect—as used in Xerox's Ethernet, AT&T's 3B-Net and Starlan, and IBM's PC Network), token passing (as used in the token-ring and token-bus networks), and address contention resolution (as used in AT&T's ISN). A large number of additional channel access algorithms have been studied in the literature including Expressnet, tree algorithms, urn models, and hybrid models. If the channel is multicast (or multihop), then the analytic problem becomes much harder.

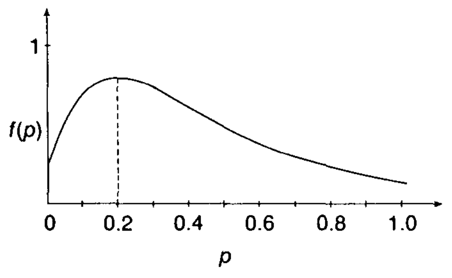
But what if the processors in our distributed environment are allowed to communicate with their peers in very limited ways? Can we endow these processors (let us call them automatons for this discussion) with an internal algorithm that will allow them to achieve a collective goal? Tsetlin [20] studied this problem at length and was able to demonstrate some remarkable behavior. For example, he describes the Goore game in which the automatons possess finite memory and act in a probabilistic fashion based on their current state and the current input. They cannot communicate with each other at all and are required to vote YES or NO at



There are many systems that degrade badly when pushed too hard. They can even degrade to a situation of deadlock. Examples include thrashing in virtual memory systems, deadlocks in computer networks, and bumper-to-bumper traffic in highway systems.

FIGURE 10. A Dangerous Throughput Profile

certain times. The automatons are not aware of each other's vote; however, there is a referee who can observe and calculate the percentage, p, of automatons that vote YES. The referee has a function, $f(p)$ (such as that shown in Figure 11), where we require that $0 \leq f(p) \leq 1$. Whenever the referee observes a percentage, p, who vote YES, *he or she will, with probability*, $f(p)$, reward each automaton, independently, with a one dollar payment; with probability $1 - f(p)$ he or she will punish an automaton by taking one dollar away. Tsetlin proved that no matter how many players there may be in a Goore game, if the automatons have sufficient memory, then for the payoff probability shown in the figure, exactly 20 percent of the automatons will vote YES with probability one! This is a beautiful demonstration of the ability of a distributed-processing system to act in an optimum fashion, even when the rules of the reward function are unknown to the players and when they can neither observe nor communicate with each other. All they are allowed is to vote when asked, and to observe the reward or penalty they receive as a result of that vote. In this work we see the beginnings of a theory that may be able to explain how the colony of ants performs its tasks.



The Goore game rewards each member of a set of automatons independently with a probability given by the function $f(p)$, where p is the fraction of the set that votes YES at a given time. The automatons are completely unaware of the other automatons, do not know the function $f(p)$, and, remarkably, will collectively vote in a way that maximizes the payoff to all.

FIGURE 11. The Goore Game

NEEDED UNDERSTANDING AND TOOLS

In the previous section, we discussed a few of the things known about distributed-systems performance and behavior. A few isolated facts are indeed known, but overall theory and understanding are still lacking.

For instance we need considerably sharper tools to evaluate the ways in which randomness, noise, and inaccurate measurements affect the performance of distributed systems. What is the effect of distributed control in an environment where that control is delayed, based on estimates, and not necessarily consistent throughout the system? What is the effect on performance of scaling some of the system parameters? We need a common metric for discussing the various system resources of communications, storage, and processing. For example, is there a processing component to communications? We also need a proper way to discuss distributed algorithms and distributed architectures.

A microscopic theory that deals with the interaction of each job with each component of the system is likely to overwhelm us with detail and will fail to lead us to an understanding of the overall system behavior. It is similar to the futility of studying the many-body problem in physics in order to obtain the global behavior of solids. What is needed is a macroscopic theory of distributed systems, such as thermodynamics has provided for the physicist. In fact, Yemini [21] has proposed an approach for a macroscopic theory based on statistical mechanics that will lead to better understanding the global behavior of distributed systems without the need for a detailed, fine-grained analysis.

Another fruitful approach that also avoids the horrible details of any specific system structure must be credited to Shannon [18]. In analyzing the behavior of error-correcting codes for noisy communication channels, Shannon used the brilliant device of studying *all possible codes simultaneously*. This enabled him to average out the detailed structure of any given code. He could then take exquisite advantage of the law of large numbers in order to arrive at a precise statement regarding the error behavior of codes. It is likely that such an approach will allow us to study the behavior of "typical" topologies and algorithms in distributed systems.

LIKELY FUTURE DEVELOPMENTS

These are exciting times. Researchers in universities and laboratories around the world have begun to focus their attention on distributed systems. They come to this field from diverse disciplines ranging from queueing theory to neuroanatomy in which they are the experts. Thus, we have the ingredients for an enormously rich soup of separate ideas that have only just begun to blend.

