On The Performance of Parallel Computers
Order Statistics, Amdahl's Law, and Their Applications

Tao Zhang
Euler Solutions
Lester Lipsky
University of Connecticut
Yiping Ding
BGS Systems

We examine the time it takes to execute a number of tasks in parallel if the individual task execution times are not known, but are taken from some distribution, such as Uniform, Exponential and Power-tail. Using Amdahl's law as the framework for comparison, it is clearly shown that system performance and efficiency for massively parallel systems *depend strongly on the distribution. Therefore no general statements about speedup are possible without some specific knowledge of the distribution of task times. We also examine the much more difficult problem of parallel systems where the number of tasks is greater than the number of processors, and queueing must result. It is shown that only when the ratio of tasks to processors is large, does speedup approach its optimum. As an application, we also show how the framework developed in this paper can be used to model the process of running benchmarks for parallel computers or computer systems.*

1 Introduction

In recent years, the word *Parallel* has become one of the most used words in computer science research. Every area from *Parallel Operating Systems* to *Parallel Architectures* to *Parallel Algorithms* to *Parallel Languages and Compilers*, every branch seems to be dominated by it. There certainly are at least two good reasons for this interest. First there is the availability of very inexpensive and relatively fast CPU chips. Second there is the recognition that development of faster and faster single-processors is getting increasingly expensive, and in any case is approaching its upper bound. There is the general belief that increasing the number of processors is in some way equivalent to increasing the speed of a single processor in the same proportion, and in many cases this seems to be true.

For systems where there are many independent jobs competing for limited resources, the performance expectations have been realized (e.g., computers in MP mode), and the analytical models have worked well. But now the emphasis is on breaking a single job into many tasks, and hoping that the tasks can and will be executed simultaneously. The degree to which this assumption holds is dependent upon the application, and in most cases does not necessarily result in a linear increase in speed. One simple, and valid reason was given by Amdahl long ago, which has now become a *law*, widely known, understood, and ignored. Another reason for less than optimal performance is a lack of understanding of, what is called in probability theory, *Order Statistics* and one of its parallel algorithm manifestations, the so-called *fork-join construct*. If k tasks start simultaneously, the time until the last one is done could be much longer than the mean time for their average. Obviously, the time to do the k tasks sequentially is *exactly* k times their mean. But, by definition, a job is not done until all its tasks are finished, so the savings in time-to-completion may be nowhere near that hoped for.

Up to now much of the research and development on parallel systems has been done either with a very specific application in mind, or without a clear picture of what the intended user profile will be. Therefore, the models and simulations that have been developed are subject to question as to their general conclusions. What measurements have been made of the running of real programs (e.g., PVM and LINDA) report a typical speedup on the order of only 3 to 5 when using 15 to 20 CPU's. The "explanation" as to why the speedup is so low usually is given as "communication cost". Yet, even a crude anal-

ysis which takes order statistics into account, will give the same order of speedup and efficiency. The analysis strongly depends upon the viewpoint.

For instance, consider any one processor. When it finishes a task, it initiates an I/O interrupt, and sends its results to the appropriate receiving node, together with a request for more data. It then waits for the new data to arrive. If one assumes that communication is the bottleneck, then the entire delay will be charged to communication delay. But what if the next set of data has not been prepared yet, i.e., it is still being computed? Then the problem has to do with a lack of computational synchronization. One view for improving performance says "Put in more and/or faster data links." The other says "Divide the computing load more evenly." Who is right? Only more detailed measurements can decide. Whether we admit it or not, the model one has of a given system (whether analytic, or intuitive, or simulation) helps determine what measurements should be made, and in turn interprets what those measurements mean. Therefore, it is important that researchers in the field get a better understanding of parallel behavior so that better measurements and models can be constructed.

In this paper, we will give a short review of Amdahl's law, and order statistics, and provide some examples of what realistic speedups to expect. In particular, we will look at Uniform, Exponential, and Power-tail distributions in detail. Then we will examine the much more difficult problem where the number of tasks, k, exceeds the number of processors, C. In this case the tasks must queue up for service. This is equivalent to *Mean time to drain* of a G/C queue with no new arrivals, and in Reliability Theory, to *Mean time to failure, with hot and cold backup*. We will also present some examples of this, showing how it can be used to model benchmarking for parallel systems. For consistency in understanding, we will put the results in the framework of extensions to Amdahl's Law.

We use the following terminology. A *job* is a self-contained entity which can be broken up into *tasks*, some of which can run independently and some of which must be run in some order. When necessary, we will call the former *parallel tasks* and the latter *sequential tasks*. The *length* of a job or task is the time it takes to execute the job or task alone on a single processor.

2 Amdahl's Law

In 1967 [AMDA67], G. M. Amdahl made a simple and insightful argument about the limits of speedup from parallelization. He recognized that no matter what the job, some things must be done sequentially. i.e., some tasks just have to be done before others. Define the following:

T_t := Total time for a job to execute on one processor;

T_p := Time of that part of job which can be broken into independent tasks;

T_s := Time for that part of job which must run sequentially;

By definition then, $T_t = T_p + T_s$. Suppose that the parallel portion can be broken into k equal and independent parts. Next suppose that at least k processors are available, and that all k parts run in parallel. In general, let $T(k)$ be the total time it would take for a job to execute in the enhanced system (e.g., installing k parallel processors). Then, $T(1) = T_t$, and according to Amdahl's argument,

$$T(k) = T_s + T_p/k,$$

One parameter which gives a measure of improvement of performance made by parallelizing is the ***Speedup***, defined as

$$\mathcal{S}(k) := \frac{T_t}{T(k)}$$

and with Amdahl's assumption yields his law:

$$\mathcal{S}(k) = \frac{T_t}{T(k)} = \frac{1}{(1-\beta) + \beta/k}, \quad where \quad \beta := \frac{T_p}{T_t}.$$

Obviously, $\mathcal{S}(1) = 1$ and $1 \leq \mathcal{S}(k) \leq k$. But what this simple formula tells us (and what Amdahl intended to convey) is that no matter how many processors are used, the speedup can never exceed $1/(1-\beta)$. So, for instance, if $\beta = .95$, we can only gain at most a factor of 20 in speedup. Only if $\beta = 1$ (no sequential part), will the speedup equal k.

