Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

DISTRIBUTED SYSTEMS AND TRANSIENT PROCESSORS

Willard Robert Korfhage

September 1989
CSD-890057

UNIVERSITY OF CALIFORNIA

Los Angeles

Distributed Systems and Transient Processors

A dissertation submitted in partial satisfaction of the

requirement for the degree of Doctor of Philosophy

in Computer Science

by

Willard Robert Korfhage

1989

The dissertation of Willard Robert Korfhage is approved.

[illegible]

R. Baker

[illegible]

M. Ercegovac

[illegible]

E. Gafni

[illegible]

B. Rothschild

[illegible]

Leonard Kleinrock, Committee Chair

University of California, Los Angeles

1989

To Mom and Dad, and all my family.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

Oh, so many people to thank! Truly do our friends make us rich. At the top of the list is Dr. Kleinrock for providing inspiration and guidance ever since I arrived at UCLA. I am proud to have been part of his research group. Also I would like to thank the other dissertation committee members: Eli Gafni, Milos Ercegovac, Richard Baker, and Bruce Rothschild.

Just as important are my fellow graduate students, past and present, who provided much help and interesting conversation. Richard Gail deserves special note for starting events that lead to my taking position at Brooklyn Polytechnic University. My thanks also go to recent graduates and current students Jau Huang, Farid Mehovic, Bob Felderman, Chris Ferguson, Shioupyn Shen, Joy Lin, Simon Horng, Donna Neilsen, and John Lu. Special thanks go to the people who slaved over the Benevolent Bandit Laboratory (BBL), principally Eve Schooler and Bob Felderman.

The administrative staff for the group deserves my thanks for putting up with me while I've been here. Lily Chien and Cheryl Childress were always around to help with problems or fill out LVO's, and, when I first came to UCLA, Lillian Larijani kept the group running.

Thanks to the Advanced Research Projects Agency of the Department of Defense for funding this research (under contract MDA 903-82-C0064, Advanced Teleprocessing Systems, and contract MDA 903-87-C0663, Parallel Systems Laboratory), and further thanks to Hewlett-Packard and the American Electronics Association for their fellowship for doctoral students who intend to teach.

Over the years, the folks in the Locus and Tangram groups provided warm friendships, wonderful hikes, home-brewed beer, and help putting a new engine in my car. They are truly a multi-talented group.

Getting away from Computer Science, I would like to thank the UCLA sailing program for providing many wonderful hours, and to all the people who sailed the *Bruin Knight* for being great friends. Thanks to Tommy Jordan and the members of Cocu for showing me a side of LA I normally wouldn't have seen, and for letting me photograph them.

Haruko Shino deserves mention for making my last year at UCLA, and especially my final summer, quite wonderful.

And finally, but most importantly, my warmest thanks and appreciation go to my family, for it was their upbringing that enabled me to accomplish all I have done.

For the curious, this dissertation was typeset using TEXtures on a Macintosh II. Pictures were drawn in Canvas 2.0 and Aldus Freehand 2.0. Mathematica was used for mathematical analysis, and to generate the basic plots, which were then spruced up in the drawing programs.

VITA

August 22, 1960	Born, Ann Arbor, Michigan
1982	B.S. Princeton University
1985	M.S. University of California Los Angeles

PUBLICATIONS AND PRESENTATIONS

Kleinrock, L, and Korfhage, W., "Collecting Unused Processing Capacity: An Analysis of Transient Distributed Systems," *Proceedings of the* 9th *International Conference on Distributed Computing Systems*, June 5-9, 1989.

Korfhage, W., "Distributed Election in Unidirectional Eulerian Networks, With Application," Masters Thesis, UCLA Computer Science Department, 1985.

Korfhage, W. and Gafni, E., "Distributed Election in Unidirectional Eulerian Networks (Summary)," 22nd Annual Allerton Conference, 1984.

ABSTRACT OF THE DISSERTATION

Distributed Systems and Transient Processors

by

Willard Robert Korfhage

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1989

Professor Leonard Kleinrock, Chair

Distributed systems frequently have large numbers of idle computers and workstations. If these could be harnessed, then considerable computing power would be available at low cost. One can envision an environment in which large distributed programs, simulations, for example, use idle workstations to achieve a considerable speedup over the same program running on only one workstation.

This dissertation develops three models of such systems. First, we assume that we have a fixed amount of work to finish, W. We have M processors in a network, and each processor alternates between being *available* (for our work) and *non-available*. We find that in equilibrium, the number of available processors is binomially distributed. For systems in which $M = 1$, we find the Laplace transform of the distribution of finishing time, and, in a separate analysis as a cumulative, alternating renewal process, we find that the asymptotic (after a long time) distribution of the finishing time has a normal distribution.

To model the more realistic *multiprocessor* case ($M > 1$), we model the amount of work completed over time as Brownian motion with drift, using the asymptotic results of the single processor case, scaled to M processors, to supply the mean and variance of the Brownian motion. This yields the distribution of the time to complete W seconds of work, which is verified through simulation and by comparison with the single processor models. We develop extensions to this that allow a variety of different characteristics among the processors.

Following this, we turn to information theory and sorting algorithms. We examine four sorting algorithms (bubblesort, mergesort, quicksort, and radix sort) using information theory to understand how entropy decreases as the sorts progress, and why mergesort and quicksort are such efficient algorithms.

Finally, we examine some parallel versions of bubblesort, to see what potential there is for speedup and possible techniques that we may use.

CHAPTER 1

Introduction

1.1 Background.

Networks of computers are fairly common in business and research environments throughout the world. Originally motivated by a desire to ease data and device sharing, many networks have grown in speed and sophistication to the point that distributed processing can be performed on them. These networks vary in size from a handful of personal computers on a low-speed network, to networks consisting of thousands of workstations and a variety of larger machines on a high-speed, fiber-optic network. As a typical example, consider a network of workstations on a high-speed network in a research laboratory. Not only are there many machines, well connected by the network, but the users are likely to demand more and more computing power.

Networks of workstations have grown in spite of theoretical considerations that would discourage them. It is well known in queueing theory [Kle85] that a single server of large capacity shared by many users provides better response time than many smaller servers with the same total capacity. In terms of computers, this says that a mainframe will provide faster service to its users than a network of workstations. Of course, one may argue that for the same amount of money one can buy more workstation capacity than mainframe capacity, but in many real-world circumstances, if you compare systems of equal cost, then the network is still slower. Clearly factors other than speed drive the purchase of workstations, but we will not discuss these reasons. Instead, what we would like to find a way to use the network as a single large, shared server and gain back the power of large, mainframe systems without sacrificing the benefits of having workstations.

The key to this is idle time. On these networks, we often have the situation that many of the personal computers and workstations are sitting idle, waiting for their users, and thus being wasted. If we could recover this wasted time for useful processing, then we would have considerable computing power available to us at low cost. We refer to these processors, which are sometimes busy and sometimes not, as *transient* processors.

Whether this is technically feasible or not depends on a variety of factors, such as the properties of the communications medium, the properties of the computers, and the statistical characteristics of the user population.[1] In all systems of this type, one concern is that the "owner" of a machine should not see any degradation in performance because of the background programs. Any background computation will be aborted when user activity is detected, and not restarted until the system is sure that the machine is idle. Several systems to

[1]There are also other important but non-technological factors, such as people's resistance to the use of "their" machine, that would determine if and how a distributed system would be implemented. This dissertation will not examine such matters.

perform background distributed computations have been implemented.

At UCLA, the Benevolent Bandit Laboratory (BBL) [FSK89] runs distributed computations under MS-DOS on a network of IBM PC-ATs. A special shell runs on each machine, and when a machine is at the operating system prompt level (as opposed to running a program), it is available for use. If someone runs a program on a machine currently being used in a background, distributed computation, the system can select and start a replacement machine from the pool of idle processors. Because the system was also intended for the investigation of distributed algorithms, special features, such as the ability to mimic any topology, and some distributed debugging facilities, have been built in.

Condor ([LLM88], [ML87a], [ML87b]) is a very successful system running on workstations at the University of Wisconsin. It allows users to execute jobs at otherwise idle workstations.

The Butler system [Nic87], running on Andrew workstations at Carnegie-Mellon executes programs remotely on idle workstations. The system uses this to run *gypsy servers*, which are network servers that run on idle workstations instead of a fixed machine.

Worms [SH82] were developed at Xerox PARC for similar reasons. Worms prowled the network, collecting idle workstations and using them to perform some action, typically displaying a message or running a diagnostic program.

Note that of these four systems, only BBL and the worms were developed as distributed systems, whereas Condor and Butler are, at this time, more like remote job execution facilities.

There have also been ad-hoc attempts to use the idle time on processors. Dr. Tim Shimeall [Shi89], during his dissertation research, wrote a program "polite" that ran a software analysis program on workstations when no one was logged in and suspended the program when the workstation was being used. He finished 10 CPU years of work in 6 months on 20 workstations using this program. But again, this was very much a simple remote job execution facility, put together out of need, and never was it analyzed.

1.2 Outline.

This dissertation contains three topics. The first subject is the analysis of BBL-type systems, the second subject is an examination of sorting algorithms using information theory, and the third subject is bubblesort and some parallel versions of it.

In chapter 2, we examine BBL-type systems, developing three models of these systems and analyzing them to find the distribution of time to finish a fixed amount of work.

The first model is a single processor with general *available* and *non-available* times. We examine the number of non-available periods interrupting a program, and from this we find the distribution of the time to finish a program, its Laplace transform, and from the latter, the distribution's mean and variance.

The second model is also a single processor, but it is analyzed as a cumulative, alternating renewal process. We find that the asymptotic distribution of the accumulated work (over a long period of time) is normal, with simple expressions for it mean and the variance.

The third model handles multiple processors and views the amount of work done over time as Brownian motion with drift. We scale to M processors the asymptotic mean and variance of the accumulated work (from the second model), and use this as the mean and variance of the Brownian motion. From this we get the probability density of the time to finish a fixed amount of work. The mean and the variance agree very closely, for $M = 1$, with the first model.

We close this chapter by using the results for the finishing time distribution in models of the network as a whole.

In chapter 3, we examine sorting algorithms by considering sorting as a method of decreasing entropy. Bubblesort, mergesort, quicksort, and radix sort are examined to bound the decrease in uncertainty as these sorts progress, and from this we learn where the efficiencies and inefficiencies of the various algorithms lie.

Finally, chapter 4 examines some parallel versions of bubblesort, to see what potential there is for speedup, and what techniques may be of use in this algorithm.

CHAPTER 2

Transient Processors

2.1 Introduction.

Supposed that we wish to run distributed programs on a network of transient processors. This chapter analyzes the performance characteristics of such programs; by doing so, we can discover the potential that awaits us in our now-underutilized networks of workstations.

We first discuss our model of the network and program in section 2.1.1 and define characteristics of interest to us. Following this, section 2.2 examines the number of available machines in the network, and we show that this has a binomial distribution. Next we analyze the behavior of a single program in a network of transient processors and find the probability density of its finishing time using three models. The first two models (section, 2.4.3 and 2.4.4) examine a network consisting of only a single processor; the third model (section 2.5) extends this to multiple processors by using Brownian motion. We then ease some of our assumptions is section 2.5.9 and develop extensions to the multiprocessor model that allow more complex models of programs, and processors with differing characteristics.

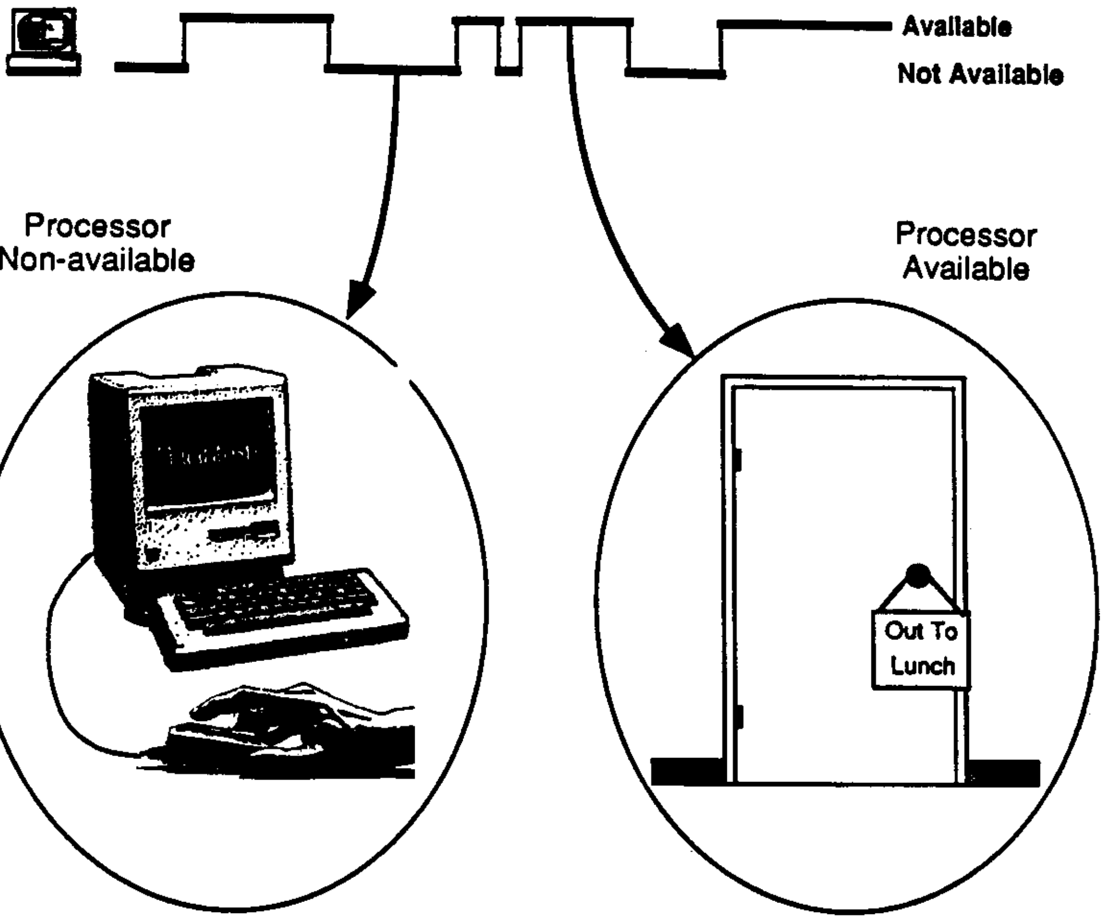


Figure 2.1: A processor alternates between available and non-available periods.

2.1.1 The Model.

2.1.1.1 The Network.

Assume that we have a network of M identical processors, each of which has a capacity to complete one minute of work per minute. A processor alternates between a ***non-available state*** (signified by n or na), when the owner is using it (e.g. typing at the keyboard), and an ***available state*** (signified by a or av), when it is sitting idle. The lengths of non-available periods are independent and identically distributed (i.i.d.) random variables from distribution $N(t)$, with mean t_n, variance σ_n^2, and corresponding density $n(t)$; we allow any general distribution for $N(t)$, unless otherwise specified. Likewise, available periods are i.i.d. random variables from a general distribution $A(t)$ with mean t_a, variance σ_a^2, and density $a(t)$. At the end of this chapter, we extend the analysis to allow for non-identical processors.

2.1.1.2 The Distributed Program.

In general, a program consists of multiple stages of work, each of which must be completed before the start of the next (Figure 2.2). The time to finish a program is the sum of the times to complete the individual stages. The time to finish a stage depends only on the amount of work in that stage, and is independent of the other stages. This means that the probability distribution of the *total* time to finish a program is the convolution of the distributions of the individual stage finishing times. Assuming that the network characteristics do not change during the execution of the program, then we only need to analyze the time to finish a single stage requiring W minutes of work. Using this result we find the finishing time probability density function of all other stages, and from there, the finishing time probability density function of the program as a whole.

We make the simplifying assumption that the work in any stage is infinitely divisible — it can always be divided evenly among all available processors. We

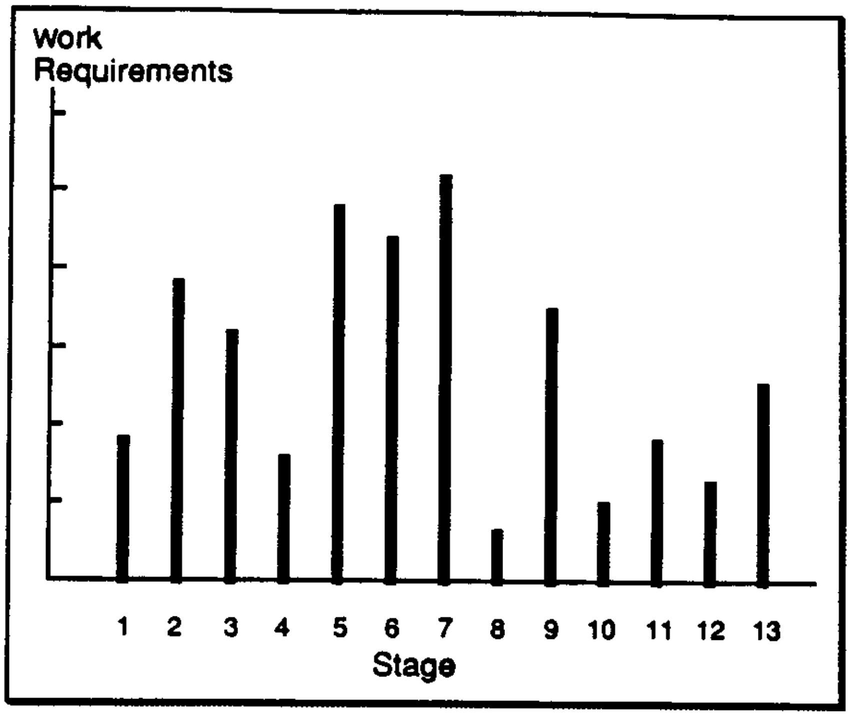


Figure 2.2: Execution time profile of an algorithm.

also ignore overheads that occur in a real system (e.g. communication delays, processing delays), and thus our model provides an optimistic bound on system performance. In the conclusion of this dissertation, we discuss a method for including overhead in our models.

2.1.2 What We Seek.

The purpose of this chapter is to find the function $f(t)$ that is the pdf of a program's finishing time. Also of interest are its mean, $\bar{f}$, and its variance, σ_f^2.

In the process of finding these, we will need another function, the pdf of the amount of work accumulated by a processor or network of processors over time. We denote this by $w(u \mid t)$, the probability that after t minutes of time have elapsed, the processor or processors under consideration has accumulated u minutes of work. This function has mean $\overline{Y_t}$ and variance $\mathrm{Var}[Y_t]$.

2.1.3 Notation.

We use the abbreviations "PDF" to stand for "probability distribution function" and "pdf" to stand for "probability density function." Typically we use capital letters for a PDF and lower case letters for a pdf. If $F(x)$ is a PDF, then $f(x) = \frac{\partial}{\partial x} F(x)$.

Define the following symbols:

$a(t)$ the pdf of the length of an available period.

$A(t)$ the PDF of the length of an available period.

$\mathrm{Av}^*(s)$ the Laplace transform of the available period pdf.

t_a the average length of an available period.

σ_a^2 the variance of the length of an available period.

$n(t)$ the pdf of the length of a non-available period.

$N(t)$ the PDF of the length of a non-available period.

$\mathrm{Na}^*(s)$ the Laplace transform of the non-available period pdf.

t_n the average length of a non-available period.

σ_n^2 the variance of the length of a non-available period.

$p_a \; = t_a/(t_a + t_n)$,

$p_n \; = 1 - p_a = t_n/(t_a + t_n)$,

M the number of processors.

W the total amount of work to do.

$w(u\,|\,t)$ the pdf of accumulated work.

$\overline{Y_t}$ the mean accumulated work in t minutes.

$\mathrm{Var}[Y_t]$ the variance of accumulated work in t minutes.

$f(t)$ the pdf of a program's finishing time.

$\overline{f}$ the mean program finishing time.

σ_f^2 the variance of a program's finishing time.

We take the unit time to be 1 minute.

2.1.3.1 Example Parameters.

Mutka and Livny, [ML87a], made actual measurements of a network of transient processors, and they developed models for the available and non-available period densities. From these measurements, we derive two examples that we use throughout this chapter.

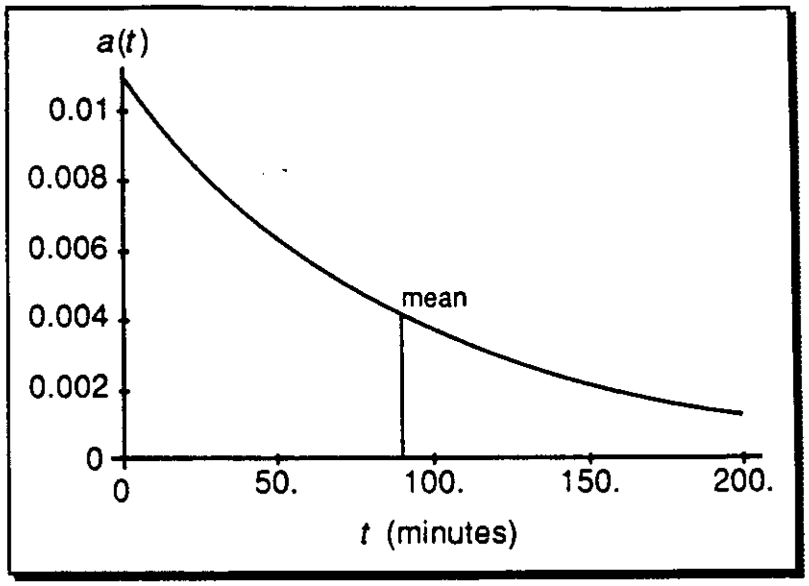


Figure 2.3: Mutka and Livny's available time pdf.

For the available time PDF, they use a 3-stage hyperexponential distribution:

$$\begin{aligned} A(t) &= \text{P[length of an available period} \leq t] \qquad (2.1) \\ &= 0.33(1 - e^{-(t/3)}) + 0.4(1 - e^{-(t/25)}) + 0.27(1 - e^{-(t/300)}) \quad t \geq 0, \end{aligned}$$

which has a mean of $t_a = 91$ minutes and a variance of $\sigma_a^2 = 40225$ minutes^2. Figure 2.3 shows the corresponding density.

The non-available time, $N(t) = \text{P[length of a non-available period} \leq t]$, is a 2-stage hyperexponential distribution:

$$N(t) = \begin{cases} 0.7(1 - e^{-(t/7)}) + 0.3(1 - e^{-(t/55)}) & \text{if } t \geq 7 \\ 0 & \text{if } 0 \leq t < 7 \end{cases} \qquad (2.2)$$

which has mean $t_n = 31.305$ minutes and variance of $\sigma_n^2 = 2131.83$ minutes^2. The 7 minutes shift in the distribution arises because a processor was not declared

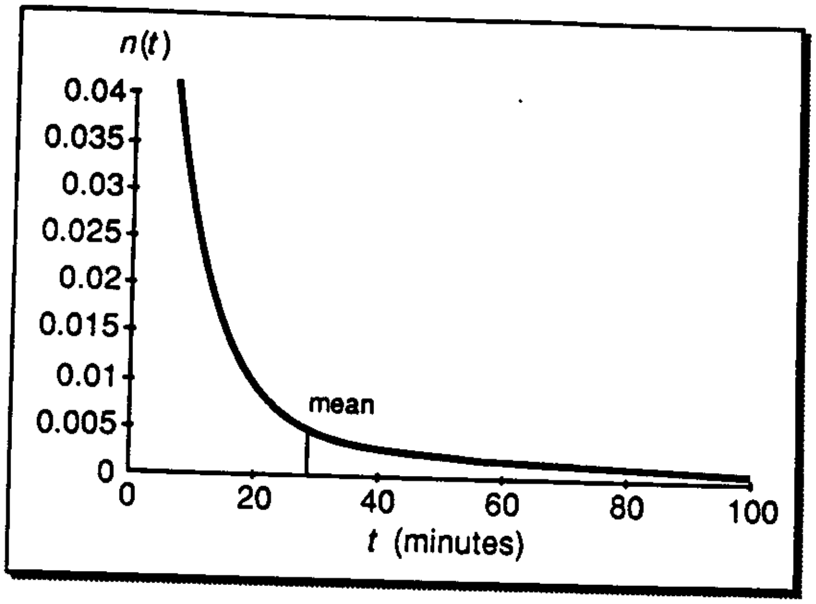


Figure 2.4: Mutka and Livny's non-available time pdf.

idle until 7 idle minutes had elapsed. Figure 2.4 shows the corresponding density.

We will use Mutka and Livny's distributions where ever possible, but frequently we assume exponentially distributed available and non-available periods. At such times, we will take the means of these exponential distributions to be the numbers given above. The use of exponential distributions instead of hyperexponential distributions will not affect any means that we derive, but any variances that we find will be lower than if we used Mutka and Livny's distributions.

Regardless of which distributions we use, we take W = 1000 minutes and $M = 1$ for single processor examples, and, W = 10000 minutes (almost 7 days) and $M = 100$ for most multiple processor examples. The reason for the large values of W is explained later in this chapter.

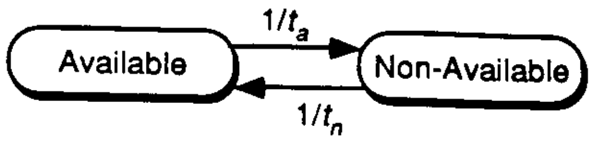


Figure 2.5: Two state model of a processor.

2.2 Distribution of Number of Available Processors.

One question of immediate interest to us is the probability density of the number of available processors in our network.We will show that any state-based model of a processor ultimately leads to a binomial density of the number of available processors.

2.2.1 Two-State Model of a Processor.

The simplest model of a processor is the one given in the introduction: a processor alternates between available and non-available states at constant rates (Figure 2.5). Each processor in the network is available a fraction of the time, $p_a = t_a/(t_a + t_n)$. The probability that k out of M processors are available is

$$\mathrm{P}[N = k] = \begin{pmatrix} M \\ k \end{pmatrix} (p_a)^{M-k}(p_n)^k$$

which is the binomial density. Note that these arguments have been made without reference to the distribution of available or non-available times, only to the average times. This result, therefore, holds for general available and non-available time distributions.

2.2.2 Multi-State Model of a Processor.

We may wish to make a more complex model of a processor, with many states and transitions, as in Figure 2.6. Given transition rates between states, we could solve for the steady state probabilities p_i that a processor is in any particular state i. For our purposes, a processor is ultimately either available or non-available, so suppose we partition the states into two groups: those states in which the processor is available, and those states in which the processor is non-available. If we then think of each group as a state whose probability is the sum of the probabilities of its constituent states, we have reduced our multi-state processor model to a two-state model, which we analyzed in the previous section. When we do this, the rate between the available and non-available states is equal to the sums of the rates between each group. We can compute this quantity by first finding p_a, the sum of the p_i's for all available state, and similarly p_n for non-available states. To compute the rate of going from available to non-available, for each available state, i, multiply the rate from state i to non-available states by p_i, sum over all available states, and divide by p_a (to normalize). In Figure 2.6, this would be

$$\text{Av to Na rate} = \frac{p_1}{p_a}r_1 + \frac{p_3}{p_a}(r_4 + r_9).$$

We then do likewise for the non-available states.

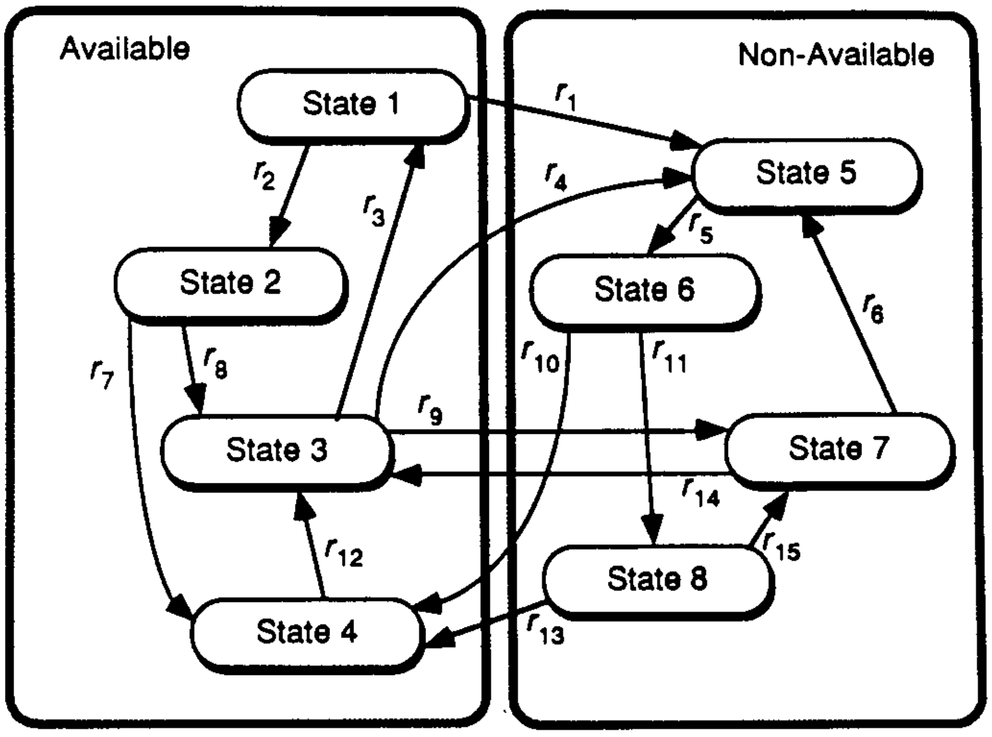


Figure 2.6: Multi-state model of a processor.

2.3 Time to Finish a Program: Quick Means.

With simple reasoning, we can find the mean cumulative work over time, and the mean finishing time for a program. Over a long period of time, a processor is available a fraction of the time $p_a = t_a/(t_a + t_n)$, and non-available the remaining fraction of the time, $p_n = t_n/(t_a + t_n)$. Over a period of t seconds, the amount of work a processor does is equal to the fraction of time it is available, and thus we have

$$\overline{Y_t} = \frac{t_a}{t_a + t_n}\, t. \tag{2.3}$$

Similarly, it takes $(t_a + t_n)/t_a$ seconds to accumulate one second of work, so the average finishing time for a program on a single processor is

$$\overline{J} = \frac{t_a + t_n}{t_a}\, W. \tag{2.4}$$

In an M processor network, we accumulate work M times faster and finish in $1/M$ of the time. Thus:

$$\overline{Y_t} = \frac{t_a M}{(t_a + t_n)}\, t. \tag{2.5}$$

$$\overline{J} = \frac{t_a + t_n}{t_a M}\, W. \tag{2.6}$$

We will use these as a check on the other analyses.

2.4 The Distribution of Finishing Time for One Processor.

2.4.1 Introduction.

We would now like to find the pdf of finishing time for any particular algorithm or program. Figure 2.7 shows the individual behavior of five processors as time progresses, along with a graph of the total number of available processors vs. time. As an algorithm runs on the system, it accumulates processor time in an amount equal to the area under the plot in the figure. We wish to know the pdf of the time it takes to complete the algorithm. This is also known as the *first passage time*, the first time at which the accumulated work is greater than some particular amount (W minutes in our case).

In this section, we analyze the behavior of a program on a single, transient processor using two methods. The direct method, in section 2.4.3 yields $f(t)$ for general distributions, but unfortunately, it does not extend to multiple processors because these analysis depends upon the system being either fully available or fully non-available. In a multiprocessor system, we usually have partial availability: some of the machines are available and some are not. We also do not derive the pdf of accumulated work using the direct analysis, but by analyzing the problem as a cumulative, alternating, renewal process (section 2.4.4), we find the asymptotic probability density of the cumulative work, $w(u \mid t)$, as $t \to \infty$, and we use this later in the Brownian motion analysis.

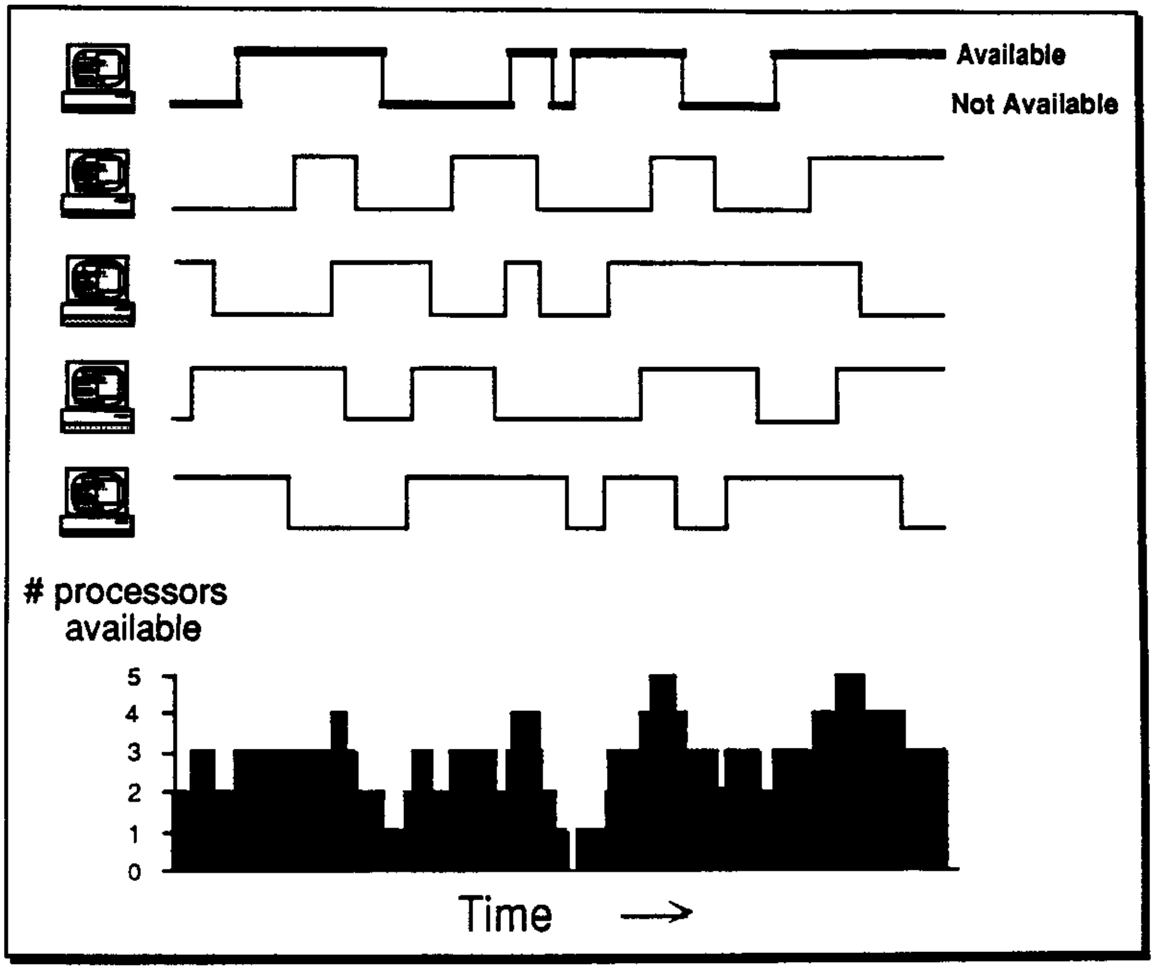


Figure 2.7: Number of available processors as time progresses.

2.4.2 Related Work.

We can use a variety of approaches to analyze a program on a single transient processor. In this section, we discuss three such approaches, and their drawbacks: preemptive priority queueing systems, queueing systems with vacations, and reliability analysis.

2.4.2.1 Preemptive Priority Queues.

We may model a transient processor using a preemptive priority queueing system with two classes. The high priority class, representing non-available periods, interrupts the service of the low priority class, representing the distributed programs. Such systems have been extensively analyzed (see, for example, [Kle76], or [Sev77] as discussed in [Agr85]), but ultimately this model fails us because it allows queueing of non-available periods. This is not a realistic model of a transient processor.

2.4.2.2 Queues with Vacations.

In a queueing system with vacations, the queueing server is subject to randomly occurring stoppages lasting for random amounts of time. There are many varieties of such systems depending upon what restrictions we put on the vacations (see [Dos86] for a survey). For our model, we require vacations to occur preemptively and at any time (as opposed to vacations that occur only when the processor is busy). The first analysis of such systems is in [WC58] and [Thi63], but Gaver ([DPG62]) derives the Laplace transform for the finishing time by assuming exponentially distributed available periods and generally distributed non-available periods. Federgruen and Green ([FG86]) extend the analysis to generally distributed available periods, but then find only the first two moments of the finishing time, and not its distribution.

2.4.2.3 Reliability Models.

Reliability analysis concerns itself with the availability of a system over time, and some work has been done on cumulative availability, the cumulative amount of time that a system is available. This corresponds exactly to accumulation of work in a network of transient processors. For a system with two states, available and non-available, Donatiello and Iyer [DI87] find the transform of the cumulative availability, and they derive a closed-form expression when the time in each state is exponentially distributed. However, their closed-form expression is very complex and unintuitive, and we derive a simpler, more useful, expression later in this chapter. Also, their two-state model applies only to one processor, and can not model a network of processors. Silva and Gail [dSeSG89] discuss the calculation of cumulative availability in systems which can be modeled as homogeneous Markov chains. They use *randomization* to derive a general technique for calculating cumulative availability, and further find good methods of numerical solution for their technique. The SAVE (System Availability Estimator) program [Goy87] implements these ideas. If we were interested in numerical, not analytic,

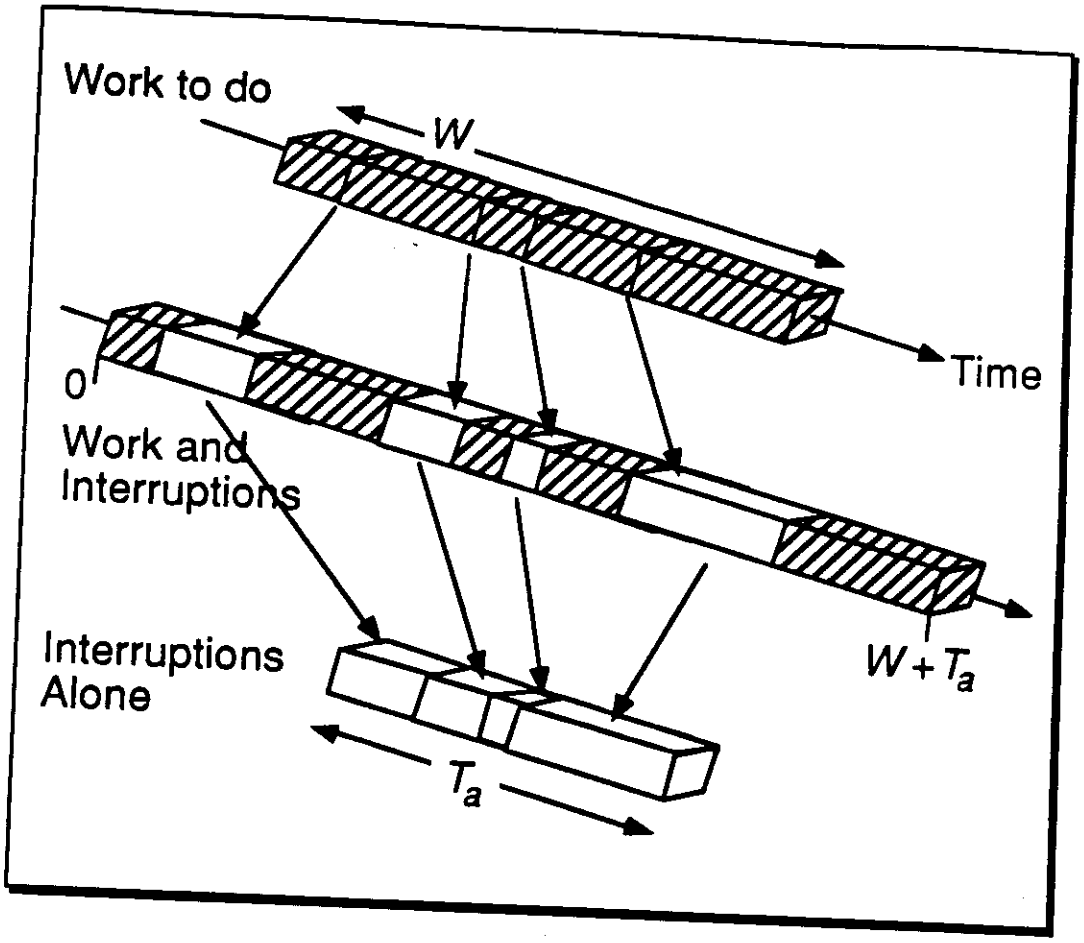


Figure 2.8: Time for one node to finish W units of work.

results, this would be an excellent approach.

