Teletraffic Analysis and Computer Performance Evaluation
O.J. Boxma, J.W. Cohen, H.C. Tijms (Editors)

PERFORMANCE MODELS FOR DISTRIBUTED SYSTEMS

Leonard Kleinrock

Computer Science Department
University of California, Los Angeles
Los Angeles, California
U.S.A.

We discuss the relatively sad state of affairs regarding performance evaluation of distributed systems. We introduce some of the popular models, discuss the architecture and algorithm modelling, and then proceed to present a scattering of known results. Finally, a number of open areas are discussed which offer directions through which we hope to gain more understanding and insight into the operation and principles of these systems.

1. INTRODUCTION

We have finally seen the convergence of two giant technologies, namely, data processing and data communications. For years they had each been developing individually at a breakneck pace fueled largely by the needs demanded by the end user and by the enormous cost/performance improvements brought about by the semiconductor industry. Now they have merged, and they have created a technological environment whose potential is staggering.

The environment is one which is heavily distributed. We are beginning to see systems with distributed communications, distributed storage, distributed processing and distributed control. We are moving headlong into such structures in response to the user demands and in response to the manufacturers' desire to roll out products as rapidly as possible.

Unfortunately, we lack an understanding of the basic principles of distributed systems, principles which are badly needed to predict performance, to explain behavior and to establish design methodologies. We are even lacking the basic models to describe these systems. And the systems we are constructing are running into various performance problems in the form of high cost, poor response time, low efficiency, deadlocks, difficulty in growth, unproved operational algorithms, etc.

We are implementing too quickly for the theory, on the one hand, and too slowly for the applications, on the other. The situation is frustrating, due, in part, to the rapid rate at which technological change is occurring relative to the ability of our analytical science to keep pace with it. Even more frustrating is the recognition that Nature seems to have implemented some exquisite distributed systems which defy our understanding and analysis; further, these systems appear to be structured very very differently from the way we build our distributed systems today.

In this paper, we discuss some of the issues and performance measures, comment on architectures and algorithms, present some of the meager and scattered results which have been established, and then offer some open areas and problems.

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

2. ISSUES AND PERFORMANCE MEASURES

A distributed system consists of sources which are distributed and which create demands for service from a set of resources which themselves are distributed. As usual, more than one source may demand service from the same resource at the same time, causing a conflict. As queueing theorists, we know how to resolve such conflicts: we ask the demands to form an orderly queue to be served in some order, usually one at a time. This turns out to be a wonderfully efficient mechanism for resolving the conflict; indeed, it implements the demand access procedure known in the field of telecommunications as "statistical multiplexing."

Unfortunately, in a distributed environment, one cannot form a queue without incurring some overhead costs; this is the case because an individual source cannot automatically "see" who is on the queue and must therefore devote some effort to discover this information. This gives rise to the study of multiaccess algorithms, a field which has received some attention in the last few years. In spite of this work, there is still no comprehensive theory of multiaccess algorithms which explains the fundamental principles and observed phenomena. We will not devote much effort to that subject in this paper. Instead, we will focus here on the area of distributed processing. (The long term goal is still to understand the interaction of communication, storage, processing and control, but that global understanding is not yet within our grasp.)

One of the major issues in distributed processing is that of concurrency. Concurrency is a measure of the number of resources which can be utilized simultaneously. For example, in computer networks, a number of different communication channels can be transmitting messages at the same time. In packet radio systems, one introduces the notion of spatial reuse of a channel.

In distributed processing, the concurrency we discuss refers to the number of processors which can be active at a given time. The average number of busy processors is known as the "speedup," S, for a given problem. S measures how much faster a job can be processed using multiple processors, as opposed to using a single processor.

Another key measure of interest is the efficiency with which the processing resources are being used (i.e., the utilization factor). The utilization factor is proportional to the throughput, γ, of the system. The throughput (or carried traffic) is a function of the blocking probability, B, and the offered traffic, λ, through the relationship $\gamma = \lambda B$.

Clearly, the mean response time, T, is a major performance measure for our systems. The mean-response-time-versus-throughput profile is perhaps one of the most common and important profiles used in the study of congestion systems. Therefore, we see that the speedup, S, is also the ratio of the mean response time using a single processor to the mean response time using multiple processors.