Another measure of improvement is the *Efficiency*, defined as $\mathcal{E}(k) := \mathcal{S}(k)/k$, which according to Amdahl's Law is

$$\mathcal{E}(k) = \frac{\mathcal{S}(k)}{k} = \frac{1}{\beta + (1-\beta)k}.$$

This tells a similar story. Only if $\beta = 1$ does $\mathcal{E} = 1$. Suppose instead that $\beta = .95$ and $k = 20$. Then $\mathcal{S}(20) = 10.26$, and $\mathcal{E}(20) = .513$. Which of the two parameters one should use depends on the application. For instance, if the computing resources are limited, then $\mathcal{E}(k)$ is the appropriate parameter. But if throwing processors at a problem is not an issue, then $\mathcal{S}(k)$ is the appropriate parameter. In this example, for instance, as k increases, the system approaches a speedup of 20 with an efficiency of 0.

Even if we were able to deal exclusively with CPU-bound jobs, and we had tightly coupled, memory sharing processors, use of Amdahl's Law in this simple form is doubly flawed. First of all, Amdahl's assumption that one could get a k-fold speedup of the parallel portion of a job is, more often than not, a gross exaggeration. But it encourages researchers and developers to focus too much on how to push β closer and closer to 1. So, for instance, many researchers in parallel algorithms have searched for ways to parallelize inherently sequential processes, using *Massively Parallel Computers.*

Our purpose here is not to criticize such research, but rather to refocus on the other issue. To do this, we modify Amdahl's Law in the following way. Let $T_p(k)$ be the actual time it takes to execute the parallel portion in a k-parallel system. Clearly, $T_p(1) = T_p$, and ideally, we would hope that $T_p(k) = T_p\,/k$. Instead, we have the *Quality Parameter*,

$$\mathcal{Q}(k) := \frac{k\,T_p(k)}{T_p}. \tag{1}$$

By definition $\mathcal{Q}(1) = 1$, and ideally, $\mathcal{Q}(k)$ should be 1 for all k. But in practice it is greater than 1 (it would be extraordinary if $\mathcal{Q}(k)$ could be less than 1). Then Amdahl's Law becomes

$$\mathcal{S}(k) = \frac{1}{(1-\beta) + \beta\,\mathcal{Q}(k)/k},$$

or

$$\mathcal{E}(k) = \frac{1}{\beta\mathcal{Q}(k) + (1-\beta)k}. \tag{2}$$

In the following sections we will examine $\mathcal{Q}(k)$ under different assumptions, and then give some graphical comparisons.

3 Massively Parallel Systems

One of our purposes in this paper is to show the affects of delays because multiple tasks executed in parallel must wait for each other to finish before going on to the next set of tasks. Thus we are assuming that computation times are much larger than communication times, so the distinction between shared-memory, and message-passing local memory systems is unimportant. This is probably the opposite of what many researchers are concentrating at present.

For the purpose of this paper we use the following definitions, thereby separating the two types of mathematical procedures which must be used.

Definition 1 A *Uniprocessor* is a system in which all tasks must be done sequentially.

Definition 2 A *Multiprocessor* or *Parallel Processor* is a system which allows tasks to be carried out in parallel. A *k-Processor* can handle a maximum of k tasks simultaneously.

Definition 3 A *Massively Parallel System* is one in which all tasks, no matter how many, can be done in parallel.

Note that these definitions refer to the hardware system. If a job is submitted that is programmed to do everything in series, then the three systems would behave identically. If a job cannot be parallelized to ever have more than k simultaneous tasks, then a k-processor and a massively parallel system would behave identically. A massively parallel system is a k-processor system where $k = \infty$, and a uniprocessor is a k-processor with $k = 1$.

3.1 Order Statistics

Suppose there are a very large number of tasks that might be performed, and they are of widely varying lengths. The *Probability Distribution Function* (*PDF*) of their lengths can be thought of in the following way. Suppose a task is taken at random from the set of tasks. Then the probability that it will take less than time t to execute is $F(t)$. When the total number of tasks is very large, this corresponds to the fraction of tasks which have length less than or equal to t. Mathematically it is written as follows. Let T be a *Random Variable* which denotes the length of a task selected at random. Then

$$F(t) := Pr(T \leq t).$$

The *Reliability Function* associated with these tasks is defined as

$$R(t) := Pr(T > t) = 1 - F(t),$$

and the *probability density function* (*pdf*) is defined as

$$f(t) := \frac{dF(t)}{dt}. \tag{3}$$

Then the *Expectation Value of* T is given by

$$\mathrm{E}(T) := \int_0^\infty t \cdot f(t)\,dt.$$

We would *expect* that the *mean* (average) length of a large number of tasks selected at random to be close to $\mathrm{E}(T)$. In general, the n^{th} *moments* of $f(t)$ (if they exist) are given by

$$\mathrm{E}(T^n) := \int_0^\infty t^n \cdot f(t)\,dt.$$

Now suppose that a job selects $k > 1$ tasks at random from the large set of tasks, and we want to know how long it would take to execute all of them (i.e., how long until the job is done). As long as they are executed one-at-a-time, our intuition agrees very well with reality. We know that their sum should be close to $k\,\mathrm{E}(T)$.

Suppose there are k processors available and all k tasks start executing simultaneously. Let $\{T_j | j = 1, \cdots k\}$ denote the execution times for each of the j tasks. How long will it take now until they are all finished? The job is done when every task is done. Or, put somewhat differently, the job is not done until the task that takes the longest, is done. Obviously (except for the *Deterministic Distribution*, where all tasks take exactly the same time), some tasks take less than average, and some take more than average. Therefore, the time for the job to complete will be longer than the mean time for a task. How much longer requires some careful analysis. We give some more definitions so as to be consistent with standard texts on the subject (e.g., [TRIV82]). Let $Y_1, Y_2, \cdots, Y_j, \cdots, Y_k$ be the same set of random variables as the T_j's, except that they are in size place. That is, Y_1 is the smallest of the T_j's, Y_2 is the second smallest, and Y_k is the largest. We have,

$$Y_j \leq Y_{j+1} \quad \text{for} \quad 1 \leq j < k.$$

In particular, we can write

$$Y_1 = \min_j T_j \quad \text{and} \quad Y_k = \max_j T_j.$$

The study of the various properties of the Y_j's is called *Order Statistics*, of which we are primarily interested in two, namely Y_1 and Y_k.

Let $R_{Y_1}(t)$ be the reliability function for the shortest task. This is the probability that the shortest task is not finished by time t. Clearly, this must be the same as the probability that *none* of the tasks are finished by time t. Since the tasks are independent, and described by the same distribution, we have

$$R_{Y_1}(t) = [R(t)]^k. \tag{4}$$

In a similar fashion we can argue that the probability that the longest task will be finished by time t is the same as the probability that *all* tasks are finished by time t, or

$$F_{Y_k}(t) = [F(t)]^k. \tag{5}$$

We are finally ready to determine the mean time for the job to finish when all k tasks run simultaneously.