2.4.3 Direct Analysis.

We make a direct analysis of the single-processor problem by counting the number of non-available periods that interrupt our program before it completes. If our program starts when the processor is available, as shown in the middle of Figure 2.8, it will finish at time $W + T_a$, where T_a is the additional time the program spends in the system because of interrupting non-available periods.

Because we must finish the program in an available period, and because we assume work starts during an available period, then none of the non-available periods are truncated, and it is relatively easy to analyze the total length of the non-available periods during the time $T_a + W$.

We first find the number of non-available periods in $T_a + W$ by examining the arrival process of these periods. Note that the arrival process for non-available periods stops during these (non-available) periods. Because the duration of such periods does not affect the rate at which they arrive, we may take non-available periods to have 0 length. As shown in Figure 2.9, the number of real non-available periods arriving in $T_a + W$ minutes is the same as the number of zero-length non-available periods arriving in W minutes. We let $p(k \mid W)$ be the probability density that k zero-length non-available periods arrive in W minutes.

Given k non-available periods, the probability density of the amount time spent in such periods is the k-fold convolution of the pdf of time in one period:

$$\frac{dP_{T_a}(t|k)}{dt} = \underbrace{n(t) \otimes n(t) \otimes \cdots \otimes n(t)}_{k \text{ times}}, \tag{2.7}$$

where $\otimes$ is the convolution operator. Unconditioning on k, we get that the density of T_a is

$$\frac{dP_{T_a}(t)}{dt} = \begin{cases} p(0 \mid W) & \text{if } t = 0 \\ \sum_{k=1}^{\infty} \underbrace{n(t) \otimes n(t) \otimes \cdots \otimes n(t)}_{k \text{ times}} \; p(k \mid W) & \text{if } t > 0 \end{cases} \tag{2.8}$$

When we add the W seconds necessary to do the program's work, the pdf of

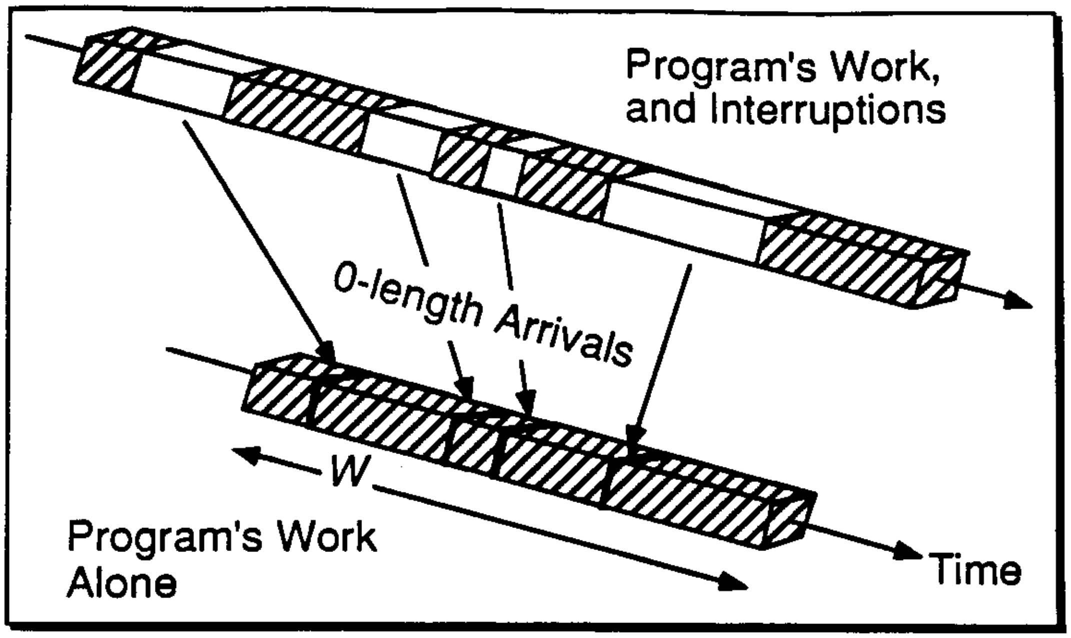


Figure 2.9: Converting non-available periods to 0 length.

the finishing time for the program is:

$$f(t) = \begin{cases} p(0 \mid W) & \text{if } t = W \\ \sum_{k=1}^{\infty} \underbrace{n(t-W) \otimes n(t-W) \otimes \cdots \otimes n(t-W)}_{k \text{ times}} \; p(k \mid W) & \text{if } t > W \end{cases} \tag{2.9}$$

Let us now derive the Laplace transform of the finishing time using an analogous process. The Laplace transform of the density of the length of k non-available periods is $[\mathrm{Na}^*(s)]^k$. The conditional Laplace transform of the finishing time is

$$F^*(s \mid k) = e^{-Ws}[\mathrm{Na}^*(s)]^k. \tag{2.10}$$

Unconditioning, we get

$$F^*(s) \;=\; e^{-Ws}\sum_{k=1}^{\infty}[\mathrm{Na}^*(s)]^k \, p(k \mid W)$$

$$= e^{-Ws}\mathrm{P}(\mathrm{N}a^*(s)), \tag{2.11}$$

where $\mathrm{P}(z) = \sum_{k=0}^{\infty} p(k \mid W)z^k$ is the z-transform of $p(k \mid W)$.

To find the mean and variance of the finishing time, we could either take derivatives of the expression above, or view the time spent in non-available periods as a random sum of random variables. Following the latter course, let $\overline{p}$ and σ_p^2 be the mean and variance of of $p(k \mid W)$; we leave their dependence upon W as implicit. The time to finish is the sum of a constant W minutes, plus many i.i.d. random variables representing the non-available periods. Thus, the mean time to finish W seconds of work is

$$\overline{f} = \overline{p}\,t_n + W = W\frac{t_a + t_n}{t_a}, \tag{2.12}$$

where $\overline{p} = W/t_a$ is the average number of zero-length non-available periods (with an average interarrival time of t_a minutes) that arrive in W minutes. The variance is

$$\sigma_f^2 = \sigma_n^2\overline{p} + t_n^2\sigma_p^2 \tag{2.13}$$

The central limit theorem assures us that when we add many random variables, the resulting distribution tends toward a normal distribution. We may note an important consequence of this: asymptotically, for large W, the finishing time density is normal with the mean and variance given in Equations 2.12 and 2.13.

2.4.3.1 Example: Exponential Distributions

As a concrete example, let us use exponentially distributed available and non-available periods.

Because the available periods are exponentially distributed, non-available periods arrive according to a Poisson process with rate $1/t_a$. The probability that k non-available periods interrupt our program before it completes is given by:

$$p(k \mid W) = \frac{(W/t_a)^k}{k!} e^{-(W/t_a)}, \tag{2.14}$$

which has mean and variance both equal to W/t_a.

Because the non-available periods are exponentially distributed, the convolution of the distributions of k such periods is a k stage Erlang distribution with mean $k\,t_n$.

We may graphically examine the constructions of the finishing time pdf. Figure 2.10 plots the Erlang densities that correspond to different numbers of interruptions, and also plots the Poisson weighting function (Equation 2.14) for these curves. We multiply the Erlangs by the Poisson and this results in Figure 2.11. Summing over the number of interruptions (the number of stages in the Erlang densities) we get the final density of T_a, also shown in these figures.

The density of T_a is:

$$\frac{dP_{T_a}(t)}{dt} = \begin{cases} e^{-W/t_a} & \text{if } t = 0 \\ \sum_{k=1}^{\infty} \left(\frac{(1/t_n)(t/t_n)^{k-1}}{(k-1)!} e^{-t/t_n} \right) \left(\frac{(W/t_a)^k}{k!} e^{-W/t_a} \right) & \text{if } t > 0 \end{cases} \tag{2.15}$$

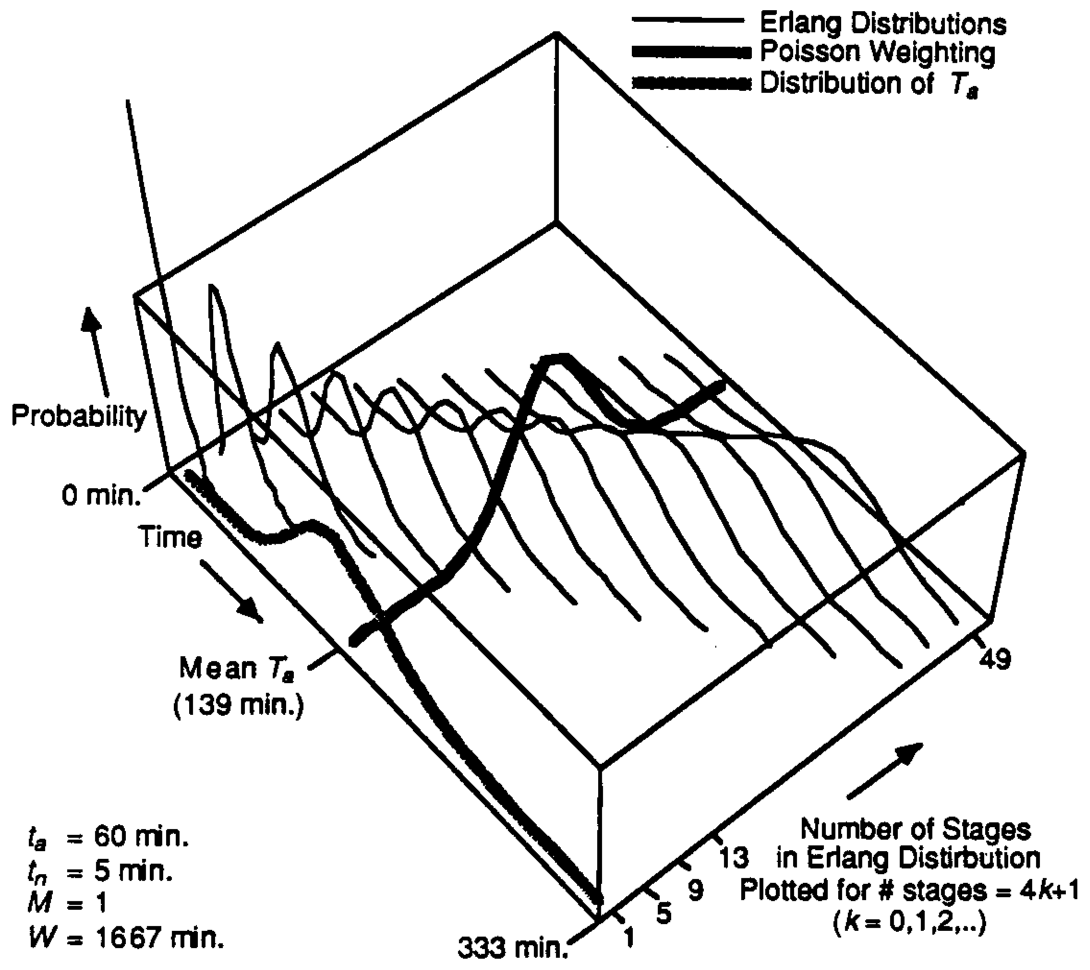


Figure 2.10: Erlang densities, the Poisson weighting function, and the finishing time.

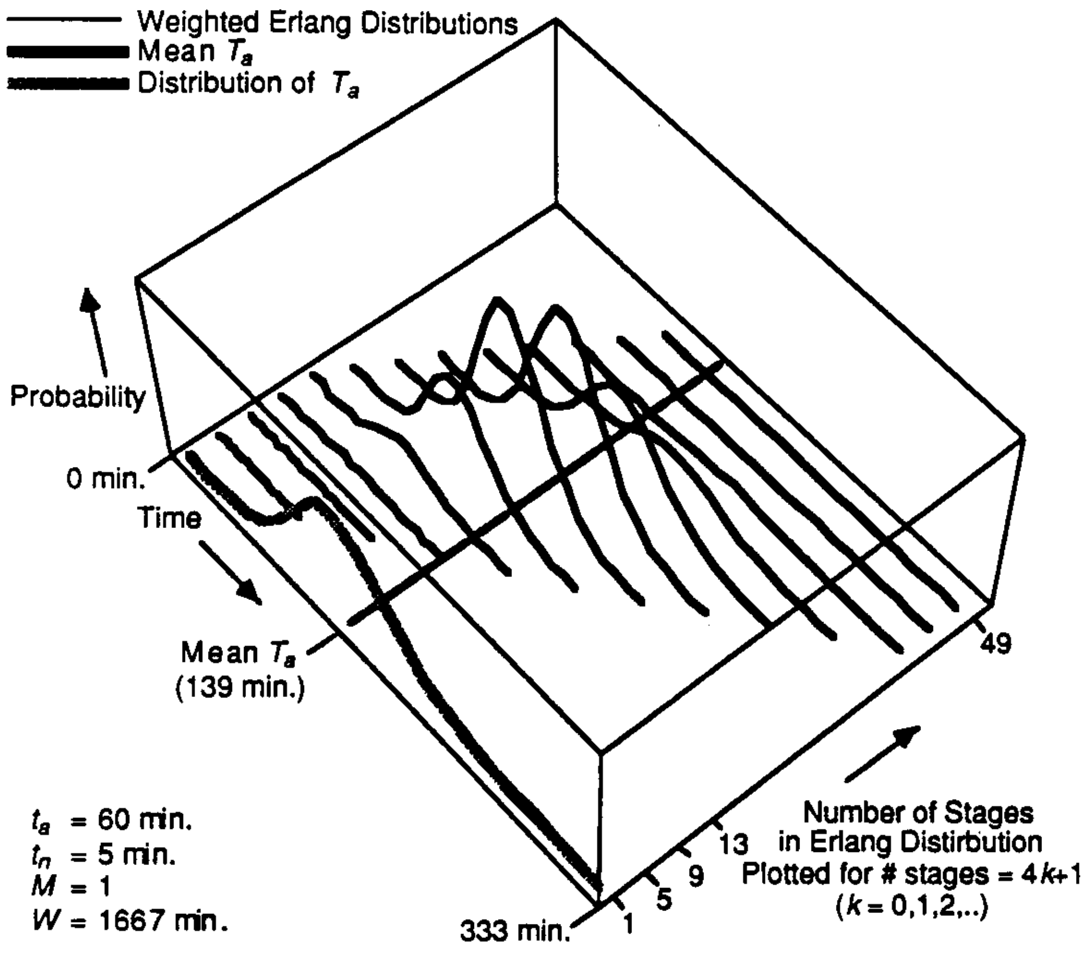


Figure 2.11: Weighted Erlang densities and the finishing time.

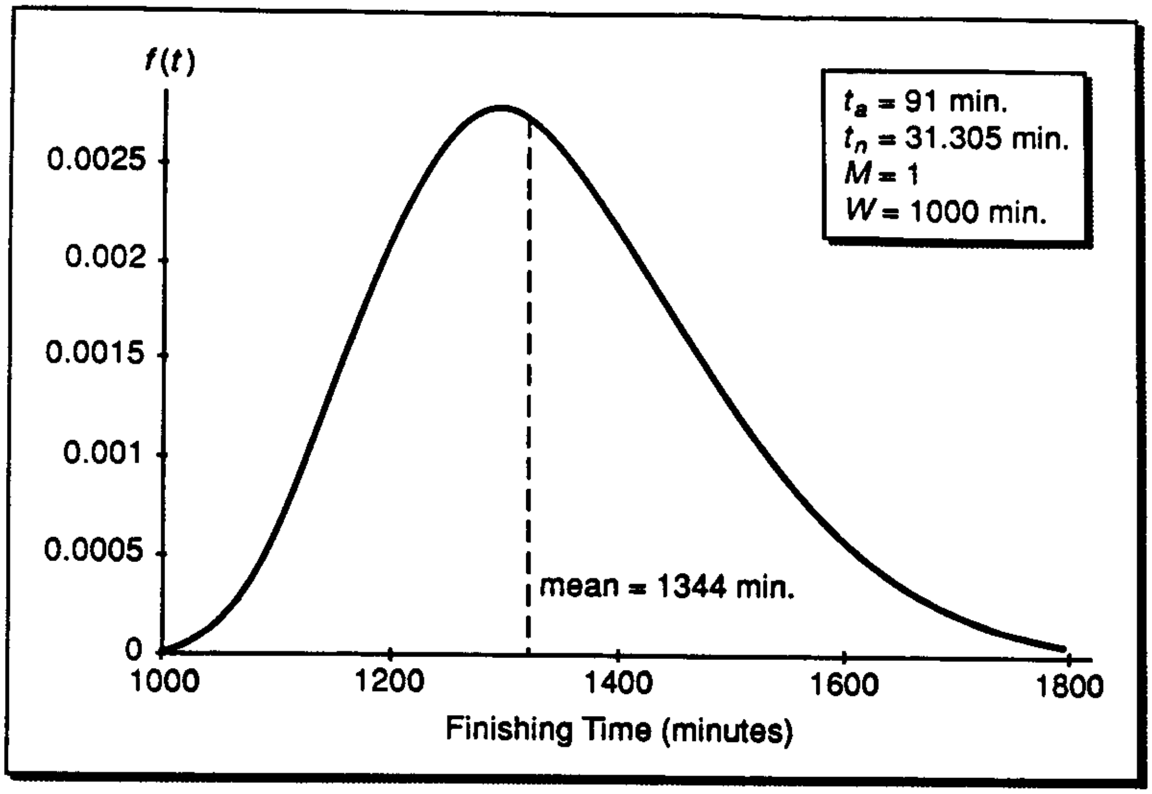


Figure 2.12: Probability density of finishing time for direct analysis.

The term $\frac{(1/t_n)(t/t_n)^{k-1}}{(k-1)!}e^{-t/t_n}$, a k-stage Erlang density, is the density of T_a if we have k non-available periods. For $t > 0$, this may be further reduced to

$$\frac{dP_{T_a}(t)}{dt} = \frac{1}{t_n}\sqrt{\frac{tW}{t_a t_n}}e^{-t/t_n}e^{-W/t_a}I_1\left(2\sqrt{\frac{tW}{t_a t_n}}\right) \qquad t > 0 \qquad (2.16)$$

where $I_1(z)$ is the modified Bessel function of the first kind of order 1.

The finishing time density is:

$$f(t) = \begin{cases} e^{-W/t_a} & \text{if } t = W \\ \sum_{k=1}^{\infty}\left(\frac{(1/t_n)((t-W)/t_n)^{k-1}}{(k-1)!}e^{-(t-W)/t_n}\right)\left(\frac{(W/t_a)^k}{k!}e^{-W/t_a}\right) & \text{if } t > W \end{cases} \qquad (2.17)$$

Figure 2.12 illustrates this density using Mutka and Livny's parameters. The mean finishing time is

$$\overline{f} = W\frac{t_a + t_n}{t_a}, \tag{2.18}$$

and its variance is

$$\sigma_f^2 = \frac{2t_n^2 W}{t_a}. \tag{2.19}$$

As mentioned before, if W is sufficiently large, we expect many non-available periods, and under these conditions the distribution of the number of non-available periods approaches a normal distribution.

The finishing time time density looks similar to a normal density, but it is asymmetrical. For t less than the mean first passage time, it rises sharply to a peak before the mean, then drops into a stretched-out tail for large t. This asymmetry is more apparent for small W, and as W grows, the density becomes similar to a normal density, as one would expect from the central limit theorem.

2.4.3.2 Modifying the Analysis to Compute System Time for Job.

The preceding analysis assumed that the program started during an available period. In real life, however,the program may arrive during a non-available period, and we should account for this. Let the true time in the system be known as the ***system time***. With probability $p_a = t_a/(t_a + t_n)$, the job arrives during an available period, and the distribution of system time is the same as that of the finishing time from the preceding analysis. With probability $p_n = 1 - p_a$, the job arrives in a non-available period, and waits the residual life of the non-available period before starting. We define the probability density of this residual life as $r(t) = (1 - \int_{x=0}^{y} N(x)dx)/t_n$, with i^{th} moment r_i and Laplace transform $R^*(s) = (1 - \text{Na}^*(s))/(st_n)$.

We account for this residual life by adding $p_n r(t)$ to the pdf of $T_a(t)$, and in the transform domain multiplying $T_a^*(s)$ by $p_n R^*(s)$. The mean of the system time is then

$$\overline{s} = W\frac{t_a + t_n}{t_a} + t_n r_1 \tag{2.20}$$

and its variance is

$$\sigma_s^2 = \sigma_f^2 + p_n(r_2 - r_1^2) \tag{2.21}$$

Note that as W gets large, the terms that account for the contribution of the possible initial non-available period become negligible, and the mean and variance become identical to Equations 2.12 and 2.13.

2.4.3.3 Conclusion.

In this section we derived an expression for the pdf of a program's finishing time, and the Laplace transform of this density. From this expression, we derived its mean and variance. Unfortunately, this analysis will not extend to multiple processors because it depends upon the system being fully available or fully non-available. With multiple processors the system is usually partially available.

To continue, we must find the pdf of accumulated work, which we do in the next section by using a cumulative, alternating renewal process in our analysis.

2.4.4 Cumulative, Alternating Renewal Theoretic Analysis.

We continue the single processor analysis using a cumulative, alternating renewal process. Cox, in his book on renewal processes [Cox62], discusses this type of process, and this section follows his analysis. We find the asymptotic ($t \to \infty$) pdf of accumulated work for generally distributed available and non-available periods, but we do not find the finishing time density because this is known from the previous analysis.

We can form a renewal process from the alternating states of a transient processor by letting a renewal period be a non-available period followed by an available period. In Figure 2.13 the heavy dots indicate the beginning of each renewal period. Let X_i be the duration of the i^{th} renewal period, and within this, let X_i' be the duration of the i^{th} unavailable period, and let X_i'' be the duration of the i^{th} available period. The time between renewal points is $X_i = X_i' + X_i''$, and has mean $\overline{x} = t_a + t_n$, variance $\sigma_x^2 = t_a^2 + t_n^2$, and Laplace transform $X^*(s)$. The X_i''s are i.i.d. random variables, likewise the X_i'''s are i.i.d., and the X_i''s and the X_i'''s are mutually independent. We assume, for the moment, that $t = 0$ occurs at one of the renewal points.

To form a cumulative process from this, let $W_i = X_i''$, with mean $\overline{w}$, variance σ_w^2, and Laplace transform $\mathrm{Av}^*(s)$, and let

$$\begin{aligned} Y_t &= \sum_{i=1}^{N_t} W_i \quad (N_t = 1, 2, \ldots) \\ &= 0 \qquad (N_t = 0) \end{aligned} \tag{2.22}$$

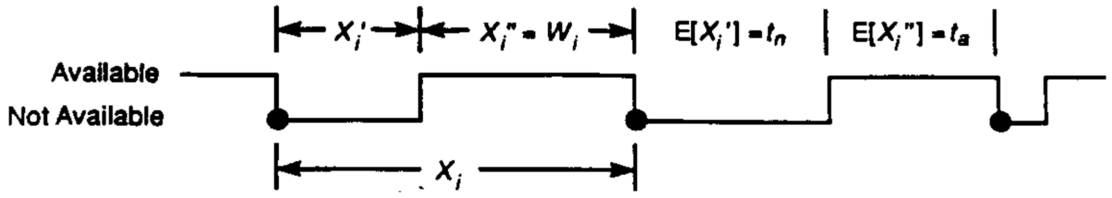


Figure 2.13: Cumulative renewal process.

where N_t is the number of renewals up to time t. Y_t is a random variable equal to the sum of all the available time up to time t, excepting the current available period, if the process is in such a period at time t. As time goes to infinity, the asymptotic distribution of Y_t has the same properties as the process that accumulates the true total available time up to time t, but Y_t's analysis is more tractable.

Letting ρ_{yx} be the correlation coefficient between W_i and X_i. Cox shows that

$$\mathrm{E}[Y_t] \approx \frac{\overline{w}}{\overline{x}} t \tag{2.23}$$

$$\begin{aligned} \mathrm{Var}[Y_t] &\approx \left[\frac{\sigma_w^2(1-\rho_{yx}^2)}{\overline{x}} + \frac{\sigma_x^2}{\overline{x}^3}\left(\overline{w} - \rho_{yx}\frac{\sigma_w \overline{x}}{\sigma_x}\right)^2\right] t \\ &= \left(\frac{\sigma_w^2}{\overline{x}} + \frac{\sigma_x^2 \overline{w}^2}{\overline{x}^3} - \frac{2\sigma_x\sigma_w\rho_{yx}\overline{w}}{\overline{x}^2}\right) \end{aligned} \tag{2.24}$$

We find ρ_{yx}:

$$\begin{aligned} \rho_{yx} &= \frac{\mathrm{Cov}[XY]}{\sqrt{\sigma_x^2\sigma_y^2}} \\ &= \frac{\mathrm{E}[XY] - \overline{x}\overline{y}}{\sqrt{\sigma_x^2\sigma_y^2}} \\ &= \frac{\mathrm{E}[(X' + X'')Y] - (t_a + t_n)\overline{y}}{\sqrt{\sigma_x^2\sigma_y^2}} \end{aligned}$$

$$\begin{aligned}
&= \frac{\mathrm{E}[X'Y] + \mathrm{E}[X''Y] - (t_a + t_n)t_a}{\sqrt{\sigma_x^2\sigma_y^2}} \\
&= \frac{t_n t_a + \sigma_y^2 + \overline{y}^2 - t_a^2 - t_n t_a}{\sqrt{\sigma_x^2\sigma_y^2}} \\
&= \frac{\sigma_y^2}{\sqrt{\sigma_x^2\sigma_y^2}} \\
&= \sigma_a/\sqrt{\sigma_a^2 + \sigma_n^2}. \qquad (2.25)
\end{aligned}$$

Applying ρ_{yx} to Equations 2.23 and 2.24 yields

$$\mathrm{E}[Y_t] \approx \frac{\overline{y}}{\overline{x}}t \approx \frac{t_a}{t_a + t_n}t \qquad (2.26)$$

$$\begin{aligned}
\mathrm{Var}[Y_t] &\approx \frac{\sigma_y^2\overline{x}^2 + \sigma_x^2\overline{y}^2 - 2\overline{x}\,\overline{y}\sigma_y^2}{(t_a + t_n)^3}t \\
&= \frac{\sigma_a^2 t_n^2 + t_a^2\sigma_n^2}{(t_a + t_n)^3}t. \qquad (2.27)
\end{aligned}$$

For exponentially distributed available and non-available periods, the mean remains the same and the variance may be rewritten as

$$\mathrm{Var}[Y_t] \approx \frac{2t_a^2 t_n^2}{(t_a + t_n)^3}t. \qquad (2.28)$$

Of particular interest to us is the fact that the asymptotic pdf of Y_t (for large t) is normal with mean and variance given by Equations 2.26 and 2.27. Y_t is a sum of random variables, and the Central-Limit Theorem [MGB74] tells us that as the number of random variables in the sum approaches infinity, the pdf of Y_t converges to a normal pdf. Thus Y_t is well approximated by a normal pdf if many renewal periods have occurred, or equivalently, $t \gg t_a + t_n$. This normal pdf will be the basis of the Brownian motion model in the next section.

Let us examine a special case of Y_t: suppose that Y_t is scaled by a factor c, so it's mean is $c\,t_a$, for some $c > 0$. The distributions of X_i' and X_i'' are unchanged. The Y_t's have a new mean and variance:

$$\overline{y'} = c\,\overline{y}$$

$$\sigma_{y'}^2 = c^2\sigma_y^2$$

Using these, we find that the value for $\rho_{y'x}$ is still σ_y/σ_x.Applying these values to Equations 2.26 and 2.27, we get:

$$\overline{Y_t} = \frac{c\,t_a}{t_a + t_n}t \tag{2.29}$$

$$\mathrm{Var}[Y_t] = \frac{c^2(\sigma_a^2 t_n^2 + t_a^2\sigma_n^2)}{(t_a + t_n)^3}t. \tag{2.30}$$

The mean is now c times Equation 2.26, and the variance is c^2 times Equation 2.27. We will use this result in the Brownian motion analysis to model networks of non-identical processors.

2.4.5 Conclusion.

The important result of this section is that the asymptotic pdf (for large t) of the amount of work accumulated over time is normal with the mean and variance given in Equations 2.26 and 2.27.

2.5 The Distribution of Finishing Time for M Processors.

In the preceding sections we analyzed the behavior of a program on a single transient processor. Now we expand our analysis to encompass a network of M

processors.

2.5.1 Brownian Motion Approximation.

Brownian motion concerns the random movement of a particle through space. Its namesake is Robert Brown, who described the motion of pollen suspended in water [Bro28]; the collisions of water molecules made the pollen move in an erratic fashion. Some years later, Einstein published the first theoretical approach to Brownian motion [Ein26], and many researchers have expanded the field since then. See, for example,[Wie76], [IHPM65], [KT75], [Hid80], and [Har85] for complete treatments of the subject of Brownian motion.

A stochastic process, $Y(t)$, that describes Brownian motion has two basic properties. The first is that $X(t)$ has ***independent increments***: $Y(t_1) - Y(t_0)$ and $Y(t_3) - Y(t_2)$ are independent for $0 \leq t_0 < t_1 < t_2 < t_3 < \infty$. Movement of the particle in one interval is independent of its movement in another interval. The second property is that each increment in the process, $Y(t_1) - Y(t_0)$, is normally distributed with a mean and variance proportional to $t_1 - t_0$. If the normal distribution has mean 0 and variance equal to $t_1 - t_0$, then the process describes ***standard Brownian motion***, which is also known as a *Wiener process*. Brownian motion with a non-zero mean is known as *Brownian motion with drift*.

In our model, the stochastic process $Y(t)$ represents the amount of work accumulated by a network of transient processors up to time t. In section 2.4.4, we found that over a long period of time (much longer than $t_a + t_n$), the amount of work done by one transient processor is asymptotically normal with mean and variance given in Equations 2.26 and 2.27, respectively. If we have a network of M such processors, and all the processors are assumed to be independent and identical, then, asymptotically, the amount of work done by time t is the sum of M independent, (approximately) normally distributed random variables, and this is itself (approximately) normally distributed. The mean amount of work done by time t is:

$$\mu = \frac{t_a}{t_a + t_n} Mt = p_a Mt \tag{2.31}$$

and variance of the amount of work done by time t is:

$$\sigma^2 = \frac{2(\sigma_a^2 t_n^2 + \sigma_n^2 t_a^2)}{(t_a + t_n)^3} \, Mt. \tag{2.32}$$

Thus, Brownian motion with drift is a natural model of our system. From μ and σ^2 above, we define $\bar{b}$ and σ_b^2 (b signifies Brownian), the mean and variance of the amount of work accumulated per unit time:

$$\bar{b} = \frac{t_a}{t_a + t_n} M = p_a M \tag{2.33}$$

$$\sigma_b^2 = \frac{2(\sigma_a^2 t_n^2 + \sigma_n^2 t_a^2)}{(t_a + t_n)^3} \, M. \tag{2.34}$$

We use $\bar{b}$ and σ_b^2 later in this analysis.

We must still assure ourselves that our stochastic process indeed has independent increments. On a short term scale, this is clearly not true. Two consecutive one-minute intervals are likely to have the same, or at least similar, numbers of available processors in both intervals, and hence similar amounts of work accumulated in those intervals. However, in two one-minute intervals separated by several hours, the number of available processors is quite unrelated (unless the network has some very unusual statistical properties), and the work accumulated in one interval is quite independent of the work accumulated in the other interval. Thus we conclude that the Brownian motion model is reasonable only over a long span of time, and we insure this by specifying that $t_a \ll W$ and $t_n \ll W$. Note, too, that we are using the asymptotic results of section 2.4.4, and these are valid only for a long span of time, which also requires a large W relative to t_a and t_n.

The Brownian motion model does allow some behavior that seemingly cannot occur in a real network: the process is allowed to move in the negative direction, implying that we can lose work that we have already done. In a network of transient processors, this is, in fact, a reasonable assumption. It is possible that if a portion of a program is executing on a processor when it becomes non-available, some or all of the work completed may be lost. Thus, negative movement of the particle should not bother us.

What is more troublesome is the fact that there is some non-zero probability that the amount of work done by a particular time is negative. The program cannot accumulate less than 0 minutes of work, so this aspect of our model is clearly incorrect. Given $\overline{b}$, σ_b^2, and t, and using the fact that cumulative work is normally distributed, we can compute the probability of negative cumulative work as:

$$\Phi\left(\frac{-\bar{b}t}{\sqrt{\sigma_b^2 t}}\right) = \frac{1}{\sqrt{2\pi}}\int_{-\infty}^{-(\bar{b}t/\sqrt{\sigma_b^2 t})} e^{-\frac{1}{2}x^2}\,dx \tag{2.35}$$

where $\Phi(x)$ is the cumulative density function for a standard normal distribution. For t near 0, $\Phi(-\bar{b}t/\sqrt{\sigma_b^2 t}) \approx .5$, meaning that for very small t, our model says that almost 50% of the time the program has accumulated a negative amount of work. Clearly, Brownian motion is a poor model of networks of transient processors for very small t. If we manipulate the expression $\frac{-\bar{b}t}{\sqrt{\sigma_b^2 t}}$ we find it is equal to $-\sqrt{\frac{Mt_n}{2(1-p_a)}t}$, and the coefficient of $\sqrt{t}$ is less than -1 for any reasonable values of M, t_a, and t_n. Thus as time passes and t moves away from 0 and becomes large, $-\sqrt{\frac{Mt_n}{2(1-p_a)}t}$ becomes quite negative and $\Phi(\frac{-\bar{b}t}{\sqrt{\sigma_b^2 t}})$ shrinks to near 0. In fact, for $t = 3\sqrt{\sigma_b^2}/\bar{b}$, the probability that we have negative work is approximately 0.0023 and drops rapidly thereafter to negligible amounts as t grows. This is just further confirmation that our model is valid only for relatively large W that requires more than a short time to complete.

Using $\bar{b}$ and σ_b^2 as the parameters for our Brownian motion, and using results in Karlin and Taylor [KT75], we find the probability density of the time, t, that it takes for M processors to finish W minutes of work is

$$f(t) = \frac{W}{\sqrt{2\pi\sigma_b^2 t^3}} \exp\left[-\frac{(W-\bar{b}t)^2}{2\sigma_b^2 t}\right] \tag{2.36}$$

This has mean

$$\bar{t} = \frac{W}{\bar{b}} = \frac{W}{M}\frac{(t_a+t_n)}{t_a} \tag{2.37}$$

and variance

$$\sigma_f^2 = \frac{W}{\bar{b}}\frac{\sigma_b^2}{\bar{b}^2}. \tag{2.38}$$

Note that for the case $M = 1$, the mean and variance agree with the direct analysis of section 2.4.3.

2.5.1.1 Example: Exponential Distributions.

If we assume that both available and non-available periods are exponentially distributed, then the mean and variance of the accumulated work per unit time are:

$$\bar{b} = \frac{t_a}{t_a + t_n}M = p_a M \tag{2.39}$$

$$\sigma_b^2 = \frac{2(t_a t_n)^2}{(t_a + t_n)^3}M = \frac{2p_a^2(1-p_a)M}{t_n}. \tag{2.40}$$

Applying this to Equations 2.37 and 2.38 yields the mean of the finishing time:

$$\bar{f} = \frac{W}{\bar{b}} = \frac{W}{M}\frac{(t_a + t_n)}{t_a} \tag{2.41}$$

and variance

$$\sigma_f^2 = \frac{W}{\bar{b}}\frac{\sigma_b^2}{\bar{b}^2} = \frac{2W}{M^2}\frac{t_n^2}{t_a}. \tag{2.42}$$

Note that for $M = 1$, these results reduce to those of section 2.4.3. Figure 2.14 shows the finishing time density for both the direct and the Brownian motion analyses with $t_a = 3600, t_n = 300, M = 1$, and $W = 10^5$. We note the good concordance between the two analyses.

When we have $M = 100$ processors, Figure 2.15 shows the pdf of finishing time for various t_n with $t_a = 91$ minutes and $W = 10^4$ minutes. Using $t_a = 91$

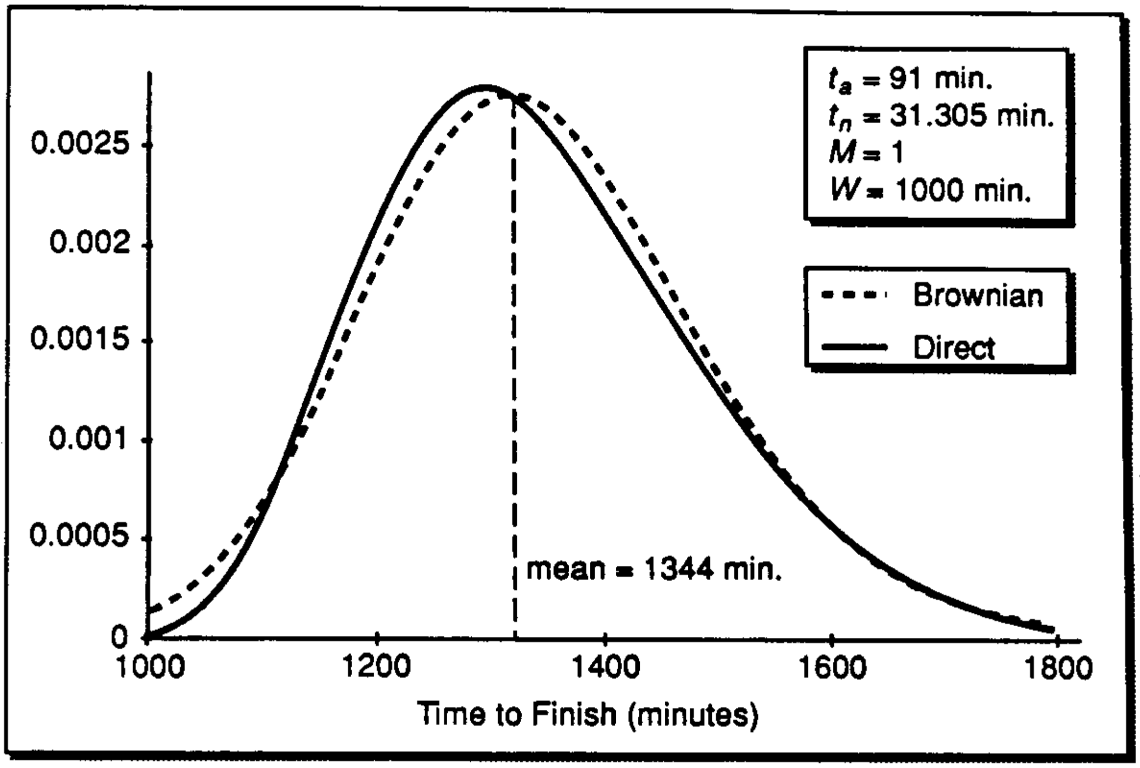


Figure 2.14: Finishing time densities for direct and Brownian motion analyses.