As the theoretical frontiers are being assaulted, so too are the practitioners busily building systems. This is a double-edged sword. On the one hand, the implementation of real distributed systems in the hands of the designers and users provides us with a strong motivation for progress in understanding, as well as a magnificent test bed in which we can experiment. On the other hand, these systems are massively expensive and are being implemented without the benefit of the principles we seek. As a result, they may be colossal failures! The reality is that there is no way we can prevent their proliferation as manufacturers respond to the frenzied demand from the user community. In a sense

UNDERLYING PRINCIPLES OF DISTRIBUTED-SYSTEMS BEHAVIOR

- Developing innovative architectures for parallel processing
- Providing better languages and algorithms for specification of concurrency
- More expressive models of computation
- Matching the architecture to the algorithm
- Understanding the trade-off among communication, processing, and storage
- Evaluation of the speedup factor for classes of algorithms and architectures
- Evaluation of the cost-effectiveness of distributed-processing networks
- Study of distributed algorithms in networks
- Investigation of how loosely coupled self-organizing automatons can demonstrate expedient behavior
- Development of a macroscopic theory of distributed systems
- Understanding how to average over algorithms, architectures, and topologies to provide meaningful measures of system performance

we are all responsible for the current craziness, because we have been "promising" these miraculous systems to the user for almost a decade.

In the face of these developments, we can foresee some of the likely developments that will take place over the next decade or two. Let us first consider the likely technology developments in hardware-type resources. One of the most exciting of these is the huge data bandwidth projected for fiber-optic technology. These fibers are being used for point-to-point communication pipes at rates on the order of hundreds of megabits per second. Bell Laboratories and Japan have been leapfrogging each other in setting world records for the largest data rates transmitted over the longest distances. Earlier this year, Bell Laboratories established a new record by transmitting at the rate of 4 billion bits per second at a distance of 117 km without any repeaters! The product of data rate times distance has been doubling every year since 1975, and based on the limits imposed by physics, there are still five orders of magnitude to go (16 years of doubling left). The tiny glass fiber is so clear that, if the oceans of the world were made of this glass, one could see the bottom of the deepest trench in the ocean floor from the surface. If we consider a 1-mW laser and a requirement of 10 photons to detect 1 bit of information (high-quality detection), then a single strand of fiber should be able to support a data bandwidth of 10^{15} bits per second. That would provide, for example, a 100-Mbit-per-second channel to each of 10 million users—all on one thin strand! This light-wave technology is being installed across the United States right now. The Los Angeles 1984 Olympics video was transmitted from the games' remote locations to satellite transmitters using a fiber-optic network installed by Pacific Bell—perhaps the most well-known application to date. This technology is being applied to local area networks by a number of vendors, but the technology for this application is not yet mature because we have yet to develop an efficient way to optically tap into the light pipes at low loss. As soon as that problem is resolved (in the next two or three years), we are likely to see a rapid deployment of fiber-optic channels in our local network environment. As discussed earlier, enormous bandwidths are necessary, but not sufficient, for many tightly coupled systems. The latency introduced due to propagation delay can inhibit tight control. (E.g., if we transmit data into a 1-Gbit-per-second light pipe spanning the United States, the 15,000-microsecond propagation delay is such that the first bit will come out of the other end only after 15 million bits have been pumped in!)

This planet is currently laced with many types of computer/communications networks at all levels. There are wide area networks, packet-switched networks, circuit-switched networks, satellite networks, packet radio networks, metropolitan area networks, local area networks, cellular radio networks, and more; and they are mostly incompatible within each type and across types. At the same time, the end user's facility consists of telephones, data terminals, Host machines, PBX switches, alarm systems, video systems, FAX machines, etc. The incompatibility problem escalates! What is needed in a distributed system is a standard digital communication service to connect the many user devices with one another across the room or across the world. Fortunately, there is a worldwide movement to define and adopt an integrated solution to this problem, which has given rise to the Integrated Services Digital Network (ISDN). The ISDN service defines a customer interface (a plug in the wall) to which the user's devices can attach and gain access to the worldwide integrated digital network. We are not likely to see much definition and penetration of ISDN until the end of this decade and, possibly, into the next decade (and most likely it will first appear at the local network level).

What all this should tell us is that we are approaching a time when massive connectivity among devices and systems will exist. Such connectivity is necessary if we are to derive the full benefits from distributed systems.

At the processor technology level, perhaps the most dramatic development is the gathering momentum in the proliferation of personal workstations. They are spearheading the drive toward distributed systems. At the other end of the spectrum, parallel machine architectures are being proposed all over the world to increase the processing capacity that can be applied to a single problem. Both of these technologies are moving very rapidly and are putting pressure on distributed-systems research and development. We are seeing the development of massively distributed architectures that can be configured as tightly coupled, loosely coupled, or even hierarchically structured systems.

Massively distributed and massively connected systems with enormous computational capacity are likely to appear in the next 10 years. Unless we pay very careful attention to the user interface, users will be hopelessly lost and ineffective. At the very least, we must provide users with languages that allow them to take advantage of the distributed architecture and to write application code quickly and in a way that allows the application package to be modified and maintained easily. Moreover, the complexity of the system should be transparent to users. Users need to interface with a systemwide operating system that offers the use of a single logon (with a networkwide name and password) and that provides access to file servers, database servers, automatic backup, processing servers, mail servers, application packages, education and help functions, etc.