We must compute $\mathrm{E}(Y_k)$. There are two ways to do this. One is to find $f_{Y_k}(t)$ by differentiating (5), and then using (3). The other approach is to integrate (3) by parts, coming up with the general formula

$$\mathrm{E}(T) = \int_0^\infty R(t)\,dt, \tag{6}$$

which for us becomes

$$T_p(k) = \mathrm{E}(Y_k) = \int_0^\infty \left(1 - [F(t)]^k\right)\,dt. \tag{7}$$

As simple as this formula looks, it cannot be used directly, since it is the integral of the difference of two terms, each of which integrates to ∞. Which formula to choose depends on the function to be integrated, and the ingenuity of the integrator. We present some examples in the following sections.

3.2 Tasks With Uniformly Distributed Service Times

The *Uniform* distribution is a convenient function to use here since it is so easy to integrate, and since it *does* occur in computer science applications. For instance, the *Linear Search* algorithm has a time for completion taken from a uniform distribution. Also, the time to find the beginning of a file on a fixed-head disc is uniformly distributed. Let $2\bar{T}$ be the maximum time the task can take. Then the mean time for the task is $\bar{T}$. The pdf for this process is

$$f(t) = \begin{cases} 1/(2\bar{T}) & 0 \leq t \leq 2\bar{T} \\ 0 & \text{otherwise} \end{cases}$$

It follows that

$$F(t) = \begin{cases} 0 & t \leq 0 \\ t/(2\bar{T}) & 0 \leq t \leq 2\bar{T} \\ 1 & 2\bar{T} \leq t \end{cases}$$

(7) can be used here because the integrand is 0 for $t > 2\bar{T}$, so we get

$$T_p(k) = \mathrm{E}(Y_k) = \int_0^{2\bar{T}} \left[1 - \left(\frac{t}{2\bar{T}}\right)^k\right] dt$$

$$= 2\bar{T} - \frac{1}{k+1}\frac{(2\bar{T})^{k+1}}{(2\bar{T})^k} = \frac{2k}{k+1}\bar{T}. \tag{8}$$

When this result is put into (1), remembering that $T_p = k\bar{T}$, we have the quality parameter

$$Q(k) = \frac{k}{k\bar{T}}\frac{2k}{k+1}\bar{T} = \frac{2k}{k+1}. \tag{9}$$

We see that $Q(k) > 1$ for $k > 1$, and approaches 2 as k increases. In other words, when there are a large

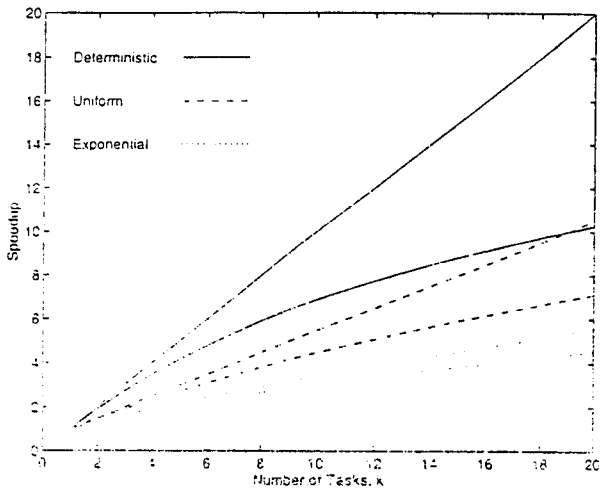

Figure 1: Speedup In Massively Parallel Systems, As a Function of k, the Number of tasks. The uppermost curve corresponds to tasks which are all exactly the same size (Deterministic distribution). The other two distributions are the uniform, and exponential. As expected, if any part of a job must be done sequentially ($\beta < 1$), all speedups reach a maximum value of $1/(1-\beta)$.

number of tasks, at least one of them will need close to the maximum time to complete. So Equations (2) yield the usual saturation values when $\beta < 1$ but even if $\beta = 1$ the speedup is

$$S(k) = \frac{k+1}{2}$$

and the efficiency approaches

$$\lim_{k\to\infty} \mathcal{E}(k) = \frac{1}{2}.$$

We see then, even with this rather well-behaved function, only about one-half the speedup hoped for will be realized, and at 50% efficiency. It is characteristic of distributions with finite upper limits that $\mathrm{E}(Y_k)$ should approach that upper bound for large k.

Figure (1) compares the ideal case ($\mathcal{Q}(k) = 1$) with the uniform and exponential distributions, for $\beta = .95$ and 1. The exponential distribution is described in detail in the next subsection. It is characteristic of those distributions which have unbounded upper limits for the task time (such as the exponential) to *usually* finish in a reasonable time. For instance, for the uniform distribution, 1/2 the time a task will finish in less than the mean time. On the other hand, for exponential distributions, almost 2/3 of the time a task will finish in less than its mean time. But it also means that some non-negligible number of tasks must be much larger than average. When several tasks run in parallel, it is hard to avoid having one of them being very large. And of course, the more there are, the larger will be the largest. Hence the poor speedup characteristics for large k.

3.3 Tasks With Exponentially Distributed Service Times

The exponential distribution is very useful in so many areas of analytic modeling because it has mathematical properties which allow certain classes of problems to be solved which would be insoluble for other distributions. Is it realistic in the context used here? Sure. There are problems which are well represented by exponential distributions, particularly those which involve random searches (with repeats) of large sample spaces. Even if they are not very realistic, combinations of them can be used to approximate the real system. Even when a poor approximation is made (most of the time, one doesn't know what the distribution is, anyway) one can get a qualitative answer, where *none* is available otherwise. A surprising property concerning conditional properties described in Feller [FELL71] implies that if a job is broken up into k pieces in a random way, then the mean size of the largest piece depends on k in the same way as that for exponential service times! This in turn, implies that automatic parallelizing compilers may not be able to yield speedups better than that described in this section.