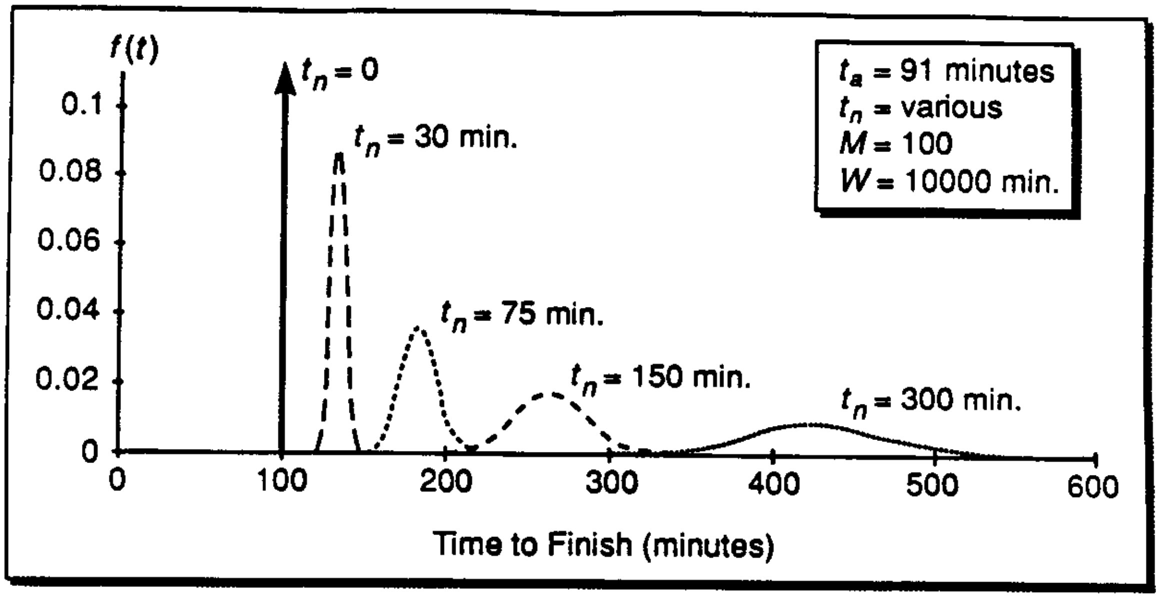


Figure 2.15: Finishing time densities for Brownian motion model for various t_n.

minutes and $t_n = 31.305$ minutes (the standard multiprocessor example), we have $\bar{b} = 74.4t$ and $\sigma_b^2 = 887.2t$, which leads to $\bar{f} = 0.0134W$ and $\sigma_f^2 = 0.00215W$. Our example job of 10^4 minutes would take about a week to run on a single, dedicated processor. When run on a network of 100 transient processors, it would take 134.06 minutes, or about 2.25 hours. This particular finishing time pdf is in Figure 2.16, which shows the density both enlarged and plotted on a full time axis (starting at 0). Note that although $W \gg t_a$ and $W \gg t_n$, we have that W/M, the finishing time if the processors were fully dedicated to the program, is close to the average length of an available period, and it is remarkable that the curve is still so symmetrical.

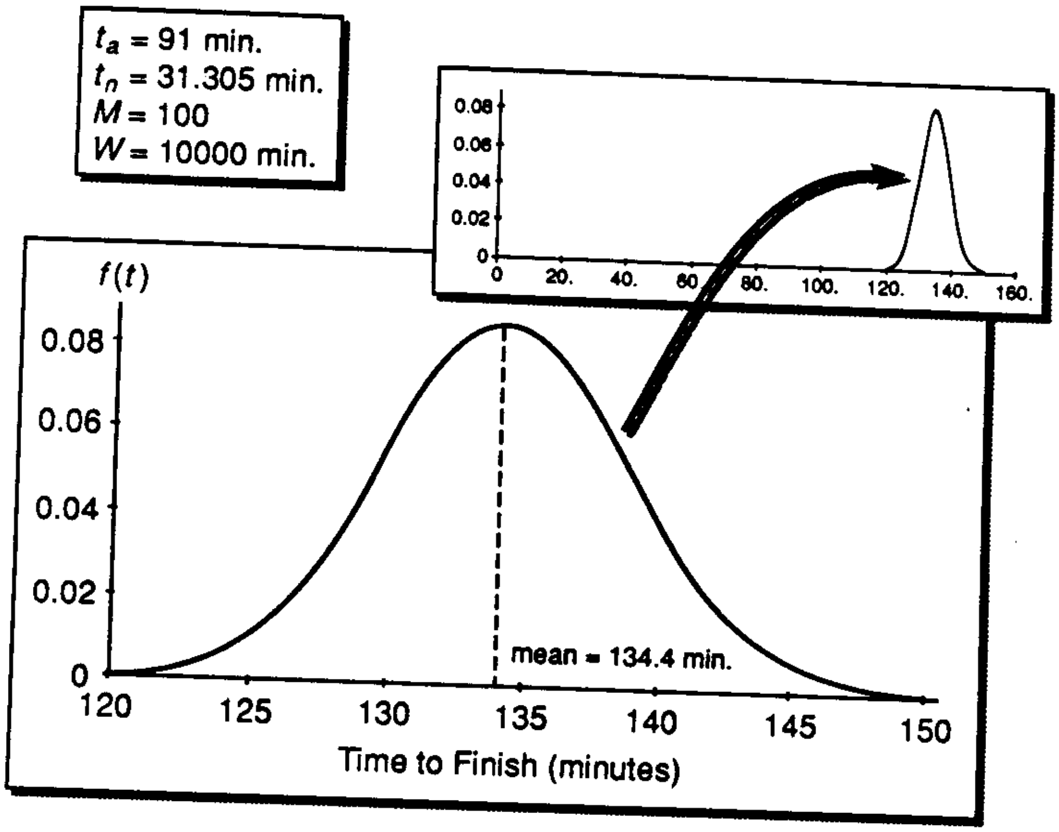


Figure 2.16: Finishing time density for Brownian motion analysis.

2.5.2 Example: Mutka and Livny's Distributions.

Let us use the distributions measured by Mutka and Livny. We have $t_a = 91$ minutes, $\sigma_a^2 = 40225$ minutes^2, $t_n = 31.305$ minutes, and $\sigma_n^2 = 2131.83$ minutes^2. Plugging these into Equations 2.33 and 2.34, we find:

$$\bar{b} = 74.4t \tag{2.43}$$

$$\sigma_b^2 = 6239t. \tag{2.44}$$

This leads to a finishing time mean and variance of:

$$\bar{f} = 0.0134W \tag{2.45}$$

$$\sigma_f^2 = 0.0151W. \tag{2.46}$$

We note that the finishing time variance using Mutka and Livny's distributions is almost an order of magnitude more than for exponential distributions.

2.5.3 The Ratio $\sigma_f/\bar{f}$.

It is instructive to examine the ratio of σ_f to $\bar{f}$, namely:

$$\frac{\sigma_f}{\bar{f}} = \frac{\sqrt{\sigma_b^2}}{\sqrt{\bar{b}W}} \tag{2.47}$$

We note immediately that this ratio goes to zero as W increases. Consequently, for sufficiently large W, it may be accurate enough to consider the finishing time distribution as an impulse at the mean finishing time (in the spirit of the law of large numbers).

Assume that the available and non-available periods have exponential distributions. Then the ratio becomes:

$$\frac{\sigma_f}{\bar{f}} = \sqrt{\frac{2t_n}{W}}\,\frac{\sqrt{t_n/t_a}}{1+t_n/t_a} \tag{2.48}$$

Because we assumed $t_n \ll W$, this ratio tends to be less than 1. If we fix t_n/W and let t_n/t_a go to infinity (which implies $t_a \to 0$), the ratio goes to 0. We explain this by noting that for small t_a, it takes very many available–non-available cycles before the work is finished. The law of large numbers insures that the finishing time density, which is the sum of these many periods, will then be tight about its mean.

If, on the other hand, we let $t_a \to \infty$, the ratio of the standard deviation to the mean goes to zero once again. When t_a is large relative to t_n, the non-available periods become negligible, as if the processors are always available. Again, the finishing time density becomes very tight about its mean because non-available time periods add little variability to the finishing time, and under some circumstances we may consider the finishing time density as an impulse located at $t = W/M$. Using the standard multiprocessor example again, we find $\sigma_f^2 = 21.53$, and approximating $f(t)$ as a normal density (discussed below), we find that 90% of the time, programs requiring 10^4 minutes of work will finish within 7.6 minutes of the 134.4 minute mean finishing time, which is an interval 3% on either side of the mean. This is very narrow indeed.

We find the peak of Equation 2.48 by taking the derivative with respect to

t_a:

$$\frac{\partial}{\partial t_a}\frac{\sigma_f}{\overline{f}} = \frac{t_n(t_a + t_n) - 2t_a t_n}{\sqrt{Wt_a}(t_a + t_n)^2}. \tag{2.49}$$

Setting this equal to 0 yields $t_a = t_n$ as the peak of the ratio, at which point it takes on the value $\sigma_f/\overline{f} = \sqrt{t_n/2W}$. The ratio's value at the peak is small because of our assumption that $t_n \ll W$. Note that if we assume $t_a = t_n$, but not $t_n \ll W$, then we can make the ratio as large as we want, simply by increasing t_n. If, for example, $t_a = t_n = 1$ year, then either the system is available immediately to do all our work, or else we will have to wait a very long time before it even starts. In such a case, the finishing time still has a reasonable mean but an enormous variance. Another fact to note is that M, the number of processors, does not affect the ratio $\sigma_f/\overline{f}$. Even if we have an infinite number of processors, we can still have great variance relative to the mean. Of course, both the mean and the standard deviation go to zero as M grows, but their ratio remains constant.

In Figure 2.17 we plot $\sigma_f/\overline{f}$ for $W = 10^4$, with t_n/W fixed for each curve and varying t_a to generate the curves. A companion illustration, Figure 2.18, shows finishing time densities for various t_a. The x-axis (labeled "Time, Relative to Mean") is centered about the mean and plots the distance relative to the mean (varying from 0.9 times the mean to 1.1 times the mean). We note that the density is flattest and has the greatest spread for $t_a = t_n$; at this point $\sigma_f/\overline{f} = 0.035$, which is quite small. For comparison, if we use Mutka and Livny's distributions at the same point, the ratio is 0.092, which is still small. The narrowing of the

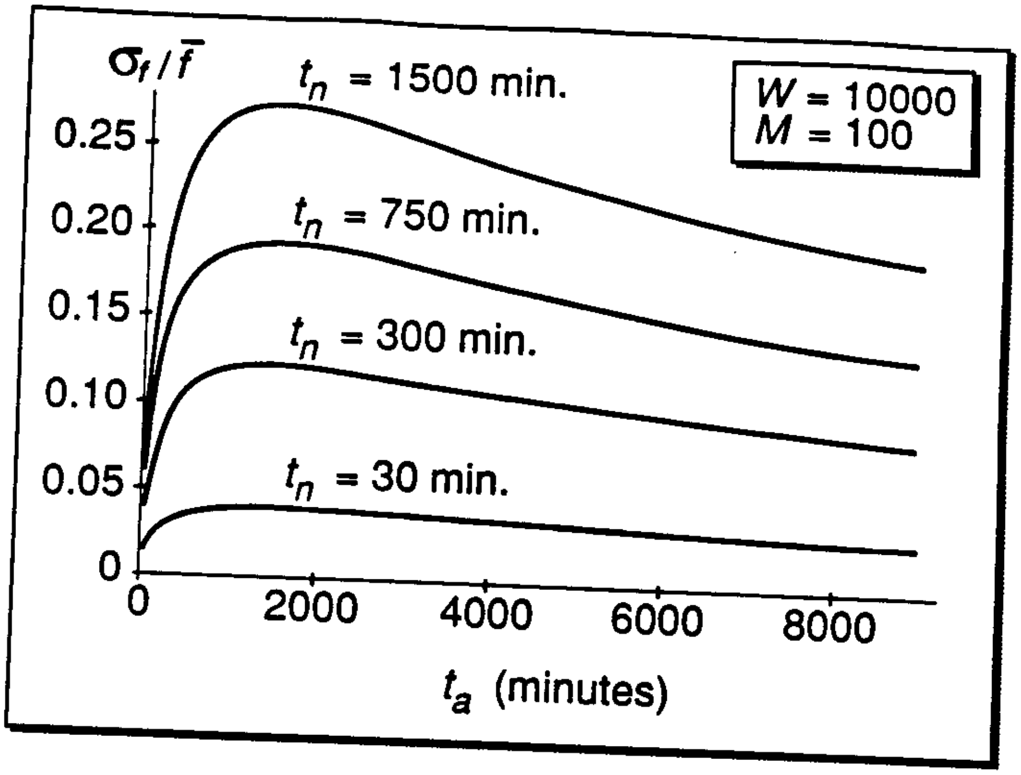


Figure 2.17: $\sigma_f/\bar{f}$ with $W = 10^6$, varying t_a and t_n.

density is also illustrated in Figures 2.19 and 2.20. The parameters for both of these plots are: varying t_a, $t_n = 31.305$ minutes, $M = 100$, and $W = 10^4$ minutes. In the Figure 2.19, the density narrows as $t_a = 31.305$, 150, 300, and 800 minutes. In Figure 2.20, we have $t_a = 31.305$, 10, 3, and 1 minute as the density narrows.

2.5.4 Normal Approximation to the Finishing Time.

The usual form of the central limit theorem states that the sum of n independent random variables tends to have a normal distribution as n gets large. Given this, we would expect the limiting distribution of $f(t)$ to be normal with

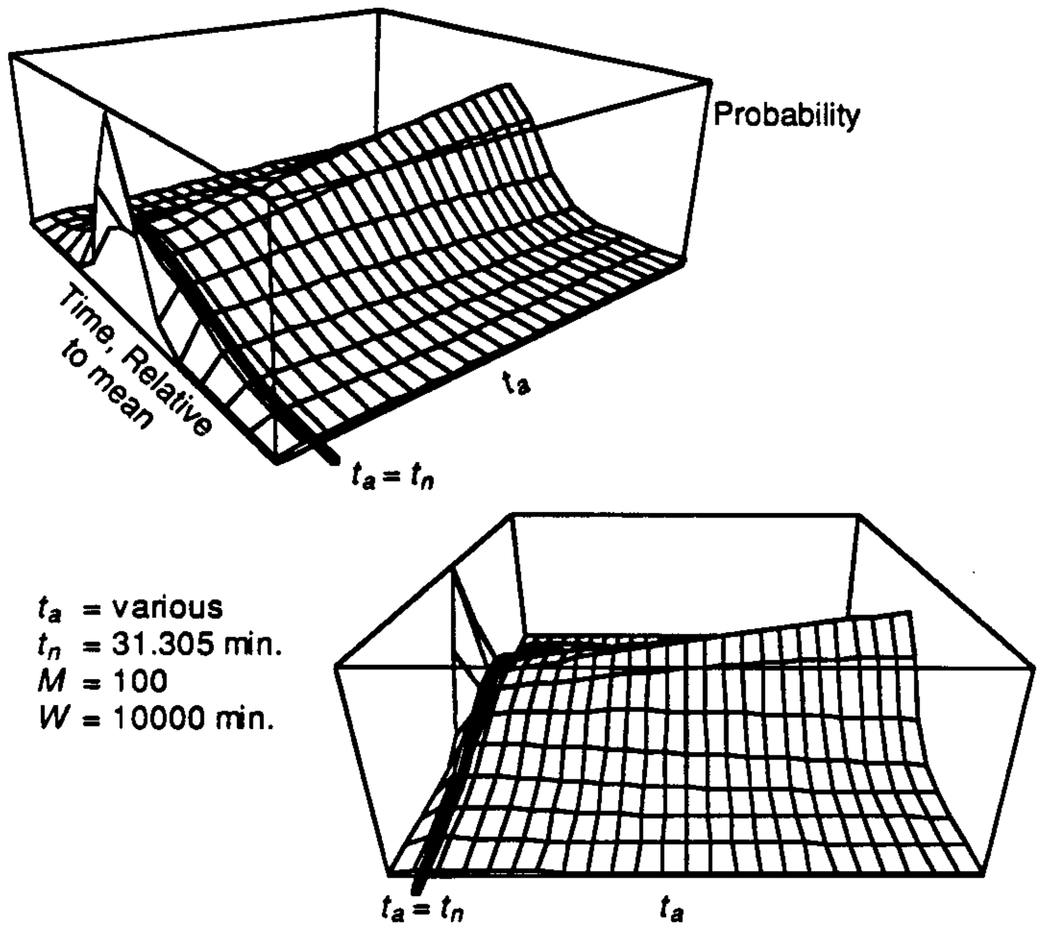


Figure 2.18: Brownian motion finishing time densities with varying t_a.

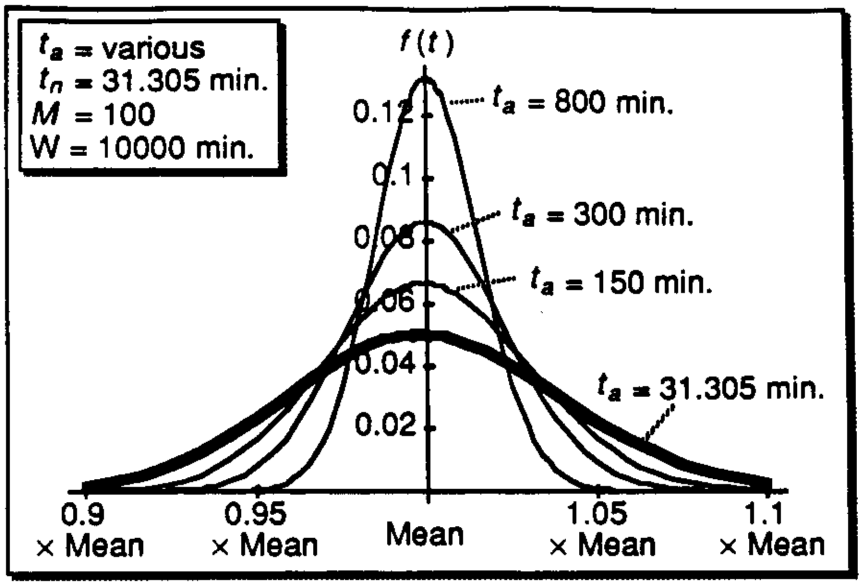


Figure 2.19: Finishing time density narrowing as t_a grows.

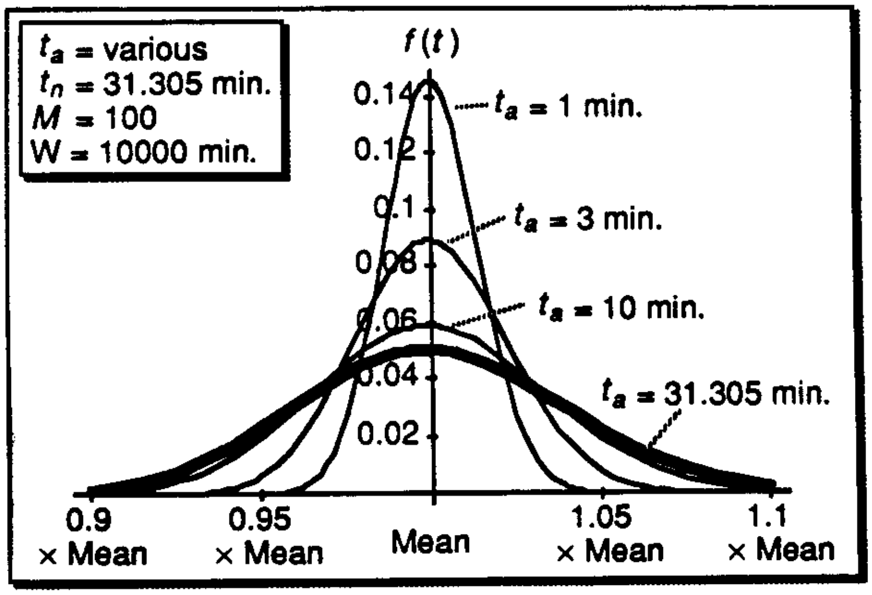


Figure 2.20: Finishing time density narrowing as t_a shrinks.

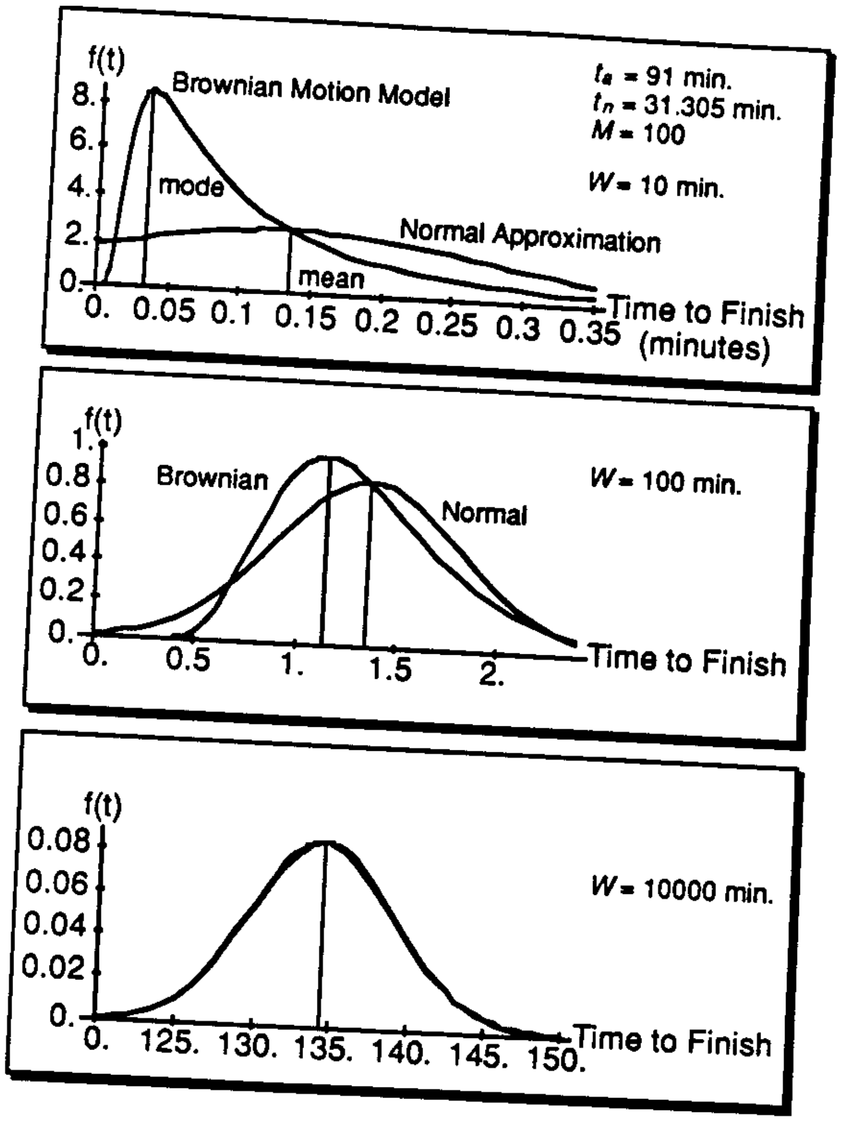


Figure 2.21: Brownian motion finishing time pdf and its normal approximation.

mean $\overline{f}$ and variance σ_f. Let us call this normal approximation $\hat{f}(t)$:

$$\hat{f}(t) = \frac{1}{\sqrt{2\pi\sigma_f^2}}\, e^{-(t-\overline{f})^2/(2\sigma_f^2)} \tag{2.50}$$

Substituting $t = \overline{f}$ shows that the finishing time and its normal approximation coincide at the mean:

$$\begin{aligned} f(\overline{f}) &= \frac{W}{\sqrt{2\pi\sigma_b^2 W^3/\overline{b}^3}}\, e^0 = \frac{\overline{b}^{3/2}}{\sqrt{2\pi\sigma_b^2 W}} \\ \\ \hat{f}(\overline{f}) &= \frac{1}{\sqrt{2\pi\sigma_f^2}}\, e^0 = \frac{\overline{b}^{3/2}}{\sqrt{2\pi\sigma_b^2 W}} \\ &= f(\overline{f}) \end{aligned}$$

Observation shows that $f(t)$ and $\hat{f}(t)$ also coincide at two more points, but these are not easily found because they are the solutions to a transcendental equation. Numerically, we find that these points appear to be separated by $\sqrt{12\sigma_f^2}$, and the distance from the lower point (smaller t) to the mean is very slightly less than half of the total distance between the two points (varying, but in the range of 49.5% of the total separation).

We need to know when $\hat{f}(t)$ is a good approximation for $f(t)$. Observation (see **Figure** 2.21) shows that the approximation is good when the mode of the finishing time is close to (within a few percent of) its mean. We find the mode by taking the derivative of $f(t)$ with respect to t, setting it equal to 0, and solving for t. We end up with a quadratic equation that has a negative and a positive root. The positive root is the mode, namely

$$t_{mode} = \frac{1}{2}\sqrt{\frac{9(\sigma_b^2)^2}{\bar{b}^4} + \frac{4W^2}{\bar{b}^2}} - \frac{3\sigma_b^2}{2\bar{b}^2} \tag{2.51}$$

By using the fact that $\sqrt{a+b} \leq \sqrt{a} + \sqrt{b}$, we show that the mode is always less than or equal to the mean:

$$t_{mode} \leq \frac{3\sigma_b^2}{2\bar{b}^2} + \frac{W}{\bar{b}} - \frac{3\sigma_b^2}{2\bar{b}^2} = \frac{W}{\bar{b}} = \bar{t}$$

Furthermore, if we observe that $9(\sigma_b^2)^2/\bar{b}^4$ is usually much less than $4W^2/\bar{b}^2$, and we use the approximation $\sqrt{a+b} \approx \sqrt{a} + b/(2\sqrt{a})$ (valid for $b \ll a$), then

$$\begin{aligned} t_{mode} &\approx \frac{W}{\bar{b}} - \frac{3}{2}\frac{\sigma_b^2}{\bar{b}^2}\left(1 - \frac{3}{4}\frac{\sigma_b^2}{W\bar{b}}\right) \\ &\approx \bar{t} - \frac{3}{2}\frac{\sigma_b^2}{\bar{b}^2} \end{aligned} \tag{2.52}$$

Under almost all circumstances, we may drop the negative term in the parenthesis, because when the Brownian motion approximation is valid we also know that $W \gg \sigma_b/\bar{b}$, and in general, $W \gg \sigma_b$. These would render the term $3\sigma_b^2/4W\bar{b}$ negligible. Only under very unusual circumstances would $W \not\gg \sigma_b^2$, and in such cases we could not drop the term. Excepting such circumstances, Equation 2.52 is quite accurate for all conditions in which our Brownian motion model is operative. Using Equation 2.52, we find that the percent difference between the mean and mode is approximately $3\sigma_b/2\bar{b}W$.

2.5.5 The Finishing Time Density and its Derivation from a Normal pdf.

We can rewrite the Brownian motion finishing time density, Equation 2.36, in a form using a normal pdf. Let $\phi_{\mu,\sigma^2}(x)$ be the probability that a random variable, normally distributed with mean μ and variance σ^2, takes on the value x. Using this, Equation 2.36 becomes

$$f(t) = \frac{W}{t}\phi_{\overline{b}t,\sigma_b^2 t}(W). \tag{2.53}$$

The normal pdf term derives from the underlying Brownian motion; it is the probability that a total of W units of work have been accumulated by time t. As for the weighting factor of W/t, no intuitive explanation has yet been found for this. Figure 2.22 illustrates the relationship between Equations 2.36 and 2.53. In this figure, the normal pdf of the amount of work done by time x, $\phi_{\overline{b}t,\sigma_b^2 t}(x)$, is plotted with thin lines for various t. The shaded plane in the figure picks out those points on the normal pdf where $x = W$; the thin line arcing down within this plane represents W/t. The thick line within the plane is the first passage time density, i.e. the product of these last two curves. The curves have been scaled differently to make them fit into one plot, so relative heights, except within the group of normal curves, are meaningless.

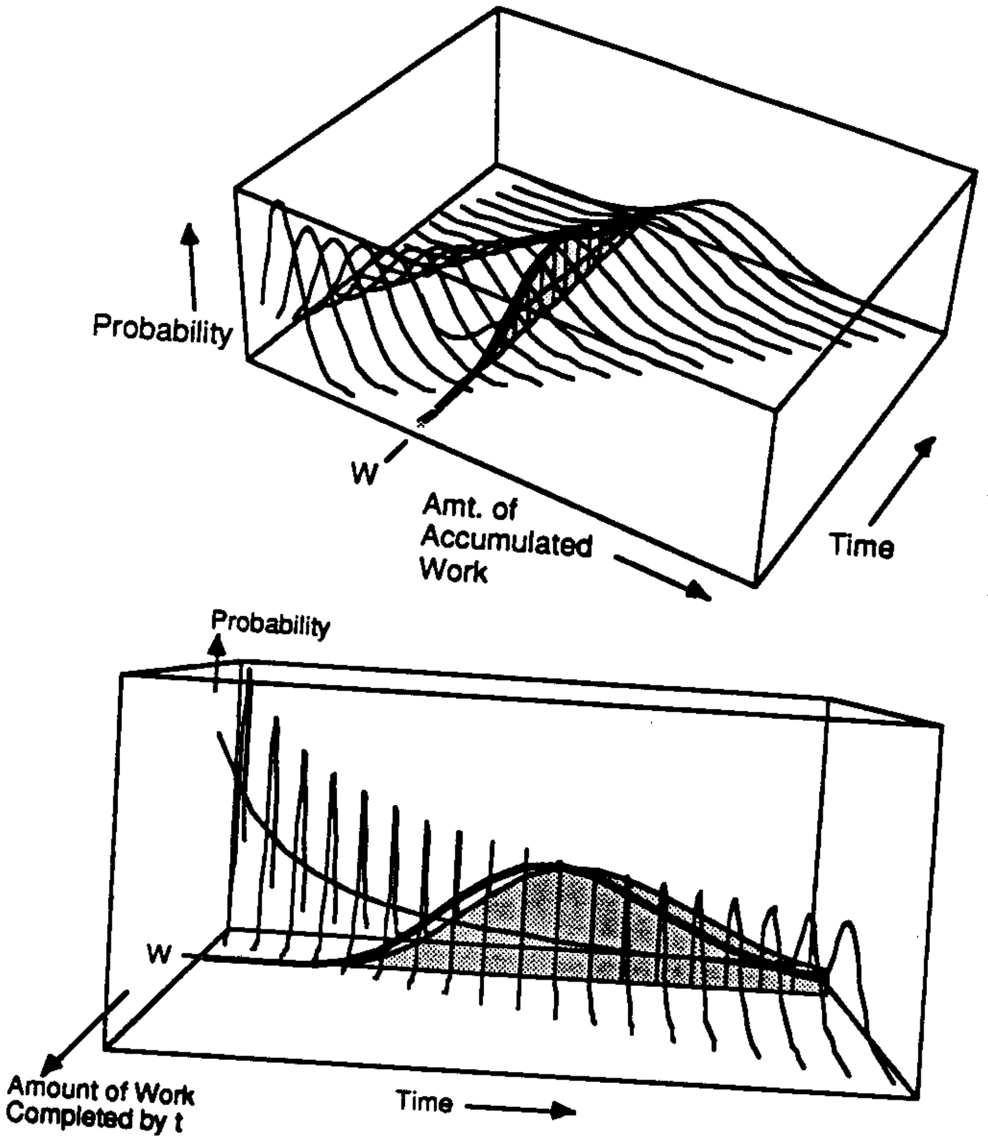


Figure 2.22: Density of finishing time.

2.5.6 Lognormal Approximation to Finishing Time.

A lognormal density provides a remarkably good approximation to Equation 2.36. If we equate the mean and variance of the finishing time pdf from the Brownian motion model to that of a lognormal pdf, then we find that the parameters for the lognormal pdf, μ_l and σ_l^2, must be

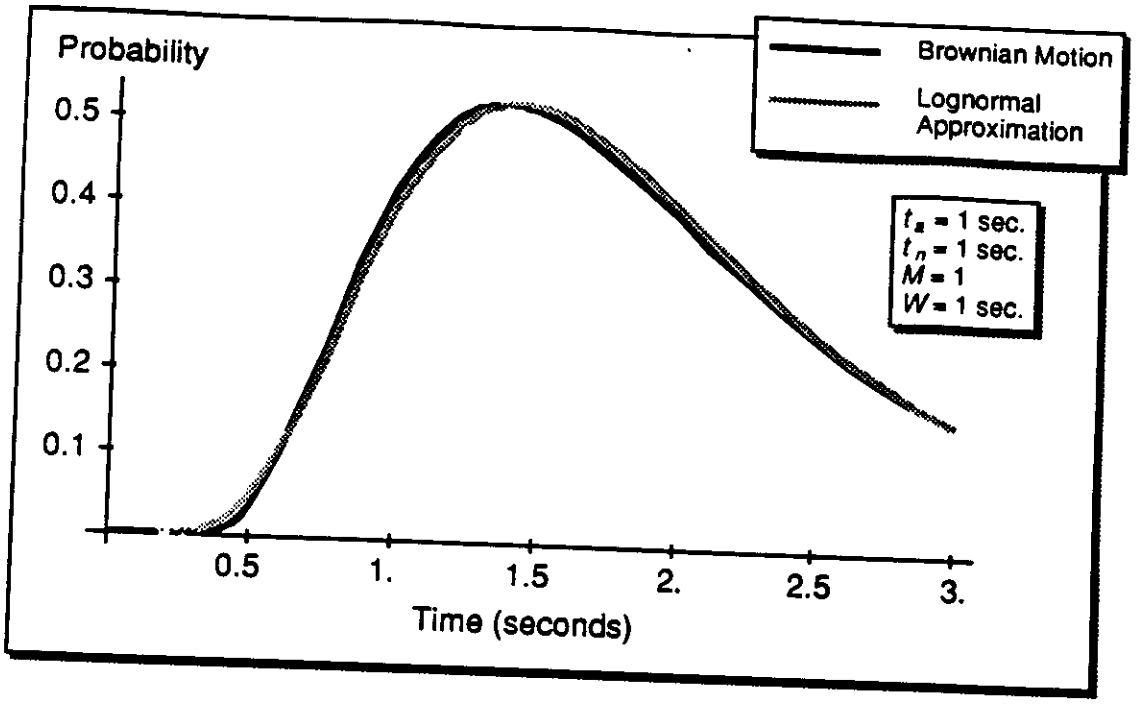
$$\begin{aligned} \mu_l &= \ln(\overline{f}) - \frac{1}{2}\sigma_l^2 \\ &= \ln\left(\frac{\overline{f}}{\sqrt{\sigma_f/\overline{f}^2 + 1}}\right) \end{aligned} \tag{2.54}$$

$$\sigma_l^2 = \ln(\sigma_f^2/\overline{f}^2 + 1), \tag{2.55}$$

where all the logs are natural logs. The actual lognormal pdf then becomes

$$l(t) = \frac{1}{t\sqrt{2\pi\sigma_l^2}} \exp\left[-\frac{(\ln(t) - \mu_l)^2}{2\sigma_l^2}\right]. \tag{2.56}$$

As shown in Figures 2.23, when both the Brownian motion finishing time pdf and the lognormal approximation are plotted, the densities are extremely close. The plot for large W does, in fact, show both curves, but they lie on top of each other and only one is visible. When they do differ, it is under circumstances where the assumptions of the Brownian motion model do not hold (e.g. W small relative to all of M, t_a, and t_n). In spite of the quality of the lognormal approximation, it is not likely to be as useful as the normal and impulse approximations to the finishing time.



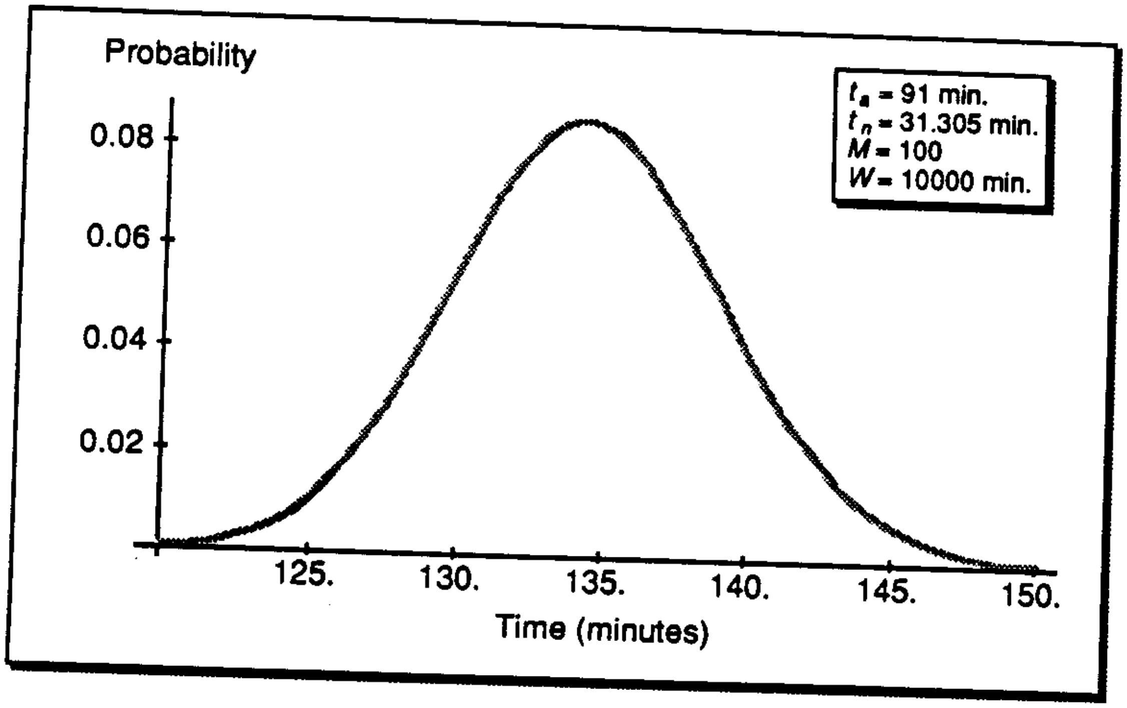


Figure 2.23: Density of finishing time and its lognormal approximation.

2.5.7 Simulation Results.

We ran simulations for the case of exponentially distributed available and non-available periods, and some results are shown in Figure 2.24. The model and the simulation agree very well for large W, as we would expect, and they deviate as W becomes small. In fact, if we examine a previous figure, Fig. 2.21, we notice that for very small W (the top two plots) the model claims considerable probability that the program will finish before the minimum possible finishing time of W/M (0.1 min. in the top plot, and 1 min. in the middle plot). The jagged lines in the plot are an artifact of the histogramming process.