It has been found convenient to define a performance measure which combines all of these other measures into a single measure which we call "power." The power, P, is defined as

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

Below, we discuss some of the fascinating properties enjoyed by this particular performance measure.

In quoting the results regarding distributed processing, we must distinguish a number of cases. We must specify whether we are talking about the response time and concurrency for a single job, or for a fixed number of jobs all of which are waiting for processing, or for a stream of jobs which arrive at random times. We must identify the number of processors which are available to work on the collection of jobs. We must know the way in which processors are assigned to jobs (i.e., scheduling and assignment). We must know if communication is instantaneous or congested, if it is unicast, multicast or broadcast, and we must know the communications connectivity among the

various system resources (e.g., processors and storage). The system description can easily get far more complex than our meager analytic tools can handle at present.

3. ARCHITECTURES AND ALGORITHMS

The world of applications, especially scientific applications, has an insatiable appetite for computing power. Any good mathematician can 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 which permit concurrent processing (a system design problem)
- optimizing compilers for detecting concurrency (a software engineering problem)
- algorithms for specification of concurrency (a language problem)
- more expressive models of computation (an analytic problem).

3.1 ARCHITECTURES

There are many ways of classifying machine architectures --- too many, in fact. We select the following classification, which we feel is appropriate for the purposes of this paper.

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 member consists of multiple processors which, collectively, can process multiple instruction streams on multiple data streams (MIMD). When the multiple processors cooperate closely to process tasks from the same job, we usually refer to this form of multiprocessing 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 the 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.

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 [1]. Two excellent, recent summaries of some of these projects are offered by Hwang [2] and by Schneck et al. [3].

3.2 ALGORITHMS AND MODELS OF COMPUTATION

The major goal in characterizing the algorithm is to identify and exploit its inherent parallelism (i.e., potential for concurrency). There are many levels of resolution at which we can attempt to find this parallelism, and we list these levels below in decreasing order of granularity [4].

- job execution
- task execution
- process execution
- instruction execution
- register-transfer
- 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, then 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, the computation we are processing. A very common model is the graph model of computation (the task graph) [5], which is normally used at the task or process level. 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 which can be exploited. See Figure 1 for an example.

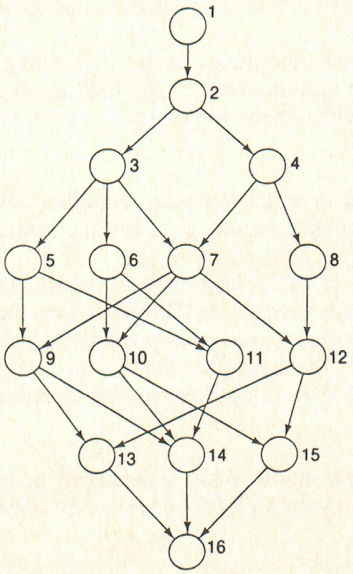


Figure 1
The Graph Model of Computation

Another common model of algorithms is the Petri Net [5]. See Figure 2 for an example. In this model, a bipartite directed multigraph is created with two sets of nodes, each of a different type.

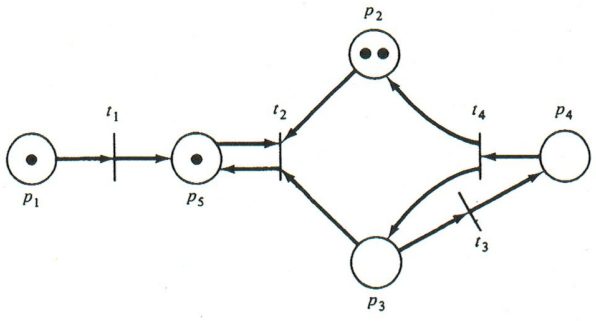


Figure 2
The Petri Net Model

Figure taken from Peterson, J.L., Petri Net Theory and the Modeling of Systems (Prentice-Hall, Englewood Cliffs, N.J., 1981, p. 17, Figure 2.11).