The PDF, pdf, and reliability functions for exponential distributions are given by

$$F(t) = 1 - e^{-\mu t}, \quad f(t) = \mu e^{-\mu t} \quad \text{and} \quad R(t) = e^{-\mu t}$$

respectively. Some properties are:

$$\mathrm{E}(T) = \frac{1}{\mu} \quad \text{and} \quad \sigma^2 = \frac{1}{\mu^2}.$$

We say that μ is the *service rate*, or the *probability rate* for completion. The most important property for our use is its memorylessness. That is, even if a task has been running for some time, but is not finished, the distribution of the time remaining is the same as if it just began. Let B be the event that the task is not finished by time $x+t$, and let A be the event that the task is not finished by time x. Then, from the law of conditional probabilities, we have:

$$Pr(B \mid A) = \frac{Pr(B \cap A)}{Pr(A)} = \frac{R(x+t)}{R(x)}$$

$$= \frac{e^{-\mu(x+t)}}{e^{-\mu x}} = e^{-\mu t} = R(t).$$

Another useful property follows. If k identically distributed exponential tasks are running in parallel, then

from (4) the reliability function for Y_1 is

$$R_{Y_1}(t) = e^{-k\mu t}.$$

That is, the time for the first task to finish is *exponentially distributed*, and

$$\mathrm{E}(Y_1) = \frac{1}{k\mu} = \frac{1}{k}\mathrm{E}(T).$$

Good use will be made of these properties soon. But first we attempt to evaluate $T_p(k)$. We have already seen that the first task finishes in a mean time of $1/k\mu$. At that moment there are $k-1$ tasks still running. By the memoryless property, it is as though they just began. So the mean time until the second one finishes is $1/[(k-1)\mu]$. We continue in this way until there are none left. Adding all this up, we get

$$T_p(k) = \frac{1}{k\mu} + \frac{1}{(k-1)\mu} + \cdots + \frac{1}{2\mu} + \frac{1}{\mu} = \frac{1}{\mu}\sum_{j=1}^{k}\frac{1}{j}.$$

The summation term is known as the *Harmonic Series*, and satisfies the following important formula

$$H(k) := \sum_{j=1}^{k}\frac{1}{j} = \log(k) + \gamma + \frac{1}{2k} - \frac{1}{12k^2} + \mathrm{O}\left(\frac{1}{k^3}\right)$$

where $\gamma = .577\,215\,664\,901\,532\,860\,606\,512\cdots$ is Euler's constant. We then see from (1), and noting that $T_p = k/\mu$,

$$\mathcal{Q}(k) = k\,\frac{H(k)}{\mu}\frac{\mu}{k} = H(k) = \log(k) + \gamma + \mathrm{O}\left(\frac{1}{k}\right).$$

So, in this case, $\mathcal{Q}(k)$ grows unboundedly with k. Therefore, the speedup will grow more slowly with increasing k, and the efficiency will go to 0, even when $\beta = 1$. That is,

$$\mathcal{S}(k) \Longrightarrow \frac{k}{\log(k) + \gamma}$$

and

$$\mathcal{E}(k) \Longrightarrow \frac{1}{\log(k) + \gamma} \text{ for } \beta = 1. \qquad (10)$$

Similar behavior is seen in analyzing ideal parallel algorithms for adding a list of numbers, and other binary tree constructs. But in those cases, one can predict which processors will be idle, and when, and therefore, those processors could possibly be assigned to something else. The example presented here represents a much broader class of algorithms, for which one cannot predict when a task will finish.

The curves for exponential task times are plotted on Figure (1), showing clearly that parallel performance for exponential distributions is much worse than for the uniform distribution. There are many plausible classes of functions which can make performance even worse (e.g., distributions with large variances, and power-tail distributions). In fact, the entire region below the ideal curves can be filled in with curves corresponding to plausible distributions.

3.4 Power-Tail Distributions

In the previous section we showed that parallel performance can be significantly degraded if the parallel tasks vary greatly in size, as is the case for exponentially distributed task times. In this section we consider the worst possible cases among those classes of jobs that are guaranteed to finish. These are known as *power-tail* distributions. Such jobs do exist, and do, or will occur more frequently than one might expect. In simplest form they have the following asymptotic property: $R(x)$ is a power-tail distribution if there exists some real number, $\alpha > 0$ such that:

$$\lim_{x\to\infty} x^{\ell}\,R(x) = \infty \quad \forall\ \ell \geq \alpha$$

and

$$\lim_{x\to\infty} x^{\ell}\,R(x) = 0 \quad \forall\ \ell < \alpha.$$

Then the density function behaves like $f(x) \propto x^{-(\alpha+1)}$ for large x. This in turn means that $\mathrm{E}(x^{\ell}) = \infty\ \forall\ \ell \geq \alpha$. As pathological as this may seem for practical distributions, this behavior has been observed in numerous applications, including distribution of CPU times [LELA86], [LIPS86], size of files stored on disc [GARG92], and arrival of packets on ethernets [LELA94]. Details about these distributions can be found in [GREI95].

As an example, let the reliability function for task times be given by:

$$R(x) = \left(\frac{\alpha - 1}{x + \alpha - 1}\right)^{\alpha}.$$

As long as $\alpha > 1$ this distribution has a mean of 1. From (7) we have

$$\mathrm{E}(Y_k) = \int_0^{\infty}\left(1 - \left[1 - \left(\frac{\alpha-1}{x+\alpha-1}\right)^{\alpha}\right]^k\right)dx$$

$$= (\alpha - 1)\int_0^{\infty}\left(1 - [1 - u^{\alpha}]^k\right)du.$$

By variable substitution, this can be further manipulated to

$$\mathrm{E}(Y_k) = \frac{\alpha-1}{\alpha}\int_0^{\infty}\frac{1 - v^k}{(1-v)^{1+1/\alpha}}dv$$

$$= \frac{\alpha-1}{\alpha}\int_0^{\infty}\frac{\sum_{\ell=0}^{k-1} v^{\ell}}{(1-v)^{1/\alpha}}dv.$$

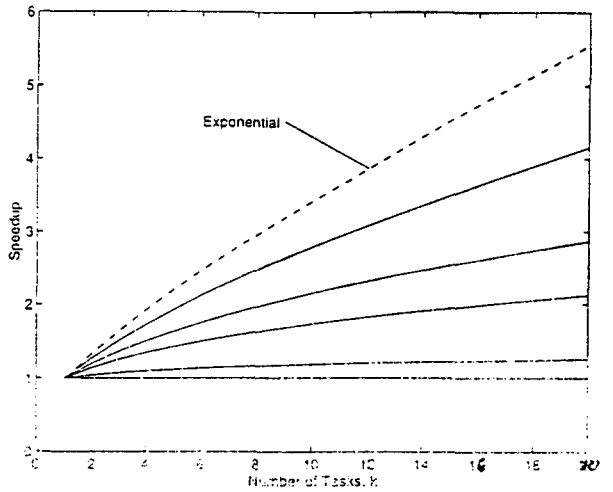

Figure 2: Speedup In Massively Parallel Systems, As a Function of k, Where Task Times Are Power-Tail distributed.