2.5.8 Heterogeneous Networks.

Up to this point, we have assumed that we have a network of homogeneous processors, i.e. all the processors have identical, unchanging characteristics. In this section we allow t_a, t_n, and processor capacity to differ among processors, and we allow t_a, t_n, and M to be chosen randomly when the program starts. We consider the scaled equations Equations 2.29 and 2.30 from the renewal analysis; these are the asymptotic mean and variance of the accumulated work for a single processor with a processing rate of c operations per second. We give these equations again below:

$$\overline{Y_t} = c\,\frac{t_a}{t_a + t_n}\,t \tag{2.57}$$

$$\mathrm{Var}[Y_t] = c^2\,\frac{2(\sigma_a^2 t_n^2 + \sigma_n^2 t_a^2)}{(t_a + t_n)^3}\,t. \tag{2.58}$$

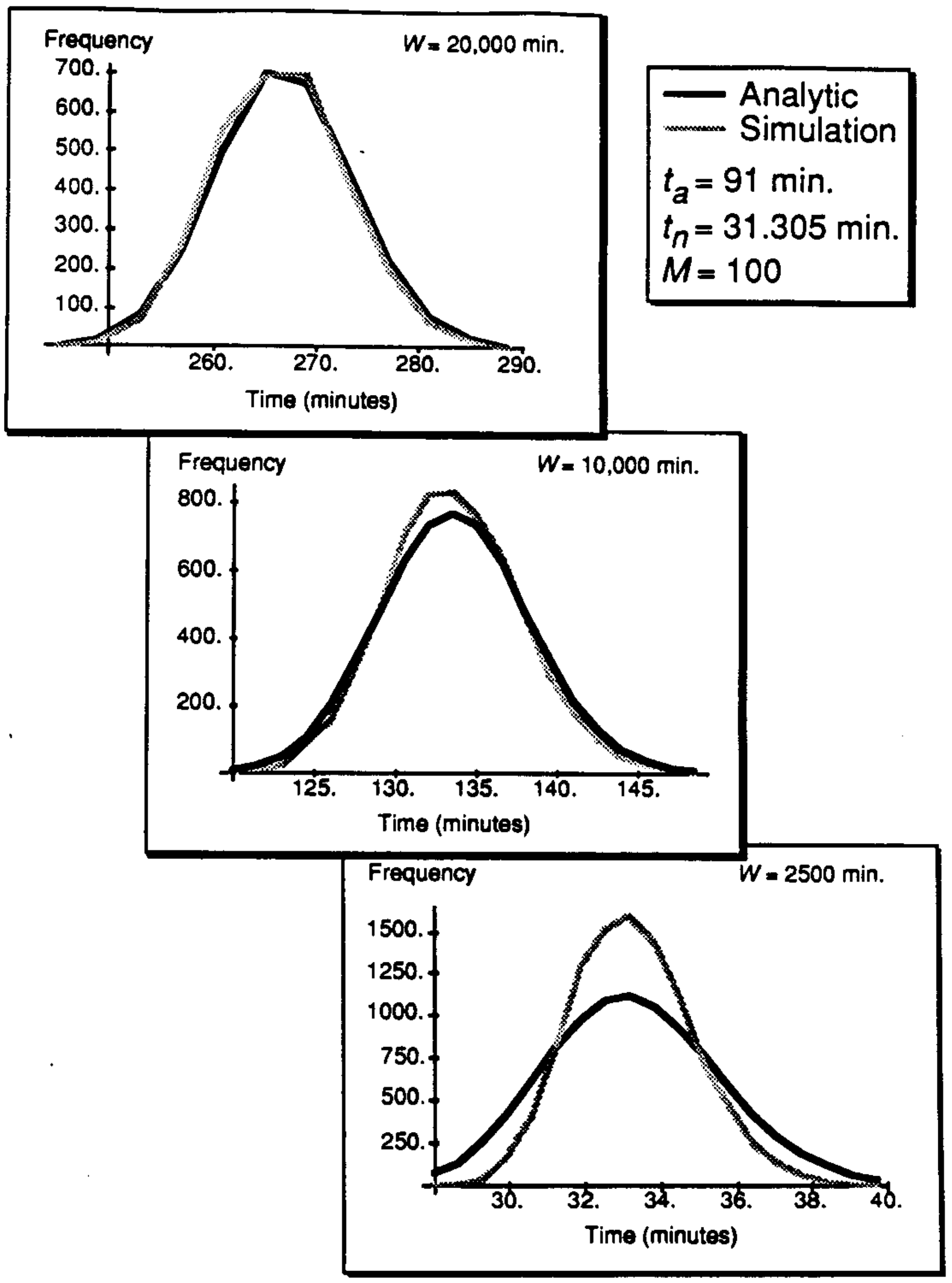


Figure 2.24: Simulation results.

Using these, we get scaled expressions for $\overline{b}$ and σ_b^2:

$$\overline{b} = c \frac{t_a}{t_a + t_n} M \tag{2.59}$$

$$\sigma_b^2 = c^2 \frac{2(\sigma_a^2 t_n^2 + \sigma_n^2 t_a^2)}{(t_a + t_n)^3} M. \tag{2.60}$$

These may then be used in the equation for the finishing time distribution

$$f(t) = \frac{W}{\sqrt{2\pi\sigma_b^2 t^3}} \exp\left[-\frac{(W - \overline{b}t)^2}{2\sigma_b^2 t}\right] \tag{2.61}$$

as before.

2.5.8.1 Non-Uniform Processor Capacity: Static Assignments.

Suppose that the processors in the network work at different rates. Let c_i be the capacity of processor i in minutes of work completed per minute. Assume all processors are identical, except in their capacity, and their characteristics do not change. We use the equations above to give us the mean and variance of a processor's accumulated work per unit time; then, to find $\overline{b}$ and σ_b^2, we sum over all M processors:

$$\overline{b} = \sum_{i=1}^{M} c_i \frac{t_a}{t_a + t_n} = \frac{t_a}{t_a + t_n} \sum_{i=1}^{M} c_i \tag{2.62}$$

$$\sigma_b^2 = \sum_{i=1}^{M} c_i^2 \frac{2(\sigma_a^2 t_n^2 + \sigma_n^2 t_a^2)}{(t_a + t_n)^3} = \frac{2(\sigma_a^2 t_n^2 + \sigma_n^2 t_a^2)}{(t_a + t_n)^3} \sum_{i=1}^{M} c_i^2. \tag{2.63}$$

If $c_i = 1$ for all i, these reduce to Equations 2.33 and 2.34.

2.5.8.2 Non-Uniform t_a and t_n: Static Assignments.

Suppose that each processor has different average available and non-available periods. Let $\overline{b}_i$ and $\sigma_{b(i)}^2$ be the mean and variance of the amount of work done per minute by processor i, and let $t_a^{(i)}$ and $t_n^{(i)}$ be the average length of available and non-available periods at processor i. In general, we can find $\overline{b}i$ and $\sigma_{\overline{b}}^2 i$ for each processor and sum over all processors to get $\overline{b}$ and $\sigma_{\overline{b}}^2$. We do not, in general, know how $t_a^{(i)}$ and $t_n^{(i)}$ affect σ_a^2 and σ_n^2, so let us pick exponential distributions for the available and non-available periods. We know from previous arguments that the amount of work processor i does over time is asymptotically normal with mean and variance

$$\overline{b}_i = \frac{t_a^{(i)}}{t_a^{(i)} + t_n^{(i)}}$$
$$\sigma_{b(i)}^2 = \frac{2(t_a^{(i)} t_n^{(i)})^2}{(t_a^{(i)} + t_n^{(i)})^3}$$

The total work done by the network is the sum of the individual processor's work, therefore:

$$\overline{b} = \sum_{i=1}^{M} \overline{b}_i = \sum_{i=1}^{M} \frac{t_a^{(i)}}{t_a^{(i)} + t_n^{(i)}}. \tag{2.64}$$

Similarly, because all the processors are independent, $\sigma_{\overline{b}}^2$ is the sum of the $\sigma_{b(i)}^2$.

$$\sigma_{\overline{b}}^2 = \sum_{i=1}^{M} \sigma_{b(i)}^2 = \sum_{i=1}^{M} \frac{2(t_a^{(i)} t_n^{(i)})^2}{(t_a^{(i)} + t_n^{(i)})^3}. \tag{2.65}$$

If all processors have identical $t_a^{(i)}$ and $t_n^{(i)}$, these reduce to the definitions of Equations 2.33 and 2.34.

2.5.8.3 Non-Uniform Processor Capacity: Dynamic Assignment.

Suppose that when a program starts, the capacities of each processor are chosen from some distribution (possibly different for each processor), and these do not change during the program's execution. We let the capacity, c_i, of processor i be chosen randomly, taking on the value x with probability $c_i(x)$, and we return to our usual assumption of generally distributed available and non-available periods. For processor i, the average amount of accumulated work per minute, given that $c_i = x$, is $x\ t_a/(t_a + t_n)$. Multiplying by $c_i(x)$ and integrating over all x to uncondition, we find that processor i accumulates, on average,

$$\overline{b}_i = \int_o^\infty c_i(x)\, x \frac{t_a}{t_a + t_n}\, dx = \overline{c_i}\, \frac{t_a}{t_a + t_n},$$

minutes of work per minute, where $\overline{c_i}$ is the average capacity of processor i. We then sum over all M processors to get $\overline{b}$:

$$\overline{b} = \frac{t_a}{t_a + t_n} \sum_{i=1}^{M} \overline{c_i} \tag{2.66}$$

Similarly for the variance, given that $c_i = x$,

$$\sigma^2_{b(i)} | (c_i = x) = x^2 \frac{2(\sigma_a^2 t_n^2 + \sigma_n^2 t_a^2)}{(t_a + t_n)^3}.$$

Uncondition by integrating over all x:

$$\begin{aligned}
\sigma^2_{b(i)} &= \int_0^\infty c_i(x)\, x^2 \frac{2(\sigma_a^2 t_n^2 + \sigma_n^2 t_a^2)}{(t_a + t_n)^3}\, dx \\
&= \frac{2(\sigma_a^2 t_n^2 + \sigma_n^2 t_a^2)}{(t_a + t_n)^3} \int_0^\infty c_i(x)\, x^2\, dx. \\
&= \frac{2(\sigma_a^2 t_n^2 + \sigma_n^2 t_a^2)}{(t_a + t_n)^3}\, \overline{c_i^2},
\end{aligned} \tag{2.67}$$

where $\overline{c_i^2}$ is the second moment of $c_i(x)$. Letting V_i be the coefficient of variation of $c_i(x)$, we can rewrite this as

$$\sigma^2_{b(i)} = \frac{2(\sigma_a^2 t_n^2 + \sigma_n^2 t_a^2)}{(t_a + t_n)^3}\, \overline{c_i}^2 (1 + V_i^2).$$

Sum over all processors to get σ_b^2:

$$\sigma_b^2 = \frac{2(\sigma_a^2 t_n^2 + \sigma_n^2 t_a^2)}{(t_a + t_n)^3} \sum_{i=1}^{M} \overline{c_i}^2 (1 + V_i^2). \tag{2.68}$$

2.5.8.4 Non-Uniform Processor Capacity (Dynamic) and Non-Uniform t_a and t_n (Static).

Let us combine the previous two analyses. For processor i, $t_a^{(i)}$ is the mean available time, $t_n^{(i)}$ is the mean non-available time, and its capacity is chosen randomly when the program starts, but has mean $\overline{c_i}$. We assume M is fixed, and that available and non-available periods are exponentially distributed. We know that to account for varying processor capacity, we multiply $\overline{b}_i$ and $\sigma_{b(i)}^2$ by the moments of c_i. Applying this to Equations 2.64 and 2.65, we get:

$$\overline{b} = \sum_{i=1}^{M} \overline{c_i} \frac{t_a^{(i)}}{t_a^{(i)} + t_n^{(i)}} \tag{2.69}$$

$$\sigma_b^2 = \sum_{i=1}^{M} \overline{c_i}^2 (1 + V_i^2) \frac{2(\sigma_a^2 t_n^2 + \sigma_n^2 t_a^2)}{(t_a + t_n)^3}, \tag{2.70}$$

where V_i is the coefficient of variation of the distribution of c_i. For $c_i = 1$ and t_a and t_n identical for all processors, this reduces to Equation 2.33 and 2.34.

2.5.8.5 M: Dynamic Assignment.

Suppose that when a program starts, the overall number of processors, M, is chosen randomly and remains at that number until the program completes. Let $m(i)$ be the probability that our network has i processors, and this pdf has mean $\overline{M}$ and variance σ_M^2. We resume, for this section, our usual assumption that all processors have identical, unchanging characteristics. We now modify $\overline{b}$ and σ_b^2 to get expressions that we can use in the usual Brownian motion analysis. Given that $M = m$, the mean amount of work done over time is $\overline{b}t = \frac{t_a}{t_a+t_n}mt$. We uncondition to find the overall $\overline{b}$:

$$\overline{b} = \sum_{i=1}^{M} m(i)\frac{t_a}{t_a+t_n}i = \frac{t_a}{t_a+t_n}\sum_{i=1}^{M} i\ m(i) = \frac{t_a}{t_a+t_n}\overline{M}. \qquad (2.71)$$

The variance of the amount of work done over time is similarly conditioned on the value of M, and its analysis is analogous to that of $\overline{b}$:

$$\sigma_b^2 = \sum_{i=1}^{M} i\ m(i)\frac{2(\sigma_a^2t_n^2+\sigma_n^2t_a^2)}{(t_a+t_n)^3} = \frac{2(\sigma_a^2t_n^2+\sigma_n^2t_a^2)}{(t_a+t_n)^3}\sum_{i=1}^{M} i\ m(i) = \frac{2(\sigma_a^2t_n^2+\sigma_n^2t_a^2)}{(t_a+t_n)^3}\overline{M}. \qquad (2.72)$$

If M always takes only one value, Equations 2.71 and 2.72 reduce to the definitions of Equations 2.33 and 2.34.

2.5.8.6 t_a: Dynamic Assignment.

Suppose that t_a is identical for all processors, but it chosen randomly when the program starts, taking on the value x ($x \geq 0$) with probability $p(x)$. The values of the other network parameters remain constant and identical for all processors, and t_a does not change while the program executes. Assume, for this paragraph, that we have exponentially distributed available and non-available periods. As before, we derive new values for $\overline{b}$ and σ_b^2.

Given that $t_a = x$, the mean amount of work done over time is

$$\overline{b} \mid (t_a = x) = \frac{x}{x+t_n}M.$$

We uncondition by integrating over all possible values of x:

$$\bar{b} = \int_0^\infty p(x)\, \frac{x}{x+t_n}\, M\, dx.$$

Let us make the substitution $u = x + t_n$. Continuing,

$$\begin{aligned}
\bar{b} &= M \int_{t_n}^\infty p(u-t_n) \frac{u-t_n}{u}\, du \\
&= M \left[\int_{t_n}^\infty p(u-t_n)\, du \;-\; t_n \int_{t_n}^\infty \frac{p(u-t_n)}{u}\, du \right] \\
&= M - Mt_n \int_{t_n}^\infty \frac{p(u-t_n)}{u}\, du. \qquad (2.73)
\end{aligned}$$

At this point we cannot get any farther without using a specific $p(x)$, so suppose t_a is exponentially distributed with parameter λ: $p(x) = \lambda e^{-\lambda x}$. We must manipulate $\int_{t_n}^\infty \frac{p(u-t_n)}{u} du$ to get it into a more useable form, and we do so with the substitution $y = \lambda u$. Let $\mathrm{Ei}(y) = -\int_{-y}^\infty \frac{e^{-t}}{t} dt$ be the exponential–integral function.

$$\begin{aligned}
\int_{t_n}^\infty \frac{p(u-t_n)}{u} du &= \int_{t_n}^\infty \frac{\lambda e^{-(u-t_n)}}{u} du \\
&= \lambda e^{\lambda t_n} \int_{t_n}^\infty \frac{e^{-\lambda u}}{u} du \\
&= \lambda e^{\lambda t_n} \int_{\lambda t_n}^\infty \frac{e^{-y}}{y} dy \\
&= -\lambda e^{\lambda t_n} \mathrm{Ei}(-\lambda t_n).
\end{aligned}$$

Therefore, if $p(x)$ is exponential with parameter λ, we have

$$\bar{b} = M + \lambda M t_n e^{\lambda t_n} \mathrm{Ei}(-\lambda t_n). \qquad (2.74)$$

We perform a similar analysis for the variance of the amount of work done over time. Because we do not, in general, know how changes in t_a will affect σ_a^2, we must a specific distribution in our analysis, and, as usual, we use exponential. Given that $t_a = x$, we find σ_b^2 to be:

$$\sigma_b^2 \mid (t_a = x) = \frac{2x^2 t_n^2}{(x+t_n)^3} M.$$

We uncondition by integrating over all possible values of x:

$$\sigma_b^2 = \int_0^\infty p(x) \frac{2x^2 t_n^2}{(x+t_n)^3} M \, dx. \tag{2.75}$$

Again, we make the substitution $u = x + t_n$.

$$\begin{aligned} \sigma_b^2 &= 2Mt_n^2 \int_{t_n}^\infty p(u-t_n) \frac{(u-t_n)^2}{u^3} du \\ &= 2Mt_n^2 \left[\int_{t_n}^\infty \frac{p(u-t_n)}{u} du - 2t_n \int_{t_n}^\infty \frac{p(u-t_n)}{u^2} du + t_n^2 \int_{t_n}^\infty \frac{p(u-t_n)}{u^3} du \right] \end{aligned}$$

We can go no farther without choosing a particular $p(x)$, and again we take $p(x)$ to be exponential with parameter λ. Continuing the analysis:

$$\sigma_b^2 = 2Mt_n^2 \lambda e^{\lambda t_n} \left[\int_{t_n}^\infty \frac{e^{-\lambda u}}{u} du - 2t_n \int_{t_n}^\infty \frac{e^{-\lambda u}}{u^2} du + t_n^2 \int_{t_n}^\infty \frac{e^{-\lambda u}}{u^3} du \right] \tag{2.76}$$

Integrating by parts, we find that

$$\begin{aligned} \int_{t_n}^\infty \frac{e^{-\lambda u}}{u^2} du &= \frac{e^{-\lambda t_n}}{t_n} - \lambda \int_{t_n}^\infty \frac{e^{-\lambda u}}{u} du \\ &= \frac{e^{-\lambda t_n}}{t_n} + \lambda \mathrm{Ei}(-\lambda t_n) \end{aligned}$$

and

$$\begin{aligned} \int_{t_n}^\infty \frac{e^{-\lambda u}}{u^3} du &= \frac{e^{-\lambda t_n}}{2t_n^2} - \frac{\lambda}{2} \int_{t_n}^\infty \frac{e^{-\lambda u}}{u^2} du \\ &= \frac{e^{-\lambda t_n}}{2t_n^2} - \frac{\lambda}{2} \frac{e^{-\lambda t_n}}{t_n} - \frac{\lambda^3}{2} \mathrm{Ei}(-\lambda t_n). \end{aligned}$$

When we substitute these back into Equation 2.76 we get

$$\sigma_b^2 = -2Mt_n^2 \lambda e^{\lambda t_n} \left[e^{-\lambda t_n} \left(\frac{3}{2} + \frac{\lambda t_n}{2} \right) + \mathrm{Ei}(-\lambda t_n) \left[\frac{\lambda^3 t_n^2}{2} + 2\lambda t_n + 1 \right] \right]. \tag{2.77}$$

2.5.8.7 t_n: Dynamic Assignment.

Now let us repeat this analysis, but with randomly chosen t_n instead of t_a. Suppose that t_n is identical for all processors, and when the program starts it takes on the value x ($x \geq 0$) with probability $q(x)$. This value remains constant throughout the execution of the program, and the values of the other network parameters remain constant and identical for all processors. As in the previous section, we must use some specific non-available period distribution, and we choose exponential. Using this information, and we derive new values for $\bar{b}$ and σ_b^2.

Given that $t_n = y$, the mean amount of work done over time is

$$\bar{b} \mid (t_n = y) = \frac{t_a}{t_a + y} M.$$

We uncondition by integrating over all possible values of y and substituting $u = t_a + y$:

$$\begin{aligned} \bar{b} &= \int_0^\infty q(y) \frac{t_a}{t_a + t_n} M \, dy \\ &= M \int_{t_a}^\infty q(u - t_a) \frac{t_a}{u} \, du \\ &= M t_a \int_{t_a}^\infty \frac{q(u - t_a)}{u} \, du. \end{aligned} \tag{2.78}$$

To continue any further, we must pick a specific $q(y)$. Let $q(y) = \lambda e^{-\lambda y}$.

$$\begin{aligned} \bar{b} &= M t_a \int_{t_a}^\infty \frac{\lambda e^{-\lambda(u - t_a)}}{u} \, du \\ &= \lambda M t_a e^{\lambda t_a} \int_{t_a}^\infty \frac{\lambda e^{-\lambda u}}{u} \, du \\ &= -\lambda M t_a e^{\lambda t_a} \mathrm{Ei}(-\lambda t_a). \end{aligned} \tag{2.79}$$

The analysis of σ_b^2 is even simpler because t_a and t_n are used symmetrically in the definition of σ_b^2 (either as $t_a + t_n$ or $t_a t_n$). Thus we may exchange t_a and t_n in Equation 2.76, yielding:

$$\begin{aligned}
\sigma_b^2 &= 2Mt_a^2 \int_{t_a}^{\infty} p(u-t_a)\, \frac{(u-t_a)^2}{u^3}\, du \\
&= 2Mt_a^2 \left[\int_{t_a}^{\infty} \frac{q(u-t_a)}{u}\, du - 2t_a \int_{t_a}^{\infty} \frac{q(u-t_a)}{u^2}\, du + t_a^2 \int_{t_a}^{\infty} \frac{q(u-t_a)}{u^3}\, du \right]
\end{aligned} \tag{2.80}$$

in general, and for t_n exponentially distributed with parameter λ:

$$\sigma_b^2 = -2Mt_a^2 \lambda e^{\lambda t_a} \left[e^{-\lambda t_a} (\frac{3}{2} + \frac{\lambda t_a}{2}) + \mathrm{Ei}(-\lambda t_a) \left[\frac{\lambda^3 t_a^2}{2} + 2\lambda t_a + 1 \right] \right]. \tag{2.81}$$

2.5.8.8 M, t_a, and t_n: Dynamic Assignment.

With the results of the preceding sections in hand, we can allow all of M, t_a, and t_n to be chosen randomly when program execution starts. Assume that all processors have identical characteristics, that available and non-available periods are exponentially distributed, and further assume that t_a and t_n are independent of M. From the results of section 2.5.8.5, we can account for M's randomness by using $\overline{M}$ in our formulas instead of M. We make this substitution, and find $\overline{b}$ conditioned on specific values of t_a and t_n:

$$\overline{b} \mid (t_a = x, t_n = y) = \frac{x}{x+y} \overline{M}.$$

We uncondition by integrating over all possible values of t_a and t_n.

$$\overline{b} = \int_0^{\infty} p(x) \int_0^{\infty} q(y)\, \frac{x}{x+y}\, \overline{M}\, dy\, dx.$$

If we substitute $u = x + y$ as before, then, using the results of the previous section:

$$\bar{b} = \int_0^\infty \overline{M}\, x\, p(x) \int_x^\infty \frac{q(u-x)}{u}\, du\, dx. \tag{2.82}$$

If, as before, we choose $q(y)$ to be exponential with parameter λ, this becomes

$$\bar{b} = -\overline{M}\lambda \int_0^\infty p(x)\, x\, e^{\lambda x}\, \mathrm{Ei}(-\lambda x)\, dx. \tag{2.83}$$

At this point, further analysis is intractable.

We follow the same procedure for the variance. Conditioning on specific values of t_a and t_n:

$$\sigma_b^2 \mid (t_a = x, t_n = y) = \frac{2x^2y^2}{(x+y)^3} M.$$

Unconditioning, and using the results of the previous sections, we arrive at:

$$\begin{aligned} \sigma_b^2 &= \int_0^\infty p(x) 2Mx^2 \left[\int_x^\infty \frac{q(u-x)}{u}\, du \right. \\ &\qquad \left. -2x \int_x^\infty \frac{q(u-x)}{u^2}\, du + x^2 \int_x^\infty \frac{q(u-x)}{u^3}\, du \right] dx. \end{aligned} \tag{2.84}$$

We can again let $q(y)$ be exponential with parameter λ, and get slightly farther:

$$\begin{aligned} \sigma_b^2 &= -\int_0^\infty p(x) 2Mx^2 \lambda e^{\lambda x} \left[e^{-\lambda x}\left(\frac{3}{2} + \frac{\lambda x}{2}\right) \right. \\ &\qquad \left. + \mathrm{Ei}(-\lambda x) \left[\frac{\lambda^3 x^2}{2} + 2\lambda x + 1 \right] \right] dx, \end{aligned} \tag{2.85}$$

but analysis is difficult beyond this point.

Notice that all these manipulations of processors capacity, t_a, t_n, and M do not change the basic form of the finishing time, just its mean and variance. This implies that the Brownian motion model is quite robust, thanks primarily to the central limit theorem.

2.5.9 Random Work and Multiple Stages.

All the preceding analysis examined programs consisting of only a single stage requiring W seconds of work; when the stage finished, the program was done. We now allow more general programs, consisting of K stages, and each stage requiring w_i units of work (for $i = 1, \ldots, K$). The finishing time distribution for a multi-stage algorithm is the convolution of the finishing time distributions of the individual stages. Let us work with our Brownian motion model in the transform domain. The Laplace transform of ft from Equation 2.36 is

$$F_1^*(s) = e^{(W/\bar{b})\left(\bar{b}-\sqrt{\bar{b}^2+2\sigma_b^2 s}\right)} \tag{2.86}$$

The transform of the finishing time pdf of each individual stage is this equation with the appropriate w_i substituted for W. Multiplying the transform of each stage results in the finishing time transform for multiple stages:

$$F_m^*(s) = e^{-\left(\sum_{i=1}^{K} w_i/\sigma_b^2\right)\left(\bar{b}-\sqrt{\bar{b}^2+2\sigma_b^2 s}\right)}. \tag{2.87}$$

We see that the number of stages has no effect on the finishing time pdf: a multistage job and a single stage job have the same finishing time pdf if their total work is the same. This is to be expected because we made the simplifying assumption that work can always be divided evenly between all the available processors — thus at the end of a stage all processors finish simultaneously, and can start the next stage without waiting for late processors, as would happen in a real system.

Suppose the work in each stage is chosen randomly when that stage starts execution. We define the random variable $\tilde{w}$ to be the amount of work in each stage. The pdf of this random variable is $w(x)$, with mean $\overline{w}$ and variance σ_w^2. Given that $\tilde{w} = w$, the probability density of the finishing time is $f(t \mid \tilde{w} = w)$. We integrate over all possible w to find the unconditional finishing time density for one stage:

$$g(t) = \int_{x=0}^{\infty} w(x)\, f(t \mid x)\, dx \tag{2.88}$$

Let us find the mean and variance of this expression. For the mean, we take the expectation with respect to t:

$$\begin{aligned} \mathrm{E}[g(t)] &= \mathrm{E}\left[\int_{x=0}^{\infty} w(x) f(t \mid x)\, dx\right] \\ &= \int_{x=0}^{\infty} w(x)\, E[f(t \mid x)]\, dx \\ &= \int_{x=0}^{\infty} w(x)\, (\overline{f} \mid x), dx \\ &= \int_{x=0}^{\infty} w(x)\, (x/\overline{b})\, dx \\ &= \overline{w}/\overline{b} \end{aligned} \tag{2.89}$$

The variance is similar:

$$\begin{aligned} \mathrm{Var}[g(t)] &= \mathrm{E}[(g(t))^2] - [\mathrm{E}[g(t)]]^2 \\ &= \mathrm{E}\left[\int_{x=0}^{\infty} w(w)(f(t \mid w))^2\, dw\right] - \left[\mathrm{E}\left[\int_{x=0}^{\infty} w(x) f(t \mid x)\, dx\right]\right]^2 \\ &= \int_{x=0}^{\infty} w(x)\, \mathrm{E}[(f(t \mid x))^2]]\, dx - \int_{x=0}^{\infty} w(x)\ [\mathrm{E}[f(t \mid x)]]^2\, dx \\ &= \int_{x=0}^{\infty} w(x)(\mathrm{E}[(f(t \mid x))^2] - [\mathrm{E}[f(t \mid x)]]^2)\, dx \\ &= \int_{x=0}^{\infty} w(x)\, \sigma_f^2\, dx \end{aligned}$$

$$
\begin{aligned}
&= \int_{x=0}^{\infty} w(x)\, x\, (\sigma_b^2/\overline{b}^3)\, dx \\
&= \overline{w}\sigma_b^2/\overline{b}^3 \qquad (2.90)
\end{aligned}
$$

It is interesting to note that only the average of $w(x)$ figures into the mean and variance of the single-stage finishing time.

Now we give each program a random number of stages, $\tilde{n}$, drawn from a density $\mathcal{N}(i)$, with mean $\overline{N}$, variance σ_N^2, and Z-transform $N(z)$. We let $F_1^*(s)$ be the Laplace transform of the density of the finishing time for a single stage. Given $\tilde{n} = i$, we have

$$F^*(s \mid \tilde{n} = i) = [F_1^*(s)]^i.$$

We uncondition by summing over all possible values of $\tilde{n}$ to get:

$$
\begin{aligned}
F^*(s) &= \sum_{i=1}^{\infty} \mathcal{N}(i)[F_1^*(s)]^i \\
&= N(F_1^*(s)) \qquad (2.91)
\end{aligned}
$$

This is the usual transform of a random sum of random variables. Letting μ_1 and σ_1^2 be the mean and variance of single stage finishing time pdf, we use [Kle75] to find the mean and variance of the finishing time for the whole program:

$$\overline{f} = \overline{N\mu_1} \qquad (2.92)$$

$$\sigma_f^2 = \overline{N}\sigma_1^2 + \sigma_N^2\mu_1^2. \qquad (2.93)$$

If each stage has a random amount of work, as in the previous paragraph, then for μ_1 and σ_1^2 we use the results of Equations 2.89 and 2.90, respectively, and finally get that

$$\overline{f} = \frac{\overline{N}\,\overline{w}}{\overline{b}} \quad (2.94)$$

$$\sigma_f^2 = \frac{\overline{N}\,\overline{w}\sigma_b^2}{\overline{b}^3} + \sigma_N^2(\frac{\overline{w}}{\overline{b}})^2. \quad (2.95)$$

For a single stage and a constant amount of work per stage, these reduce to the familiar mean and variance of our initial Brownian motion model, Equations 2.37 and 2.38, respectively.

2.5.10 Multiple Programs in a Network of Transient Processors.

All the previous work in this chapter studied one program that had a network of transient processors to itself. However, a realistic system is likely to allow multiple programs to run simultaneously, each getting a portion of idle time of the processors. In the previous sections, we found four possible models for the finishing time pdf of a program:

1. The Brownian motion finishing time density.
2. The normal approximation to Brownian motion.
3. The impulse approximation to Brownian motion.
4. The lognormal approximation to Brownian motion.

If we now view each program as a job in a queueing system, and the service requirements of each job as the corresponding program's finishing time, then we can use these results in a variety of well–analyzed queueing systems to find the overall network performance, particularly, $\mathcal{W}$, the average waiting time before the program starts.

The arrival rate to the queueing system is the arrival rate of programs to the network, and the service time distribution represents the distribution of the time the programs require from the network. We choose the number of servers to be the number of programs allowed in the network. A single server model means that each program gets exclusive access to the network, as was the situation in the preceding parts of this chapter. A variation on this is a processor sharing scheme, where each program gets M processors, but their processing power is evenly divided among all the executing programs. A model with m servers and $m = M/k$, for some k and integer > 1, represents a network where each program gets a fixed number of processors, k, to use. As a special case of this model, we will discuss bulk–arrival queueing systems. Finally, we could have an infinite server model, which means that no program ever waits for access to the network, but this model is uninteresting and we will not discuss it.

Notation. We use λ as the average arrival rate in jobs (programs) per minute. The average service time (running time of a program in minutes) is $\overline{x} = \overline{\mathcal{J}}$, and $\rho = \lambda\overline{x}/m$, where m is the number of servers in the model.

2.5.10.1 Stability Conditions.

For a queueing system to be stable, it must have $\rho < 1$. Because we expect to be running large jobs, $\overline{x}$ will be quite large, and as a consequence, λ must be fairly small. Take as an example our standard multiprocessor example, with $t_a =$ 91 min., $t_n = 31.305$ min, $M = 100$, and $W = 10^4$. Each job finishes, on average, after 134 minutes, which is about 2 hours and 15 minutes. As a consequence, we cannot have an average arrival rate of more than one job every 134 minutes, and from a practical point of view, we would prefer not to operate very near $\rho = 1$, and so an average arrival rate of, say, one job every 3 hours ($\rho = .74$) might be better. This means that we allow, on average, 8 jobs per day. This is not a large number, and in a large institution, it could easily be exceeded. Any practical system would require monitoring or control to ensure that it remains stable.

2.5.10.2 Single Server Queueing Systems — G/G/1.

A single sever queueing system models the situation where only one program at a time runs on the network, and it can use every available processor. The arrival rate process may be arbitrary, but Markovian arrival processes have served well in many other queueing system models of arrivals to computer systems [Kle75] [Kle76], so it is likely that such an arrival process would also be suitable for this situation.

The service time distribution, on the other hand, generally should not be Markovian (exponential) for best accuracy. The Brownian motion model assumed, roughly speaking, that each program had much work and would not finish too quickly, but an exponential distribution has too much probability in small service times.

We will now quote some of the general results for a single server queue. If we require that both arrival and service time distributions are general, we do not have an exact expression for the average waiting time, and the techniques of chapter 8 in [Kle75] are necessary. We do, however, have bounds on the average waiting time, $\mathcal{W}$ [Kle76]:

$$\frac{\rho^2 C_b^2 + \rho(\rho - 2)}{2\lambda(1-\rho)} \leq \mathcal{W} \leq \frac{\sigma_a^2 + \sigma_b^2}{2\overline{x}(1-\rho)}$$

where σ_a^2 and σ_b^2 are the variance of the arrival and service times, respectively, and C_b is the service time coefficient of variation. Note that the lower bound is valid only for $C_b^2 \geq (2-\rho)\rho$.

Poisson Arrivals. If our arrival process is Poisson, then we have an expression for the average waiting time:

$$\mathcal{W} = \frac{\rho\overline{x}(1 + C_b^2)}{2(1-\rho)} = \frac{\lambda\overline{x^2}}{2(1-\rho)}$$

where $\overline{x^2}$ is the second moment of the service time.

We can further modify the M/G/1 model by allowing multiprogramming of programs on the network: each program has access to all machines, and each machine is shared, via round-robin scheduling, between all programs. In a word, this is a processor-sharing model. Our average waiting time for a program needing

x minutes of processing then becomes [Kle76]:

$$\mathcal{W}(x) = \frac{\rho x}{1-\rho}$$

and the average system time (waiting time plus running time) for an x minute program is

$$T(x) = \frac{x}{1-\rho}$$

From this we conclude that for small ρ, multiprogramming does not hurt the response time very much. However, small ρ means that either programs arrive at a very low rate (and the maximum permissible rate is not very high to begin with), or the average job requires relatively little work. Given the purpose of our network, the second condition is likely to be untrue. Thus, one must analyze the job load carefully before introducing multiprogramming of distributed programs into a transient processor network.

2.5.10.3 *m* Server Queueing Systems — G/G/m.

If our queueing model has $m = M/k$ servers (k an integer > 1), we model a network in which each program runs on a fixed number (k) of processors. Why would we want to restrict the number of processors each program can use if the object of distributed processing is to use many processors in parallel?

One reason is to model real program behavior. Typically, speeding up a distributed computation by using more processors works only up to a point, and then additional processors have little effect on the finishing time. It would be very misleading to model a program as running effectively on all M processors in the network, when, in fact, it makes effective use of only k ($k < M$) processors and the remaining $M-k$ processors make no contribution to program speed (and in some cases, may even be a hindrance by causing more network traffic).

Another reason for using k processors is fairness: we want to let several programs run simultaneously and prevent one program from taking all the computing resources in the network. As an implementation of this idea, one could split the processors evenly among all the jobs in the system — if there are j programs in the system, each runs on $\lfloor M/j \rfloor$ processors. In this case of dynamic partitioning of processors, our model is really processor sharing, not multiple servers, and the round–robin results discussed in the previous paragraph are applicable. When we have fixed and equal sized partitions of processors, the G/G/m model applies.

The G/G/m queueing model is very difficult to analyze, and the best we can do is quote bounds on the average waiting time [Kle76]:

$$\frac{\rho^2 C_b^2 - \rho(2-\rho)}{2\lambda(1-\rho)} - \frac{[(m-1)/m]\overline{x^2}}{2\overline{x}} \leq W \leq \frac{\sigma_a^2 + (1/m)\sigma_b^2 + [(m-1)/m]\overline{x}^2}{2(1-\rho)/\lambda}$$

where $\rho = \overline{x}/(m\overline{t})$, and σ_a^2 and σ_b^2 are the variance of the arrival and service time distributions, respectively, and C_b^2 is the squared coefficient of variation of the service time. The lower bound, although simple, is not extremely useful because it is non-negative only for

$$C_b^2 \geq \frac{2\rho}{\rho^2 - \frac{m-1}{m}\lambda\overline{x}(1-\rho)} - 1.$$

See chapter 2 of [Kle76] for details.

If the arrival processes is Poisson, then we may use an approximation for the average waiting time (see [Mal73], [NR76], or [BCH79]):

$$\mathcal{W}_{\mathrm{M/G/m}} = \mathcal{W}_{\mathrm{M/G/1}} \frac{\mathcal{W}_{\mathrm{M/M/m}}}{\mathcal{W}_{\mathrm{M/M/1}}}, \tag{2.96}$$

where $\mathcal{W}_{\mathrm{M/G/m}}$ is the average waiting time an M/G/m system, $\mathcal{W}_{\mathrm{M/G/1}}$ is the average waiting time for an M/G/1 system with the same service time but $1/m^{\mathrm{th}}$ the arrival rate of the M/G/m system, and $\mathcal{W}_{\mathrm{M/M/1}}$ and $\mathcal{W}_{\mathrm{M/M/m}}$ are the average waiting times for systems with the same arrival rate as the M/G/1 and M/G/m systems, respectively, but with exponentially distributed service times whose means are the same as that of the general distribution.

In addition to these results, we know that when $C_b^2 \leq 1$, the total system time (waiting plus service) in a G/G/1 system is less than or equal to the system time in a G/G/m system [Kle76]. Thus to minimize the total system time under these circumstances, we should allow each program to use as many processors as it wishes. Figure 2.25 shows these bounds on the waiting time as ρ varies from 0 to 1 when we have Poisson arrivals, a 2-stage Erlang service time, 10^4 minutes of work to do, 100 processors, and each program is given $m = 20$ processors in the $\mathrm{M/E_2/m}$ case. The solid line is the upper bound for $\mathrm{M/E_2/m}$, the dashed line is the average waiting time if each program gets all the processors (a $\mathrm{M/E_2/1}$ system), and the lower bound for $\mathrm{M/E_2/m}$ lies on the x-axis.