The first kind of node is called a place, say p_i, and is drawn as a circle; it represents a condition. The second kind of node is called a transition, say t_i, and is drawn as a bar perpendicular to its incident arcs; it represents a change in state (an event). The directed arcs represent dependencies between the events and the conditions on the nodes they connect. Tokens are located in certain of the places and such a token distribution is called a marking; the marking identifies the state of the system. The graph is initialized by placing a number of tokens at certain places in the graph. When all branches leading into a transition node contain at least one token (this transition has all of its conditions satisfied), then the transition "fires"; this firing action removes one token from each of the places feeding this transition (if more than one transition can fire, one is chosen at random) and puts one token in each place fed by the transition. In Figure 2, we can see that t_1 is ready to fire. The process continues to operate as satisfied transitions fire and tokens travel around the graph, causing the system to move from one state to another. Such models are often used to check for proper termination of algorithms. A number of extensions of Petri Nets have been developed. One major extension has been to introduce the dimension of time by labelling each transition with a quantity representing the time it takes for the transition to occur. Timed Petri Nets [6], (TPN), assume deterministic values for the transition delays. Stochastic Petri Nets [7], (SPN), allow for random transition times and lead to a Markov Chain analysis. Petri Nets are capable of modelling parallelism, conflict and synchronization of processes. An excellent document summarizing much of the recent work on Timed Petri Nets may be found in the Proceedings of the International Workshop on Timed Petri Nets (July, 1985) [8]. Unfortunately, one quickly gets entangled in the state space explosion problem with these models.

We will focus mainly on the former model (the graph model of computation) in this paper. Note that one may allow the graph model for a job to be drawn randomly from a set of possible graphs. Furthermore, we must specify the distribution of time required to process each task in the graph.

3.3 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 above, then we are faced with a number of architecture/algorithm problems, namely, partitioning, scheduling, memory access, interprocess communication and synchronization. The partitioning problem refers to the decisions we must make 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. The memory access problem refers to the mechanism provided for 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 available to provide processors access to the memory modules and to other processors; this may take the form of either an interconnection network in a parallel processing system, a local area network in a local distributed processing or shared data 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 which 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 which 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 systems where one of these resources can be exchanged for the 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) face the problem 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.

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

4. RESULTS

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 [9]. Of course, the idea is that, as we apply more parallel processors to the computational job, then the time to complete that job will drop in proportion to the number of processors, P.

4.1 SPEEDUP

Recall the definition of the speedup factor, S. 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 [10] 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 $S = 0.3\,P$ for certain programs by carefully extracting the parallelism in Fortran DO loops [11]. 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) [12]. He argues that if a fraction, f, of a computation must be done serially, then the fastest that S can grow with P is

$$S \leq \frac{P}{[Pf + 1 - f]}$$

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

The actual amount of parallelism (i.e., S) achieved in a parallel processing system is a quantity which 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 [13], 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 modelled 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, and is really based on the definition of the utilization factor; in some special cases, the distribution of the number of busy processors can be found, as well.

4.2 SOME CONFIGURATION ISSUES

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 which 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 consider is the regular series-parallel structure shown in Figure 3, where we have taken a total processing capacity of C MIPS (million instructions per second) and divided it equally into mn processors, each behaving as an M/M/1 queue, and each of C/mn MIPS. On entering the system, a job selects (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.

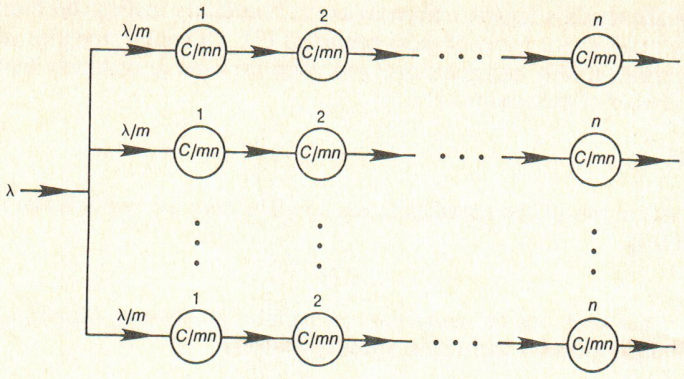


Figure 3
A Symmetrical Distributed-Processing Network

The key result for this system [14] 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! 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 aggregrate 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 aggregrate 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, $\rho =$ 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 $(\rho \to 1)$, $K = 1$. Is it the case that smaller machines are mn times as cheap as large 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 [15] 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 [16] shows that, if we consider all computers at the same time, Grosch's Law is clearly not true, as seen in Figure 4. In this figure, we see that microcomputers are a good buy. 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 currently can supply that many MIPS. Furthermore, once in the family, it pays to purchase the biggest member machine in that family (as predicted by Grosch).