This equation plainly shows that $\mathrm{E}(Y_k)$ depends on k only through the upper limit of the sum. Therefore we can write the recurrence relation:

$$\mathrm{E}(Y_{k+1}) = \mathrm{E}(Y_k) + I_k,$$

where $I_0 = 1$ and for $k > 0$

$$I_k = \int_0^\infty \frac{v^k\, dv}{(1-v)^{1/\alpha}} = \frac{\alpha k}{\alpha k + \alpha - 1} I_{k-1}.$$

The last recursive relation came from integrating by parts. It can be shown [LIPS95] [and (1)] that

$$\mathrm{E}(Y_k) = \mathcal{Q}(k) \Longrightarrow k^{1/\alpha}.$$

Comparing with (10) we can see that the speedup for large k is much worse than that for exponential distributions. In fact, for $\alpha \longrightarrow 1$ there is *no speedup at all*!

The speedup for various values of α is displayed in Figure (2), with some previous curves repeated for comparison. One should not think that this behavior is too pathological to occur. Any recursive search algorithm with an indeterminate number of nodes could exhibit this property. Also, certain simulations of systems where *time-for-first-return* is computed simultaneously over many statistical paths behave in this way (e.g., random walks). Keep in mind that as parallel technology improves, increased efforts will be made to run ever new, bigger, and more complicated problems.

4 Multiprocessor Systems

In this section we discuss the much more common situation where the number of processors available is less than the number of tasks which are ready to execute, so they must queue up. We put this subject last because it is far more difficult to analyze than the "all-at-once" case. In fact, there are no standard formulas [analogous to (7)] available to compute the mean time to complete a set of k tasks when there are only C $(1 < C < k)$ processors available. We call this time $\mathrm{E}(Y_k|C)$. From the definition, $\mathrm{E}(Y_k|k) = \mathrm{E}(Y_k)$ and $\mathrm{E}(Y_k|1) = k\bar{T}$. Looked at in this way, we know how to do it for $C = 1$ and for $C = k$ ($C > k$ is meaningless), but not in between. It must be kept in mind that C refers to the system (number of processors), and k refers to the job (number of tasks, or degree of parallelizability). Equations (1) and (2) must be modified, since they were constructed assuming that $C = k$. $T_p(k)$ becomes $\mathrm{E}(Y_k|C)$, while T_p is still $k\bar{T}$, where $\bar{T}$ is the mean time for each task. Therefore, the quality parameter becomes

$$\mathcal{Q}(C|k) = \frac{C\,\mathrm{E}(Y_k|C)}{T_p}. \tag{11}$$

Ideally, if all tasks take exactly the same time, then $\mathrm{E}(Y_k|C)$ becomes $\lceil k/C \rceil \bar{T}$, where $\lceil \cdot \rceil$ is the *ceiling function*. In this case, $\mathcal{Q}(C|k) = 1$ whenever k is a multiple of C. Similarly, Equations (2) are rewritten to

$$\mathcal{S}(C|k) = \frac{1}{(1-\beta) + \beta\, \mathcal{Q}(C|k)/C}$$

and

$$\mathcal{E}(C|k) = \frac{1}{\beta \mathcal{Q}(C|k) + (1-\beta)C}. \tag{12}$$

When $C = k$ these equations reduce to Equations (1) and (2). In the next subsection we show how $\mathrm{E}(Y_k|C)$ can be calculated if the tasks are exponentially distributed. And in the subsection following, we describe this process as a transient M/G/C queue, from which the *mean time to drain* (also described as *mean time to failure without repair*) can be computed for *Matrix-Exponential* (ME) distributions.

4.1 Tasks With Exponentially Distributed Service Times

We have already set up all the mathematics needed to calculate the mean time to process k identically distributed tasks when only $C < k$ processors are available. Suppose there are C processors available when $k > C$ tasks arrive simultaneously. The system can only process C of them at first, so in a mean time of $1/(C\mu)$ one of them finishes. Now there are $k-1$ tasks left. If $k-1 > C$ then once again, C tasks are processed. In another mean time of $1/(C\mu)$, the second task is finished. Execution continues in this way until there are C tasks

left. Then we have as many processors as we have tasks, and the rest of the job can be finished in average time, $H(C)/\mu$. This yields a total time of

$$\mathrm{E}(Y_k|C) = \frac{k-C}{\mu C} + \frac{1}{\mu}H(C) = \frac{k}{\mu C} + \frac{1}{\mu}[H(C)-1]. \quad (13)$$

Notice that as k grows larger, the most significant term by far is $k/\mu C$, so the system behaves like a single server that is C times faster. This is what one would hope for, but is only achieved by keeping C small. It is true for all distributions, but for certain distributions, k would have to be extremely large indeed. This fact was known, in one form or another, to main-frame manufacturers many years ago, as recognized by the success of systems in MP mode. The number of processors rarely exceeded 8 ($C \ll k$).

Equation (13) also tells the opposite story when C is close to k, for in that case, performance is not improved much by adding more processors. Consider two systems running the same job consisting of k exponential tasks. One system is made up of k processors, and the other has one less. Then the fraction of change is

$$\frac{\mathrm{E}(Y_k|k-1) - \mathrm{E}(Y_k|k)}{\mathrm{E}(Y_k|k)} = \frac{1}{k(k-1)H(k)}. \quad (14)$$

For example, if a 4-task job is running on a 4-processor machine, only 4% is lost by removing a processor. For 10 processors and 10 tasks, the loss is less than .4%! The reason this is so follows. The first task finishes so early, that there is plenty of time for the last task to start and finish before all the others are done. Notice how different it would be if the tasks were all close to the same size. Then it would take twice as long to finish all the tasks with one processor removed. Clearly, the performance of such systems depends critically on the distribution of the task times.

The speedup and efficiency of C-processor systems can be calculated using (11) and (12). In Figure (3) the speedup of multiprocessor systems running jobs with exponential task times is plotted as a function of the number of tasks, for various values of C. Consistent with (12) and (13), the maximum speedup that can be attained is $C/[\beta + C(1-\beta)] < C$.