The system itself could take advantage of expert-systems capability in providing these services to the user. And the system is likely to include extensive redundancy in order to provide high levels of reliability and fault tolerance. It should also be self-repairing, and even self-organizing, as the conditions and demands on it change.

Aside from the business-oriented applications and developments listed above, an enormous consumer-oriented set of products will be developed. One device that spans business and personal needs is a proper "lap" computer that will provide the user with remote access to the massive distributed network resources described in this article.

We foresee a new phenomenon whereby users are confronted with so many attractive features in new devices and software packages that they cannot possibly learn to use them all. Learning how to use the features represents an investment far beyond users' available time; and yet the features are wonderfully seductive. To coin a term, I would like to refer to this phenomenon as "FEATURE SHOCK"!

As we observe the growth of our man-made distributed systems, we wonder how the ants, bees, birds, fish, and higher animals have managed to perform so well with their distributed systems. If we are ever to achieve a level of performance anywhere near theirs, we will have to further uncover the underlying principles of distributed-systems behavior (see sidebar). We have discussed some of these in this article, but there is much new ground to be broken. Almost anywhere you dig you are likely to find pay dirt. The field is wide open for new ideas and new approaches, challenging problems remain unsolved, and the application of new results will be widespread and rapid—what lovelier environment could you seek?

REFERENCES

1. Amdahl, G.M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of AFIPS*, Vol. 30, Thompson, Washington, D.C., 1967, pp. 483–485.
2. Belghith, A., and Kleinrock, L. Analysis of the number of occupied processors in a multi-processing system. UCLA CSD Rep. 850027, Computer Science Dept., Univ. of California, Los Angeles, Aug. 1985.
3. Denning, P.J. Parallel computation. *Am. Sci.* (July–Aug. 1985).
4. Ein-Dor, P. Grosch's law re-revisited: CPU power and the cost of computation. *Commun. ACM 28*, 2 (Feb. 1985), 142–151.
5. Ercegovac, M., and Lang, T. General approaches for achieving high speed computations. In *Supercomputers*, S. Fernbach, Ed. North-Holland, Amsterdam. To be published.
6. Gerla, M., and Kleinrock, L. Flow control protocols. In *Computer Network Architectures*, Paul Green, Ed. Plenum, New York, 1982, pp. 361–412.
7. Grosch, H.A. High speed arithmetic: The digital computer as a research tool. *J. Opt. Soc. Am. 43*, 4 (Apr. 1953).
8. Grossberg, S. *Studies of the Mind and Brain; Neural Principles of Learning, Perception, Development Cognition and Motor Control.* Reidel, Hingham, Mass., 1982.
9. Hwang, K. Multiprocessor supercomputers for scientific/engineering applications. (June 1985), 57–73.
10. Hwang, K., and Briggs, F. *Computer Architecture and Parallel Processing.* McGraw-Hill, New York, 1984.
11. Kleinrock, L. *Queueing Systems, Volume 2: Computer Applications*, Chap. 4. Wiley-Interscience, New York, 1976.
12. Kleinrock, L. On flow control in computer networks. In *IEEE Proceedings of the Conference in Communication*, Vol. 2, IEEE, New York, June 1978, pp. 27.2.1–27.2.5.
13. Kleinrock, L. On the theory of distributed processing. In *Proceedings of the 22nd Annual Allerton Conference on Communication, Control and Computing*, Univ. of Illinois, Monticello, Oct. 1984, pp. 60–70.
14. Kuck, D.J. et al. The effects of program restructuring, algorithm change and architecture choice on program. In *Proceedings of the International Conference on Parallel Processing*, Aug. 1984, pp. 129–138.
15. Minsky, M., and Papert, S. On some associative, parallel and analog computations. In *Associative Information Technologies*, E.J. Jacks, Ed. Elsevier North Holland, New York, 1971.
16. Patton, C.P. Microprocessors: Architecture and applications. *IEEE Comput. Mag. 18*, 6 (June 1985), 29–40.
17. Schneck et al. Parallel processor programs in the federal government. *IEEE Comput. Mag. 18*, 6 (June 1985), 43–56.
18. Shannon, C., and Weaver, W. *The Mathematical Theory of Communication.* Univ. of Illinois Press, Urbana, 1962.
19. Stuck, B.W., and Arthurs, E. *A Computer and Communications Network Performance Analysis Primer.* Prentice-Hall, Englewood Cliffs, N.J., 1985.
20. Tsetlin, M.L. *Automaton Theory and Modeling of Biological Systems.* Academic Press, New York, 1973.
21. Yemini, Y. A statistical mechanics of distributed resource sharing mechanisms. In *Proceedings of INFOCOM 83*, 1983, pp. 531–539.

CR Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms: Design, Performance, Theory

Additional Key Words and Phrases: computer networks, distributed control, distributed processing, economics of computing power, machine architecture, multiprocessing, parallel processing, performance of distributed systems, self-organizing systems

Author's Present Address: Leonard Kleinrock, Dept. of Computer Science, Boelter Hall, UCLA, Los Angeles, CA 90024.