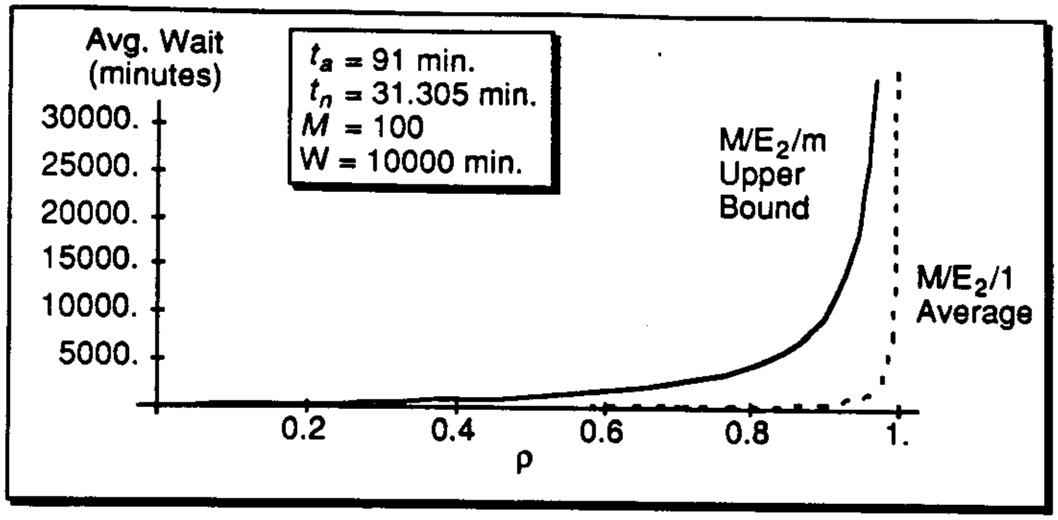


Figure 2.25: Waiting time bounds for G/G/m and G/G/1.

2.5.10.4 M Server Queueing Systems — Bulk Arrivals.

In a real distributed processing system, a program is not infinitely divisible between processors, and it is more accurate to think of the program as a collection of discrete pieces of work, each of which is not divisible. This leads to a model using a queueing system with bulk arrivals. Let a program needing a total of W minutes of work be broken into W equal-sized tasks, each needing 1 minute of work. Each bulk represents a program, and the bulk size varies randomly to model variations in amount of work each program requires. We use an M server queue to model our M processors, and we wish to find the time it takes before for the bulk starts service, and the time it takes for the bulk to complete service. Chaudhry and Templeton [CT83] analyze M/M/m bulk-arrival systems, deriving an expression for the distribution of time before the first task in a bulk (program) enters service. From this we derive an approximation for the system time of an entire bulk by letting the service time of a bulk of size $\tilde{a}$ be the distribution of the time it takes for $\tilde{a}$ tasks to finish service. This is not the true time for a bulk to finish service because tasks do not necessarily leave the system in the order they arrived, and conceivably a task from an earlier bulk could remain in service while several other bulks pass through the system. Furthermore, we do not account for precedence delays; thus, any results from this derivation will be optimistic. The final approximation for the total system time of a bulk is the distribution of time until the bulk first starts service, convolved with our approximation for the service time of a bulk. The expected system time for a bulk is sum of the expectations of both constituent distributions, which is

$$T = \sum_{d=1}^{\infty} d\overline{x} P_{d+m-1} + \beta\overline{x}$$

where $\overline{x}$ is the average service time for a single task (one fragment of the bulk), β is the average bulk size, and P_i is the probability of there being i tasks in the system (queue plus servers).

Chaudhry and Templeton also analyze the M/D/m bulk arrival system, which is better suited to our purposes because we may use our impulse approximation to the finishing time and take the service time of a task to be deterministic. Unfortunately, for our purposes they provide nothing more useful than $P(z)$, the Z-transform of the steady state probability of the number in the system. If we could invert this, then we could find the system time for a bulk as follows. Let $N(z)$ be the Z-transform of the number of tasks in the queueing system when a bulk arrives. Because we have Poisson arrivals, the Z-transform of the number of tasks that a bulk finds in the system is equal to $P(z)$. Multiply this by the Z-transform of the bulk size to get $N(z)$. Let $B(s)$ be the Laplace transform of the time to service a task. To find $F(s)$, the transform of the time for a bulk to finish, including its waiting time, we raise $B(s)$ to the j^{th} power, multiply by the probability of having j tasks in the system just after an arrival, and sum over all j. This represents a random sum of random variables, and results in the equation $F(s) = N(B(s))$. In theory we can invert this; regardless of its difficulty of inversion, we can find the moments of this distribution.

2.6 Conclusion.

In this chapter, we analyzed the distribution of finishing time for a distributed program in a network of transient processors. We first found the distribution of the number of available processors in the network, then made two analyses of a program running on a single transient processor. The results of this were then used as the basis for a multiprocessor model using Brownian motion, and from this we found four finishing time distributions: the actual Brownian motion finishing time distribution, and its normal, lognormal, and impulse approximations. These distributions, the result of analyzing a single program in isolation, were then used in several queueing models that allow multiple jobs to be present in the network.

Our models have several shortcomings that we would like to remove. First, the results are really asymptotic results, and valid only over a long period of time. If we have a relatively small amount of work to do (say, several hours), then our finishing time distributions are not valid. Their means are acceptable, but the variance is quite incorrect, and the distributions (except for the impulse approximation) show noticeable probability that the program will finish in less than the minimal time required (W/M). Perhaps there is some simple way to modify the variance expression in our models so they provide acceptable results for small W.

A second shortcoming is our model of a program. Modeling an algorithm as sequential, independent stages in very simplistic. Many programs do not have clear stages, but instead have a more complex internal structure that allows precedence relationships (between pieces of the program executing on different processors) which cause additional delays. Modeling these relationships is not a trivial task, although the work of Belghith [Bel87] may provide a starting point. Also, we must test the independent stages model against real systems, for it may provide reasonable results in a variety of situations, and then we need not always deal with the complexities of precedence.

Third, and of great importance, our network model does not account for the realities of communications. Communication entails delay, and our model does not address this issue. It is reasonable to put much of the burden of communications analysis into our determination of W — each program has different communication patterns and requirements, so each must be analyzed separately.

However, we should account for the global communications load because that affects the communications delays for each program. If a network is heavily loaded, all communications suffers delays and all programs are slowed. Also, in lower-capacity networks, a program may provide enough communications load that it can saturate the network by itself. In the conclusion of this dissertation we discuss an approach to this problem.

In spite of these difficulties, the very simplicity of the models in this chapter makes them appealing, and, at least for large W, they do capture the basic behavior of transient, distributed systems.

CHAPTER 3

Information Theory and Single-Processor Sorting

3.1 Introduction.

Sorting is one of the most common uses of a computer. Anyone who keeps a database of any size, large or small, or who even just keeps a simple list of information, will probably sort the information at some time. Sorting is certain to be one of the applications running on a network of transient processors. Fast sorting algorithms, efficient in both space and time, have long been sought, and many have been proposed. In this chapter we examine three of these algorithms.

To sort, we start with a list of keys in random order, and we want to finish with a list of keys in ascending order. Some sorting algorithms constrain the form of the keys (e.g. keys are 10 digit numbers), but in general, keys consist of letters and digits and have a bounded number of characters. Some of our analyses will assume that the keys are drawn from a known set without repetition, but we avoid such assumptions whenever possible.

The spectrum of sorting algorithms may be divided into two families, *internal sorts* and *external sorts*. Internal sorts manipulate data that resides entirely within the memory of the computer; External sorts manipulate data that resides on some input/output (i/o) device, typically a disk or tape. The algorithms differ because of the use of i/o devices in external sorts. Internal sorts have no i/o considerations, and minimizing the number of instructions executed maximizes the speed of the sort. External sorts, however, must deal with the speed of the i/o devices, which are typically orders of magnitude slower than the processor; for these sorts, minimizing i/o time is of paramount importance. In both families, the most interesting characteristics of a sort (from a user's standpoint) are its running time, and the amount of memory it requires.

Distributed sorts have characteristics and concerns of both internal and external sorting algorithms. Typically the information that any one processor works with will fit into its core memory, like an internal sort, but the algorithm must also deal with delays that arise from communication or task precedence. These delays, like the i/o delays of external sorts, have a very important effect on the design and performance of distributed algorithms.

In this chapter we use information theory to examine internal sorting on a single processor. In doing so, we learn what makes these algorithms efficient or inefficient. In the next chapter we discuss some distributed sorting algorithms.

Some of the more common internal sorting algorithms are *bubblesort, mergesort, quicksort*, and *radix sort*. The first three are comparison-based sorts: the basic action is to compare two data items and act according to that result. A single comparison is taken as the basic time unit of these sorts on a single processor, and minimizing the number of comparisons is the goal of an algorithm.

In distributed systems, this is still true to some degree, but we will also have to deal with communication and precedence delays if we want a fast algorithm.

Radix sort is a different type of algorithm, in which data elements are not compared directly, but are put into buckets based on the digit or character in a certain position in the key. Under the proper conditions, it can be faster than comparison-based sorts.

In the remainder of this chapter, we first define notation in section 3.1.1 and discuss information theory and entropy in section 3.1.2. Then we examine entropy reduction in four sorts: bubblesort, mergesort, quicksort, and radix sort.

3.1.1 Notation.

An algorithm is presented with a list of N items to be sorted. The actual form of the data items, be they numbers or strings, is unimportant. In general, this list can contain duplicates, but unless otherwise stated, we will ease our analysis and assume that there are no duplicates. An ordering is defined on the data items (not necessarily unique if the input list contains duplicates), and the purpose of the algorithm is to produce an ordered list of the input items. We refer to the i^{th} item on the list as $d[i]$. An algorithm starts with all N items in core memory of the *starting* processor, and ends when all processors have finished running the algorithm and the sorted list of N items is in the core memory of the starting processor. For ease of notation, processors will be numbered from 1 to M, where M is the total number of processors, and typically we will let processor 1 be the starting processor.

In pictures, a list of data is depicted either in a vertical column, indexed with data item number 1 at the top of the list, or horizontally with data item 1 toward the left side. The list is sorted when it is in increasing order, that is, $d[i] \leq d[i+1]$ for $i = 1, ..., N$, where "$\leq$" is an ordering function.

We use "Big O" notation, as it is commonly defined. A function $f(n)$ is $O(g)$ if $f(n) < Kg(n)$ for all n and some constant K.

For comparison-based algorithms, our unit time is the time for a comparison. Thus, we measure the length of these algorithms in terms of the number of comparisons. We discuss the time measure for radix sort in the section analyzing this sort. We use the notation ***comparison/swap*** (abbreviated *c/s*) to refer to a comparison of two data items, possibly followed by a swap of these two data items, depending upon the results of the comparison.

3.1.2 Information Theory and Entropy.

Sorting may be viewed as a process of reducing entropy. An algorithm starts with a list of items in an unknown order, and finishes with the list in a known order. To do this, the algorithm must discover information about the items in the list, and reduce the algorithm's uncertainty about the ordering of the items. Conceptually, the data given to a sorting algorithm may be any one of $N!$ permutations. By shuffling the data around in an organized fashion, the algorithm reduces the number of possible permutations of the data. When the data has only one possible permutation, the sort is finished. This may be formalized by using information theory. Shannon and Weaver[SW49] define entropy using the following expression:

$$E = \sum_{i=1}^{S} -p_i \log_2 p_i \tag{3.1}$$

where p_i is the probability that our system is in a particular state and S is the total number of possible states. In our problem, the set of system states is the set of all possible orderings of the input list. We will assume that inputs to an algorithm are uniformly distributed among all possible permutations of the input list; in other words, the input does not tend to have some initial order.

When an algorithm starts, the entropy is equal to $\log_2 N!$. With every action an (efficient) algorithm takes, it reduces the entropy by reducing the number of possible permutations of the data, gaining at most 1 bit of information per comparison. When the entropy reaches zero, which means there is only one possible permutation of the data, the algorithm has finished. By applying Equation 3.1, we can, at least in principle, quantify the amount of entropy remaining as an algorithm progresses. In practice this tends to be quite difficult, but we can often obtain bounds on the entropy or obtain the entropy at particular points during the algorithm's execution.

Some care must be taken in the analysis because simplistic analysis will lead to incorrect results. Specifically, for comparison–based sorts to gain one bit of information per comparison, then each comparison must halve the number of possible permutations while doubling the probability of the permutations that remain. For bubblesort-based algorithms (explained in the next section), this implies that the probability of swapping two data items after a comparison is 1/2. The converse is not true, however, and one can construct an algorithm that has probability 1/2 of swapping after each comparison, yet does not always gain one bit of information per comparison.

We may use entropy analysis to bound the performance of algorithms. It has been proven [AHU74] that the most efficient single-processor sort based on comparisons cannot do better than $O(N \log N)$ time. Typically this is proved by making a binary tree where each internal node represents a comparison, and then noting that the tree has $N!$ leaves. Because of this, its height, representing the time to finish, must be at least $\log_2(N!)$, which is $O(N \log N)$. We see that information theory makes the same conclusion in a slightly different manner. The data's $N!$ possible permutations (assumed to be equally likely) mean that the entropy of an unsorted list is $\log_2(N!)$. We cannot gain more than one bit of information per comparison, so the minimum number of comparisons is $\lceil \log_2(N!) \rceil$, which, as we know, is $O(N \log N)$. Thus, any sort in which we do not gain one bit (or at least some constant fraction of a bit) per comparison cannot be $O(N \log N)$.

Much work has been done on optimal sorting algorithms. The Ford-Johnson sorting algorithm [JJ59], also described as the Ford-Johnson merge-insertion algorithm in [Knu73], has been shown in [HL69] to require at most $N \log N -$

$1.329N + O(\log N)$ comparisons, as compared to the information-theoretic lower bound of $\lceil \log_2(N!) \rceil = N \log_2 N - 1.443N + O(\log_2 N)$. This has been improved on, most recently in [MBM89] by using the Ford-Johnson algorithm and appropriate merging algorithms. The information-theoretic lower bound can be achieved for some N , such as $N \leq 11$, but no sorting algorithm can achieve this bound for all N [Wel66]. Given the extensive work in this area, this chapter will not propose new sorting algorithms, but instead will examine some widely-used existing algorithms to find the source of their efficiency (or for bubblesort, inefficiency).

3.2 Entropy in Bubblesort.

3.2.1 Description of Bubblesort.

Bubblesort on a single processor has been discussed in a variety of publications, [HS78] and [AHU74] being typical sources.

Bubblesort works by comparing adjacent items in the input list, and swapping them if they are out of order. The algorithm makes passes through the list until it is in order. Figure 3.1 illustrates bubblesort on 8 data items. In the worst case, the data list is in reverse order, and moving the i^{th} largest data item to its proper position is done on the i^{th} pass at a cost of $N - i$ comparisons. Adding these up, we find that bubblesort takes at most $N(N-1)/2$ comparisons, which means it is $O(N^2)$. We note that the list data items gradually become sorted,

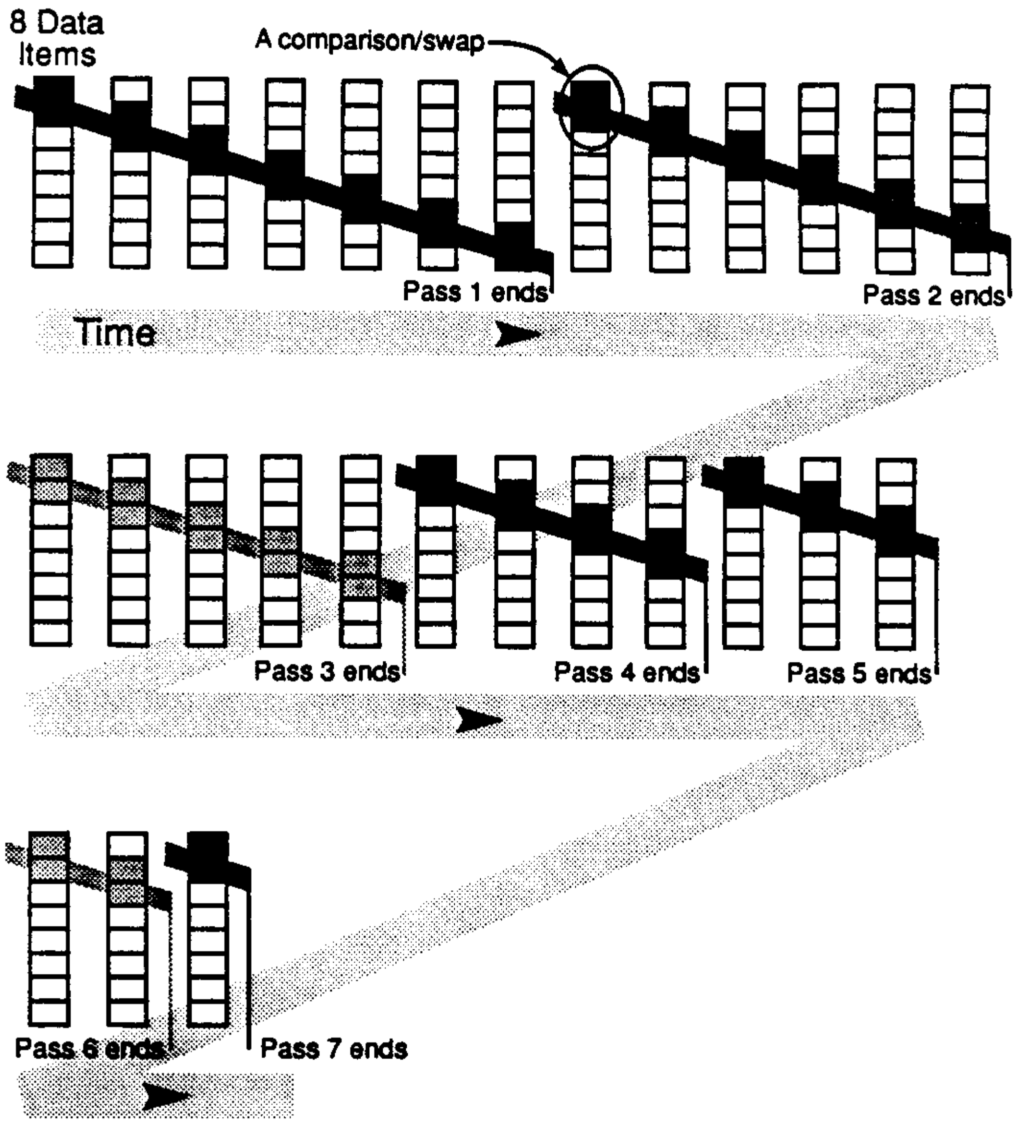


Figure 3.1: Single processor bubblesort with 8 data items.

starting with the largest items, and if the algorithm ever completes one pass without swapping any data items, it can finish immediately because subsequent passes will not make any swaps, either.

3.2.2 Notation.

To discuss data items in a symbolic fashion without reference to their actual location in the list or their particular contents, we will use capital letters (A, B, C, etc.). That is, think of a data item as a key in a box. The name of the box is A, B, etc., and its value, $Val[A]$, is the value of the key in the box. As the sort progresses, we shuffle the boxes around according to the values therein, but the boxes themselves never change names or contents, just locations. Thus we may say, for example, that data item A is greater than data item B (or $Val[A] > Val[B]$ in shortened form), which means that the contents of the data item (box) which we have symbolically named A is greater than the contents of the data item (box) symbolically named B.

When a processor starts at the top of the list and compares and swaps data items down the list until it reaches the end of the list (or until it needs to make no further comparisons because it knows the remainder of this list is sorted), we say it has completed a *pass*. Passes vary in length. In bubblesort, the first pass goes all the way down the list and requires $N-1$ comparison/swaps, the second requires $N-2$ comparison/swaps, and the last pass (pass number $N-1$) requires only 1 comparison/swap.

3.2.3 Analysis.

In this section we bound the entropy of the list at the end of each bubblesort pass. We analyze the first pass of bubblesort in detail, deriving an open form equation that tells us the entropy at any time in that pass. Early comparisons in the pass yield more information than later ones because, as the comparisons work their way down the list, each comparison is more and more likely to result in swapping the data items (because the data item we are moving down the list tends to be larger than the ones to which we are comparing it), and we gain correspondingly less and less information per comparison. The second and subsequent passes are likely to act similarly, with each comparison in a pass reducing entropy less than the preceding comparison in that pass.

We may use simple reasoning to upper bound the entropy at the end of each pass. At the end of pass i, i data items are known to be in their proper place (at the bottom of the list), so the remaining $N - i$ data items cannot be arranged in more than $(N - i)!$ ways. The maximum entropy of this is then $\log_2(N - i)!$. The upper curve of Figure 3.2 plots this upper bound for $N = 12$, and the lower curve plots the actual decrease in entropy for the first pass (obtained analytically below) and our conjecture for subsequent passes.

Let us examine the algorithm's very first comparison in detail. Initially, there are N! possible permutations of the data; let us denote a particular permutation as ABCDE..., where each letter represents one data item. Because all permu-

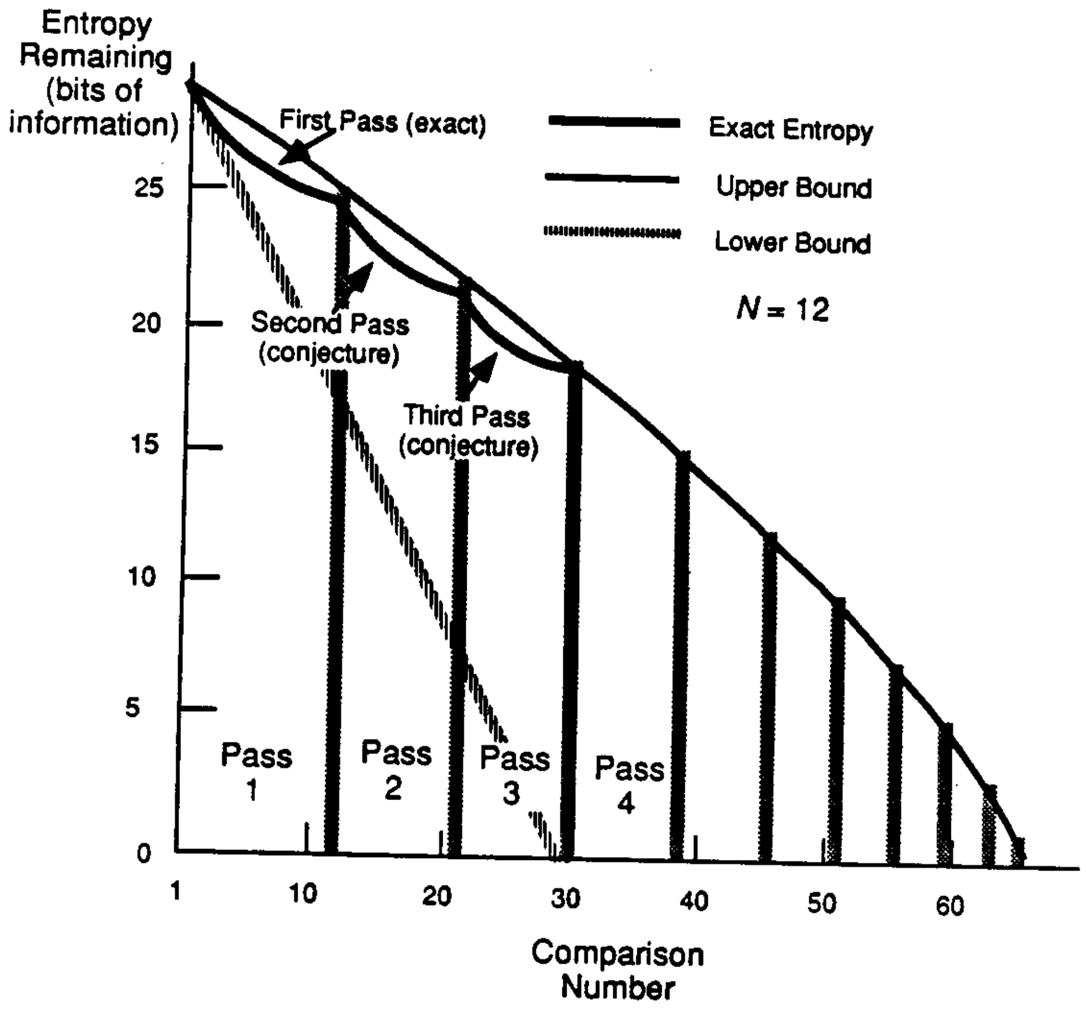


Figure 3.2: Decrease in entropy as bubblesort progresses.

tations are possible, our input data might also be arranged BACDE.... After the first comparison and swap, both these permutations cannot exist: one must map into the other. If $Val[A] > Val[B]$, then the permutation ABCDE... becomes BACDE.... Two separate permutations before the first comparison have now become one permutation that has two chances of occurring after the first comparison/swap. If we look at all $N!$ permutations of the input, we see that there are $N!/2$ pairs of permutations that differ only in the order of the first two items. After the first comparison/swap, each pair will have merged into one permutation with two chances of occurring, and we will have $N!/2$ such permutations. Let $E(i)$ be the entropy after comparison i on the first pass. The initial entropy is

$$E(0) = - \sum_{\substack{\forall N! \\ permutations}} \frac{1}{N!} \log_2 \left(\frac{1}{N!}\right) = \log_2 N!$$

because each of the permutations is assumed to be equally likely. After the first comparison/swap, we have

$$E(1) = - \sum_{\substack{\forall \frac{N!}{2} \\ permutations}} \frac{2}{N!} \log_2 \left(\frac{2}{N!}\right) = -\log_2 \left(\frac{2}{N!}\right) = \log_2(N!) - 1$$

because each of the $N!/2$ permutations has probability $2(1/N!)$ of occurring. Thus, this first comparison reduces the entropy by exactly one bit. This means that we receive the maximum amount of information from the first comparison.

Subsequent comparison/swaps are not quite so simple. First, let us look at the probability that we do not swap after the i^{th} comparison. This comparison

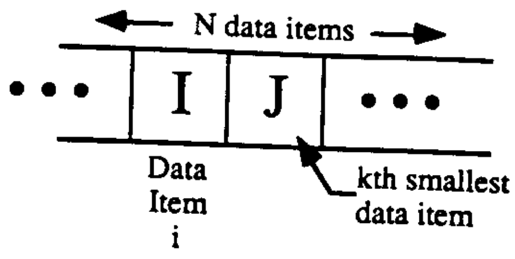


Figure 3.3: Data items i and $i+1$.

compares the i^{th} item in the list (let us call it I) and the $i+1^{st}$ item in the list (let us call it J), as illustrated in Figure 3.3. Then

$$\text{P[no swap I and J]} = \text{P[J is bigger than all data items 1 through } i\text{]}.$$

In the following paragraphs, it may also help to refer to Figure 3.7 as a numerical example.

Suppose J is the k^{th} smallest data item. For $k \leq i$, P[no swap] = 0. To see this, suppose $i = 4$ and $k = 1$, which means that J is the smallest data item and in the 5^{th} position (Figure 3.4). Obviously, all the data items in front of J are larger than J, which means that a swap will always occur after comparing I and J. The same holds true for any $k \leq i$: there aren't enough small items that could be in **front** of J to avoid a swap.

Now **examine** k = i+1, which means that the $i+1^{st}$ data item is also the $i+1^{st}$ smallest data item. There are only i data items smaller than J, so if we are to avoid a swap, all of these must have initially preceded J in the data list. What is the probability of this? The probability of data item 1 being one of the

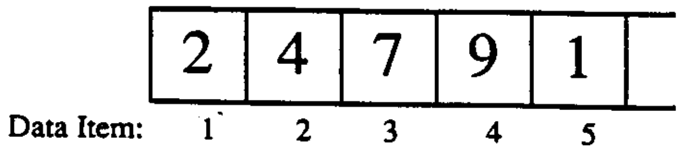


Figure 3.4: Smallest data item in 5th position.

ith smallest is $i/(N-1)$, or equivalently, $(k-1)/(N-1)$. The probability of the next data item being another of the ith smallest is $k-2/(N-2)$. One can continue this, and see that the probability of data item i being the last one of the ith smallest is $1/(N-i)$. The final probability of no swap, given that $k = i+1$, is

$$\left(\frac{k-1}{N-1}\right)\left(\frac{k-2}{N-2}\right)\left(\frac{k-3}{N-3}\right)\cdots\left(\frac{1}{N-i}\right) = \frac{(k-1)!(N-i-1)!}{(N-1)!}.$$

A similar process will give P[no swap] for $k > i+1$, and this turns out to be

$$\left(\frac{k-1}{N-1}\right)\left(\frac{k-2}{N-2}\right)\left(\frac{k-3}{N-3}\right)\cdots\left(\frac{k-i}{N-i}\right) = \frac{(k-1)!(N-i-1)!}{(k-i-1)!(N-1)!}.$$

Because $k! = 0$ for $k < 0$, we get

$$\text{P[no swap} \mid i^{\text{th}} \text{ comparison, K is } k^{\text{th}} \text{ smallest]} = \frac{(k-1)!(N-i-1)!}{(k-i-1)!(N-1)!}$$

The probability that K is the kth smallest is simply $1/N$, so

$$\begin{aligned}\text{P[no swap} \mid i^{\text{th}} \text{ comparison]} &= \sum_{k=1}^{N} \frac{1}{N}\frac{(N-i-1)!}{(N-1)!}\frac{(k-1)!}{(k-i-1)!} \\ &= \frac{(N-i-1)!}{N!}\sum_{k=1}^{N}\binom{k-1}{i} i!,\end{aligned} \tag{3.2}$$

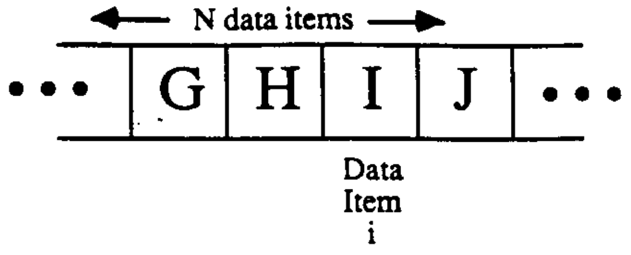


Figure 3.5: A list of data.

where the sum has non-zero terms only for $k \geq i+1$. If we apply the identity

$$\sum_{k=0}^{N} \binom{k}{m} = \binom{N+1}{m+1}$$

to the summation, then we find that

$$\text{P[no swap } \mid \text{i}^{\text{th}} \text{ comparison]} = 1/(\text{i}+1). \tag{3.3}$$

This implies that of all the permutations remaining after the $(i-1)^{st}$ comparison, $1/(i+1)$ of them remain unchanged and will still be possibilities after the next comparison. What happens to the other $i/(i+1)$?

It turns out that $1/i$ of these map on to permutations that already existed as possible permutations just before the i^{th} c/s, and the remaining $(i-1)/(i+1)$ each become permutations that did not exist just before the i^{th} c/s. To see this, let us suppose, as in Figure 3.5, that we have four consecutive data items (G, H, I, and J), suppose that I is in position i in the data list (remember, we number starting from 0), and suppose that we have done the first $i-1$ comparison/swaps. The next action will be to compare I and J. We know that a fraction $i/(i+1)$ of

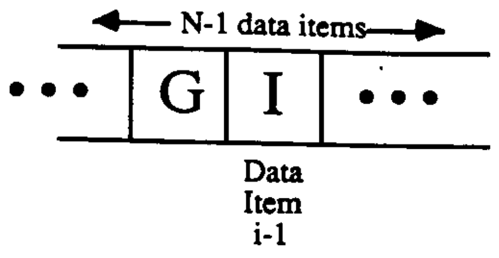


Figure 3.6: The data list with H removed.

the time we will swap H and I, giving us "G I H" in the data list. If this happens, what is the probability that G < I, or in other words, what is the probability that we map on to one of the permutations that was a possibility before the i^{th} comparison/swap? If we take H out of the original data list (after all, we already know it is bigger than G and I) and run the first pass of bubblesort on the original data so modified (as shown in Figure 3.6), then the probability that we don't swap G and I is exactly the probability that G < I. This we know to be $1/i$, so we may conclude that after the i^{th} comparison,

- $\frac{1}{(i+1)}$ of the permutations remain unchanged.
- $\frac{i}{(i+1)}\frac{1}{i} = \frac{1}{i+1}$ of the permutations map onto pre-existing permutations.
- $\frac{i}{(i+1)}\frac{(i-1)}{i} = \frac{i-1}{i+1}$ of the permutations change to "new" permutations.

To complete this, we must show that the permutations that are supposed to map onto the existing permutations actually do so. Refer once again to Figure 3.5. Because we started off with all possible permutations of data, just before the i^{th} comparison, our set of remaining permutations will contain permutations

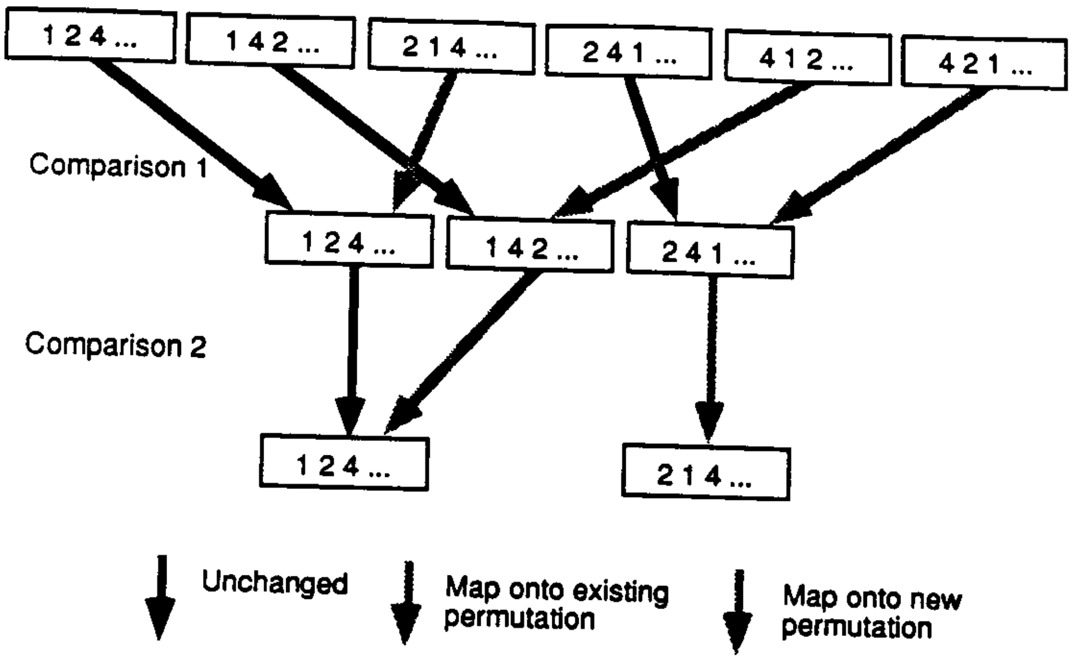


Figure 3.7: Example of permutations.

with all possible combinations of G and H such that we have both G < H, and we have all possible permutations of data to the right of H. After the i^{th} comparison, the unchanged permutations will contain all possible values of G, H, and I such that G < H < I. Of the changed permutations, $1/i$ of them will result in the order "G I H", with the condition that G < I < H, again with all possible values of G, H and I. Because both the changed and the unchanged permutations contain all possible values of three data items in increasing order at these locations, these changed permutations must be the same as the unchanged permutations. The remaining $(i-1)/i$ of the changed permutations have G > I < H, or, in other words, items $i-2$ and $i-1$ are out of order. No such permutations existed as a possibility before the i^{th} c/s, so these are the "new" permutations.

Figure 3.7 shows the first two comparison/swaps using three possible data values in their 6 possible permutations. When we do the first comparison, 1/2 of the possible permutations remain unchanged, and 1/2 map into the unchanged permutations. Note that before the first c/s, all permutations are possible, so we can not have any "new" permutations. For the second c/s, we see one unchanged permutation, one changed permutation mapping onto the unchanged one, and one "new" permutation that was not a possibility after the first c/s.

All this is well and good, but it does not yet explain how to calculate the entropy at any time; we must know the probability of the occurrence of the permutations. The key here is to note that when one changed permutation maps onto one existing permutation, the resultant permutation is twice as probable as before the mapping. This doubling effect implies that the probabilities will have the form of a power of 2 over $N!$. Define $n_{i,k}$ as the number of permutations occurring with probability $2^i/N!$ after the k^{th} comparison/swap. Initially we have $n_{0,0} = N!$, and $n_{i,0} = 0$ for $i > 0$. After the first comparison/swap, we have $n_{1,1} = N!/2$ ($N!/2$ permutations, each having probability of occurrence $2/N!$), and $n_{i,1} = 0$ for $i \neq 1$. The results given previously imply that after the second comparison/swap, 1/3 of the permutations remain unchanged, 1/3 map onto the unchanged ones, doubling their probability, and 1/3 change to something new. As seen in Figure 3.8, this is indeed what happens: we end up with $N!/6$ permutations having probability $4/N!$, and $N!/6$ permutations having probability $2/N!$. In terms of the n's, we have

$$n_{1,2} = \frac{N!}{6}$$

$$n_{2,2} = \frac{N!}{6}.$$

Look now at the next comparison/swap. Of the high probability permutations, 1/4 remain unchanged, and 1/4 map onto the unchanged, giving us $N!/24$ permutations with probability $8/N!$. Half of the high probability permutations map onto something new but still have probability $4/N!$. Of the low probability permutations, 1/4 remain unchanged and 1/4 map onto those, giving us $N!/24$ new permutations having probability $4/N!$. Half of the low probability permutations map onto new permutations but retain probability $2/N!$. Again, in terms of the n's we have

$$\begin{aligned} n_{1,3} &= \frac{N!}{12} \\ n_{2,3} &= \frac{3(N!)}{24} \\ n_{3,3} &= \frac{N!}{24} \end{aligned} \tag{3.4}$$

We can see a recursion forming here. The permutations with probability $2^k/N!$ after the i^{th} comparison are composed of 1) some of the permutations that had probability $2^k/N!$ after the previous comparison, and 2) those permutations which were half as probable after the $i-1^{\text{st}}$ comparison but which doubled in

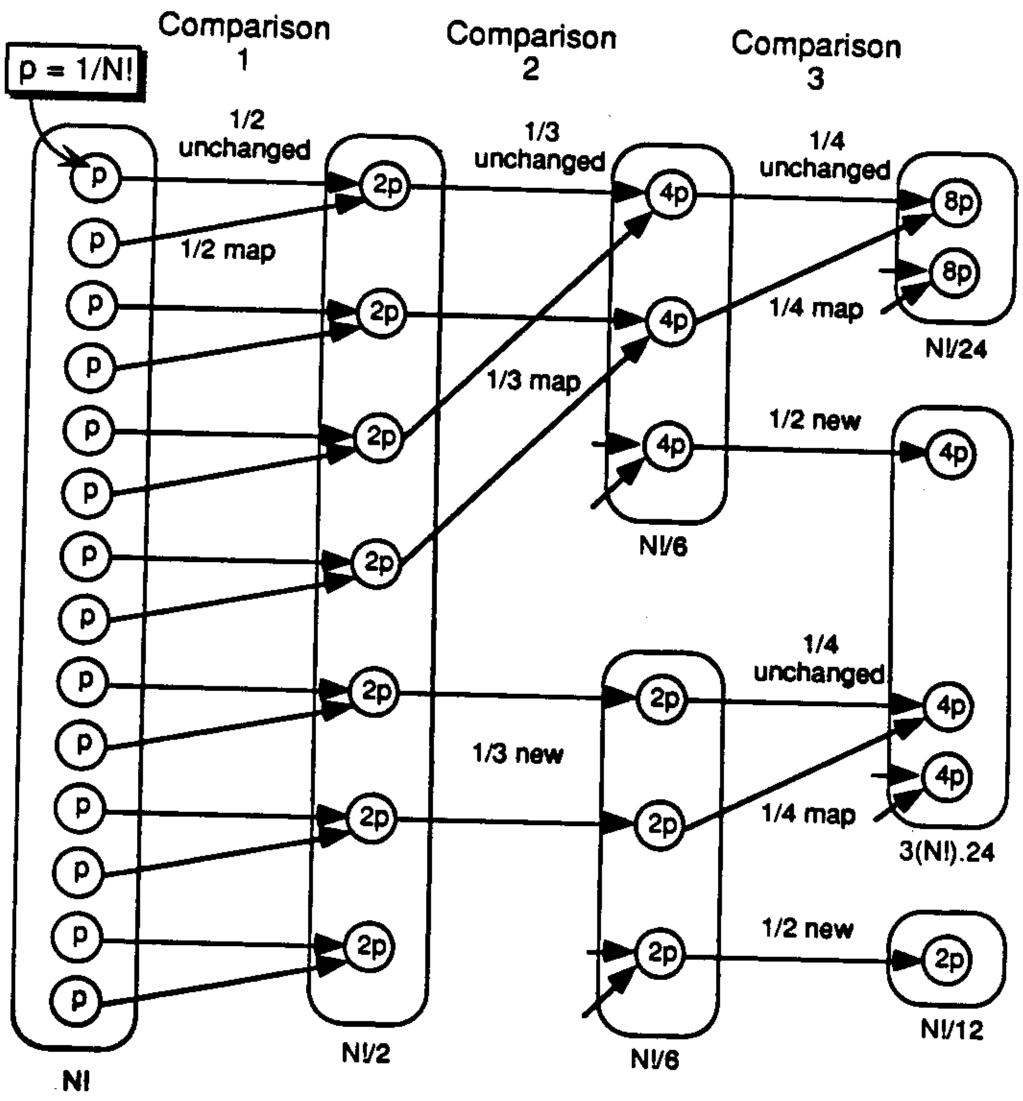


Figure 3.8: Merging permutations in first pass of bubblesort.

probability because of the mapping process. More precisely,

$$n_{i,k} = \frac{i-1}{i+1} n_{i-1,k} + \frac{1}{i+1} n_{i-1,k-1}.$$

The final expression for the entropy at any comparison k during the first pass of single-processor bubblesort is thus

$$\begin{aligned} E(k) &= -\sum_{i=0}^{k} n_{i,k} \frac{2^i}{N!} \log_2 \frac{2^i}{N!} \\ &= \sum_{i=0}^{k} n_{i,k} \frac{2^i}{N!} (\log_2(N!) - i) \end{aligned} \tag{3.5}$$

Plotting this (Figure 3.2) we see that we gain the most information at the start of this (first) pass, but this gain is less and less as the pass continues.