4.2 Other Distributions and M/G/C Queues

As long as we were dealing with tasks with exponential or deterministic distributions, the performance of a C-processor system could be calculated with relative ease. But all other distributions require such difficult calculations that they can only be done with some extensive

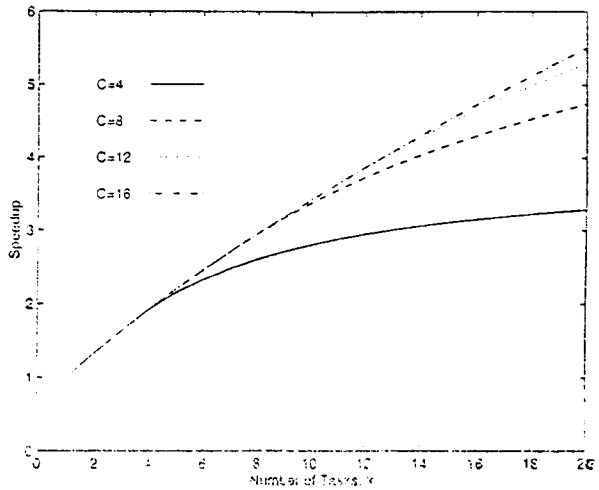

Figure 3: **Speedup of C-Processor Systems Running Jobs with Exponential Task Times, as a Function Of Number Of Tasks**: If $\beta = 1$ speedup approaches C as k increases. But when $\beta < 1$, speedup approaches $C/[\beta + C(1-\beta)] < C$.

coding. The difficulty comes in describing the state of those tasks remaining after one has finished. For exponential service times we could assume that they started all over again. But no other distribution has this memoryless property. So when a new task starts, the other tasks are in varied states of completion. The problem is virtually intractable when dealt with analytically. However, these problems have been treated in another guise, namely, the *Linear Algebraic approach* to queueing theory (LAQT) [LIPS83], for which programs have already been written [TEHR83], [ZHAN93]. We give a short description of the LAQT approach to M/G/C queues here, and refer the reader to [LIPS92] for further details.

4.2.1 Matrix Representation of Distribution Functions

Every distribution function can be approximated arbitrarily closely by some m-dimensional vector-matrix pair $< \mathbf{p}, \mathbf{B} >$ in the following way[1]. The PDF is given by

$$F(t) = 1 - \mathbf{p}\exp(-t\mathbf{B})\epsilon',$$

where ϵ' is an m-column vector of all 1's, and $\mathbf{p}\epsilon' = 1$. The matrix function, $\exp(-t\mathbf{B})$ is defined by its Taylor Series expansion, namely

$$\exp(-t\mathbf{B}) := \sum_{n=0}^{\infty} \frac{(-t)^n}{n!}\mathbf{B}^n.$$

[1]The material in this section can be skimmed over without loss in understanding the succeeding sections.

Note that if any $m \times m$ matrix $\mathbf{X}$ is multiplied from the left by a row m-vector, and from the right by an m-column vector, the result is a scalar (real or complex number). Since it appears often in LAQT, we define the specific product of this type as

$$\Psi[\mathbf{X}] := \mathbf{p}\,\mathbf{X}\,\epsilon'.$$

This clearly is a scalar, and therefore, $B(t)$ is a scalar function of t. From this it follows that the pdf is given by

$$b(t) = \Psi[\exp(-t\mathbf{B})\,\mathbf{B}],$$

and the reliability function is given by

$$R(t) = \Psi[\exp(-t\mathbf{B})],$$

It also follows that

$$\mathrm{E}(T^n) = n!\,\Psi[\,\mathbf{V}^n\,],$$

where $\mathbf{V} := \mathbf{B}^{-1}$. Because these formulas look so similar to the exponential distributions (replace μ with $\mathbf{B}$), the functions derived from them are called *Matrix Exponential* (ME) functions.

Imagine that the state of a task, as it is executed, can be represented by some row m-vector, say $\pi(t)$. Then at $t = 0$ its state is $\mathbf{p}$, and at some time t later, its state is given by $\pi(t) = \mathbf{p}\exp(-t\mathbf{B})$. The probability that it is still running at that time is simply $\pi(t)\epsilon' = R(t)$, as can be seen by the expression given above for the reliability function. The task is surely done when $\pi(t) = \mathbf{0}$. These vectors allow us to keep track of the evolution of a system as events occur (completion of tasks and initiation of new tasks).

Using only $\mathbf{p}$ and $\mathbf{B}$ one can construct matrices which represent the behavior of several tasks in various stages of simultaneous execution. For instance, the set of matrices, $\{\mathbf{B_j} \mid j = 2, \cdots C\}$ generate the completion of the next task when there are j tasks simultaneously executing, in a direct generalization of the above equations ($\mathbf{B_1} := \mathbf{B}$). For instance, let $\pi_\mathbf{j}$ be the composite state of the system at some time, then the state of the system some time t later (given that no new tasks have started) is $\pi_\mathbf{j}\exp(-t\mathbf{B_j})$, and the mean time until a task finishes is given by $\pi_\mathbf{j}[\,\mathbf{V_j}\,]\epsilon'_\mathbf{j}$, where $\mathbf{V_j} = \mathbf{B_j}^{-1}$. For our purposes here we introduce two more families of matrices. First, we have $\{\,\mathbf{R_j}\,\}$ ($\mathbf{R_1} := \mathbf{p}$). If there are $j-1$ tasks running at some time, represented by the vector $\pi_{\mathbf{j-1}}$ and a new task starts, the system is transformed to state $\pi_{\mathbf{j-1}}\mathbf{R_j}$. Next we have the set of matrices, $\{\,\mathbf{Y_j}\,\}$. These modulate what happens when a task completes. Suppose again that $\pi_\mathbf{j}$ is the state of the system at some moment, and eventually one task completes without any new task starting, then $\pi_\mathbf{j}\mathbf{Y_j}$ is the composite state of the remaining $j-1$ active tasks immediately afterwards.

Now we are ready to set up the equations needed to compute completion times. Suppose we have a computer system made up of C processors, and we wish to compute a job made of $k > C$ tasks. First, C tasks are started up simultaneously, putting the system in state

$$\mathbf{p_C} = \mathbf{p}\,\mathbf{R_2}\,\mathbf{R_3}\cdots\mathbf{R_C}.$$

The mean time until the first task is completed is given by

$$\mathrm{E}(Y_1) = \mathbf{p_C}\,\mathbf{V_C}\,\epsilon'. \tag{15}$$

Immediately after that, the system is in state $\mathbf{p_C}\,\mathbf{Y_C}$, and a new task begins execution $[\mathbf{p_C}\,\mathbf{Y_C}\,\mathbf{R_C}]$. The mean time for the next task to finish is given by

$$\mathrm{E}(Y_2) = \mathrm{E}(Y_1) + \mathbf{p_C}\,\mathbf{Y_C}\,\mathbf{R_C}\,\mathbf{V_C}\,\epsilon'.$$