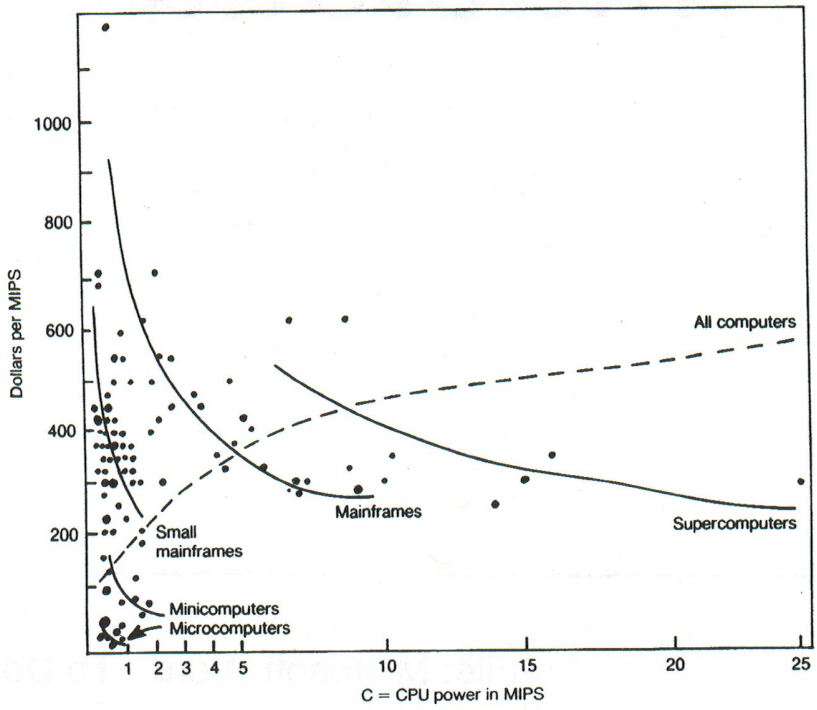
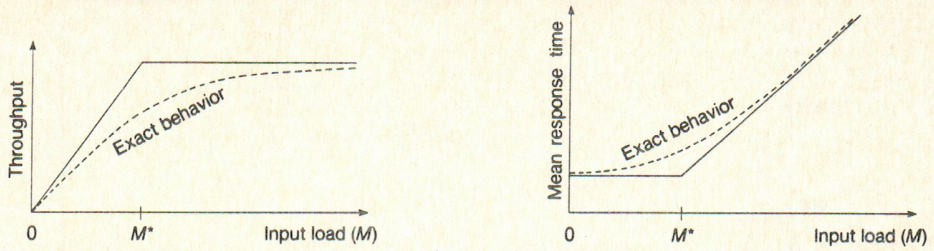


Figure 4
Economics of Computer Power

Figure taken from Ein-Dor, P., Grosch's law re-visited: CPU power and the cost of computation, Commun. ACM 28, 2 (Feb., 1985), 142-151.

4.3 DEADLOCKS AND POWER

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 multi-processor 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 which must be processed by certain of the system 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 this request is t_0 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 5. In this figure, the quantity M^* is defined as the ratio of the mean cycle time $T_0 + t_0$ to the mean time x_0 required on the critical resource in a cycle; T_0 is the mean response time when $M = 1$, and the critical resource is that system resource that is most heavily loaded [17]. To find the exact behavior (as shown in dashed lines in the figure) rather than the bounds, one must specify the distributions of the service time required by jobs at each resource in the system as well as the queueing discipline at each. That is, we must set up the closed queueing network model. Using the bounds or the exact results, one can easily see the effect of parameter changes on the system behavior. For example, one can examine the accuracy of the common rule-of-thumb which suggests that the proper mix of microprocessor speed, memory size and communication bandwidth is in the proportion 1 MIPS, 1 megabyte and 1 megabit-per-second; some suggest that we will soon see a 10,10,10 mix instead of this 1,1,1 mix.