3.2.4 Conclusion.

The decrease in entropy during bubblesort displays a very interesting pattern during bubblesort's first pass, and our analysis gives us the entropy at each comparison in that pass. Subsequent passes become more difficult to analyze, but we can bound the entropy at the end of each pass. Bubblesort's general inefficiency comes from the way it makes its comparisons: toward the end of each pass, you get very little information for each comparison.

3.3 Mergesort

3.3.1 Description.

Mergesort is based on the method of divide-and-conquer: split the data items into two equal length sublists, sort them separately (using the same procedure recursively), then merge the results into one list. Let $N = 2^k$. We then split the data lists k times to reduce it to N lists of length 1. A list of length 1 is already sorted, so we merge these lists into $N/2$ sorted sublists of length 2, then merge length 2 sublists into sorted sublists of length 4, and continuing until we merge two sublists of length $N/2$ into the final sorted data list of length N.

3.3.2 Analysis.

We will not examine every comparison in detail, but instead we derive the entropy at particular times during the sort, specifically at those points when we have just created $N/2^i$ sublists of length 2^i by merging 2^{i+1} lists half that length. Let us say that we are at "level i" at such a time. Thus we start at level 0, when we have split the data down to lists of length 1 but not sorted anything, and we end at level k.

When we are at level i, there are $\left(\begin{array}{c} N \\ \underbrace{2^i, 2^i, \ldots, 2^i}_{N/2^i times} \end{array} \right) = \frac{N!}{(2^i!)^{N/2^i}}$ ways to arrange the data into 2^i sorted list of equal length. All these arrangements are equally probable, so the entropy at this level is $\log_2(N!) - (N/2^i)\log_2((2^i)!)$. To merge the lists and move up a level requires a total of N comparisons, so the entropy after comparison iN is

$$E(iN) = \log_2(N!) - (N/2^i)\log_2((2^i)!). \tag{3.6}$$

We plot this in Figure 3.9 for $N = 1024$.

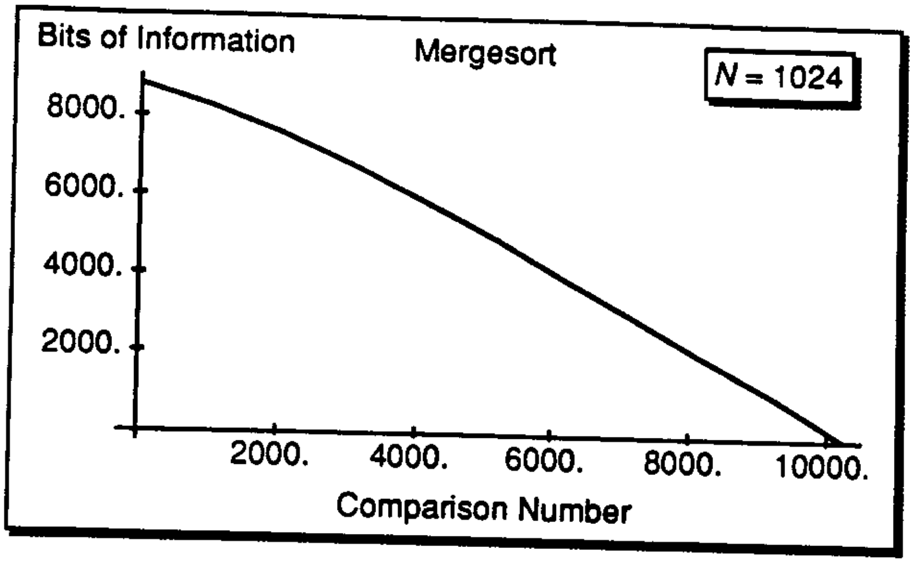


Figure 3.9: Decreasing entropy during mergesort.

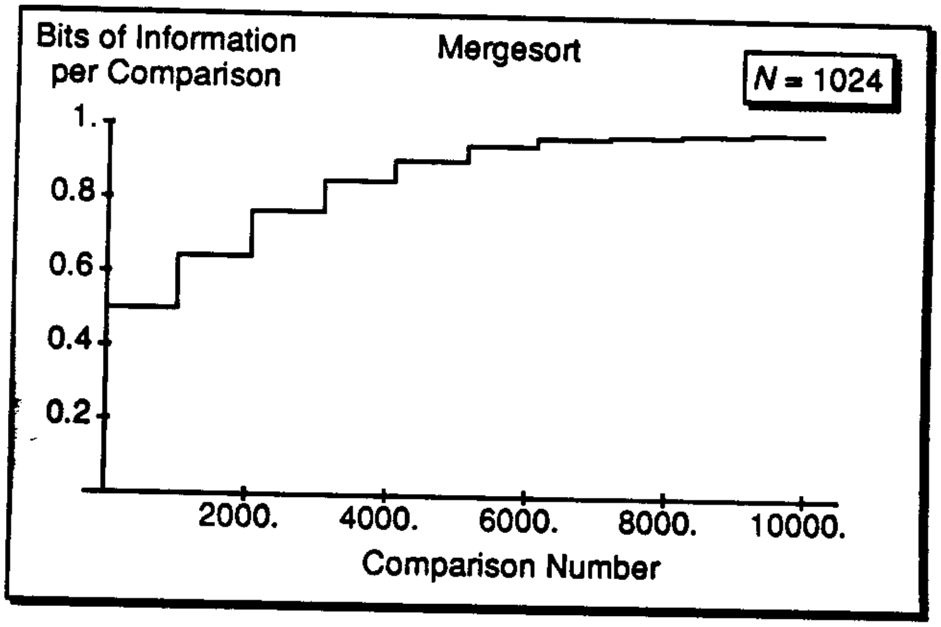


Figure 3.10: Bits of information per comparison during mergesort.

To find the average information gained per comparison at each level, we find the amount of entropy decreases in a particular level, and divide by N. At level i , this is

$$\overline{E_i} = \frac{1}{2^i} \log_2 \left(\frac{2^i!}{(2^{i-1}!)^2} \right),$$

which we plot in Figure 3.10. At the beginning of the sort we gain only 1/2 bit per comparison, but by the final level we gain almost 1 bit per comparison.

3.3.3 Conclusion.

In mergesort we gain information in the merging operation. The efficiency of this operation arises because each merge operation takes relatively little time, but greatly reduces the number of possible permutations. It is important, though, that the merges be relatively balanced (e.g. merging equal-sized lists). The work involved in merging is proportional to the total number of data items to be merged, but the amount that entropy is reduced is related to the relative sizes of the two lists, with the greatest reduction occurring for equal-sized lists, and less reduction if the lists are not balanced. Because mergesort never gains less than some constant number of bits per comparison (in this case, never less than 1/2 bit), it finishes in $O(N \log_2 N)$.

3.4 Quicksort.

3.4.1 Description.

Quicksort is a divide-and-conquer algorithm, like mergesort, but it operates slightly differently. The basic step is to split the data into two sublists by comparing the data to some special splitting value, say A. Data items smaller than A are put in one sublist, in no particular order, and data items with greater than or equal to A are put in the other sublist. Sort these two sublists (by recursively repeating this basic step), then concatenate the two sorted sublists to yield a completely sorted list of data.

3.4.2 Analysis.

The characteristics of splitting values are very important because quicksort reduces the number of permutations fastest by splitting the data lists in half. To see this, suppose that we have a list of N items, and we want to split it. If we split it exactly in half, then we could have a total of $\frac{N}{2}!\frac{N}{2}!$ permutations of the items in the list. If we now take a data item from one list and put it in the other, we find the number of permutations by dividing the expression in the last sentence by $N/2$ (to remove an item from one list) and multiply by $1+(N/2)$ (to add an item to the other list). Because $1+(N/2)$ is greater than $N/2$, the net result is to increase the number of permutations. If we continue to shrink one list and expand the other in the same fashion, we find that the number of permutations continues to grow, reaching a maximum of $(N\text{-}1)!$ when we split the data into one list of size 1 and one list of size N-1.

Suppose that we always pick optimum splitting values. Let $N = 2^k$ and let us say that at level i of the sort we take 2^i lists of length 2^{k-i} and split them into twice as many lists each half as long. Thus at level i the number of possible permutations of the data is $(N/2^i)!^{2^i}$ and the equivocation is the logarithm of this number. Each split requires a total of N comparisons, so the uncertainty at comparison iN (at the end of level i) is

$$E(iN) = 2^i \log_2 \left(\frac{N}{2^i}!\right) \tag{3.7}$$

We plot this in Figure 3.11 as the best case curve, and the bits per comparison in Figure 3.12. Note that quicksort is, in many ways, the reverse of mergesort. In mergesort, the information gained per comparison rises from 1/2 bit to near 1 bit as the sort progresses; in quicksort, the first comparisons net nearly 1 bit of information per comparison, and it falls to 1/2 bit at the end of the sort.

Now let us suppose that we pick the worst possible splitting values, and whenever we divide a list of length l, it splits into one list of length 1 and one list of length $l-1$. Each split then acts like one pass of bubblesort, which implies that quicksort's behavior in this situation is $O(N^2)$. After the i^{th} split, $\log_2((n-i)!)$ bits of information remain to be discovered because $N - i$ data items remain unsorted. Expanding this, we find that

$$\log_2((N-i)!) \quad = \quad \log_2(N) + \log_2(N-1) + \cdots + \log_2(N-i+1)$$

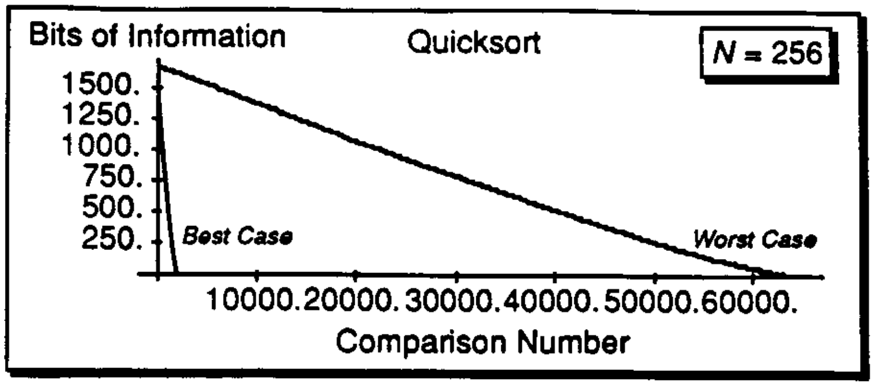


Figure 3.11: Decreasing uncertainty during quicksort, best and worst cases.

$$= \sum_{k=0}^{i-1} \log_2(N-k),$$

and thus the i^{th} split yields $\log_2(N-i+1)$ bits of information. Figure 3.11 also shows the decrease in entropy for this worst case of quicksort.

3.4.3 Conclusion.

Quicksort gains its efficiency in the splitting operation, and its performance can be anywhere from $O(N \log_2 N)$ to $O(N^2)$, depending upon how the splitting values are chosen. In the best case, the splitting value is the median, and some quicksort algorithms sample the data to estimate the median (e.g. [LL89]). However, the median is not the only possible splitting value for $O(N \log_2 N)$ quicksort. Horowitz and Sahni [HS78] prove that quicksort with random splitting values is $O(N \log_2 N)$. In the worst case of quicksort, each split separates only one data item, and then the algorithm's performance is similar to bubblesort. Thus, the technique for finding splitting elements is very important to this sort.

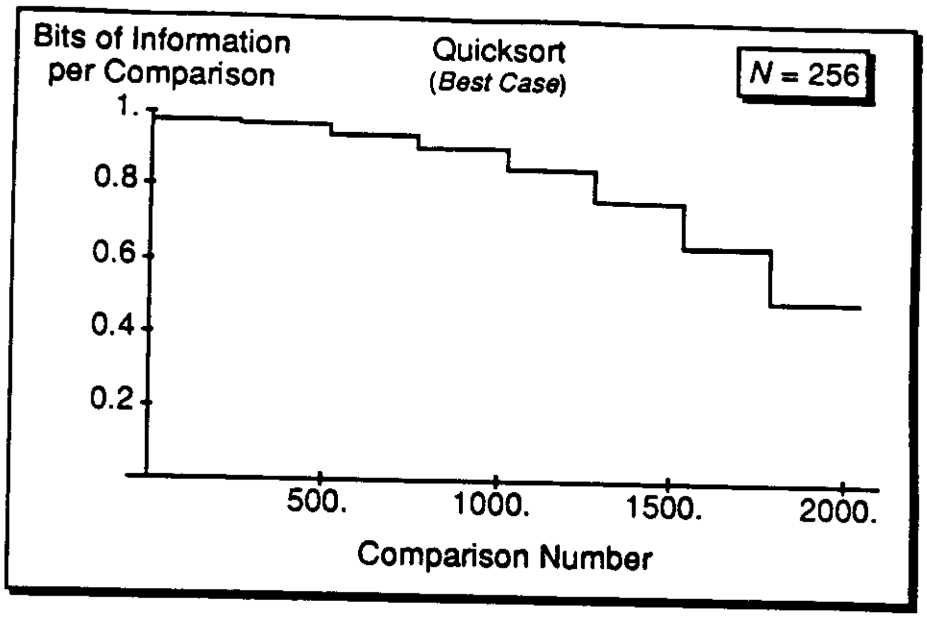


Figure 3.12: Bits of information gained per comparison during quicksort.

3.5 Radix Sort.

3.5.1 Description.

Radix sort operates very differently from the previous algorithms. The sort never makes direct comparisons of data items; instead, it puts data items in "buckets" based on the character or digit at a particular position in the data item. Let us suppose that we have unique numeric data items, l digits long (with digit 1 being the leftmost and l the rightmost), and each digit takes on one of c values, from 1 to c . We use c buckets in the sort. The basic step of radix sort is to examine one digit, say i, of each data item. If the character at that digit is x, then append the data item to the contents of bucket x. When all the data items have been examined, concatenate the buckets in order from 1 to c to form a new list of N data items, and then clear the buckets. To fully sort these data items, repeat this step, starting with the least significant digit, and working on the next most significant digit at every step. As a short example, suppose we have a list of 3 data items:

213

312

113

We first put the data items into buckets on the basis of the least significant digit:

bucket 1	bucket 2	bucket 3
	312	213
		113

When concatenating the bucket results in the data item list:

312

213

113

We now put the data items into buckets on the basis of the middle digit. In this case, all the second digits are 1, so all the data items end up in bucket 1:

bucket 1	bucket 2	bucket 3
312		
213		
113		

This results in the same data item list as after the previous step. To complete the sorting, we now put the data in buckets on the basis of the most significant digit:

bucket 1	bucket 2	bucket 3
113	213	312

Concatenating the buckets results in the final sorted list of data items:

113

213

312

We note that after step i of the algorithm, the data item list is sorted if you look at only the i least significant digits of the data items. Thus the algorithm requires l steps to complete, and each step requires the manipulation of every data item, so if we count putting a data item in a bucket as an operation, then algorithm takes $O(lN)$ operations. Previous algorithms were timed in terms of comparisons, and, although it depends upon the implementation, it is reasonable to assume that putting a data item in a bucket takes a constant factor times longer (or shorter) than a comparison. Thus we can compare the big O results for each sort, and we conclude that radix sort can be faster than a comparison-based sort. In particular, for N large and l small, radix sort should perform quite well, whereas for N small and l large, say 100 data items, each 100 digits long, its performance will be terrible. Radix sort does have the additional restriction that we must know ahead of time the number of buckets to allow, and all data items must have equal length (or be padded out to equal length). Thus, we might wonder if these restrictions limit the entropy of the sorted list to less than $\log_2(N!)$ bits.

3.5.2 Analysis.

As with the previous sorts, radix sort starts with N data items in one of $N\,!$ permutations (each equally likely), and ends up with N data items in the one, sorted permutation. Thus the decrease in the number of permutations corresponds to a decrease in entropy.

Uncertainty arises in radix sort because at any given time (excepting the end of the algorithm) we have examined only a portion of the digits of each data item. Thus, if k data items' i least significant digits are identical, then there are $k\,!$ possible arrangements of the data items that we cannot distinguish when we have examined only the i least significant digits.

Let a set of data items of whose i least significant digits are identical be known as a "level i group," and let g_i be the number of level i groups and $n_{i,j}$ be the number of data items in the j^{th} level i group. After step i of the radix

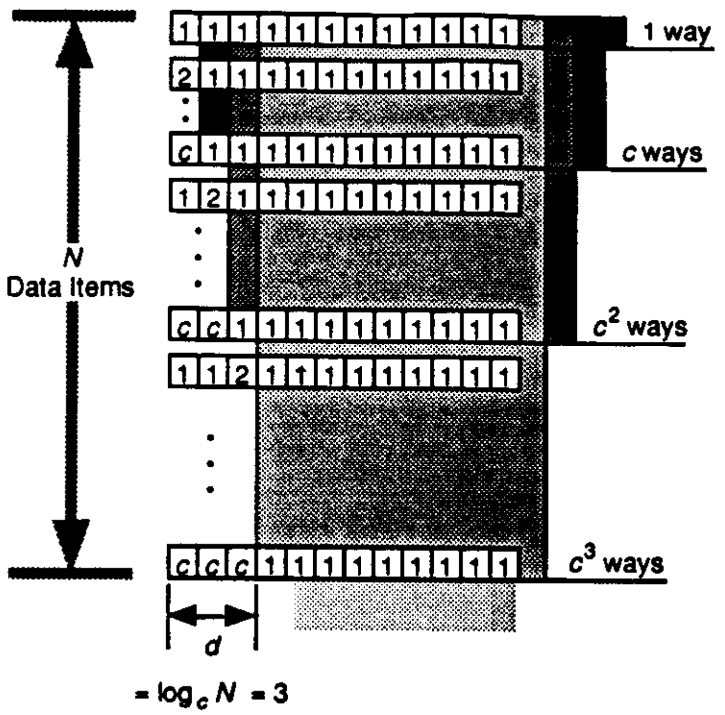


Figure 3.13: The data items that reduce entropy the slowest.

sort algorithm, the number of possible permutations still existing is $\prod_{j=1}^{g_i} n_{i,j}!$. This is highly dependent upon the particular data being sorted. Let us find the data items that reduce entropy the slowest.

If we have N data items, then we need at least $\lceil d = \log_c(N) \rceil$ digits in our data items. For simplicity of analysis, assume that $N = c^d$. If l is greater than d , then we maximize the number of permutations by distinguishing the data items only by their most significant d digits, and making the least significant $l-d$ digits all identical and equal to, say, 1. Thus for the first $l - d$ steps of radix sort, we always have $g_i = 1$ and $n_{i,j} = N$, which mean there are N ! permutations.

Of the most significant d digits, we would like to use as many 1's as possible while still distinguishing the data items. Figure 3.13 provides an example for $d =$ 3. One data item can have 1's in all its d most significant digits. Another c -1 data items can have 2 through c as the most significant digits, and 1's every place else. The next $c^2 - c$ data items can have any digits for two most significant, except those already used, and 1's every place else. Continuing this way, we construct a set of data items such that we have c^{d-i} data items with 1's in digits d +i through l , for $i = 1, \ldots, d$. Using these data items, we can calculate the entropy, $E(i)$ at the end of every step i :

$$E(i) = \begin{cases} \log_2(N!) & \text{for } i = 1, \ldots, l-d \\ c^{\,i+d-l} \log_2(c^{l-i}!) & \text{for } i = l-d+1, \ldots, l \end{cases} , \qquad (3.8)$$

where, it should be remembered, that we have assumed $N = c^d$. If this last condition does not hold, we can still build a data item set for slowest entropy reduction by using the same technique described above. This provides a lower bound on the decrease in entropy. For most sets of data items, the entropy will decrease much faster. In fact, it will often be the case that entropy will decrease to 0 before the sort finishes. If, for example, the lower d digits of all data items are unique, then after d steps, there is only one possible permutation of the data. However, this permutation is not necessarily sorted, and the algorithm must continue to completion to ensure the data items' sorting. This is fundamentally different behavior from the previous three algorithms we discussed.

3.5.3 Conclusion.

Radix sorts operates quite differently from the previous sorting algorithms. Because the sort is not based on comparisons, the notion of entropy that we used before does not seem to be an appropriate measure of the sort's progress. In many instances the entropy will reach 0 before the data items are sorted. Unlike the other algorithms whose lists of data items gradually become sorted, the ordering of data items at intermediate stages in radix sort bears little relationship to the data's final ordering. This may be related to a phenomenon from the "Rubik's Cube" puzzle: it was observed that the fastest method of ordering the cubes (in terms of number of movements) was not to put them in place one at a time, but was instead a series of non-obvious, seemingly random, moves from which order appeared only during the final few movements. Given Rubik's Cube's relationship to mathematical notions such as groups, there many be interesting underpinnings to radix sort.

3.6 Conclusion.

We have examined four well-known sorting algorithms using information theory to see where their efficiency, or inefficiency lies. Bubblesort makes too many comparisons that yield little information. This is particularly true toward the end of the sort, when the list of data items tends to be highly ordered. Mergesort gains its efficiency because the merging process takes relatively little time but reduces the entropy greatly (if you are merging equal-sized lists). Quicksort's efficiency arises in the splitting operation, and the splitting value is critically important. If the splitting value is the median, then entropy is reduced the fastest. Finally, radix sort operates quite differently from the other sorts, and entropy is not a reasonable measure of this sort's progress.

CHAPTER 4

Bubblesorts on Multiple Processors

4.1 Introduction.

In this chapter, we examine four parallel versions of bubblesort. Two of these versions simply use multiple processors to make the passes found in the standard, single-processor bubblesort. Another version splits the data among the processors (but overlapping data between adjacent processors) and the processors continuously bubblesort their own data. The final version lets processors make random comparisons.

Single processor bubblesort and the notation we use was described in Section 3.2. For the distributed sorts in this chapter, we assume that core memory on a processor is large enough to hold the data it deals with, that the computer runs only one program at a time, and that communication is reliable, with messages arriving in the order they were sent and with no delay.

The time to finish an algorithm using N processors is denoted by $T(N)$, and the speedup of an algorithm using N processors is $S(N) = T(1)/T(N)$.

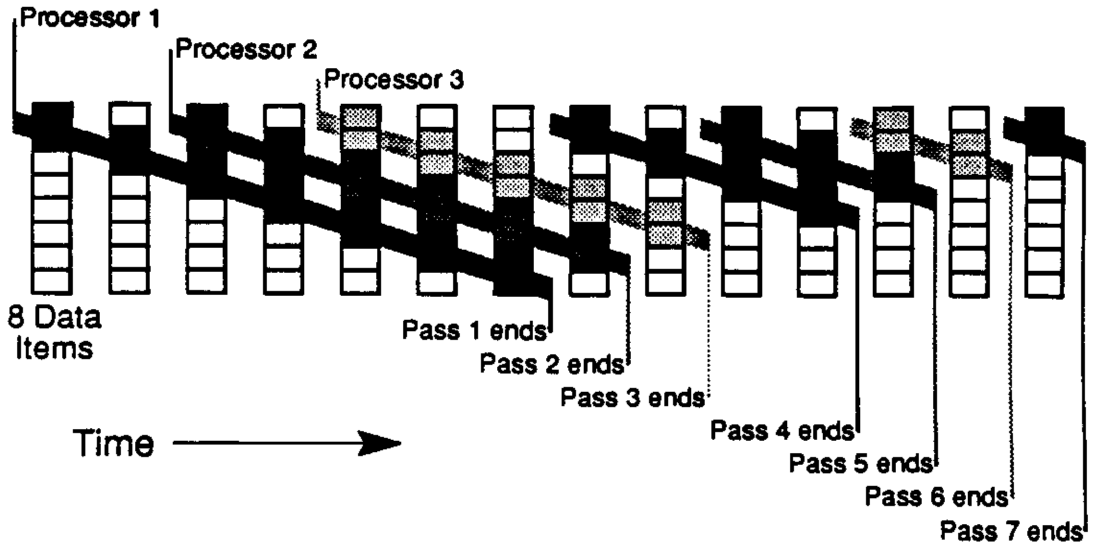


Figure 4.1: Depth-first bubblesort with 3 processors and 8 data items.

4.2 Depth-First Bubblesort.

4.2.1 Description.

Our first parallel bubblesort algorithm is very much like the standard, single-processor bubblesort, except each processor works on a different pass through the data. For example, in Figure 4.1, processor 1 makes the first pass through the data, and processors 2 and 3 follow behind, making the second and third passes. When a processor finishes its pass, it goes to the top of the list and starts another. This process continues until the list is sorted. The sort is called "depth-first" because each processor takes each pass as deep into the list as it can before starting a new pass; this is in contrast to the "breadth-first" scheduling of the algorithm in the next section.

We do not allow two processors to work on one data element simultaneously, which means that consecutive passes operate on consecutive, non-overlapping pairs of data items. To enforce this, some form of locking or access control is necessary, but this is easily done if each pass monitors the progress of the preceding pass and does not overtake it.

4.2.2 Speedup.

To compute the speedup of the algorithm as the number of processors is varied, we examine the processor that takes the first pass. For purposes of illustration, assume this is processor 1. Define a superpass as beginning every time the processor 1 starts a new pass, and finishing when processor 1 makes the last comparison/swap of that pass. Let a complete superpass be a superpass in which all processors participate (i.e. all processors start a pass). An incomplete superpass is a superpass in which not all processors start a new pass, and will occur at the end of the algorithm, when there is not enough work for all the processors.

In terms of superpasses, the algorithm consists of a number of complete superpasses, followed by a possibly incomplete superpass, plus some extra time for any remaining processors to finish. All processors are kept busy for a number of superpasses, but there may not be enough data items remaining for the last superpass to use all the processors. Figure 4.2 shows an example. This picture contains three complete superpasses and one incomplete superpass. All

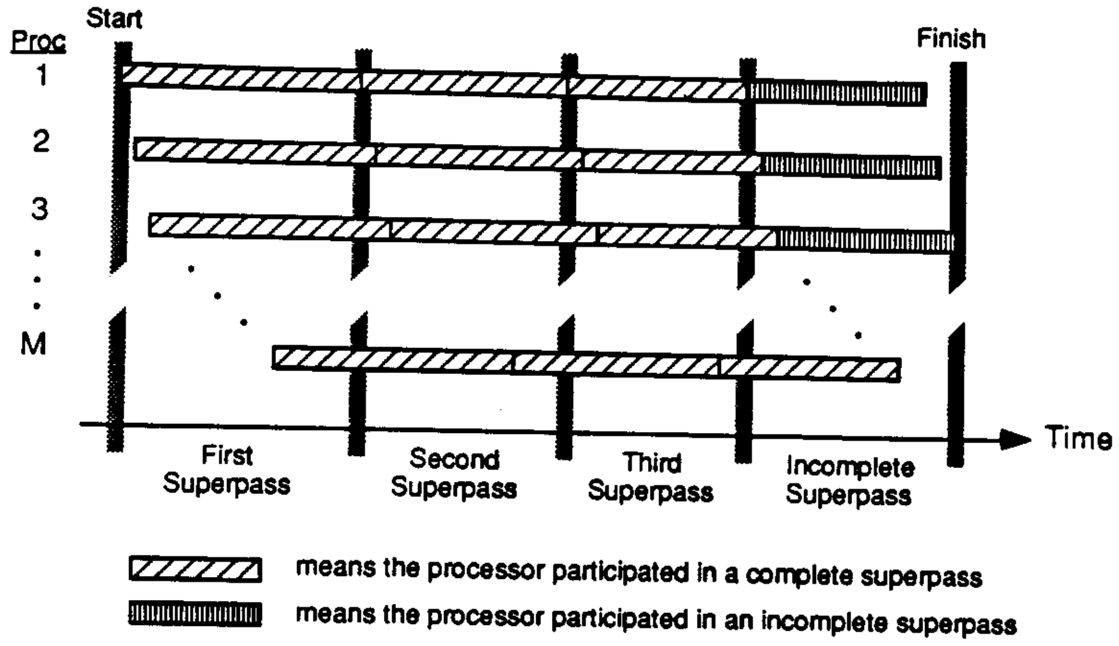

Start
Proc
1
2
3
M
Finish
Time
First Superpass
Second Superpass
Third Superpass
Incomplete Superpass
means the processor participated in a complete superpass
means the processor participated in an incomplete superpass

Figure 4.2: Superpasses.

M processors participate in the complete superpasses, but only the first three processors participate in the last superpass.

To find the algorithm's running time, we compute how long it takes processor 1 to finish each superpass, and add to it the time for the other processors to finish the last superpass. We ignore the time required for non-first processors to finish the initial superpasses because this time is overlapped with the beginning of processor 1's the next superpass. Also, we should note that the first processor will never have to wait for other processors before beginning a pass, except for possibly the last superpass, because until that time, there will always be enough unsorted data items (more than $2M$) to have all the processors working simultaneously when the first processor ends its pass. However, the first processor

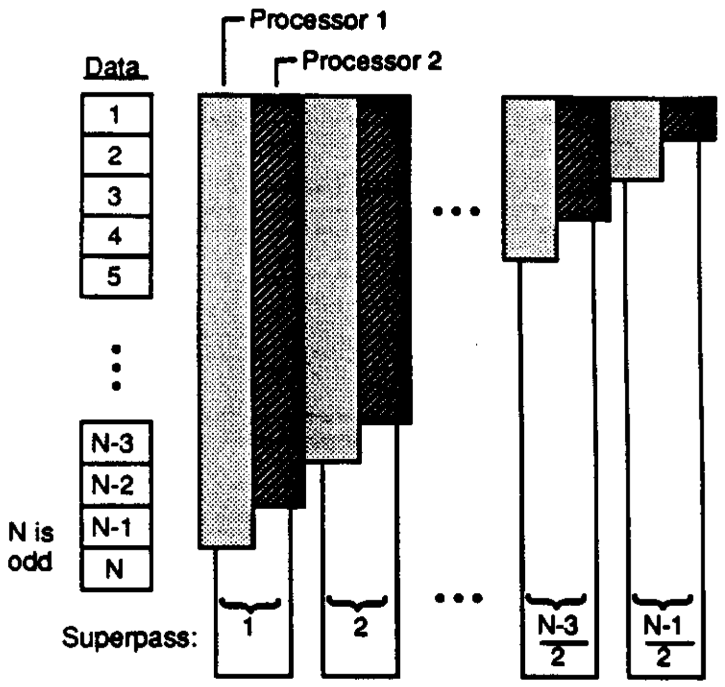


Figure 4.3: Superpasses for 2 processors, N odd.

might have to wait before beginning the last superpass, so this time must also be accounted for. Therefore, we may finally say that

Time to finish algorithm =

time for first processor to finish all superpasses

+ time first processor waits before beginning last superpass

+ time for other processors to finish last superpass.

Let us suppose that we have two processors and N data items; this means that the first processor makes every other pass, and for odd N, all superpasses are complete. If N is odd (Figure 4.3), then the first superpass will take $N-1$ time units, the second $N-3$ time units, and continuing until the final superpass,

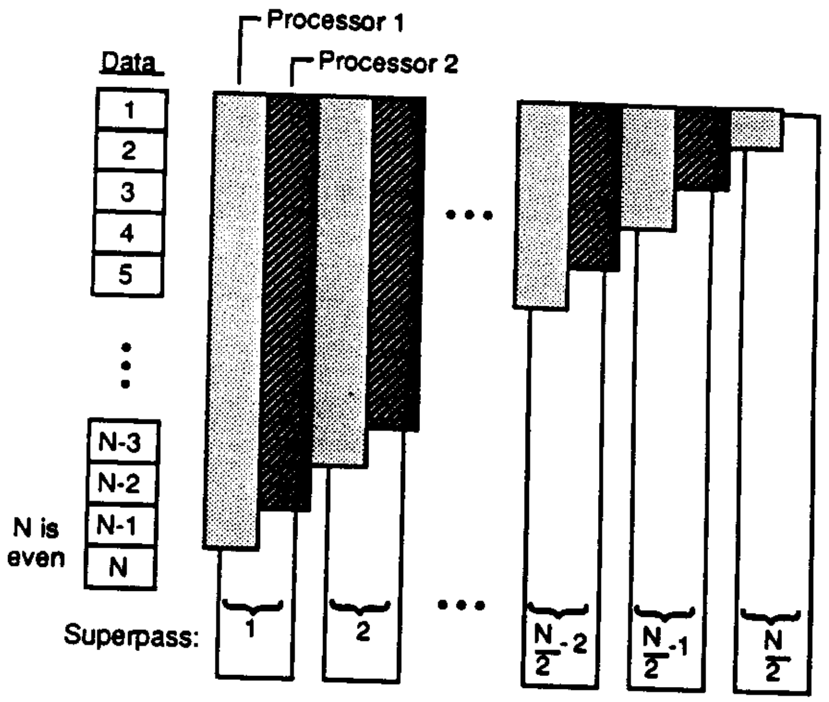


Figure 4.4: Superpasses for 2 processors, N even.

which takes two time units. Processor 1 does not have to wait to finish the last superpass, and processor 2 takes finishes 1 time unit after one, so we have

$$T(N) = \left[\sum_{i=1}^{(N-1)/2} 2i\right] + 1 = \frac{(N-1)(N+1)}{4} + 1, \tag{4.1}$$

which is proved below.

For N even (Figure 4.4), we have that the first superpass takes $N-1$ time units, the second takes $N-3$ times units, and continuing until the final superpass takes 1 time unit. However, processor 1 must wait 1 time unit before beginning the last superpass, and the algorithm finishes as soon as processor 1 completes the last pass. This gives us

$$\begin{aligned} T(N) &= \left[\sum_{i=1}^{(N-1)/2} 2i-1\right]+1 \\ &= \frac{N}{2}\left(\frac{N}{2}-1\right)-\frac{N}{2}+1 \\ &= \frac{(N)(N)}{4}+1. \end{aligned} \tag{4.2}$$

If we examine the case of $M = 3$ processors, we find the following:

If $N = 3k$, for some $k > 0$, then

$$T(N) = \frac{1}{6}(N)(N+1)+2.$$

If $N = 3k+1$, for some $k > 0$, then

$$T(N) = \frac{1}{6}(N-1)(N+2)+2.$$

If $N = 3k+2$, for some $k \geq 0$, then

$$T(N) = \frac{1}{6}(N+1)(N)+2.$$

In the general case of M processors, and if $N = Mk + c$, for some $k \geq 0$ and $c = 0, 1, \ldots, M-1$ such that $N > 1$,

$$T(N) = \frac{1}{2M}(N)(N+M-2)+(M-1) \quad \text{for } c=0 \tag{4.3}$$

$$T(N) = \frac{1}{2M}(N+c-2)(N+M-c)+(M-1) \quad \text{for } c>0 \tag{4.4}$$

It turns out that Equation 4.3 is 1 plus Equation 4.4, with c set equal to 0. When $k = 1$, neither result applies, because we cannot use all the processors during the sort. We can correct this by using $M = \lfloor N/2 \rfloor$, which is the maximum number of useful processors.

If we look at the general behavior, we see that it is $O(N^2/M + M)$, which means that we can get linear speedup for small numbers of processors, but performance gain falls off as we throw more processors at the problem. When we reach $M = N/2$, then we cannot make use of any additional processors — there will never be enough work for all of them. The plot in Figure 4.6 shows the behavior of the speedup as the number of processors increases.

To prove Equations 4.3 and 4.4, we break time into three pieces, as mentioned before: the time required for the k complete superpasses, the time the first processor must wait before beginning the last complete superpass, and the time required for the other processors to finish after the first processor is done.

The first superpass of processor 1 takes $N-1$ time units because it must make $N-1$ comparisons. Following this, every complete superpass takes M time units less than the preceding one. Thus superpass j ends at time

$$t_j = \sum_{i=1}^{j}(N-1-iM) = j(N-1) - \frac{j(j-1)}{2}M \qquad (4.5)$$

We compute the time the first processor spends waiting before beginning its last superpass by noting a structure to the movement of the processors: because two processors cannot compare the same data item simultaneously, we end up with processors comparing adjacent pairs of data items (as in Figure 4.1) Thus, we need a minimum of $2M$ data items if we are to have all the processors busy.