In general, as long as there are still tasks in the queue waiting to start up, the mean time for the next one to leave is

$$\mathrm{E}(Y_{j+1}) = \mathrm{E}(Y_j) + \mathbf{p_C}(\mathbf{Y_C}\,\mathbf{R_C})^j\,\mathbf{V_C}\,\epsilon', \tag{16}$$

for $j \leq k - C$. Eventually there will only be C tasks remaining. Then, when one of them finishes, no new task will begin, and the state of the remaining $C-1$ tasks is $\mathbf{p_C}(\mathbf{Y_C}\,\mathbf{R_C})^{k-c}\,\mathbf{Y_C}$. The mean time until one of them leaves is given by

$$\mathrm{E}(Y_{k-C+2}) = \mathrm{E}(Y_{k-C+1}) + \mathbf{p_C}(\mathbf{Y_C}\,\mathbf{R_C})^{k-C}\,\mathbf{Y_C}\,\mathbf{V_{C-1}}\,\epsilon', \tag{17}$$

and so on until they are all done. As complicated as this looks, it is easy and efficient to compute. The time for each departure can be computed from the previous one by one matrix multiplication, and one vector dot product.

Of course, if the tasks are exponentially distributed, these equations reduce to those already given for exponential tasks. But the results for the uniform distribution cannot be reproduced exactly here because there is no exact ME representation of the uniform distribution. Good approximations are available, however. There is a computational problem in attempting to use these formulas to model ever larger parallel systems. The dimension of the largest matrices is given by

$$D(C,m) = \binom{m+C-1}{C}$$

which can be extremely large if both m and C are even moderate in size. For instance, $D(C,2) = C+1$, a very

manageable size. But for $m = 5$, $D(10,5) = 252$, and $D(20,5) = 15504$. This last number corresponds to a matrix which takes up almost one gigabyte of memory.

Even if the tasks making up a job are taken from different distributions, the above equations are valid, but now the dimension of the largest matrices is m^C, which for $m = 5$ and $C = 10$ is 9,765,625, or 400,000 gigabytes! So, in practice, this (or any other) procedure is limited to small values of m and C for heterogeneous tasks. Even if $m = 1$ (i.e., exponential servers with different service times) the problem can get rather big, since the identities of the individual tasks must be kept.

In the next section we show how the framework developed here can be used to model the process of running benchmarks for parallel computers.

5 Application: A Model for Benchmarking Parallel Computers

Benchmark programs are often used in the computer industry to compare the relative performance of a variety of computer systems. A mix of tasks (jobs) is used to test their performance. These tasks include integer processing, floating point arithmetic processing, file/database access, network access, and graphics processing. Some industry standard benchmark programs (e.g., SPECint95 and SPECfp95 from Systems Performance Evaluation Cooperative), mainly measure the CPU performance, while others, (e.g., TPC Benchmark A, B, C, and D from Transaction Processing Performance Council), measure the overall system performance under certain hardware and software configurations.

Although the benchmarks may vary by design for different testing purposes, they all have the following properties:

1. They consist of a fixed set of data and tasks.
2. The tasks are run on a targeted system, which normally idles before the benchmark tasks start and does not accept other arrivals until all the tasks finish.
3. The performance values of the system are then derived based on the duration of executing those tasks. The duration is called *the drain time*, or *the mean time to drain*.

In other words, a benchmark measures how long it takes for a set of tasks, which show up together at an idle system, to finish without other tasks entering and running in the system at the same time. Based on the benchmark properties and the framework developed in the previous section, we can establish a model for benchmarking and computing the drain time of parallel computers.

Since a non-trivial benchmark consists of many tasks and they are all ready to run, the order in which these tasks are run can have a considerable effect on the measured performance. Some operating systems may schedule the tasks based on their service times, while others may simply dispatch them randomly. A normal benchmark run can only indicate the system performance under a particular running sequence for the tasks. The analytical model proposed below takes into account the ordering effect and corresponds to the average of all possible orderings.

Assume that a benchmark has k tasks that can be run in parallel on a parallel computer system. We also assume that k is greater than the number of processors, C. (This is a very realistic assumption, since one does not want to test a system that has C processors with a benchmark that has less than C tasks.) From equations (15), (16), (17) and so on the mean time until the last task to complete, $\mathrm{E}(Y_k|C)$, which is $T_p(k)$, can be computed from the following equation:

$$T_p(k) = E(Y_k|C) = \sum_{i=1}^{k-C+1} \mathbf{p_C}(\mathbf{Y_C}\,\mathbf{R_C})^{i-1}\,\mathbf{V_C}\,\epsilon' +$$

$$\sum_{j=1}^{C-1} \mathbf{p_C}(\mathbf{Y_C}\,\mathbf{R_C})^{k-C} \left[\prod_{i=0}^{j-1} \mathbf{Y_{C-i}}\right] \mathbf{V_{C-j}}\,\epsilon'_{c-j}.$$

Using the $T_p(k)$ computed above, we can then derive the Quality Parameter $\mathcal{Q}(C|k)$ and the Speedup $\mathcal{S}(C|k)$ or the Efficiency $\mathcal{E}(C|k)$ from equations (11) and (12), respectively.

In addition to deriving the mean time to drain formula shown above, we can also use the equations introduced in the previous section to study the interdeparture times of individual tasks of the benchmark. The interdeparture times reveal more detailed information on how long it takes for a task to leave the system, especially at the beginning and the end of the benchmark runs.

The interdeparture times between tasks j and $j+1$ can be computed by

$$d_{j+1} = \mathrm{E}(Y_{j+1}) - \mathrm{E}(Y_j) =$$

$$\mathbf{p_C}(\mathbf{Y_C}\,\mathbf{R_C})^{j}\,\mathbf{V_C}\,\epsilon',$$

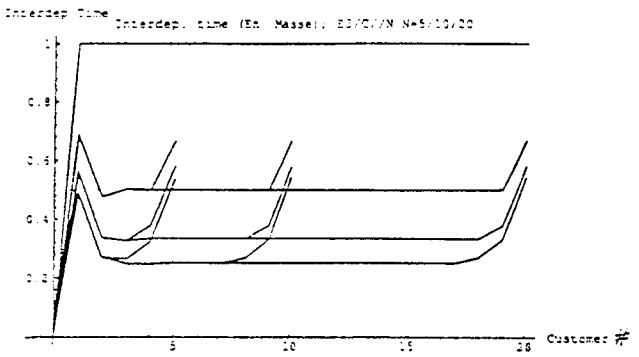

Figure 4: The interdeparture times of the tasks for the number of processors $C = 1, 2, 3$, and 4 with E_3 service time distribution.