(a) Bound on Throughput (b) Bound on Mean Response Time

Figure 5
Bounds on Throughput and Response Time

One can also interpret profiles such as that shown in Figure 5a as the carried traffic (γ) versus the offered traffic (λ). This function, $\gamma(\lambda)$, is often referred to as the "flow control" function in systems analysis.

Of course, we usually wish to display the mean response time, T, as a function of the throughput, γ, as shown, for example, in the familiar curve of Figure 6. As usual, we note the trade-off between these two performance measures.

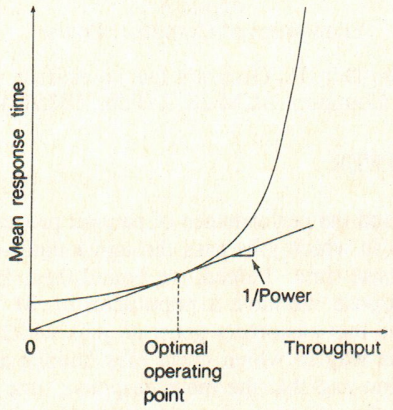


Figure 6
The Key System Profile and Power

We are immediately compelled to inquire about the location of the "optimal" operating point for a system. The answer depends upon how much you hate delay versus how much you love throughput. One way to quantify this love-hate choice is to adopt our earlier definition of power, $P = \frac{\gamma}{T}$, as our objective function. The operating point that optimizes (i.e., maximizes) the power (large throughput and small delay) is located at that throughput such that

$$\frac{dT}{d\gamma} = \frac{T}{\gamma}$$

when the derivative exists. Even in the discontinuous case, this optimum occurs where a straight line (of minimum slope) out of the origin touches the throughput-delay profile (usually tangen-

tially); such a tangent and operating point are shown in Figure 6. *This result holds for all profiles and all flow control functions*. Moreover, for all M/G/1 systems, 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 [18]. In the case of an M/M/1 queue, the optimum power point occurs at half the maximum throughput (50% efficient) and twice the minimum delay. It turns out that the average number in system is a natural invariant to consider when one maximizes power.

5. OPEN AREAS

In the previous section, we listed a few of the known results in performance evaluation of distributed systems. Other results do exist, and they come from many diverse fields. Nevertheless, one has the uneasy feeling, as stated earlier, that we have only scratched the surface. The underlying results have still not been uncovered. Much work needs to be done.

In this section, we present some intriguing results and ideas which perhaps will lead to some fundamental understanding.

A large general area of interest is that of distributed control systems. 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 IBM 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, for example, Expressnet, tree algorithms, urn models, and hybrid models. If the channel is multicast (or multi-hop), 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 automata for this discussion) with an internal algorithm which 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 automata 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 certain instants of time. The automata are not aware of each other's vote; however, there is a referee who can observe and calculate the percentage, p, of automata that vote YES. The referee has a function, $f(p)$ (such as that shown in Figure 7), where we require that $0 \le f(p) \le 1$. Whenever the referee observes a percentage, p, who vote YES, he will, with probability $f(p)$, reward each automaton, independently, with a one dollar payment; with probability $1 - f(p)$ he 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 automata have sufficient memory, then for the payoff probability shown in the figure, exactly 20% of the automata 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 which may be able to explain collective phenomena.