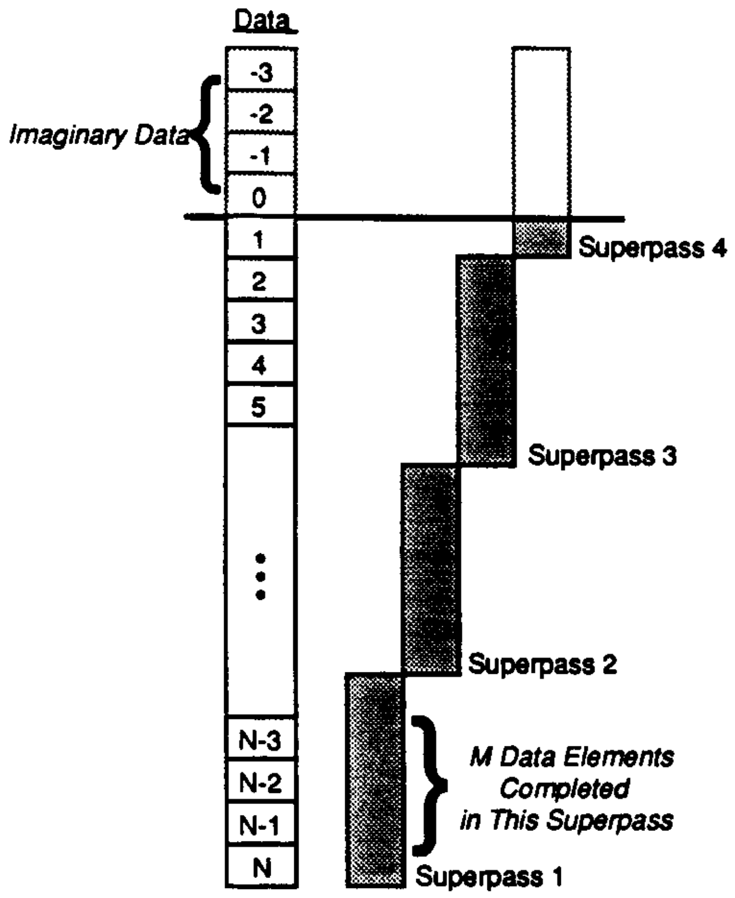


Figure 4.5: Computing $T(N)$.

When processor 1 finishes the second-to-last superpass, it puts the $(M+c)^{th}$ data items in its place, leaving $M+c-1$ data items still unsorted ($2M-1$ unsorted if $c=0$). This is not enough to keep all the processors busy, so processor 1 (and possibly others) must wait. We may find this waiting time by imagining that we do have $2M$ data elements (the data elements labeled 0 and smaller in Figure 4.5). Time spent in any comparisons with imaginary data elements is time that would be wasted. If we have $c=0$, we need one imaginary data element, which means we wait one time unit. If $c=M-1$, we need two imaginary data elements, which means we wait two time units. In general, we wait $M-c+1$ time units, given $c \neq 0$.

Finally, we must compute the time required for all the other processors to finish the remaining incomplete superpass. Note that after processor 1 finishes its pass, one pass finishes each time unit until all M processors have returned to the top of the list. Therefore, if we have c data items, $c>0$, it takes $c-1$ time units for processor 1 to finish its pass, and $(c-1)-1$ time units for the remaining processor to finish their passes, giving a total of $2c-3$ time units for the last passes. For the case of $c=0$, Equation 4.5 already counts the time for processor 1 to finish its last pass. The other processors must finish $M-1$ remaining data elements, which takes $M-2$ time units after processor 1 finishes.

Adding all this information together, we find

$$T(N) \quad = k(N-1) - \tfrac{k(k-1)}{2}M + M - 1 \qquad \text{for } c=0 \tag{4.6}$$

$$T(N) = k(N-1) - \tfrac{k(k-1)}{2}M + M + c - 2 \quad \text{for } c > 0 \tag{4.7}$$

From Equation 4.6, we get

$$\begin{aligned}
T(N) &= k(N-1) - \frac{k(k-1)}{2}M + M - 1 \\
&= \frac{Mk(k+1)}{2} - k + M - 1 \\
&= \frac{N(k+1) - 2k}{2} + M - 1 \\
&= \frac{N(Mk+M) - 2Mk}{2M} + M - 1 \\
&= \frac{N(N+M-2)}{2M} + M - 1
\end{aligned}$$

Similarly from Equation 4.7

$$\begin{aligned}
T(N) &= k(N-1) - \frac{k(k-1)}{2}M + M + c - 2 \\
&= \frac{Mk(k+1)}{2} - k(c-1) + M + c - 2 \\
&= \frac{Mk(Mk+M)}{2M} + \frac{2Mk(c-1)}{2M} + c - 1 + M - 1 \\
&= \frac{(Mk + 2(c-1))(Mk+M)}{2M} + M - 1 \\
&= \frac{(Mk + c + c - 2)(Mk + c + M - c)}{2M} + M - 1 \\
&= \frac{(N + c - 2)(N + M - c)}{2M} + M - 1
\end{aligned}$$

This proves Equations 4.3 and 4.4.

To compute the speedup, $S(N)$, we divide the time it takes one processor to finish the sort by the time it takes N processors to finish the sort. One processor takes

$$\sum_{i=1}^{N-1} i = \frac{N(N-1)}{2}$$

comparison/swaps. Dividing this by Equations 4.3 and 4.4, we get

$$S(N) = \frac{MN(N-1)}{N(N+M-2)+2M(M-1)} \quad \text{for } c = 0 \tag{4.8}$$

$$S(N) = \frac{MN(N-1)}{(N+c-2)(N+M-c)+2M(M-1)} \quad \text{for } c > 0. \tag{4.9}$$

Figure 4.6 plots Equation 4.9 for $N = 2000$ and M varying from 1 to 1000 (you get no further speedup for $M > N/2$). This data is identical for breadth-first bubblesort, which will be discussed in the next section. A little manipulation shows that $S(N)$ is $O(M/(1 + M^2/N^2))$, which is almost linear for small M, but gives decreasing performance as M grows.

4.2.3 Processor Utilization.

The processor utilization varies with time. Initially, not all the processors can be used, and every two time units a new processor can enter the data list. This causes the utilization to ramp up until all processors are being used. When the processors reach the end of their passes, one processor finishes a pass every time unit, and goes back to the top of the list to start again. Note, however, that although processors finish their passes at the rate of one per time unit, they **start** new passes at the rate of only one every two time units, and thus we expect some processors will have to wait before starting a pass. This behavior may be seen in the plot of processor utilization versus time (Fig. 4.7), where the downward spikes in utilization are caused by one series of passes ending and a new series of passes beginning. As the passes get shorter, these spikes get closer

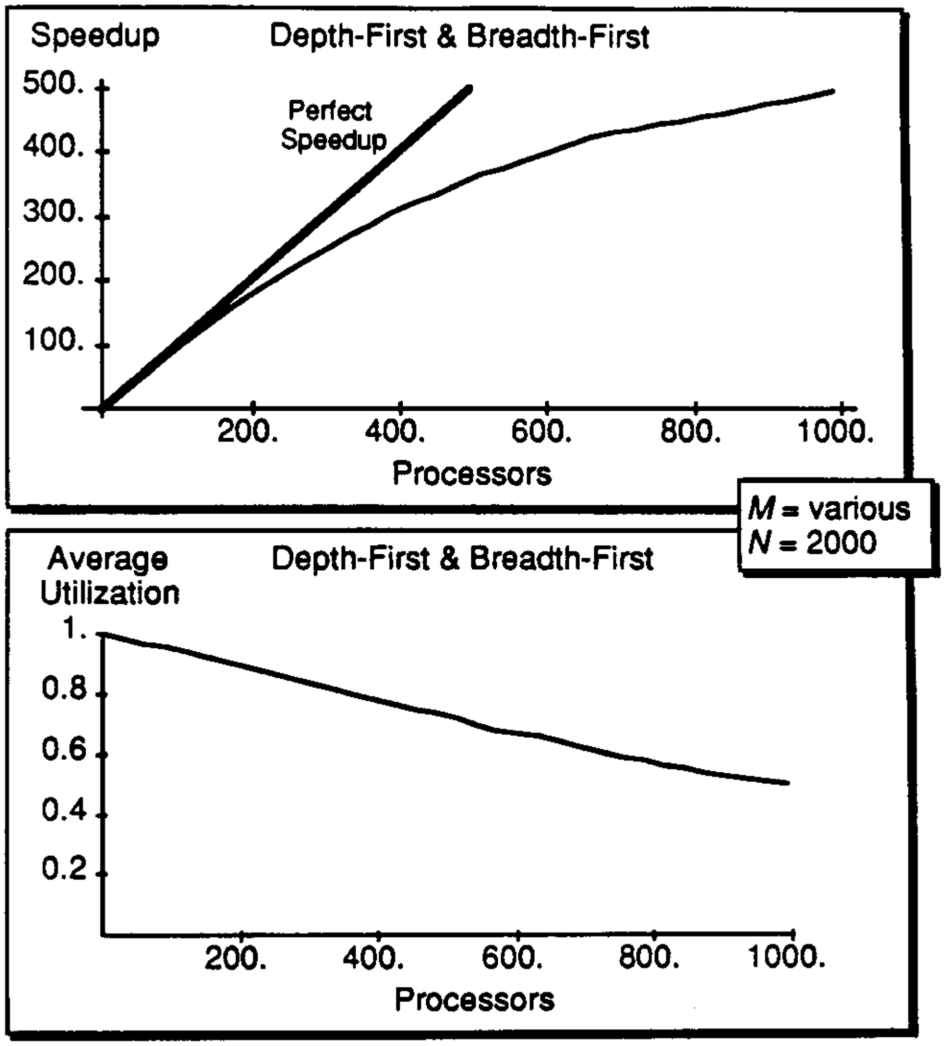


Figure 4.6: Speedup of depth-first and breadth-first bubblesort.

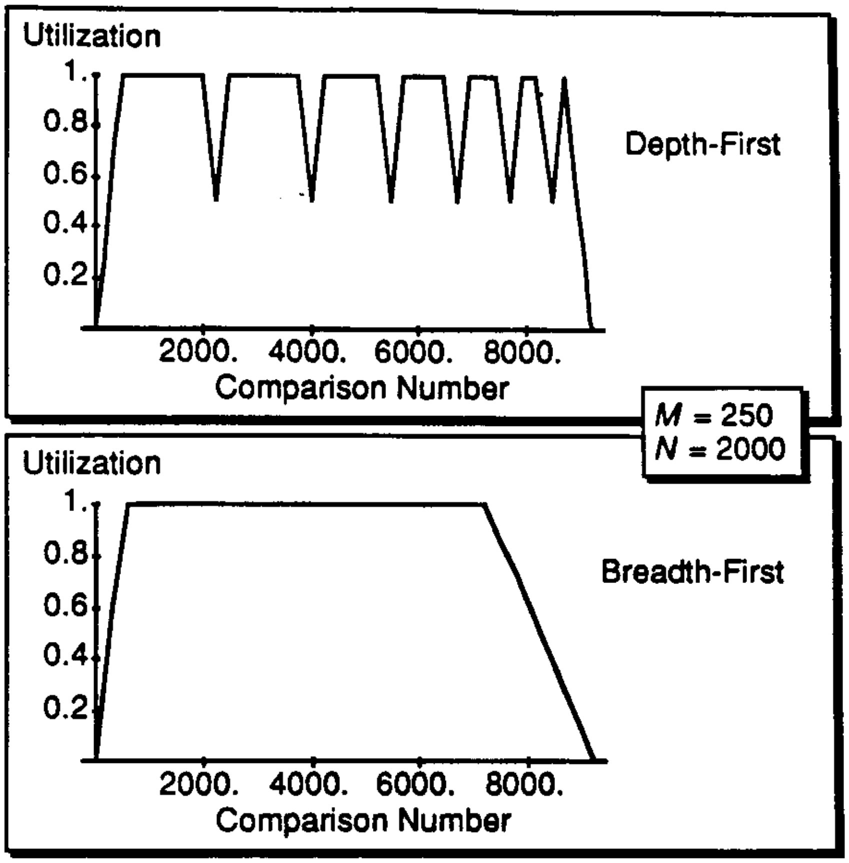


Figure 4.7: Utilization vs. time for 250 processors.

together, although the backlog remains the same. At the bottom of each spike, we are using only half of the processors (N processors have finished, but only $N/2$ processors have started the next pass). As some point, there is so little data left that not all the processors can be used, and the utilization gradually declines until the algorithm ends. We will discuss the plots for breadth-first bubblesort in the next section. If $M = N/2$ then we get no spikes (Figure 4.8) because we can use all the processors for only one comparison, and thereafter utilization declines.

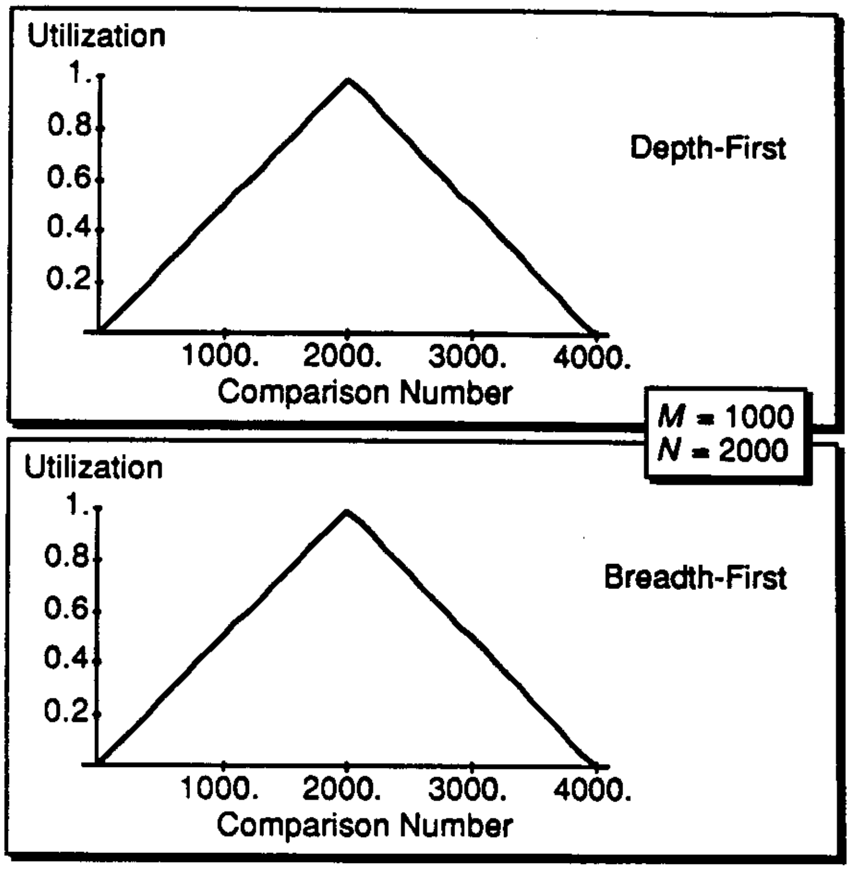


Figure 4.8: Utilization vs. time for 1000 processors.

4.2.4 Conclusion.

From a practical point of view, this algorithm is quite hopeless in a loosely coupled distributed system. The need for a global data structure, and the small amount of data transfered in each one of many messages makes the algorithm quite inefficient. Even in a shared memory system, which is much more suited to this type of algorithm, there are other algorithms just as easy to implement that run considerably faster. Akl [Akl85], for example, gives a parallel quicksort algorithm that is $O(N^b \log N)$ using N^b processors, where $0 < b < 1$. Thus, the algorithm described here is of only theoretical interest.

4.3 Breadth-First Bubblesort.

4.3.1 Description.

Our second algorithm is another version of bubblesort, called *breadth-first bubblesort.* It is similar to depth-first bubblesort, and in fact has the same running time, but the processors are used differently. In the depth-first sort, each processor takes a pass and works on that pass until it is completed; in the breadth-first algorithm processors start a new pass whenever possible, rather than continuing an older, unfinished pass. One may think of the processors as trying to stay at the top of the list in breadth-first bubblesort, whereas they try to run to the bottom of the list in depth-first bubblesort. Figure 4.9 illustrates this sort. Note that as soon as it can, processor 1 stops working on the first pass and starts the

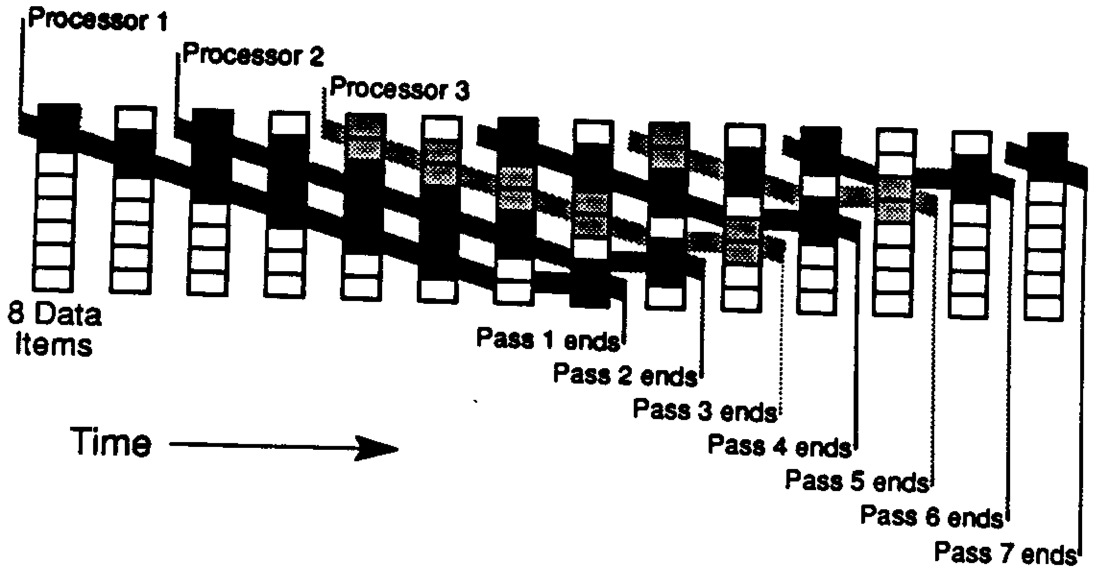


Figure 4.9: Breadth-first bubblesort with 3 processors and 8 data items.

fourth pass. Processor 2, working on pass 2, eventually runs into the incomplete pass 1, and must advance pass 1 by one data item before the progress of pass 2 can resume.

For a given number of processors and a given number of data items, this algorithm takes the same time as depth-first bubblesort. What changes is the utilization. Instead of there being spikes in the utilization as processors end a pass and start a new one, the utilization ramps up as fast as depth-first bubblesort, then stays constant with all the processors fully utilized, and finally ramps down as the passes become so short that not all the processors can be used.

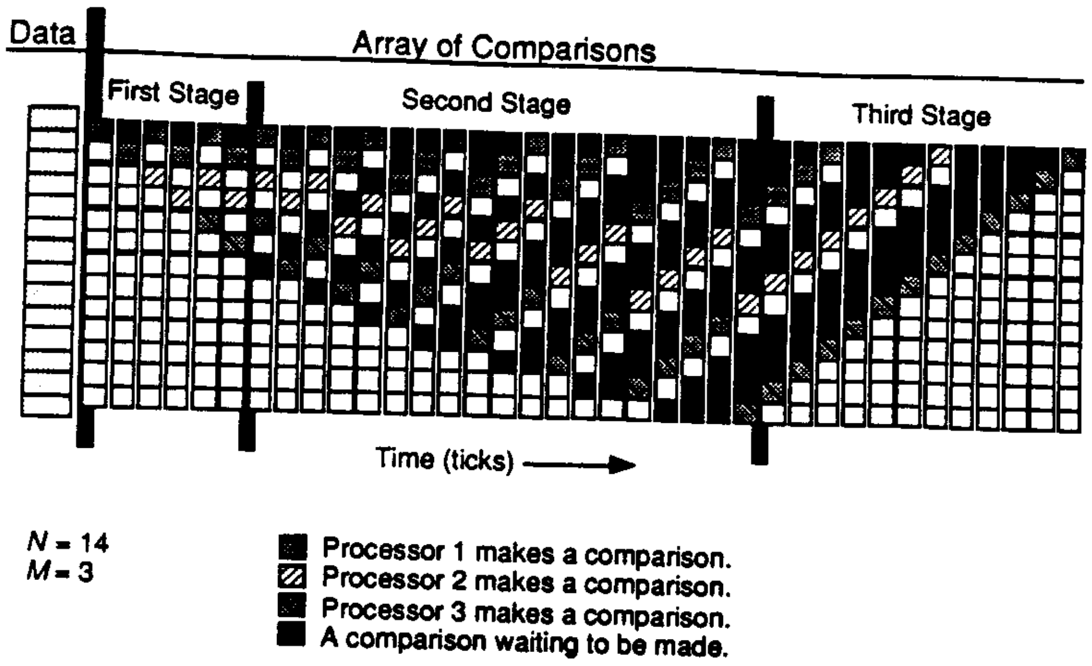

Data
Array of Comparisons
First Stage
Second Stage
Third Stage
Time (ticks)
N = 14
M = 3
Processor 1 makes a comparison.
Processor 2 makes a comparison.
Processor 3 makes a comparison.
A comparison waiting to be made.

Figure 4.10: "Trains" developing in breadth-first bubblesort.

4.3.2 Speedup.

To analyze the speedup, we examine three distinct stages in the algorithm. In the first stage, processors enter the algorithm. This is similar to the starting bottleneck noted in the depth-first algorithm. One processor enters every other time unit until the first pass has made its $2M^{th}$ comparison/swap. Once this occurs, the second stage of the algorithm starts, in which we get "trains" of passes growing longer and longer. This stage ends when the first pass ends. In the final stage of the algorithm, one pass ends every time unit, and fewer and fewer processors are used. Refer to Figure 4.10 for the example below.

For convience during the analysis, let $N = Mk_b + c_b + 1$ where $k_b = 0, 1, 2, \ldots$ and $c_b = 1, 2, \ldots, M$ such that $N \geq 1$. This differs from the definition of the previous section but makes the analysis clearer. We will show that the speedup results in this section are the same as the depth-first speedup results.

Also, assume the processors are synchronized, and at every "tick" of a clock, each processor makes a comparison/swap.

Because processors cannot compare the same data element simultaneously, we will observe "trains" of passes. A "train" is defined as a number of passes, each of which cannot proceed until the one ahead proceeds. The pass at the head of a train is not blocked, and may proceed whenever it wishes.

We will examine the time to finish only for $N > 2M+1$. If $N \leq 2M$, it takes $N-1$ ticks for the first pass to finish, and it takes $N-2$ ticks for the remaining $N-2$ passes to finish, for a total $T(N) = 2N-3$.

4.3.2.1 First Stage: Forming $M+1$ Trains of Length 1.

We define the first stage as the time required to form $M+1$ trains of length 1. For the $2M-2$ ticks of the algorithm, processors are still entering the algorithm. On tick $2M-1$, M processors compare items 1 & 2, 3 & 4, 5 & 6, ..., $2M-1$ & $2M$; creating one train of length 2 at the top of the list, and $M-1$ trains of length 1. Another tick and the heads of the trains all move down one, creating $M+1$ trains of length 1, ending this stage. The total time in this stage is $2M$.

4.3.2.2 Second Stage: Trains Advance.

What we want to find in this stage is the number of ticks it takes for the first pass to finish, which is the first time that a processor compares data items $N-1$ and N.

What interests us now is the number of ticks it takes for the leading train (with pass 1 at its head) to move down one comparison. Let us generalize and suppose that we have a leading train of length 1 and M trains of length i (see Figure 4.11). Only the top M trains move at the each tick, because we have only M processors, so the bottom train stays where it is. After i ticks, the top M trains have completely shifted forward by one comparison, and the lowest of these "ran into" the stationary train of length 1. At this point we have a lowest train of length $i+1$ and $M-1$ trains above it of length i. One more tick allows the lowest comparison to move forward, for a total of $i+1$ ticks to move the lowest comparison down one.

This gives us a configuration of: one train of length 1, a following train of length $i+1$, and $M-1$ trains of length i (again, refer to Figure 4.11). After i more ticks to shuffle the top M trains forward (again, the second train merges with the first), and one more tick to move the lowest pass forward, we find that we have two trains of length $i+1$ (the lowest of the top M trains). As we continue this process, every $i+1$ ticks we expand another length i train into length $i+1$. Thus, it takes a total of $M(i+1)$ ticks to convert M trains of length i and one

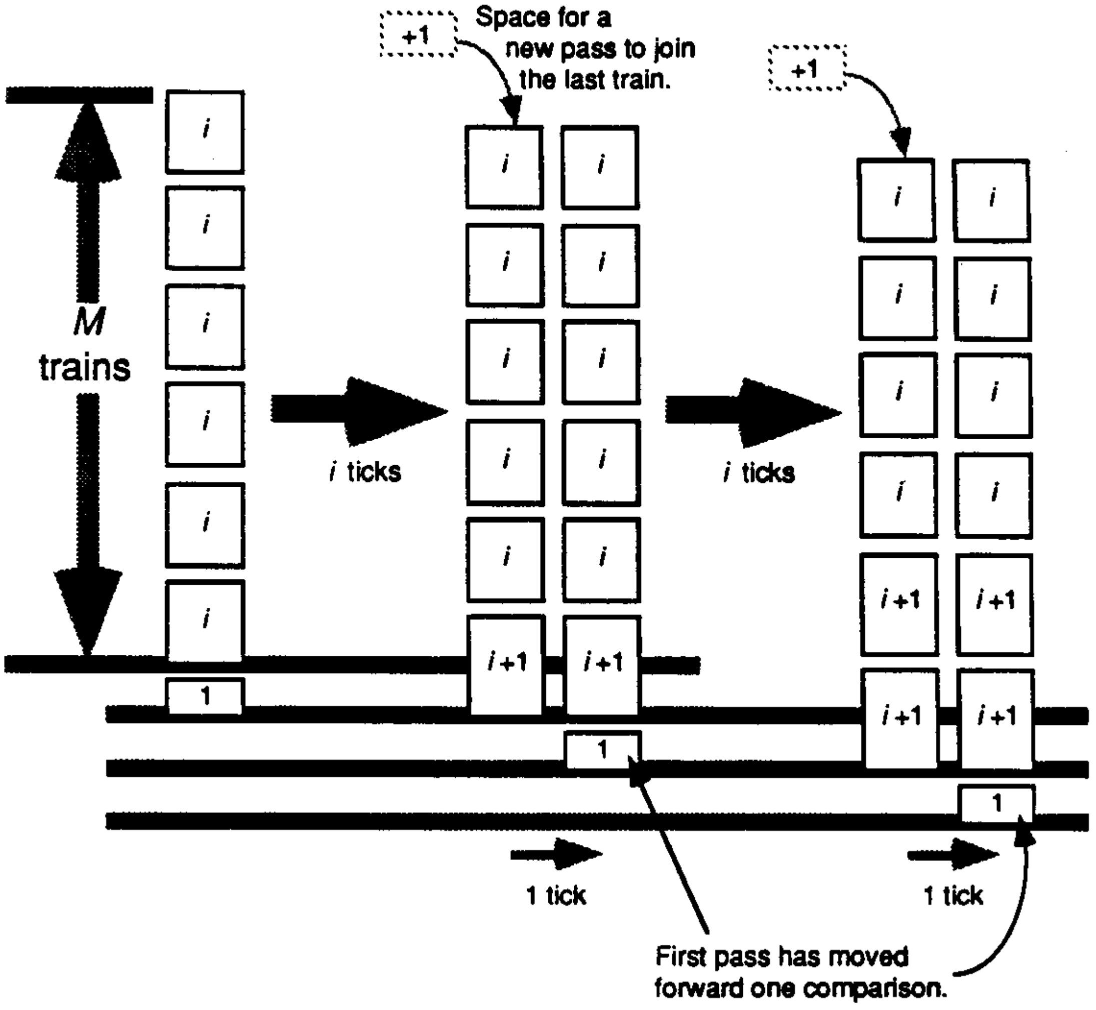

+1
Space for a new pass to join the last train.
+1
M trains
i
i+1
1
i ticks
i ticks
1 tick
1 tick
First pass has moved forward one comparison.

Figure 4.11: "Trains" developing in breadth-first bubblesort.

train of length 1 into M trains of length $i+1$ and one train of length 1, moving the lowest pass down one comparison every $i+1$ ticks in the process.

When the second stage begins, we have trains of length $i = 1$, which means that moving the first pass down another M comparisons (it has already made $2M$ comparisons) will take $2M$ ticks. At this point, we have trains of length $i = 2$, and to move the first pass another M comparisons will take $3M$ ticks, and so on, with $i = 3$, $i = 4$, etc. Remembering that $N = Mk_b + c_b + 1$, we see that it takes $M + \sum_{i=1}^{k_b-1} iM$ ticks to make Mk_b comparisons, plus $k_b c_b$ ticks to move the lowest pass an additional c_b comparisons and finish the second stage.

4.3.2.3 Third Stage: Finishing Passes.

After the first pass finishes, passes end at the rate of one every clock tick during this final stage. By definition the first pass has completed, so $N-2$ passes remain, and it takes a total of $N-2$ clock ticks to finish this stage.

4.3.2.4 Total Time for All Stages.

The total time for all stages is

$$\begin{aligned} T(N) &= M + \sum_{i=1}^{k_b-1} iM + k_b c_b + N - 2 \\ &= M + \frac{Mk_b(k_b-1)}{2} + k_b c_b + N - 2 \end{aligned} \tag{4.10}$$

We now verify that this is identical to the time to finish for depth-first bubblesort with the same number of processors and data (Equations 4.3 and 4.4).

The case of $N = Mk$, $k = 1, 2, \ldots$, in the depth-first section should correspond to the case of $c_b = M - 1$, $k_b = k - 1$ in this section. Let us verify this:

$$\begin{aligned}
T(N) &= M + \frac{M(k-1)(k-2)}{2} + (k-1)(M-1) + N - 2 \\
&= \frac{N^2 - 3NM + 2M^2}{2M} + \frac{4NM}{2M} - k - 1 \\
&= \frac{N(N+M-2)}{2M} + m + k - k - 1 \\
&= \frac{N(N+M-2)}{2M} + m - 1,
\end{aligned}$$

which is the same as Equation 4.3.

For $N = Mk + 1$ in the depth-first section, we have $c_b = M$ and $k_b = k - 1$ in this section. We verify their equality:

$$\begin{aligned}
T(N) &= M + \frac{Mk(k-1)}{2} + (k-1)M + N - 2 \\
&= \frac{M^2(k^2 - 3k + 2 + 2k)}{2M} + \frac{2M(N-2)}{2M} \\
&= \frac{M^2k^2 - M^2k + 2MN - 2M}{2M} + M - 1 \\
&= \frac{Mk(Mk + M)}{2M} + M - 1 \\
&= \frac{(N-1)(N+M-1)}{2M} + M - 1,
\end{aligned}$$

which is the same as Equation 4.4 for $c = 1$.

Finally, for all other values of $N = Mk + c$ in the depth-first section, we can use $c_b = c - 1$ and $k_b = k$. We verify the equality of the time to finish depth-first and breadth-first sorts:

$$T(N) = M + \frac{Mk(k-1)}{2} + k(c-1) + N - 2$$

$$\begin{aligned}
&= M + \frac{(N-c)((N-c)/M-1)}{2} + \frac{2Mk(c-1)}{2M} + N - 2 \\
&= \frac{(N-c)(N-c-M)}{2M} + \frac{2Mk(c-1)}{2M} + \frac{2MN-2M}{2M} + M - 1 \\
&= \frac{N^2 + N((M-c)+(c-2)) + (c-2)(M-c)}{2M} + M - 1 \\
&= \frac{(N+c-2)(N+M-c)}{2M} + M - 1,
\end{aligned}$$

which is the same as Equation 4.4. Thus we have, in rather lengthy fashion, shown that depth-first and breadth-first bubblesorts take the same amount of time. Refer to Figure 4.6 for speedup and average utilization plots.

4.3.3 Processor Utilization.

Processor utilization vs. time is plotted in Figures 4.7 and 4.8, along with the corresponding plot for depth-first bubblesort. Examining the plot for breadth first sort, we can see that the utilization goes through three stages, and these correspond to the stages we used in our speedup analysis.

In the first stage, utilization ramps up. We have already noted in a previous section that we can add only one processor every two time units. If we are using many processors, this could take a fair amount of time.

In the second stage, we have 100% utilization of processors. Because each processor works on its own train of passes, there is never any conflict that requires one processor to be idle.

In the third stage, utilization ramps down. We can use only as many processors as we have trains, and, conceptually, we shift trains off the top of the list as

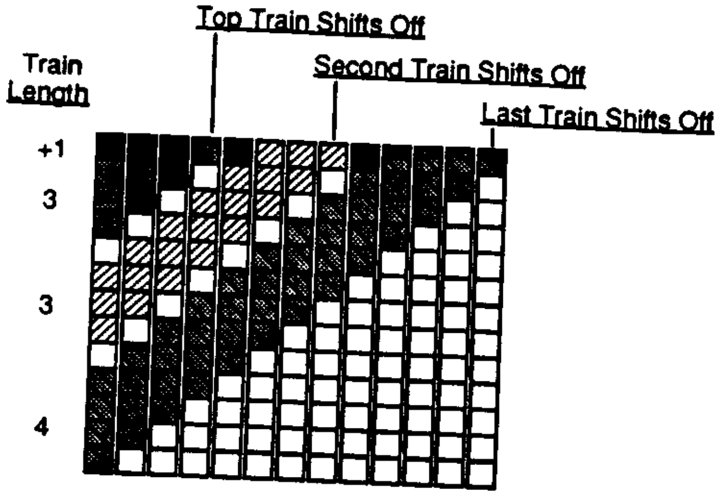


Figure 4.12: Trains shifting off.

the algorithm winds down, and therefore processor utilization must decrease.

When the first pass is poised to make its last comparison, we have c_b trains of length k_b and $M - c_b$ trains of length $k_b - 1$, plus one additional comparison in the top train because there is always a pass waiting to start (see Figure 4.12). The total length of these trains, plus the intervening spaces, should equal $N - 1$, the maximum number of comparisons in one pass. As a check:

$$c_b k_b + (M - c_b)(k_b - 1) + M - 1 + 1 = Mk_b + c_b = N - 1.$$

As we make comparisons, we may think of the trains as shifting up, off the top of the data list, and the algorithm finishes when the last train shifts off. Thus it takes k_b ticks to shift off the topmost train of length $k_b - 1$ (remember, the top train is always lengthened by one because of the waiting pass), at which point we have only $M - 1$ trains left and one processor must be idle. After another k_b ticks, another train has shifted off, and we need only $M - 2$ processors. After the top c_b trains have shifted off, it then takes $k_b + 1$ ticks to shift off each train, and the process continues until the final train shifts off the top.

Referring back to the graph in Figure 4.7, we notice that both depth-first and breadth-first bubblesort start in the same fashion, but once all processors are busy, breadth-first bubblesort makes more efficient use of them, and it can release processors sooner in the last stage of the algorithm. Thus, we would prefer this algorithm to its depth-first version.

4.3.4 Conclusion.

Compared to depth-first bubblesort, breadth-first bubblesort makes better use of processors because it does not leave processors idle, waiting for other processors to move passes forward. However, it does not sort any faster, and remains an $O(N^2/M + M)$ sort.

4.4 Split Bubblesort.

4.4.1 Description.

One problem with the depth-first and breadth-first bubblesorts is that accessing individual data pairs for comparing and swapping is inefficient if communications takes a relatively long time. Furthermore, synchronizing processors so that they make comparisons in the proper sequence requires additional time and communication. In general, we prefer that processors work mostly on data that is local to themselves and communicate with others only occasionally. Let us then

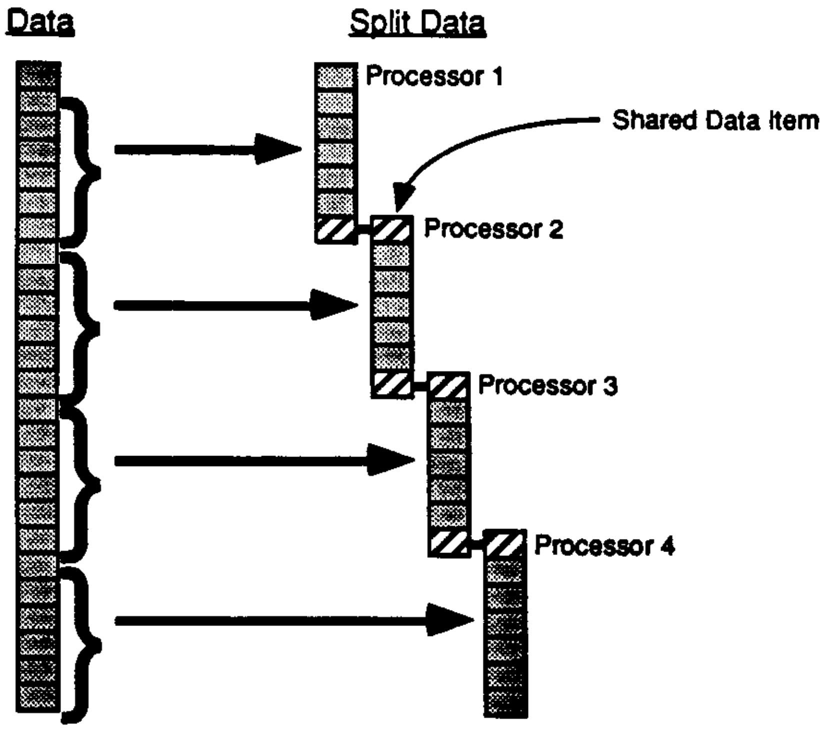


Figure 4.13: Data split 4 ways.

split the data among the processors and let each processor perform bubblesort on its own data, as in Figure 4.13. To allow communication, these lists overlap with shared data: the data item at the bottom of one processor's list is the same as the data item on the top of the next processor's list. If one processor changes a shared data item, it is communicated to the other processor. To sort all the data, each processor continuously sorts its data, and the sort terminates when an external mechanism finds that all the data is sorted, at which point all the data must be sent to processor 1 and concatenated.

We choose to make the processor's lists as even as possible. Possibly we would want to distribute the data on the basis of the amount of activity in each section of the list. In particular, the ends of the data list see less activity than the middle (as the ends become sorted, less data passes through them), so we may wish to concentrate processors in the middle of the list. This is a reasonable optimization in a practical system, but it does not change the fundamental nature of the algorithm, and we will not analyze this alternative.

4.4.1.1 Concurrency Control.

We cannot allow two processors to swap the same data item simultaneously, and thus we must have some form of concurrency control on overlapped data items. We can use either optimistic or pessimistic concurrency control. In the former case, if a processor swaps a shared data item, then it sends the new data item to the other processor; unless otherwise notified, it assumes that the new data was accepted. In the pessimistic case, a processor must lock the shared data item before it is examined, and release it when it has done. When communications takes a relatively long time, as it does in a distributed system, then the optimistic method is better by far. It may take several orders of magnitude more time to request and release a lock than it does for a processor to make a pass through its data, and ultimately most processors spend their time waiting for communications. We implemented split bubblesort using pessimistic concurrency control on the BBL system [FSK89], and it took intolerably long to run.