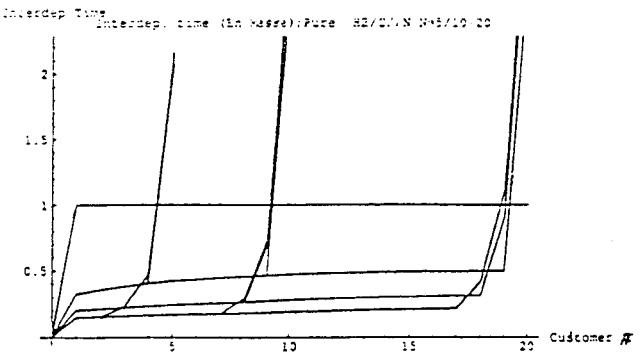

Figure 5: The interdeparture times of the tasks for the number of processors $C = 1, 2, 3$, and 4 with H_2 service time distribution.

for $1 \leq j \leq k - C$, and by

$$d_{j+1} = \mathrm{E}(Y_{j+1}) - \mathrm{E}(Y_j) =$$

$$\mathbf{p_C}(\mathbf{Y_C}\,\mathbf{R_C})^{k-C}\left[\prod_{i=0}^{j-k+C-1}\mathbf{Y_{C-i}}\right]\mathbf{V_{k-j}}\,\epsilon'_{k-j},$$

for $k - C + 1 \leq j \leq k - 1$.

As numerical examples, we calculate the interdeparture times for each of the $k = 5, 10$, or 20 tasks, assuming that the system has $C = 1, 2, 3$, or 4 processors, respectively. We selected two non-exponential distributions to represent the service times which have the same mean, but different variances, σ^2. One is the Erlang-3 distribution, E_3, with a mean of 1. The other is a Hyperexponential distribution, H_2, also with a mean of 1.0, but with a $\sigma^2 = 2.01939$. (This distribution has three free parameters, so the third parameter was fixed by setting the branch probabilities to 0.1 and 0.9.)

Figures (4) and (5) show the curves for the interdeparture times of the tasks with four different processor numbers $C = 1, 2, 3$, and 4. It is interesting to see that the interdeparture times approach constants after the initial tasks leave the system. After that the interdeparture times stay steady until the number of remaining tasks is less than the number of processors. From that point on, the interdeparture times increase. The increase in interdeparture times when the number of tasks is less than the number of processors is quite intuitive: The chance for the next task to finish at a given time reduces as the number of tasks remaining in the system becomes less. Therefore the interdeparture times elongate.

Note that for a single processor system ($C = 1$), the interdeparture times remain a constant, which is the mean service time, independent of the number of tasks in the system.

Note also that by knowing the interdeparture times for each of the tasks, we can easily compute the $T_p(k)$ ($= \mathrm{E}(Y_k|C)$), which is the sum of the interdeparture times plus the mean time for the first task to complete, i.e.,

$$T_p(k) = \mathrm{E}(Y_1) + \sum_{j=2}^{k} d_j.$$

The formulas as given here and in the previous section are also applicable to single-class Jackson Networks with C active customers, i.e., a system with MPL $= C$. It is known that a Jackson network with multiprogramming level (MPL) constraint invalidates the product-form solution. Our formulas, however, give the correct results.

6 Concluding Remarks

In this paper we described Amdahl's law, and through it examined just what is necessary for speedup to be proportional to k, the number of tasks which can run simultaneously. We made a conceptual distinction between *Massively Parallel* systems and *Multiprocessor* systems. We showed how the *Order Statistics* for the former can be calculated by standard probabilistic techniques, but the latter requires rather extraordinary, but not impossible, effort. The results indicated that if $C \ll k$, where C is the number of processes, then speedup approaches C. But if $C = k$ (the massively parallel case) then speedup will almost surely be much less than k, and efficiency will get worse with increasing k. As an application, we also presented a method to model the process of running benchmarks for parallel computers. The method and the computations involved can also be applied to solving Jackson networks with multiprogramming level (MPL) constraints.

References

[AMDA67] G. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities", *AFIPS Conference Proceedings* **30**, pp.483-485.

[FELL71] William Feller, *An Introduction to Probability Theory and its Applications*, Vol. II, John Wiley and Sons, New York, 1971.

[GARG92] Sharad Garg, Lester Lipsky and Maryann Robbert, "The Effect of Power-Tail Distributions on the Behavior of Time Sharing Computer Systems", *1992 ACM SIGAPP Symposium on Applied Computing*, Kansas City, MO, March, 1992.

[GREI95] "The Importance of Power-tail Distributions for Telecommunication Traffic Models", (Submitted for publication).

[LELA86] Will E. Leland and Teunis Ott, "Load-Balancing Heuristics and Process Behavior", *Proceedings of ACM SIGMETRICS 1986*, May 27 - 30, 1986, pp. 54-69. (The proceedings appeared as vol 14, no 1, Performance Evaluation Review, May, 1986.)

[LELA94] Will E. Leland, Murad S. Taqqu, Walter Willinger and Daniel V. Wilson, "On the Self-Similar Nature of Ethernet Traffic (Extended Version)", *Proc. of IEEE/ACM Trans. on Networking*, **2**,1, Feb. 1994.

[LIPS83] Lester Lipsky, "Explicit Solutions of M/G/C//N-Type Queueing Loops with Generalizations", *OPERATIONS RESEARCH*, 33, pp. 911-927, July 1985.

[LIPS86] Lester Lipsky, "A Heuristic Fit of an Unusual Set of Data", Bell Communications Research Report, January 1986.

[LIPS92] Lester Lipsky, *QUEUEING THEORY: A Linear Algebraic Approach*, MacMillan and Company, New York, 1992.

[LIPS95] Lester Lipsky and Pierre Fiorini, "Auto-Correlation of Counting Processes Associated with Renewal Processes", Technical Report, Booth Research Center, University of Connecticut, August 1995.

[TEHR83] Aby Tehranipour, *Explicit Solutions of Generalized M/G/C//N Systems Including an Analysis of Their Transient Behavior*, Ph.D. Thesis, Department of Computer Science, University of Nebraska, Lincoln, December 1983.

[TRIV82] Kishor S. Trivedi, *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*, Prentice-Hall,Englewood Cliffs, NJ, 1982.

[ZHAN93] Tao Zhang, *Transient Properties of Parallel Queues*, MS Thesis, Department of Computer Science and Engineering, University of Connecticut, December 1993.