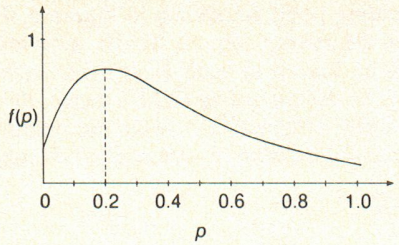


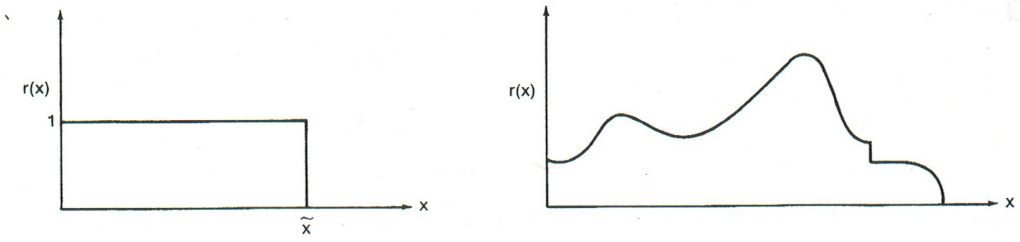
Figure 7
The Goore Game

Another fascinating study of late has to do with the subject of "common knowledge" [21]. It is best exemplified by the problem of two friendly armies camped on opposite hills overlooking an enemy army camped in the valley between them. Neither of the friendly armies can defeat the enemy by itself, but both together surely can. The general of the first friendly army ("army A") sends a messenger to the general of the second friendly army ("army B") one night, informing him that they are to attack the enemy at dawn. Unfortunately, the messenger must travel through the enemy valley and may not reach his destination. Now, suppose the messenger is successful. Will army B's general attack at dawn? Clearly not, since he must acknowledge to army A that he got the message in order for army A to be willing to attack. So he sends the messenger back; let us assume the messenger makes it again. Now will army A attack? No, since army B is not sure that they (army A) got the acknowledgement and if they did not get it, army B should not proceed. And so it continues, with acknowledgements of acknowledgements of acknowledgements, etc, ad infinitum. Neither army will ever attack! Neither has common knowledge which is defined to be knowledge that A knows that B knows that A knows that B knows .... all the way. Common knowledge is an important concept in distributed systems. In fact, it is related to our earlier comment in Section 2: the problem of forming a queue in a distributed environment is one of common knowledge, to a degree.

Most of us know how to play the game of 20 questions. Suppose I asked you to determine which number I was thinking of from the set 0,1,2,...,15. You could clearly determine the answer with only four yes/no questions by first determining which half of those contained my selection by asking if the number was less than 8. Based on the answer to the first question, you would then split the eligible 8 into two sets of 4, etc, finally converging on the answer in 4 questions. Now, could you simply ask four questions in parallel (i.e., write down the four questions to be answered)? Specifically, you could not make any of the questions depend on the answer to any of the others. The answer is yes (and it is trivial to see - it is left as a fun exercise for the reader). The fact that it is possible says that this type of problem lends itself very nicely to parallel processing with lots of concurrency. Some recent work has been reported in this area in [22].

The degree of coupling between processes affects their ability to take advantage of multiple processors. If they are completely uncoupled, then one can utilize as many processors as one has processes. If each process depends upon the other in some sequential fashion, the amount of concurrency may be severely limited. The models described in Section 3.2 expose some of this coupling. The challenge is to find a proper, useful measure of coupling among processes that can be used to predict the speedup they can achieve.

We have mentioned the trade-off between processing and communications. Indeed, we have been able to show [23] that the use of broadcast communications can assist significantly in the following situation. Suppose we have K sorted sublists in each of K distributed processors and we wish to merge and sort the entire list. There is a simple way to transmit the sorted sublists using broadcast communications in such a way that the receiving processors will accept the list elements in their properly sorted order; thus there will be no global sorting required after the merging (i.e., the transmission) has taken place.



(a) Classical Queueing (b) New Queueing Class

Figure 8
Desired Rate of Service as a Function of the Elapsed Time in Service

We close this section by introducing a new class of queueing problem which the author and two of his students (Abdelfettah Belghith and Jau-Hsiung Huang) have been studying with relatively little analytical success. Consider the task graph shown in Figure 1. Let us assume that each task requires a random amount of time to be processed (for simplicity, let us assume that each is exponentially distributed with rate μ. Further, assume that no more than one processor may work on a given task. When node (task) 1 is being processed, the job proceeds at a rate μ. When it completes, node 2 begins and again the job proceeds at a rate μ. When it finishes, nodes 3 and 4 both become activated and, assuming at least two processors are available, the job then proceeds at a rate 2μ, etc. What we see is that the rate at which a job can absorb work depends upon where it is in its processing cycle (i.e., which tasks are currently being processed) and also depends upon how many processors are available to work on the job (other jobs may be competing for a finite number of processors). Of course, we would like to handle the case of an arbitrary distribution of task times. The classic queueing model, shown in Figure 8a, assumes that the rate at which a job can absorb work is constant, and that the total amount of work is a random variable, $\tilde{x}$. Specifically, we plot $r(x)$, the desired rate of service (seconds per second) as a function of the elapsed time in service (assuming no competition for service). The total service time is the area under the curve. In Figure 8b, we show a general picutre for our new class of problem where the rate varies with elapsed time.