However, optimistic concurrency control as described above may be a bit too optimistic because it requires keeping information so it can undo data swaps. This brings with it the possibility of cascading undos: processor 1 must undo a swap, invalidating a swap with processor 2 , which must in turn invalidate a swap that processor 2 made with processor 3, and so on. When the sorting first starts, it is likely that most shared data item swaps will have a conflict, and at this point, pessimistic concurrency control would work better then optimistic. Later in the sort, such swaps are unlikely to conflict, and optimistic concurrency control would then be the choice. As a compromise, one could limit a processor's actions after swapping shared data, making it pause if it tries to do anything affecting another processor, but this includes acknowledging swaps initiated by other processors. Although this prevents cascading undos, we would get the same sort of delays as locking: processor 2 waits for an acknowledgement from processor 1 that its swap was ok, and meanwhile processor 3 wait for 2, 4 waits for 3, and so on. We will, it seems, always have delays, be they for locking or for undos.

4.4.1.2 Termination.

Depth-first and breadth-first bubblesort require global knowledge of the progress of the passes to know when the sort has finished. There is no such global data knowledge in split bubblesort, and we must resort to other methods. If all the processors are synchronized, and make comparisons simultaneously, then we can compute a maximum time to finish, as in the speedup section below. However, we would prefer not to rely on this because it is unlikely that processors work in true synchronization, given varying delays for concurrency control, and it is quite likely that the data will require less than the maximum time to be sorted. Thus, we must turn to more active methods of detecting termination.

Much work has been done in the area of termination detection ([AS87], [Mis83], [DS80], [Fra80], [FR82]). Misra's marker algorithm [Mis83] is particularly well suited for this sort. The algorithm depends on "painting" the processors: white if they are sorted, black if they are not sorted. A marker traverses the processors, keeping a count, m, of the number of white processors it sees. When this count reaches $2(N-1)$, the data is sorted. The algorithm is quite simple:

1. Every processor is initially black. Processor 1 starts the marker with $m = 0$.
2. Assume processor i receives the marker. If this processor is black, set $m = 0$. Otherwise set $m = m + 1$. In either case, send the marker to the next processor only when this processor becomes sorted. We want the marker to travel from processor 1 to 2 to ... to N, and then in reverse, to $N-1$ to ... to 2 to 1, repeating this sequence *ad infinitum*. Therefore, send the marker to processor $i-1$ or $i+1$, as is appropriate to maintain this sequence.
3. If processor i ever becomes unsorted, it paints itself black.

It is important to note that when a shared data item is in the process of being swapped, both processors sharing the data item cannot guarantee that they are sorted, and therefore, they must paint themselves black.

Misra proves that this algorithm guarantees termination if the marker passes over all the links in the network and finds that all the processors have been continuously white (sorted). We count each shared data item as two unidirectional links between adjacent processors, and by bouncing back and forth between processors 1 and N, we continuously traverse all the links. When the marker count reaches $2(N-1)$, then all the processors have remained sorted during one complete pass over the links, and the entire list of data it sorted.

4.4.2 Speedup.

The actual time to sort the data is highly dependent upon the initial distribution of the keys. We will look at a worst case: every data item must bubble to its proper location from the first (top) position in the list of data. This is the worst case because, as sorted data fills up the bottom of the list, the remaining unsorted data becomes more orderly – data items cannot help but be pushed toward their proper locations. Once i data items have been sorted into the bottom i positions in the data list, the farthest we can force the $i+1^{st}$ item to travel is through $N-i-1$ comparisons, and we do this by starting that data item from position 1.

Let $N = (l-1)M+1$, which means that each of M processors has a local data list containing l data items total (including shared data items). Studying general N would not change our basic conclusions, but would complicate the analysis. Also for analytic tractability, assume that processors are synchronized, and all make a comparison at the tick of a clock.

Each processor requires $l - 1 = (N - 1)/M$ comparison/swaps to make one pass through its local data, and assume, for simplicity, that each processor always makes complete passes through its data (after all, it does not know when the top or bottom data item will change). To bubble the largest data item from the top of the list to the bottom requires one pass through each of M processors. After the largest data item leaves the first processor, the second largest data item can start downward through the first processor, and it arrives at its final location in the next pass of the processor. Subsequent data items act similarly, and we have a pipeline effect (Figure 4.14), with one data item finishing in each pass. Thus we require M passes of each processor through its local data to put the largest data item in place, and we require $N - 1$ additional passes to put the rest of the data items in place. The total time to finish is $T(N) = (M + N - 1)(l - 1) = (M + N - 1)(N - 1)/M$, which is $O(N^2/M + M)$. The worst case finishing time of the split bubblesort algorithm is, unfortunately, of the same order as the other parallelized bubblesorts we have discussed. To find speedup, we divide $T(1)$ by $T(N)$ and find:

$$S(N) \leq \frac{NM}{2(N + M - 1)}, \tag{4.11}$$

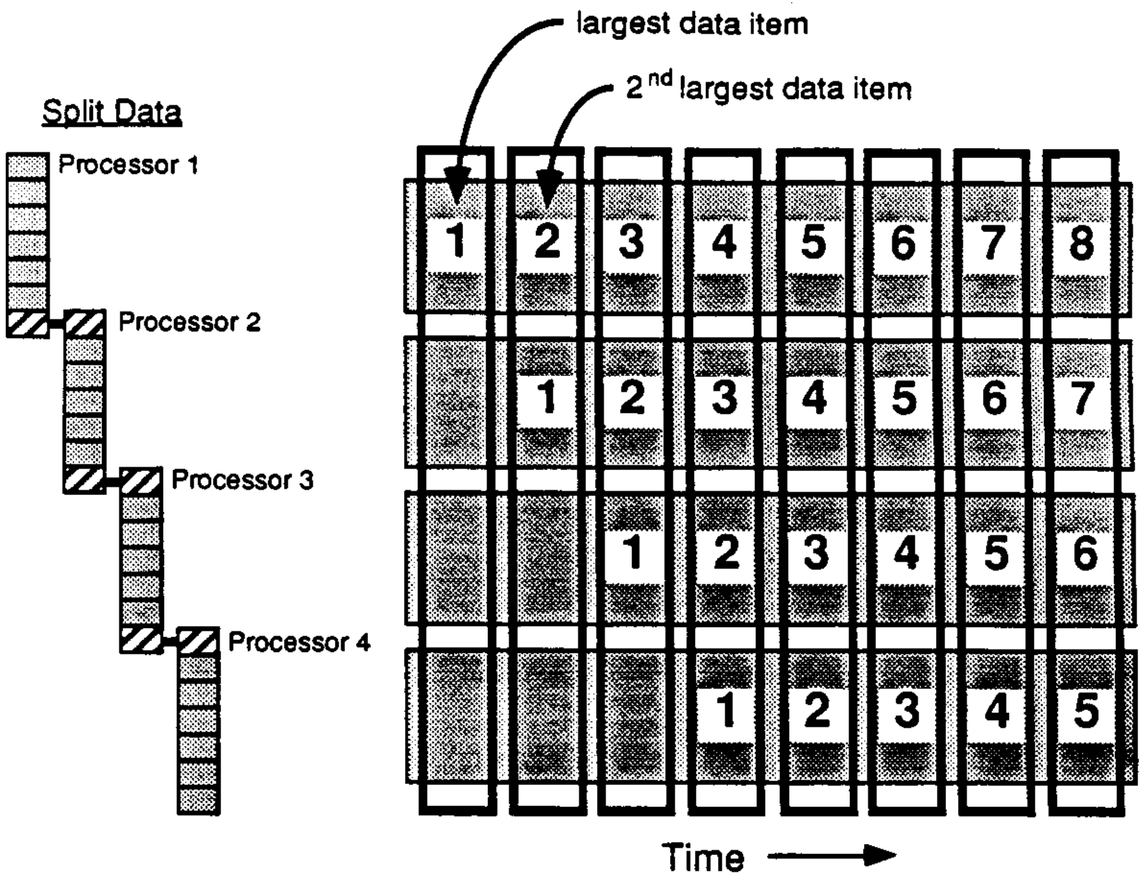


Figure 4.14: Pipelining of data movement.

and we note that if $N \gg M-1$, then this expression becomes approximately $M/2$. Thus, in an ideal situation, with no delays for communication or concurrency control, speedup is roughly linear until the number of processors is $N/2$ (beyond which further processors are not useful), and simulation results (Figure 4.15) bear this out.

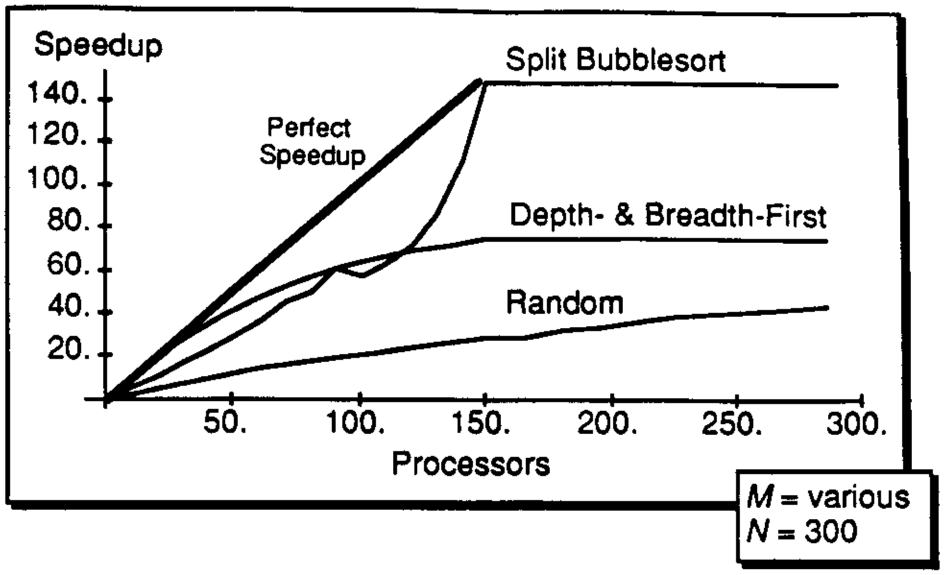


Figure 4.15: Speedup (from simulations) of split bubblesort, depth-first bubblesort, and random bubblesort.

4.4.3 Processor Utilization.

Processor utilization appears to be very high. According to the algorithm, processors are never idle, but this is misleading. Some of the processor work is "wheel spinning" — a processor continues to make passes through its data even when nothing changes, and the processor might as well remain idle. Furthermore, a real system will have processors idled because of concurrency control (e.g. waiting for locks or acknowledgements).

4.4.4 Conclusion.

Split bubblesort is attractive, as bubblesorts go. It requires no global data structures, unlike depth-first and breadth-first bubblesort. Much work is local to each processor and can be done without communciations. Finally, each processor communicates with no more than two adjacent processors, and a linear or ring topology is ideally suited to this pattern. Simulation results do show that this sort can run twice as fast as depth- and breadth-first bubblesorts when communication is instantaneous, and relative performance should be even better when accounting for real communications delays. Unfortunately, there is a need for concurrency control, which can cause considerable slowing in a real system. Furthermore, messages sent in this sort will tend to be small, leading to relatively low communications efficiency. Split bubblesort is a good distributed bubblesort algorithm, but other distributed sorting algorithms, such as distributed quicksort, are inherently faster.

4.5 Random Bubblesort.

4.5.1 Description.

From a theoretical point of view, it would be interesting to let processors pick random pairs of adjacent data items to compare and swap, rather than force them to follow the normal patterns of bubblesort phases. Such a sort avoids the need for coordinated action among processors, but now requires a termination mechanism. To detect a sorted list it is sufficient to have one processor check the entire list periodically; however, to prevent the checking processor from missing a swap, we must force all processors to stop swapping during the check. This would be bad in a real system, but because this algorithm is really for comparison with the other bubblesort algorithms, and not a seriously proposed implementation, we won't let its overall poor performance deter us from some analysis.

We assume that the probability of picking any pair of adjacent data items to compare and swap is uniform over all choices. For better performance, we might want to vary this distribution: as the sort progresses, the ends of the data list tend to be more sorted than the center, so it makes sense to have more processors working on the center data items than on the ends. However, we will retain our simple uniform assumption.

As before, we cannot allow two processors to compare and swap the same data item simultaneously. For purposes of simulation and analysis, we assume synchronous processors that all make (or attempt to make) a comparison/swap at the tick of a clock. However, at each tick, we sequentially assign the processors to random comparisons, and if any processor would conflict with a previously assigned processor, then one of these processors is idle for this clock tick. We repeat this procedure at every clock tick.

4.5.2 Speedup.

It is, unfortunately, difficult to derive an expression even for the expected time to finish a random bubblesort. There are many possible sequences of comparison/swaps that are guaranteed to lead to a sorted list, and there are many ways each sequence can occur during the random comparison/swaps of the participating processors. If we could determine the various minimal sequences of comparisons necessary to guarantee that the data is sorted, then we could statistically add the extra, unnecessary comparisons that will occur in a real system, and use this to find the finishing time. However, the number of minimal valid sequences that guarantee a sorted list seems very large and difficult to count. As an example, if comparisons occur in the same order as standard single-processor bubblesort, then we are guaranteed that the list is sorted. However, we may also sort the list by completely sorting data items 2 through N, and then comparing data items 1 and 2, then 2 and 3, ..., then $N-1$ and N (thus bubbling data item 1 into its proper place). There are yet more sequences of comparisons that guarantee the list is sorted, but they get very complex, and counting them is difficult. Instead, we must rely upon simulations to get an idea of performance.

Figure 4.16 shows the speedup (over standard, single processor bubblesort) and average utilization for $N = 500$ and varying numbers of processors. We notice that as the number of processors becomes large, speedup of the random bubblesort gradually rises to about half that of depth-first bubblesort. This is because we get ideal depth-first bubblesort speedup when we make $N/2$ comparisons at every clock tick, but we must make "odd comparisons" (compare data items 1&2, 3&4, etc.) on one clock tick, and the "even" comparisons (compare data items 2&3, 4&5, etc.) on the next clock tick. When we make random comparisons with many processors, we will make close to $N/2$ comparisons, but, lacking a mechanism to ensure that the processors make the necessary compar-

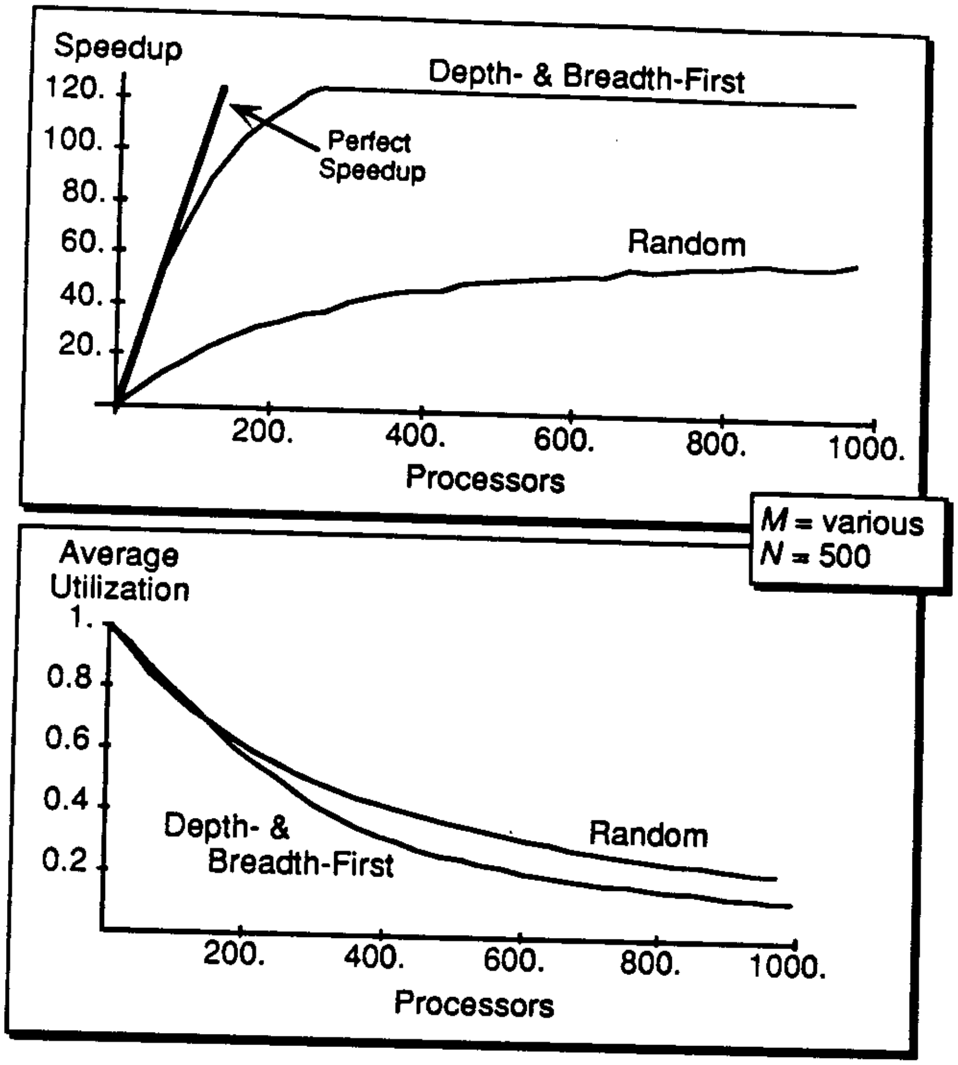


Figure 4.16: Speedup of random and depth-first bubblesorts.

isons, only about half of the comparisons will be useful. Thus we would expect random bubblesort's speedup to approach half the maximum speedup of depth-first bubblesort.

4.5.3 Processor Utilization.

Processor utilization varies randomly with time, but we can estimate it by assuming a synchronous system, with processors making a comparison/swap simultaneously, at the tick of a clock. Given N data items, then we have $N-1$ possible comparisons to assign to each processor. We make the maximum number of comparisons, N_{cmax}, by assigning one processor to every other comparison, yielding $N_{cmax} = \lceil (N-1)/2 \rceil$. However, if initial processors are assigned every third comparison, then we can make a maximum of only $N_{cmin} = \lceil (N-1)/3 \rceil$ comparisons (because a single, untouched data item is between every pair being compared). Thus the maximum number of possible comparisons is determined by the assignments of the initial processors, and this dependency makes it difficult to find a distribution for the number of busy processors. If we assume we have found the maximum number of possible comparisons, N_c, then we can find the distribution of busy processors as follows. The total number of ways to let each of M processors pick one of N_c comparisons is N_c^M. Of these, the number of ways that M processors can pick exactly i comparisons ($i \leq M$) is $(N_c)(N_c-1)\ldots(N_c-i+1)S(M,i)$, where $S(M,i)$ is the Stirling number of the

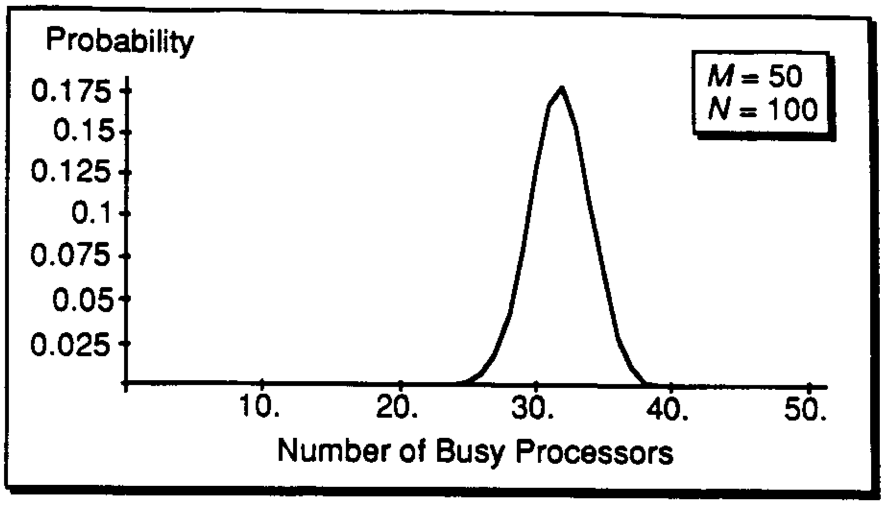


Figure 4.17: Distribution of number of busy processors.

second kind[1]. Dividing this by N_c^M, we get the probability of i busy processors.

$$\text{P}[i \text{ busy processors}] = \frac{(N_c)(N_c-1)\dots(N_c-i+1)}{N_c^M} S(M,i). \tag{4.12}$$

If we sum this for $i = 1, ..., N_c$, an identity in [GKP89] verifies that this is equal to 1. Figure 4.17 shows this probability density for $N = 100$, $N_c = N_{cmax} = 50$, and $M = 50$, which is a best-case estimate for the processor utilization.

We note that this density is quite narrow about its mean, thanks to the law of large numbers. When we plot the actual utilizations from a simulation (Figure 4.18) we see that the utilization does, in fact, vary mainly within a narrow band.

4.5.4 Conclusion.

Random bubblesort is not a realistic algorithm for implementation, but it is an interesting idea from a theoretical viewpoint. Performance is, not surprisingly,

[1] $S(M,i)$ = the number of ways to partition M elements into i non-empty subsets.

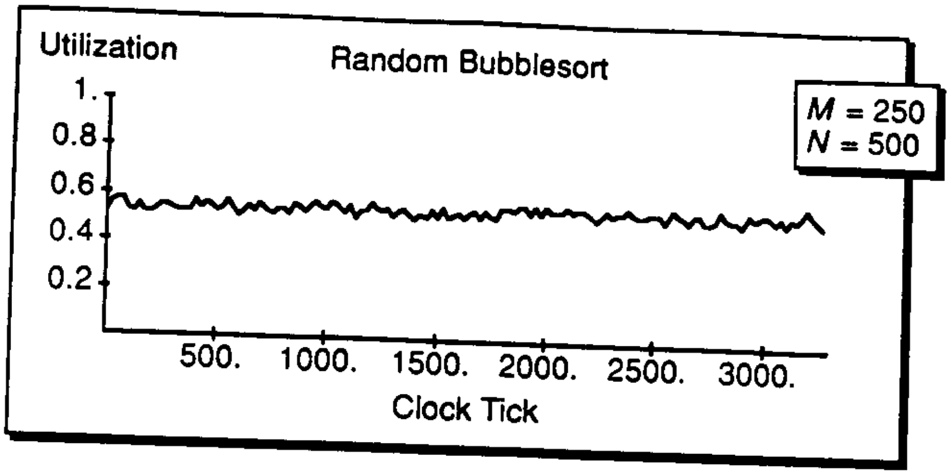


Figure 4.18: Utilization during a run of random bubblesort.

the worst of all the parallel bubblesort algorithms, and exact analysis of the algorithm is surprisingly difficult.

4.6 Chapter Conclusion.

In this chapter we examined four parallel versions of bubblesort. One of them, split bubblesort, gives impressive, almost linear speedup for up to $N/2$ processors, and it has the desirable property that each machine's interaction is limited to one or two other machines. The other sorts generally performed worse, and are only of theoretical interest. It remains to be seen how split bubblesort compares to distributed versions of inherently efficient sorts, such as quicksort, but it is likely that quicksort and its ilk will always perform better than any version of bubblesort.

CHAPTER 5

Conclusion

In this dissertation, we have examined three topics:

1. Models for the distribution of time to finish a program on a network of transient processors.

2. The application of information theory to several sorting algorithms.

3. Parallel bubblesort algorithms.

The first of these topics still has a variety of interesting extensions that we discuss in the next section. The second topic is interesting for the illumination it gives to familiar algorithms, but single-processor internal sorting is a well trodden field with, it seems, mainly esoteric work remaining. There are sorting algorithms that are extremely close to the information theoretic lower bound for comparison-based sorting, certainly close enough for any practical purpose, but there is still room for minute improvement on these and other researchers continue to work on this topic. Finally, the last topic, parallel bubblesort is, quite clearly, not very practical, and any further work in this area should start with algorithms that are inherently faster than bubblesort, such as quicksort and mergesort. A distributed radix sort would also be interesting, and distributed *external* sorting is a topic that has not been widely addressed.

5.1 Future Work with Transient Processors.

There are four areas in which to extend the analysis of transient processors:

- Better Brownian motion approximations.
- Analytic models of networks of transient processors, and not approximations.
- Models of overhead in distributed processing.
- Measurements to compare the models to real processors and networks.

We discuss each in turn.

5.1.1 Better Brownian Motion Approximations.

There are several ways to extend the Brownian motion model. The first is to find a new expression for the variance of the underlying Brownian motion (σ_b^2) so that the model provides reasonable results for small W. We know that under such conditions the Brownian motion model indicates more variance in the finishing time than actually exists, and there may be some method to reduce this excess.

A second extension is to allow multiprogramming on the processors in the network. Instead of getting either all the processor or none of it, one may receive a fraction of its capacity. If the available capacities are a finite set (e.g. 0, f_1, f_2, and 1, where $0 < f_1, f_2 < 1$), then this can probably be done by reducing the capacity of all processors to the lowest non-zero fraction available (e.g. f_1) and increasing the total number of processors to M/f_1 so that the total capacity remains constant. If the available fraction is a continuous function, then other techniques must be used. This should be compared to the results of G/G/1 processor-sharing analysis.

5.1.2 Analytic Models of Transient Processors.

Deriving analytic models of transient processors is not a simple problem because, when viewed as a whole, the network has a varying service rate. The system point method, discussed in [BP77] and [BP81] may well be useful in this situation. Brill and Green, in [BG84], analyzed jobs that required a random number of servers (again, using the system point method), and this may be another starting point for modeling.

5.1.3 Models of Overhead in Distributed Processing.

Overhead is an unfortunate fact of life that detracts from the rosy picture our analysis has given us. Although we potentially have enormous computational resources available to us, because of overhead and the resulting delays, we might get relatively little use out of these resources. This inefficiency arises from both the communications medium and the program structure, and from the way these two entities interact. All these inefficiencies we refer to as *overhead*, and we briefly examine them to see how much delay they cause us and lay out a path for future research.

We attack the problem of overhead by analyzing algorithm performance. Among other things, a distributed algorithm is typically analyzed for its speedup, which we have defined the ratio of the time to execute on one processor to the time to execute on M processors. The speedup thus found is usually quite idealized because the analysis does not account for communications delays or the realities of processing. This is quite understandable because communications delays vary not only from network to network, but they also very over time on any particular network. It would be impossible for a general algorithm analysis to account for such unknowns, therefore authors frequently ignore them. What we now do is take the idealized speedup function for an algorithm, and determine how to modify it to account for the characteristics of a particular network, yielding a more realistic prediction of algorithm performance.

The first task is to separate processing from communications. A distributed program spends its time either communicating or processing[1]. Define a program's ***critical path*** as the sequence of events that determines $T(M)$, a program's execution time on M processors. Of this time, the algorithm spends $T_p(M)$ seconds processing, and $T_c(M)$ seconds communicating, thus $T(M) = T_p(M) + T_c(M)$.

[1] We subsume all non–communications delays, such as disk I/O, into the processing time.

The speedup function then becomes:

$$S(M) = \frac{T(1)}{T_c(M) + T_p(M)}. \tag{5.1}$$

Although we must ultimately analyze the realistic communications delay, let us assume, for the moment, that we have an ideal network with capacity C_n bits per second, and that the time to transmit a packet of K bits is K/C_n seconds. If we know that the number of bits transmitted by events along the critical path is given by a function $B(M)$ (because communications might vary with the number of processors), then we can claim that $T_c(M) = B(M)/C_n$ seconds (assuming communications and processing are not overlapped for events on the critical path). This provides us with a restatement of the previous equation:

$$S(M) = \frac{T(1)}{B(M)/C_n + T_p(M)}. \tag{5.2}$$

We must consider, though, what capacity we really want to use for C_n. In typical networks that connect workstations (e.g. Ethernet and token rings), the basic network has far more capacity than any one machine can use. Because of their computation requirements, the high–level protocols (e.g. TCP/IP, X.400) limit the throughput at any one machine to a fraction of the network's maximum possible throughput, and if the network is run well, this is the primary limitation on communication, not the raw network capacity. Thus, it is better to use the protocol's limits and capacities as our model of the network, rather than capacity of the underlying real network. Through measurement or analysis, one may find the maximum speed, in bits per second, of communication between two processors, C_p, and use this in our formula for speedup:

$$S(M) = \frac{T(1)}{B(M)/C_p + T_p(M)}. \tag{5.3}$$

In deriving C_p, we assume that there is no other traffic on the network (we account for that in the next paragraph), and we assume that C_p is the same between any pair of processors. Note that we need not model the protocol as a queueing system (although we might do so in order to find C_p), because any waiting is caused by the transmission of other data (on the critical path), and this time is counted in the term $B(M)/C_p$.

We do not, in general, have only two processors communicating over the network, but instead have many processors, some of which are involved in the algorithm and some of which are not. We must find the total load on the network, apply this to a model of the underlying communications network (e.g. Ethernet) and determine how much the total traffic slows down communication along the critical path. Let $G_a(M)$ be the load, in bits per second, that the algorithm places on the network while executing on M processors, and assume that traffic from other processors on the network adds a constant additional load of G_o bits per second. With a load of $G_a(M)+G_o$, use the network model to find the factor by which the transmission time increases with load, and call it $L_n(M)$. What would take t seconds to transmit on an unloaded network takes $L_n(M)t$ seconds because of the actual network load. Our communications time must be increased by a like amount, so our new speedup becomes:

$$S(M) = \frac{T(1)}{\frac{B(M)}{C_p} L_n(M) + T_p(M)}. \tag{5.4}$$

We should note, though, that protocol delays are primarily caused by processing, not by network transmission, and high network loads may not affect the protocol processing times. A local area network without broadcast packets is an example of such a situation — the network hardware screens out irrelevant packets. Broadcast packets in the network, however, would require processing time that would slow down the protocol processing. We should multiply G_o by a factor f, $0 \leq f \leq 1$, that tells how much of the global load really affects transmission time, then use $G_a(M) + f\,G_o$ to determine $L_n(M)$.

We have now accounted for communications overhead. We can continue this analysis to account for transient processors and the effect of multitasking with other programs on the processors. Assume that the processors are statistically identical. We know from chapter 2 that the delay caused by transient processors is practically deterministic in many situations. If so, we may then divide the algorithm's processing time by p_a to account for the transient processors. If the processors are multitasking, and if the distributed program gets only a fraction f_m of the processing time, we should multiply the processing time by $1/f_m$ to account for this. Our final expression for speedup is:

$$S(M) = \frac{T(1)}{\frac{B(M)}{C_p} L_n(M) + \frac{T_p(M)}{p_a f_m}}. \tag{5.5}$$

With this equation in hand, we can then model real algorithms and real networks to see if it is accurate. This is one of the most important areas for future research.

5.1.4 Measurements to Compare the Models to Real World.

One noticeable gap in this dissertation is the lack of measurements comparing the transient processor models to real machines. It is not for lack of intent, because one of the purposes of the BBL system was to provide a testbed for measurements. However, MS-DOS is not a sophisticated networking environment, and it is difficult for two programs to use the network simultaneously from one machine. In particular, a widely-used remote file access program was quite incompatible with BBL, so most users could not have BBL running on their machines. Furthermore, laboratory machines, such as those used for the system, are heavily used when the lab is open and completely idle when the lab is closed, giving them markedly different usage patterns from a typical workstation in someone's office. For measurements we really need a new system, built in a good networking environment, to provide a testbed.

5.2 Closing Words.

The use of transient processors is a developing field and plenty of work remains to be done. Given the needs of many people for large amounts of computing capacity, it is a shame that systems to use transient processors are not more widespread. However, the field of distributed processing is still relatively young, and high-speed networks with many powerful machines are a relatively new feature in the computing world, so we hope to see proper use of transient processors in future networks.

Bibliography

[Agr85] Subhash Chandra Agrawal. *Metamodeling: a study of approximations in queueing models.* The MIT Series in Computer Systems. The MIT Press, 1985.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffery D. Ullman. *The Design and Analysis of Compter Algorithms.* Addison-Wesley, 1974.

[Akl85] Selim G. Akl. *Parallel Sorting Algorithms.* Notes and Reports in Computer Science and Applied Mathematics. Academic Press, 1985.

[AS87] Yehuda Afek and Michael Saks. Detecting global termination conditions in the face of uncertainty. In *Principles of Distributed Computing*, 1987.

[BCH79] O. J. Boxma, J. W. Cohen, and N. Huffels. Approximations of the mean waiting time in an m/g/s queueing system. *Operations Research*, 27(6):1115–1126, Nov.-Dec. 1979.

[Bel87] Abdelfettah Belghith. *Response Time and Parallel Processing Systems with Certain Synchronization Constraints.* PhD thesis, University of California, Los Angeles, March 1987.

[BG84] Percy H. Brill and Linda Green. Queues in which customers receive simultaneous service from a random number of servers: A system point approach. *Management Science*, 30(1):51–68, January 1984.

[BP77] P. H. Brill and J. M. Posner. Level crossings in point processes applied to queues: Single-server case. *Operations Research*, 25(4):662–674, July-Aug. 1977.

[BP81] P. H. Brill and M. J. M. Posner. The system point method in exponential queues: A level crossing approach. *Mathematics of Operations Research*, 6(1):31–49, February 1981.

[Bro28] R. Brown. A brief account of microscopical observations made in the months of june, july, and august, 1827, on the particles contained in the pollen of plants; and on the general existence of active molecules in organic and inorganic bodies. *Philos. Mag. Ann. of Philos*, New ser.(4):161–178, 1828.

[Cox62] D. R. Cox. *Renewal Theory.* Methuen and Co., Ltd., London, science paperbacks edition, 1962.

[CT83] M. L. Chaudhry and J. G. C. Templeton. *A First Course in Bulk Queues.* John Wiley and Sons, Inc., 1983.

[DI87] Lorenzo Donatiello and Balakrishna R. Iyer. Closed-form solution for system availability distribution. *IEEE Transactions on Reliability*, R-36(1):45–47, April 1987.

[Dos86] B. T. Doshi. Queueing systems with vacations — a survey. *Queueing Systems*, 1(1):29–67, June 1986.

[DPG62] Jr. D. P. Gaver. A waiting line with interrupted service, including priorities. *Journal of the Royal Statistical Society*, B24:73–90, 1962.

[DS80] Edsger W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, August 1980.

[dSeSG89] Edmundo de Souza e Silva and H. Richard Gail. Calculating availability and performability measures of repairable computer systems using randomization. *Journal of the ACM*, 36(1):171–193, January 1989.

[Ein26] Albert Einstein. *Investigations on the Theory of the Brownian Motion.* Methuen and Co., London, 1926.

[FG86] Awi Federgruen and Linda Green. Queueing systems with service interruptions. *Operations Research*, 34(5):752–768, Sept.-Oct. 1986.

[FR82] Nissim Francez and Michael Rodeh. Achieving distributed termination without freezing. *IEEE Transactions on Software Engineering*, SE-8(3):287–292, May 1982.

[Fra80] Nissim Francez. Distributed termination. *ACM Transactions on Programming Languages and Systems*, 2(1):42–55, January 1980.

[FSK89] Robert Felderman, Eve Schooler, and Leonard Kleinrock. The benevolent bandit laboratory: A testbed for distributed algorithms. *IEEE Journal on Selected Areas in Communications*, 7(2):303–311, February 1989.

[GKP89] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics.* Addison-Wesley, 1989.

[Goy87] Ambuj Goyal. System availability estimator (SAVE) user's manual version 2.0 (External). Technical Report RC 12517 (No. 56267), IBM Watson Research Center, February 1987.

[Har85] J. Michael Harrison. *Brownian Motion and Stochastic Flow Systems.* Series in Probability and Statistics. John Wiley and Sons, Inc., 1985.

[Hid80] T. Hida. *Brownian Motion*, volume 11 of *Applications of Mathematics.* Springer-Verlag, 1980.

[HL69] F. K. Hwang and S. Lin. An analysis of ford and johnson's sorting algorithm. In *Proceedings of the 3rd Annual Princeton Conference on Information Sciences and Systems*, pages 292–296, 1969.

[HS78] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Computer Algorithms.* Computer Science Press, 1978.

[IHPM65] Kiyosi Ito and Jr. Henry P. McKean. *Diffusion Processes and Their Sample Paths*, volume 125 of *Die Grundlehren Der Mathematischen Wissenschaften in Einzeldarstellungen.* Academic Press, 1965.

[JJ59] L. R. Ford Jr. and S. M. Johnson. A tournament problem. *American Mathematical Monthly*, 66(5):387–389, 1959.

[Kle75] Leonard Kleinrock. *Queueing Systems, Volume 1: Theory.* John Wiley and Sons, 1975.

[Kle76] Leonard Kleinrock. *Queueing Systems, Volume 2: Computer Applications.* John Wiley and Sons, 1976.

[Kle85] Leonard Kleinrock. Distributed systems. *Communications of the ACM*, 28(11):1200–1213, November 1985.

[Knu73] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, Mass., 1973.

[KT75] Samuel Karlin and Howard M. Taylor. *A First Course in Stochastic Processes.* Academic Press, second edition, 1975.

[LL89] W. S. Luk and Franky Ling. An analytic/empirical study of distributed sorting on a local area network. *IEEE Transactions on Software Engineering*, 15(5):575–586, May 1989.

[LLM88] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor - a hunter of idle workstations. In *8th International Conference in Distributed Computing Systems*, San Jose, CA, June 1988.

[Mal73] Erik Malløe. Approximation formulae for estimation of waiting-time in multiple-channel queueing system. *Management Science*, 19(6):703–710, February 1973.

[MBM89] G. K. Manacher, T. D. Bui, and T. Mai. Optimum combinations of sorting and merging. *Journal of the ACM*, 36(2):290–334, April 1989.

[MGB74] Alexander Mood, Franklin Graybill, and Duane Boes. *Introduction to the Theory of Statistics*. Series in Probability and Statistics. McGraw-Hill, 1974.

[Mis83] Jayadev Misra. Detecting termination of distributed computations using markers. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 290–294. ACM, August 1983.

[ML87a] Matt W. Mutka and Miron Livny. Profiling workstation's available capacity for remote execution. Computer Sciences Technical Report 697, CS Dept., Univ. of Wisconsin, May 1987.

[ML87b] Matt W. Mutka and Miron Livny. Scheduling remote processing capacity in a workstation-processor bank network. In *7th Internations Conference on Distributed Computing Systems*, page ?, Berlin, West Germany, September 1987.

[Nic87] David A. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 5–12. ACM, November 1987.

[NR76] S. A. Nozaki and S. M. Ross. Approximations in multi-server poisson queues. Technical Report ORC 76-10, Operations Research Center, UC Berkeley, April 1976.

[Sev77] K. C. Sevcik. Priority scheduling disciplines in queueing network models of computer systems. In *Proc. IFIP Congress 77*, pages 565–570, Amsterdam, 1977. North-Holland Publishing Co.

[SH82] John F. Shoch and Jon A. Hupp. The "worm" programs — early experience with a distributed computation. *Communications of the ACM*, 25(3):172–180, March 1982.

[Shi89] Timothy J. Shimeall. Personal communication, 1989.

[SW49] Claude E. Shannon and Warren Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, 1949.

[Thi63] K. Thiruvengadam. Queueing with breakdowns. *Operations Research*, 11(1):62–71, Jan-Feb. 1963.

[WC58] Harrison White and Lee S. Christie. Queueing with preemptive priorities or with breakdown. *Operations Research*, 6(1):79–95, Jan-Feb. 1958.

[Wel66] Mark Wells. Applications of a language for computing in combinatorics. In *Information Processing 65 (Proceeding of the 1965 IFIP Congress)*, pages 497–498, Amsterdam, The Netherlands, 1966. North-Holland.

[Wie76] Norbert Wiener. *Norbert Wiener: Collected Works*, volume 1, chapter 1C, pages 435–758. The MIT Press, 1976.