6. CONCLUSIONS

An overall theory and understanding is lacking for distributed systems behavior. 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 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 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, much as thermodynamics has provided for the physicist. In fact, Yemini [24] has proposed an approach for a macroscopic theory based upon statistical mechanics that will lead to better understanding the global behavior of distributed systems without a detailed, fine-grained analysis.

Massively distributed and massively connected systems with enormous computational capacity are likely to appear in the next ten years. Some of the directions which are needed to assist in the analysis and design of these systems are the following:

- 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 tradeoff 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 automata 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

It is through developments such as these that we hope to establish a unified theory for distributed systems.

The author takes pleasure in acknowledging the expert assistance and unlimited energy of Jodi L. Feiner in preparing this manuscript for publication. The many hours she devoted to this arduous task have helped produce this document.

REFERENCES

[1] Ercegovac, M., and Lang, T., General approaches for achieving high speed computations, Supercomputers, North-Holland, Amsterdam, to be published.

[2] Hwang, K., Multiprocessor supercomputers for scientific/engineering applications (June, 1985), 57-73.

[3] Schneck et al., Parallel processor programs in the federal government, IEEE Computer Magazine 18, 6 (June 1985), 43-56.

[4] Patton, C.P., Microprocessors: architecture and applications, IEEE Computer Magazine 18, 6 (June 1985), 29-40.

[5] Peterson, J.L., Petri Net Theory and the Modeling of Systems (Prentice-Hall, Englewood Cliffs, N.J., 1981).

[6] Ramchandani, C., Analysis of asynchronous concurrent systems by time Petri nets, Project MAC Technical Report MAC-RT-120, MIT, (1974).

[7] Molloy, M.K., Performance analysis using stochastic Petri nets, IEEE Trans. on Computers, vol. 31, no. 9 (1982) 913-917.

[8] Proceedings of the International Workshop on Timed Petri Nets (IEEE Computer Society Press, Maryland, 1985).

[9] Denning, P.J., Parallel computation, Am. Sci., (July 1985).

[10] Minsky, M. and Papert, S., On some associative, parallel and analog computations, in: Jacks, E.J. (ed.), Associative Information Technologies, (Elsevier North-Holland, New York, 1971).

[11] 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 (1984).

[12] Amdahl, G.M., Validity of the single processor approach to achieving large scale computing capabilities, in: Proceedings of AFIPS, Vol. 30, (1967).

[13] Belghith, A. and Kleinrock, L., Analysis of the number of occupied processors in a multiprocessing system, UCLA CSD Rep. 850027, Computer Science Dept., Univ. of California, Los Angeles, (August 1984).

[14] 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, October 1984.

[15] Grosch, H.A., High speed arithmetic: The digital computer as a research tool, J. Opt. Soc. Am. 43, 4 (April 1953).

[16] Ein-Dor, P., Grosch's law re-revisited: CPU power and the cost of computation, Commun. ACM 28, 2 (Feb. 1985).

[17] Kleinrock, L., Queueing Systems, Volume 2: Computer Applications, Chap. 4. (Wiley Interscience, New York, 1976).

[18] Kleinrock, L., On flow control in computer networks, in: IEEE Proceedings of the Conference in Communication, Vol. 2, IEEE, New York, June 1978.

[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] Halpern, J.Y., and Fagin, R., A formal model of knowledge, action, and communication in distributed systems: preliminary report, in: Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing (Ontario, August 1985).

[22] Traub, J.F., Wasilkowski, G.W., and Wozniakowski, H., Information, Uncertainty, Complexity (Addison-Wesley, Reading, Massachusetts, 1983).

[23] Dechter, R. and Kleinrock, L., Broadcast Communications and Distributed Algorithms, Technical Report, Computer Science Dept., Univ. of California, Los Angeles, 1984. To appear in IEEE Trans. on Computers.

[24] Yemini, Y., A statistical mechanics of distributed resource sharing mechanisms, in: Proceedings of INFOCOM 83, (1983) 531-539.